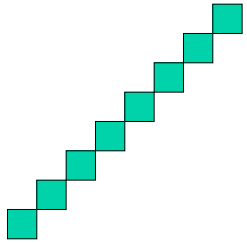
Issues

- End points may not be integral due to clipping (or other reasons)
- Brightness is a function of slope.
- Discretization problems “aliasing” (related to previous point).

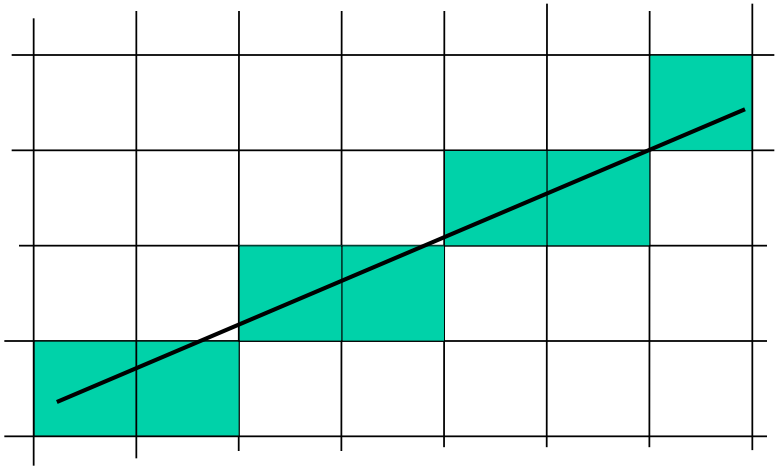
Line drawing--simple line (Bresenham) brightness issues



8 pixels per 8*sqrt(2) length

8 pixels for 8 length
(Brighter)

Line drawing--discretization artifacts (often called aliasing)

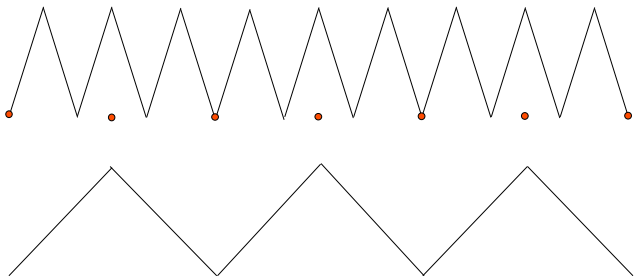


[H&B, pp 214-221]

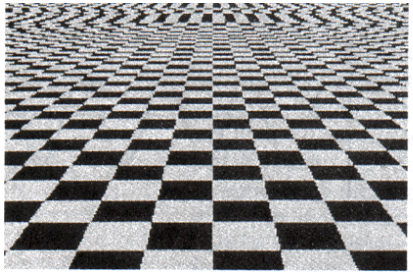
Aliasing

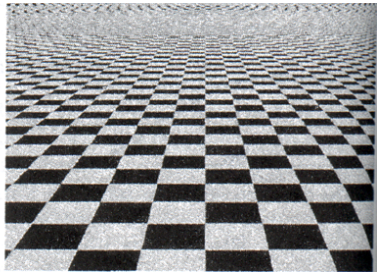
- We are using discrete binary squares to represent perfect mathematical entities
- To get a value for that that square we used a “sample” at a particular discrete location.
- The sample is somewhat arbitrary due to the choice of discretization, leading to the jagged edges.
- Insufficient samples mean that higher frequency parts of the signal can “alias” (masquerade as) lower frequency information.

Aliasing [H&B, figure 4-46]



Aliasing





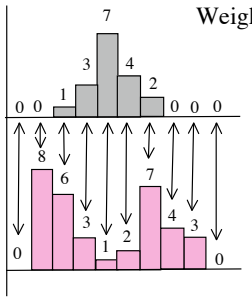
from Watt and Policarpo, The Computer Image

Aliasing (cont)

- Points and lines as discussed so far have no width. To make them visible we concocted a way to sample them based on which discrete cell was closer
- General approach to reducing aliasing is to exploit ability to draw levels of gray between black and white.
- Example--give the line some width; brightness is proportional to area that pixel shares with line
- A more principled approach (which subsumes the above) is to "filter" before sampling.

Linear Filters (background)

- General process: Form new image whose pixels are a **weighted sum** of original pixel values, using the same set of weights at each point.



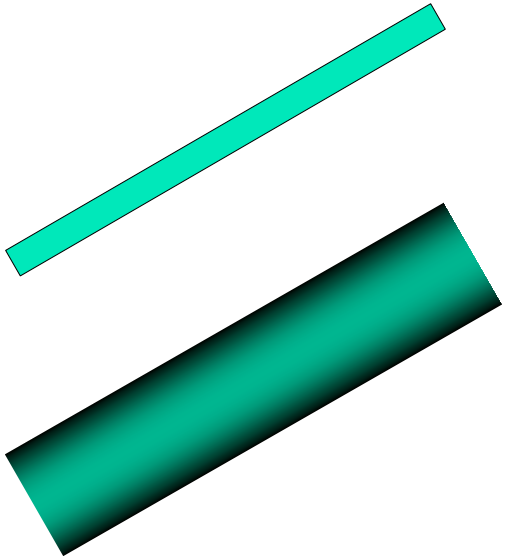
Weights (kernel of filter)

Multiply lined up pairs of numbers and then sum up to get weighted average at the filter location. Then shift the filter and do the same to get the next value.

Signal

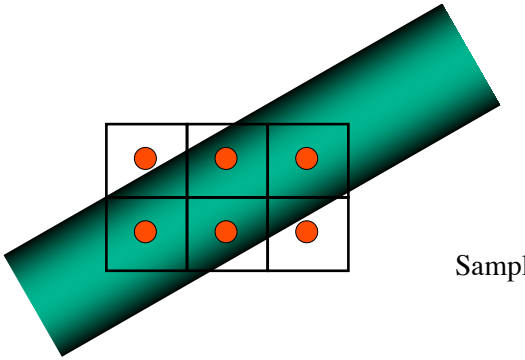
Aliasing via filtering and then sampling

- 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)

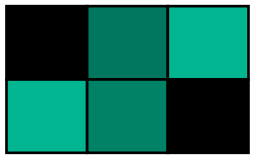


Line with width

Blurred



Sample

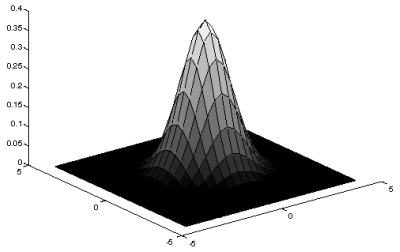


Paint with sample value

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)

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

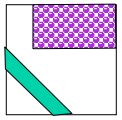
Exact expression is optional

Bonus slide (not in handout)

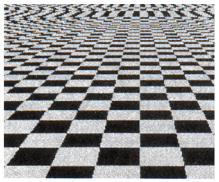
Anti-Aliasing (re-cap)

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

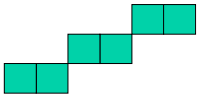
Each pixel can cover a variety of objects to various degrees



Aliasing due to limited sampling rate



Jagged edges due to discrete pixels



Bonus slide (not in handout)

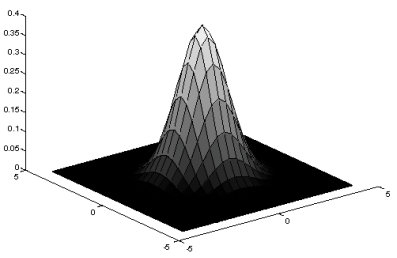
Anti-Aliasing (re-cap)

- 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).

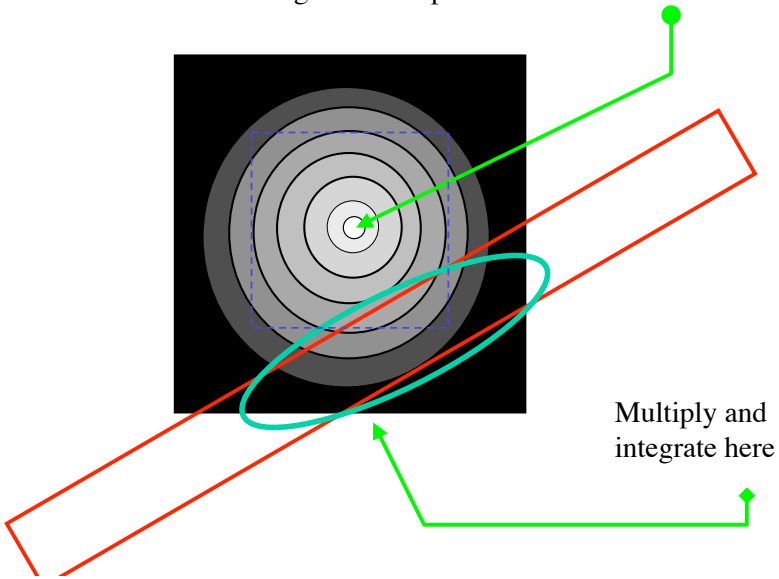
Aliasing via filtering and then sampling

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

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



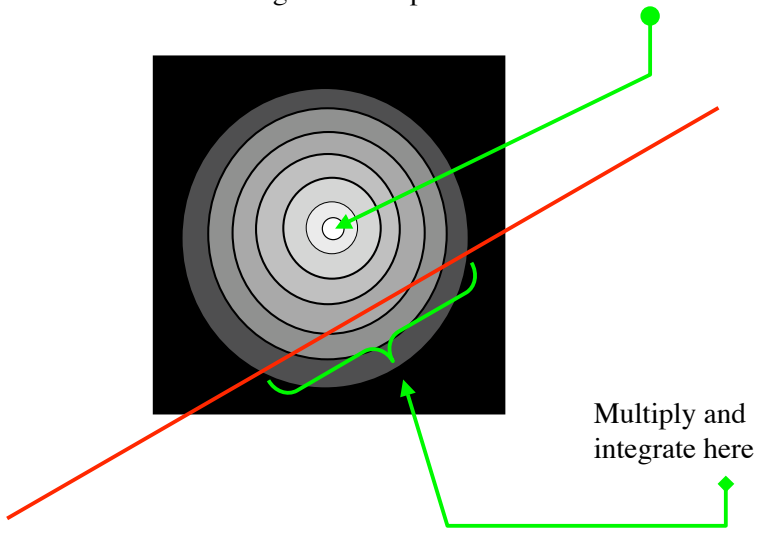
To calculate brightness for pixel with center here



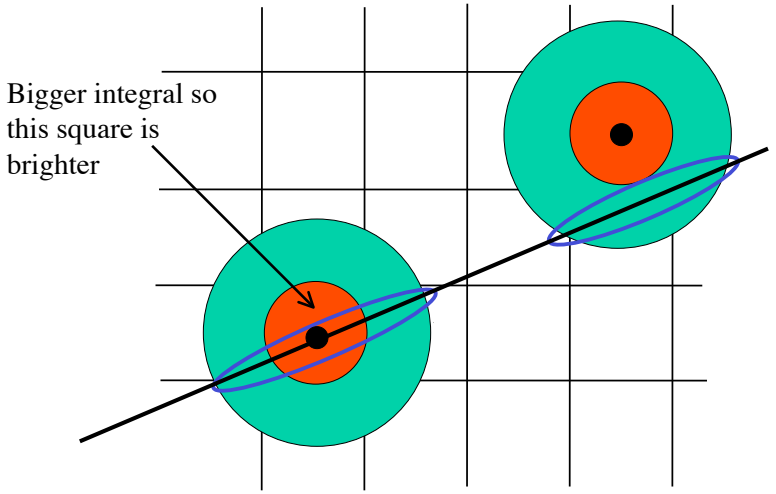
Line with no width

- 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.

To calculate brightness for pixel with center here

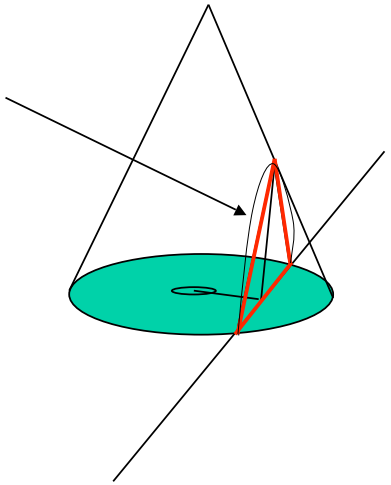


Line with cone example



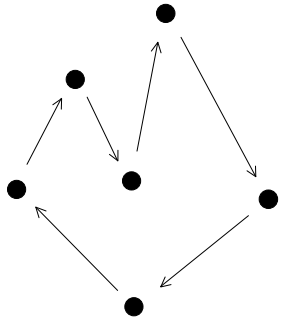
Approximating a Gaussian filter with a cone

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

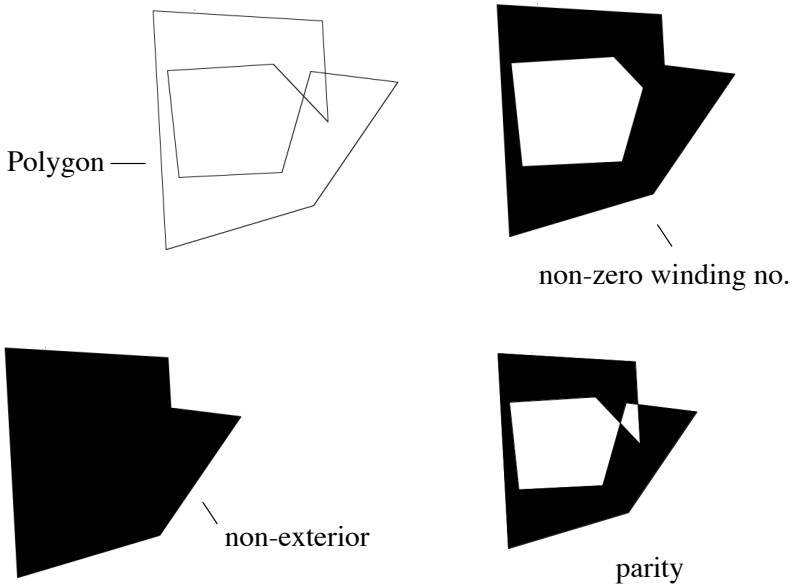
Need

Filling polygons

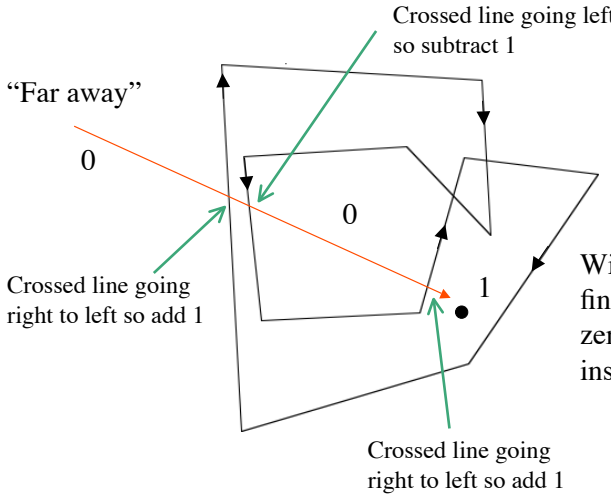
- Polygons are defined by a list of edges - each is a pair of vertices (order counts)
- Assume that each vertex is an integer vertex, and polygon lies within frame buffer
- Need to define what is inside and what is outside

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)

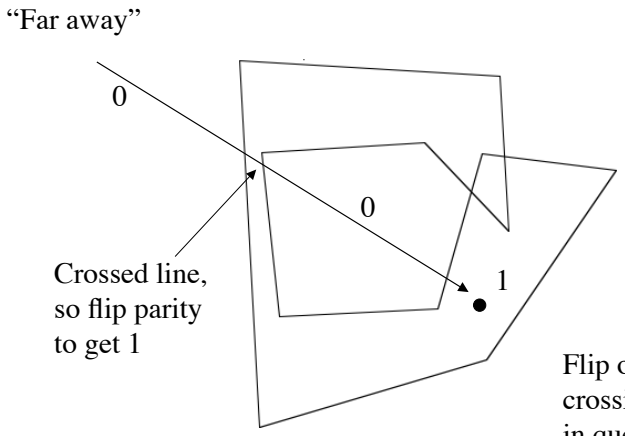


Non-zero winding number--details



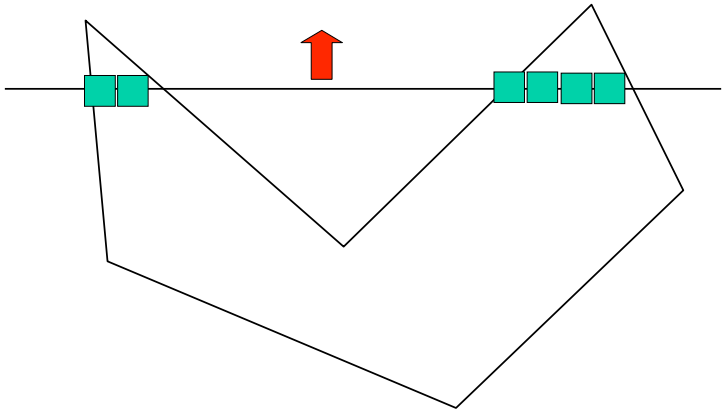
Winding number at final point is non zero, so point is inside by this rule.

Parity rule--details



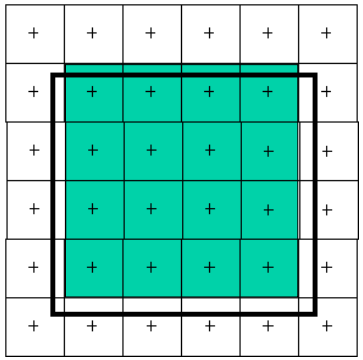
Flip once for each line crossing. Value at point in question is 1, so point is inside.

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 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).

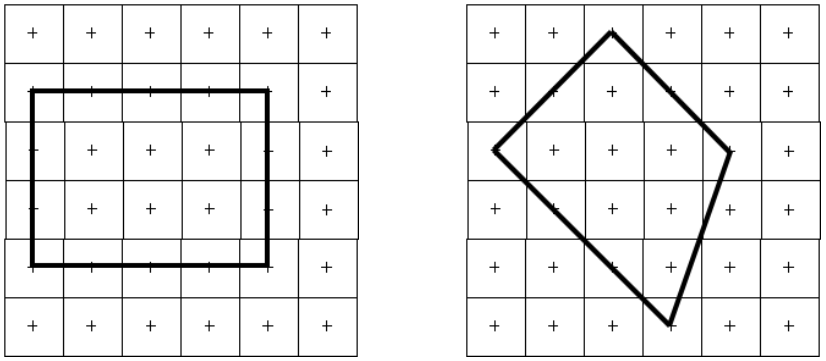
Ambiguous cases

- 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 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)

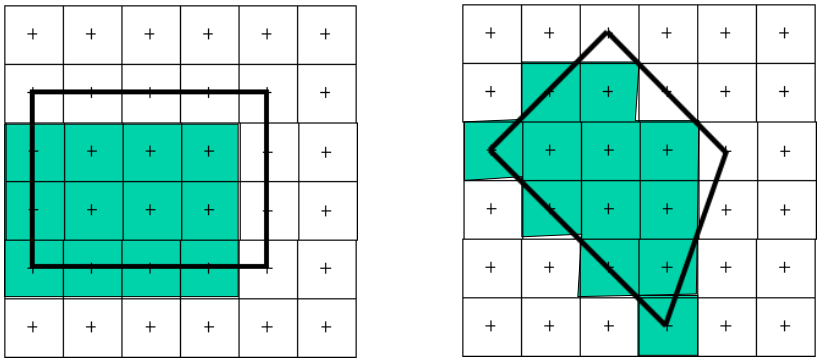
Ambiguous cases

- 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).

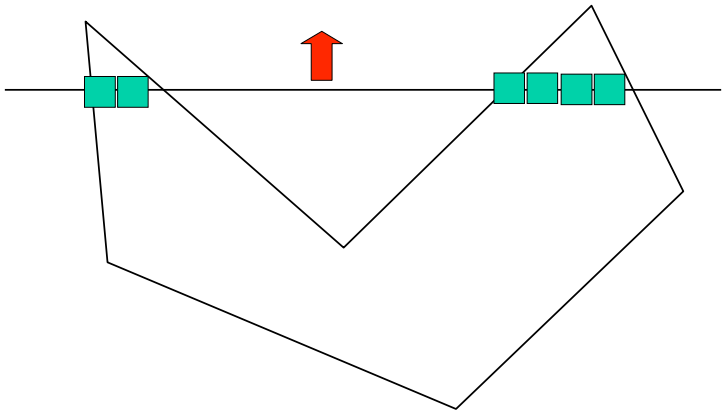
Ambiguous inside cases (?)



Ambiguous inside cases (answer)



Sweep fill

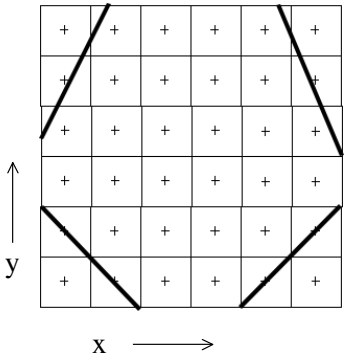


Sweep fill

- 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.

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)



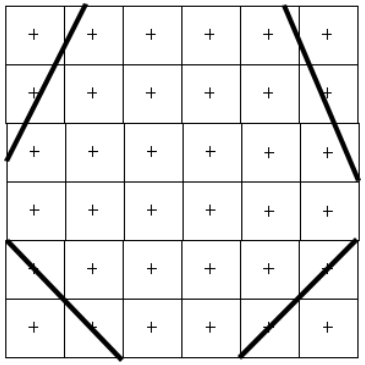
Algorithm

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

The next span - 1

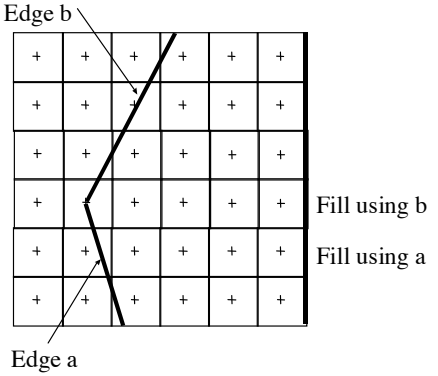
- for an edge, have y=mx+c
- hence, if $y_n=m\ x_n +c$, then $y_{n+1}=y_n+1=m\ (x_n+1/m)+c$
- hence, *if there is no change in the edges*, have:

  x += (1/m)

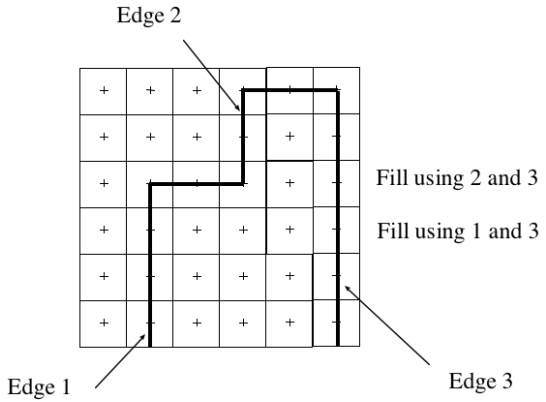


The next span - 2

- Horizontal edges are irrelevant (typically would be pruned at the outset)
- Edge becomes relevant when y >= ymin of edge (note appeal to convention)*
- Edge becomes irrelevant - when y >= ymax of edge (note appeal to convention)*



*Because we add edges and check for irrelevant edges *before* drawing, bottom horizontal edges are drawn, but top ones are not.

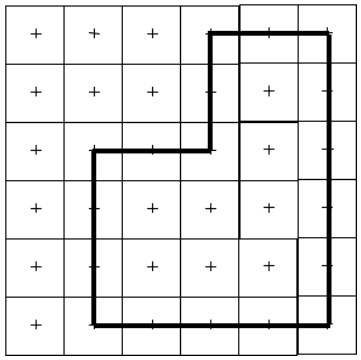


Filling in details -- 1

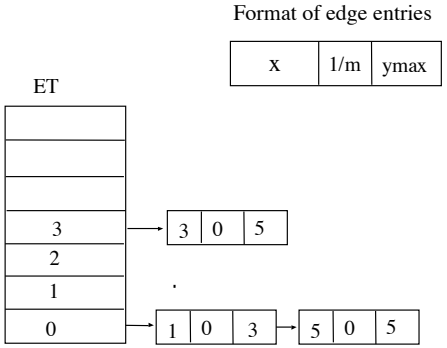
- For each edge store: x-value, maximum y value of edge, 1/m
  - x-value starts out as x value for y_{min}
  - m is never 0 because we ignore horizontal ones
- Keep edges in a table, indexed by minimum y value (Edge Table==ET)
- Maintain a list of active edges (Active Edge List==AEL).

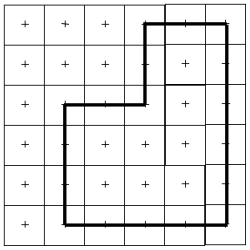
Filling in details -- 2

- For row = min to row=max
  - AEL=append(AEL, ET(row)); (add edges starting at the current row)
  - remove edges whose ymax=row
    - OK since we are assuming integral coordinates; otherwise one would use ceil(ymax)
  - sort AEL by x-value
  - fill spans
    - use parity rule
    - remember convention for integral x_{min} and x_{max}
  - update each edge in AEL
    - x += (1/m)

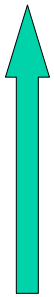


Compute the edge table (ET) to begin. Then fill polygon and update active edge list (AEL) row by row.





AEL just before filling



Row=5

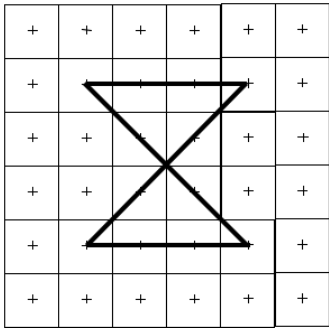
Row=4 | 3 | 0 | 5 | → | 5 | 0 | 5 |

Row=3 | 3 | 0 | 5 | → | 5 | 0 | 5 |

Row=2 | 1 | 0 | 3 | → | 5 | 0 | 5 |

Row=1 | 1 | 0 | 3 | → | 5 | 0 | 5 |

Row=0 | 1 | 0 | 3 | → | 5 | 0 | 5 |



Format of edge entries

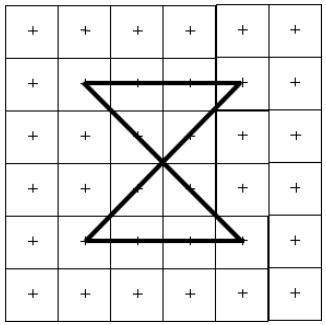
x	1/m	ymax

ET

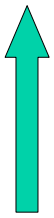
4
3
2
1
0

→ | 1 | 1 | 4 | → | 4 | -1 | 4 |

(This is all there is in the ET---why?)



AEL just before filling



Row=4

Row=3 | 2 | -1 | 4 | → | 3 | 1 | 4 |

Row=2 | 2 | 1 | 4 | → | 3 | -1 | 4 |

Row=1 | 1 | 1 | 4 | → | 4 | -1 | 4 |

Row=0

Comments

- Sort is quite fast, because AEL is usually almost in order.
- Nonetheless, OpenGL limits to convex polygons, so two and only two elements in AEL at any time, and no sorting.
- With additional logic to keep track of what color to use, can fill in many polygons at a time.
- Can be done *without* division/floating point

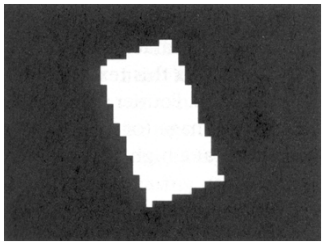
Dodging division and floating point

- 1/m=Dx/Dy, which is a rational number.
- x = x_int + x_num/Dy
- store x as (x_int, x_num),
- then x->x+1/m is given by:
  - x_num=x_num+Dx
  - if x_num >= x_denom
    - x_int=x_int+1
    - x_num=x_num-x_denom

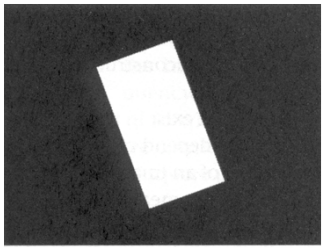
- Advantages:
  - no division/floating point
  - can tell if x is an integer or not (check x_num=0), and get truncate(x) easily, for the span endpoints.

Aliasing/Anti-Aliasing

- Analogous to the case of lines
- Anti-aliasing is done using graduated gray levels computed by smoothing and sampling



Aliasing



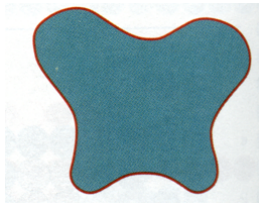
Ideal

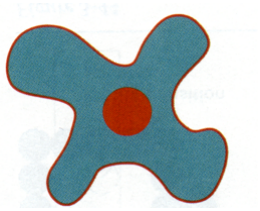
Aliasing/Anti-Aliasing

- Some anti-aliasing approaches implicitly deal with boundary ambiguity
- Problem with “slivers” is really an sampling problem and is handled by filtering and sampling.

Boundary fill

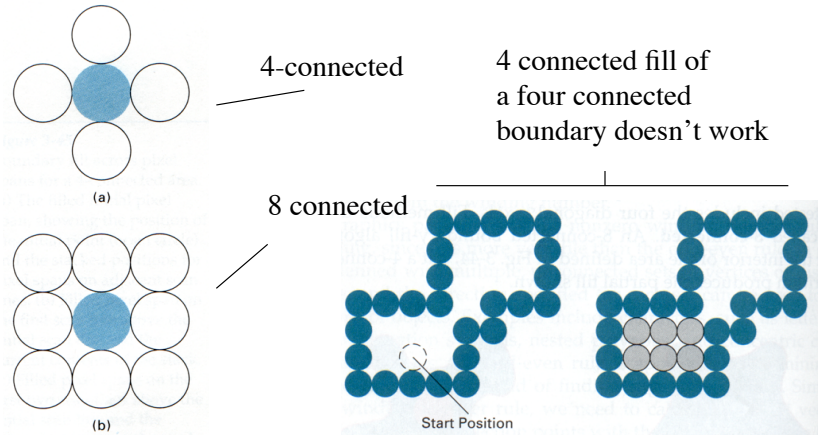
- Basic idea: fill in pixels inside a boundary





- Recursive formulation:
  - to fill starting from an inside point
    - if point has not been filled,
      - fill
      - call recursively with all neighbours that are not boundary pixels

Choice of neighbours is important



Pattern fill

- Use coordinates as index into pattern