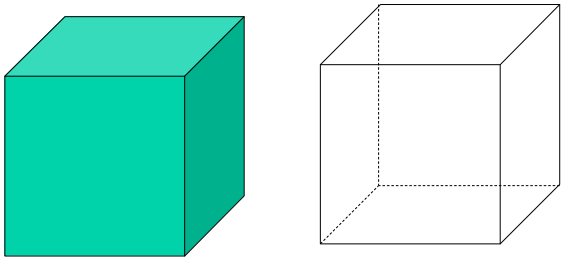
Homework One Issues

- Making movies
  - The intention of assignment one is that the "movie" capability only exists while reading the input file.
  - You can implement this by calling the drawing callback yourself.
  - After EOF, then enter event loop via
    glutMainLoop();
  - If you want to do animation after glutMainLoop() has been called (maybe useful for A2 extensions), have a look at:
    glutIdleFunc();
- Other issues?

Drawing in 2D*



*That's all there really is!

Displaying lines

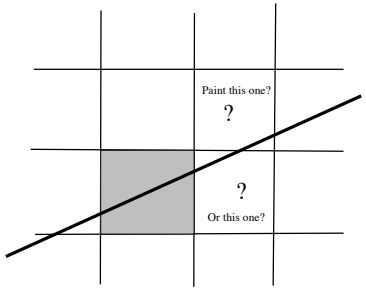
- Assume for now:
  - lines have integer vertices
  - lines all lie within the displayable region of the frame buffer
- Other algorithms will take care of these issues.
- Consider lines of the form y=m x + c, where 0<m<1
- Other cases follow by symmetry
- (Boundary cases, e.g. m=0, m=1 also work in what follows, but are often considered separately, because they can be done very quickly as special cases).

Displaying lines

- Variety of naive (poor) algorithms:
  - step x, compute new y at each step by equation, rounding
  - step x, compute new y at each step by adding m to old y, rounding
- What if we don't assume m<1?

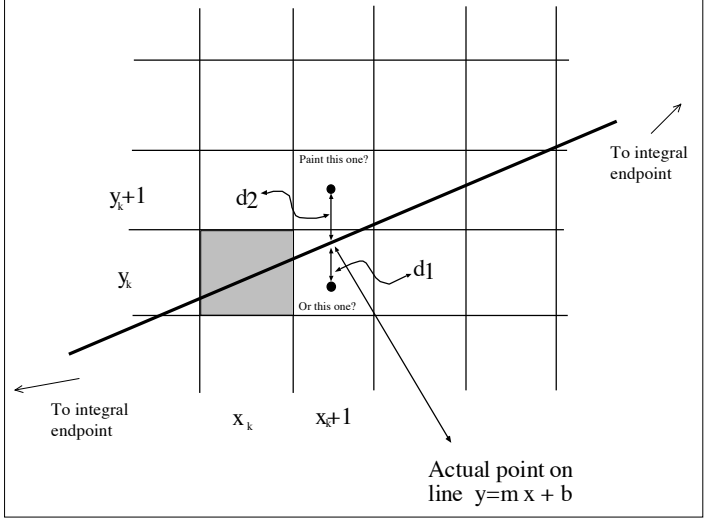
Bresenham's algorithm [H&B, pp 95-99]

- Plot the pixel whose y-value is closest to the line



Bresenham's algorithm [H&B, pp 95-99]

- Plot the pixel whose y-value is closest to the line
- Given (x_k, y_k), must **choose** from either (x_k+1, y_k+1) or (x_k+1, y_k)---recall we are working on case $0<m<1$
- We can derive a "decision parameter" for this choice that is easy to update and cheap to compute (no floating point operations if endpoints are integral).
- "decision parameter" == "determiner"

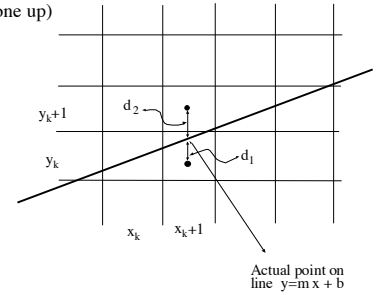


Bresenham's algorithm

- Decision parameter is $d_1 - d_2$

  $d_1 - d_2 < 0$ => plot at y_k (same level as previous)

  otherwise => plot at y_k+1 (one up)



Current integral point is (x_k, y_k)

Line goes through $(x_k + 1, y)$

Is y closer to y_k or y_{k+1} ?

$$d_1 = y - y_k \quad \textit{and} \quad d_2 = (y_k + 1) - y$$

So $d_1 - d_2 = \ ?$

Plugging in $y = m(x_k + 1) + b$

Gives:

$$d_1 - d_2 = \ ?$$

Avoiding Floating Point

From the previous slide

$$d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1$$

Recall that,

$$m = (y_{end} - y_{start}) / (x_{end} - x_{start}) = dy / dx$$

So, for integral endpoints we can avoid division (and floating point ops) if we scale by a factor of dx. Use determiner P_k.

$$\begin{aligned} p_k &= (d_1 - d_2)dx \\ &= (2m(x_k + 1) - 2y_k + 2b - 1)dx \\ &= 2(x_k + 1)dy - 2y_k(dx) + 2b(dx) - dx \\ &= 2(x_k)dy - 2y_k(dx) + 2(dy) + 2b(dx) - dx \\ &= 2(x_k)dy - 2y_k(dx) + \text{constant} \end{aligned}$$

(No division)

Incremental Update

From previous slide

$$p_k = 2(x_k)dy - 2y_k(dx) + \text{constant}$$

Express the next determiner in terms of the previous.

$$\begin{aligned} p_{k+1} &= 2(x_k + 1)dy - 2y_{k+1}(dx) + \text{constant} \\ &= p_k + 2dy - 2(y_{k+1} - y_k) \end{aligned}$$

Either 1 or 0 depending on decision on y

Bresenham algorithm

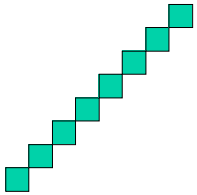
- From previous slide, $p_{k+1} = p_k + 2\,dy - 2\,dx\,(y_{k+1} - y_k)$
- Exercise*: check that $p_0 = 2\,dy - dx$
- Algorithm (for the case that 0<m<1):
  - x=x_start, y=y_start, p=2 dy - dx, **mark** (x, y)
  - until x=x_end
    - x=x+1
    - p>0 ? y=y+1, **mark** (x, y), p=p+2 dy - 2 dx
    - else y=y, **mark** (x, y), p=p+2 dy

*Hint: For p_0, (x_k, y_k) is on the line. Use formula for the line and plug into expression for p_k from two slides back: $p_k = (2m(x_k + 1) - 2y_k + 2b - 1)dx$

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).

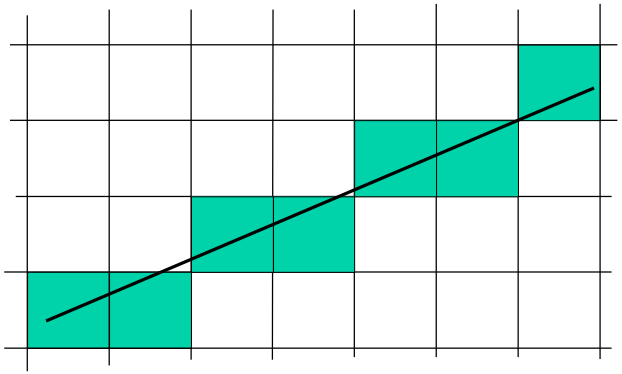
Line drawing--simple line (Bresenham) brightness issues



8 pixels per 8*sqrt(2) length

8 pixels for 8 length
(Brighter)

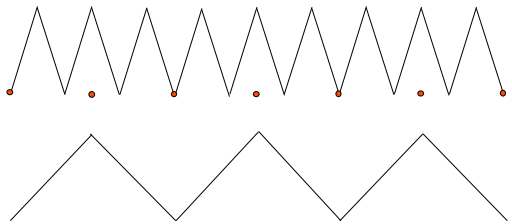
Line drawing--discretization artifacts (often called aliasing)



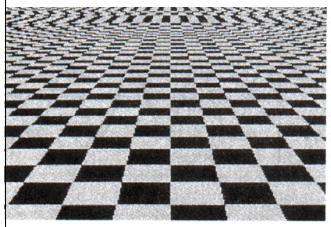
Aliasing [H&B, pp 214-221]

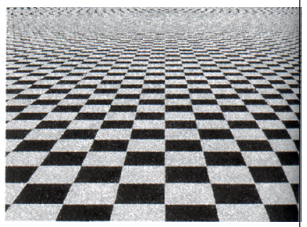
- We are using discrete binary squares to represent perfect mathematical entities
- To get a value for that that square we used a “sample” at a particular discrete location.
- The sample is somewhat arbitrary due to the choice of discretization, leading to the jagged edges.
- Insufficient samples mean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

Aliasing [H&B, figure 4-46]



Aliasing





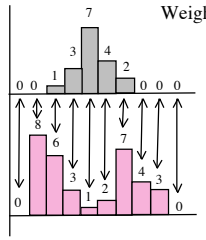
from Watt and Policarpo, The Computer Image

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.



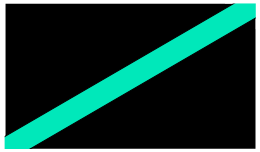
Weights (kernel of filter)

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.

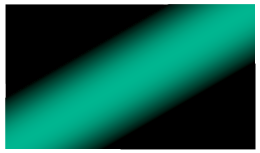
Signal

Aliasing via filtering and then sampling

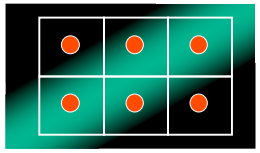
- A filter can be thought of as a weighted average. The weights are given by the filter function. (Examples to come).
- **Conceptually**, we smooth (convolve) the object to be drawn by applying the filter to the mathematical representation.
- This blurs the object, widens the area it occupies
- Now we "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)



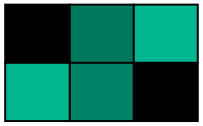
Line with width



Blurred



Sample

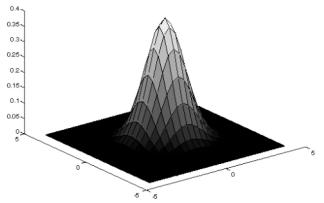


Paint with sample value

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)

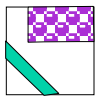
$$g \otimes h = \iint g(x-\xi, y-\eta) h(\xi,\eta) d\xi d\eta$$

Exact expression is optional

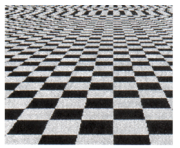
Anti-Aliasing (summary)

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

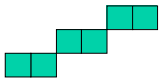
Each pixel can cover a variety of objects to various degrees



Aliasing due to limited sampling rate



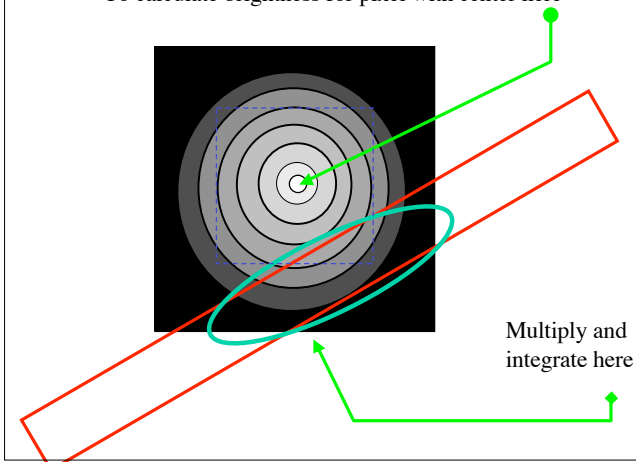
Jagged edges due to discrete pixels



Anti-aliasing via filtering

- 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).

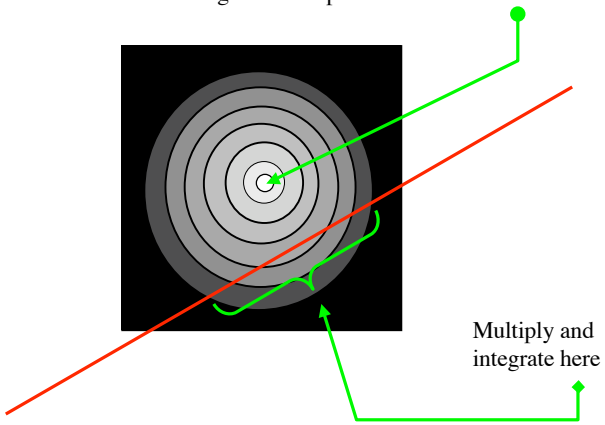
To calculate brightness for pixel with center here



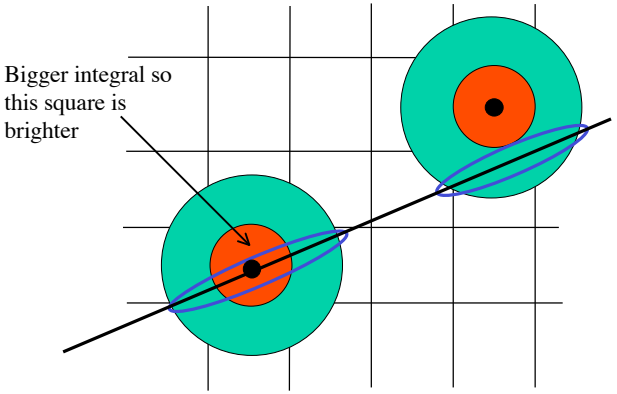
Line with no width

- 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

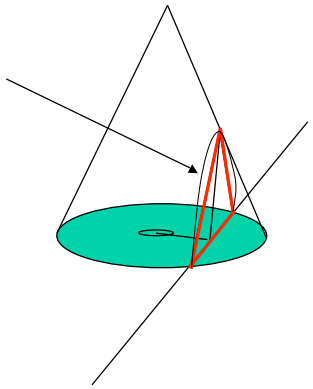


Line with cone example



Approximating a Gaussian filter with a cone

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

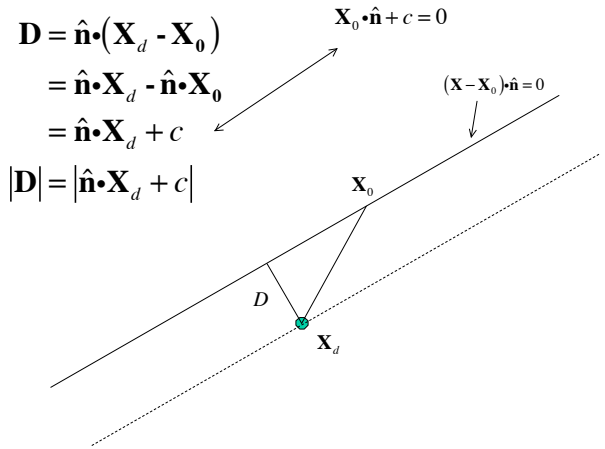
where a^2 + b^2 = 1

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

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

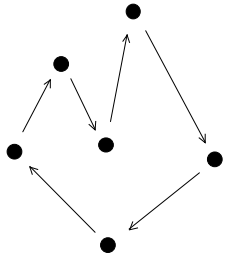
Subtracting these two we get: $(\mathbf{X} - \mathbf{X}_0)\bullet\hat{\mathbf{n}} = 0$

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



Scan converting polygons

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



Have

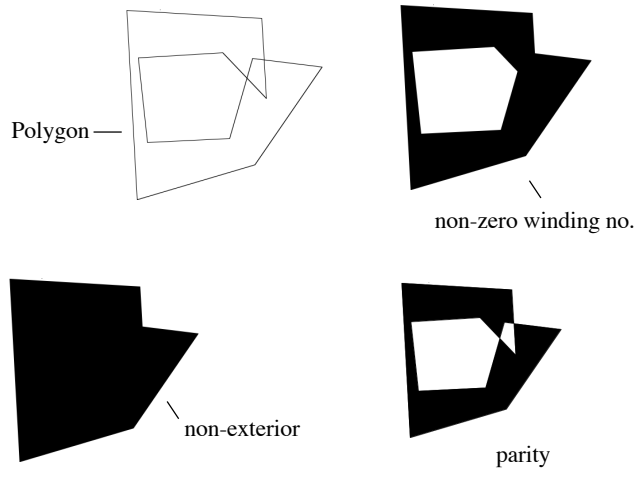
Need

Filling polygons

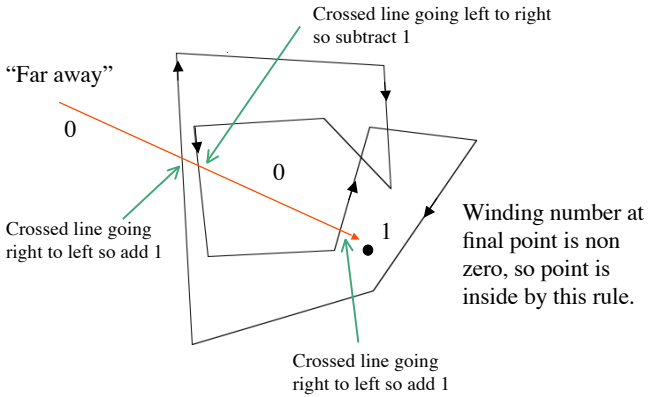
- Polygons are defined by a list of edges - each is a pair of vertices (order counts)
- Assume that each vertex is an integer vertex, and polygon lies within frame buffer
- Need to define what is inside and what is outside

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)

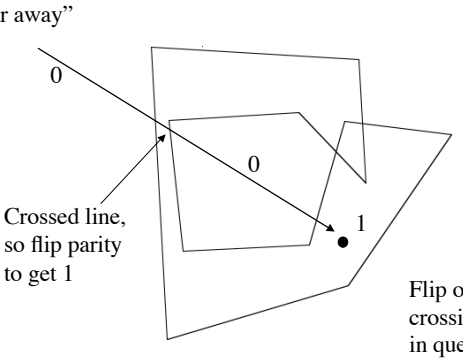


Non-zero winding number--details



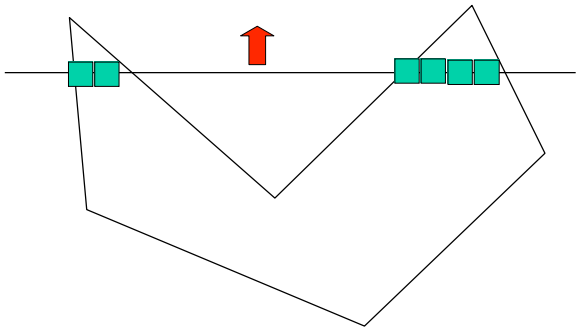
Winding number at final point is non zero, so point is inside by this rule.

Parity rule--details



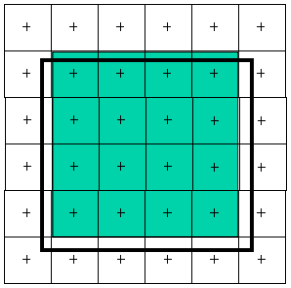
Flip once for each line crossing. Value at point in question is 1, so point is inside.

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 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).

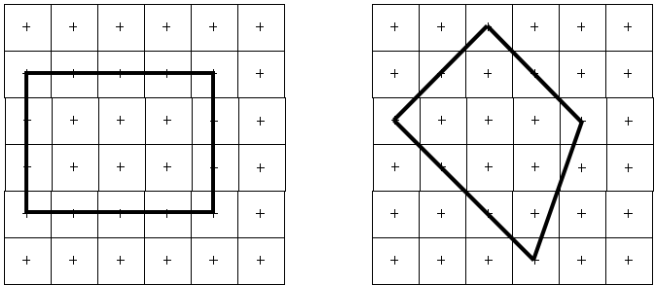
Ambiguous cases

- 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 left and bottom 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)

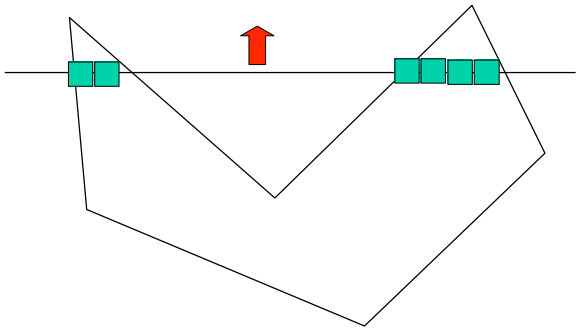
Ambiguous cases (cont.)

- What if it is a vertex between the two cases?
  - 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 (?)



Sweep fill

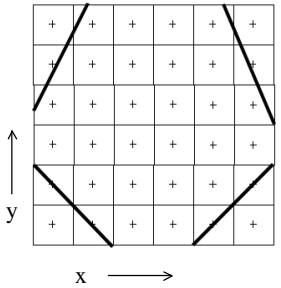


Sweep fill

- 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)



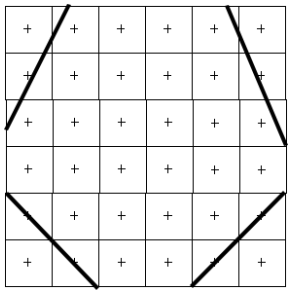
Algorithm

- 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

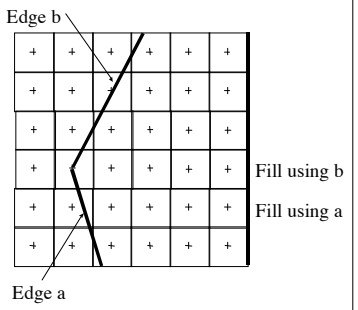
- for an edge, have y=mx+c
- hence, if y_n=m x_n +c, then y_{n+1}=y_n+1=m (x_n+1/m)+c
- hence, *if there is no change in the edges*, have:

  x += (1/m)

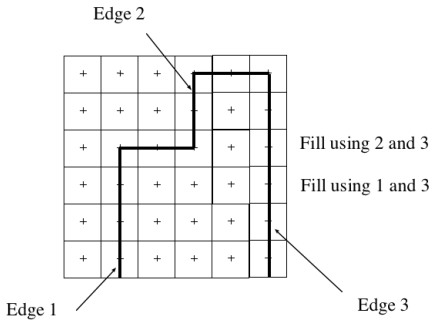


The next span - 2

- 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.

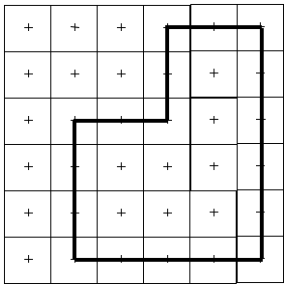


Filling in details -- 1

- 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

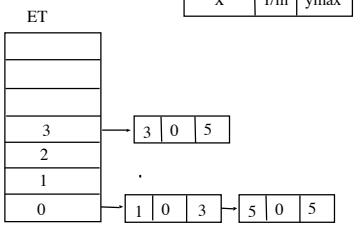
- For row = min to row=max
  - AEL=append(AEL, ET(row)); (add edges starting at the current row)
  - remove edges whose ymax=row
    - OK since we are assuming integral coordinates; otherwise one would use ceil(ymax)
  - sort AEL by x-value
  - fill spans
    - use parity rule
    - remember convention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

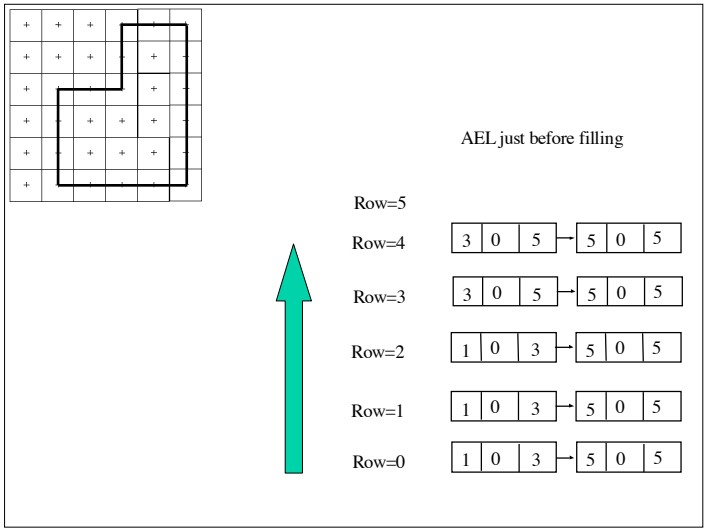


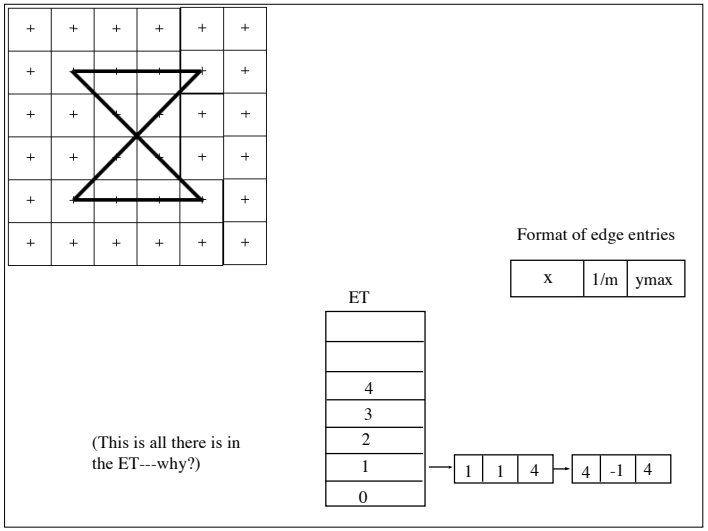
Compute the edge table (ET) to begin. Then fill polygon and update active edge list (AEL) row by row.

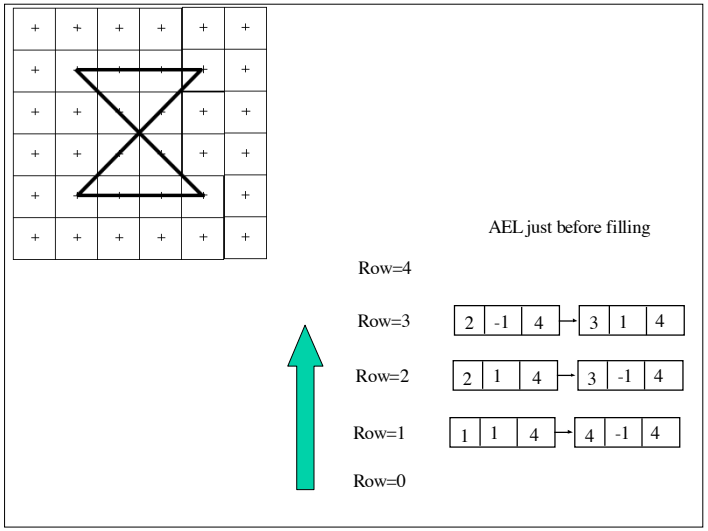
Format of edge entries

x	1/m	ymax









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

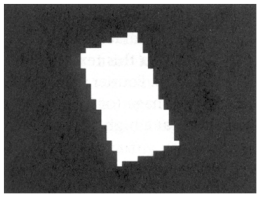
Dodging division and floating point

- 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

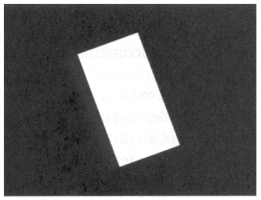
- Advantages:
  - no division/floating point
  - can tell if x is an integer or not (check x_num=0), and get truncate(x) easily, for the span endpoints.

Aliasing/Anti-Aliasing

- Analogous to lines
- Anti-aliasing is done using graduated gray levels computed by smoothing and sampling
- Problem with "slivers" is really an sampling problem and is handled by filtering and sampling.



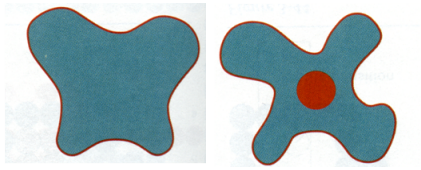
Aliasing



Ideal

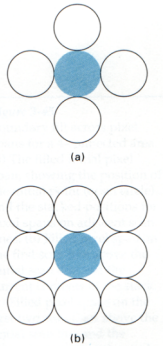
Boundary fill

- 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

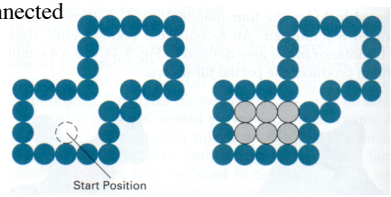
Choice of neighbours is important



4-connected

8 connected

4 connected fill of a four connected boundary doesn't work



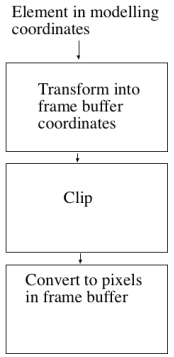
Pattern fill

- Use coordinates as index into pattern

Clipping

- 2D elements are laid out in a convenient (often user based) coordinate system and then transformed to a frame buffer coordinate system.
- Objects that are to be drawn must lie inside frame buffer, and may have to lie inside particular region - e.g. viewport.
- We want to dodge additional expensive operations on objects or parts of objects that won't be displayed.
- How do we ensure that the line/polygon lies inside a region?
- (Answer) Cut them up!

Clipping in the 2D pipeline



Clipping references

Hearn and Baker

C-S (lines): p 317
L-B (lines): p 322
N-L (lines): p 325

S-H (poly): p 331
W-A(poly): p 335

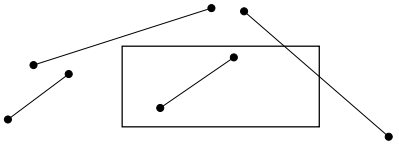
Foley at al.

C-S (lines): p 103
L-B (lines): p 107
N-L (lines): N.A.

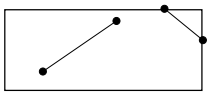
S-H (poly): p 112
W-A(poly): N.A.

Clipping lines

Have

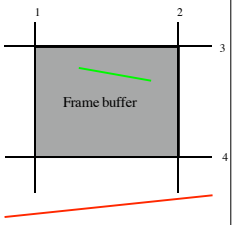


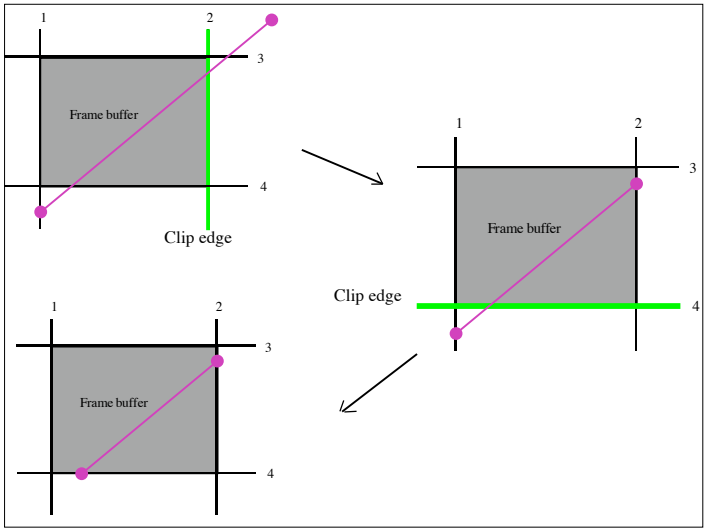
Need



Cohen-Sutherland clipping (lines)

- Clip line against convex region.
- For **each** edge of the region, clip line against that edge:
  - line all on wrong side of any edge? throw it away (**trivial reject**--e.g. red line with respect to bottom edge)
  - line all on correct side of *all* edges? doesn't need clipping (trivial accept--e.g. green line).
  - line crosses edge? **replace** endpoint on wrong side with crossing point (**clip**)

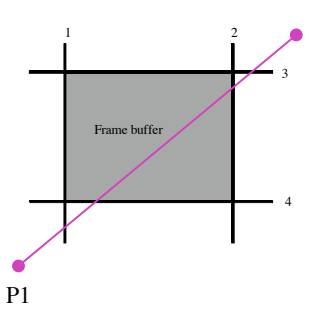




Cohen Sutherland - details

- Only need to clip line against edges where one endpoint is inside and one is outside.
- The state of the *outside* endpoint (e.g., in or out, w.r.t a given edge) changes due to clipping as we proceed--need to track this.
- Use "outcode" to record endpoint in/out wrt each edge. One bit per clipping edge, 1 if out, 0 if in.

Outcode example



Outcode for P1?

Outcode for P2?

Cohen Sutherland - details

- Trivial reject condition?
- Trivial accept condition?
- Clipping line against vertical/horizontal edge is easy:
  - line has endpoints (x_s, y_s) and (x_e, y_e)
  - e.g. (vertical case) clip against x=a gives the point?
  - new point replaces the point for which outcode() is true
- Algorithm is valid for any convex clipping region (intersections are slightly more difficult)

Cohen Sutherland - Algorithm

- Compute outcodes for endpoints
- While not trivial accept and not trivial reject:
  - clip against a problem edge (i.e. one for which an outcode bit is 1)
  - compute outcodes again
- Return appropriate data structure

Cyrus-Beck/Liang-Barsky clipping

- Parametric clipping: consider line in parametric form and reason about the parameter values
- More efficient, as we don't compute the coordinate values at irrelevant vertices

- Line is:

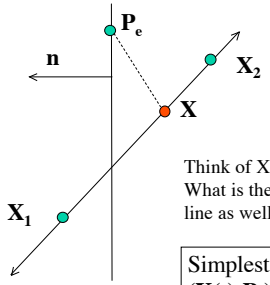
$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + t \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}$$

$$\Delta x = x_2 - x_1$$
$$\Delta y = y_2 - y_1$$

Cyrus-Beck/Liang-Barsky clipping

- Consider the parameter values, t, for each clip edge
- Only t inside (0,1) is relevant
- Assumptions
  - $\mathbf{X}_1 != \mathbf{X}_2$
  - Ignore case where line is parallel to a clip edge (has no effect, but would lead to divide by zero).
  - We have a normal, **n**, for each clip edge pointing outward
  - For axis aligned rectangle (the usual case) these are?

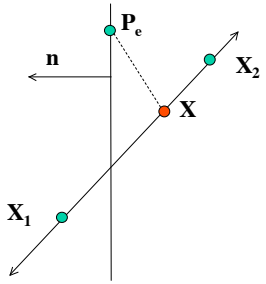
Computing t for intersection point



Think of X moving along the line shown. What is the condition that it is on the other line as well (i.e., intersects?)

Simplest to work from condition $(\mathbf{X}(t)\text{-}\mathbf{P}_e)\bullet n=0$

Computing t for intersection point



Set

$$\mathbf{D} = \mathbf{X}_2 - \mathbf{X}_1$$

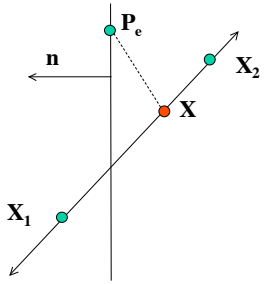
Then

$$\mathbf{X} = \mathbf{X}_1 + t\mathbf{D}$$

And condition is

$$(\mathbf{P}_e - (\mathbf{X}_1 + t\mathbf{D})) \bullet \mathbf{n} = \mathbf{0}$$

Computing t for intersection point, X



Condition

$$(\mathbf{P}_e - (\mathbf{X}_1 + tD)) \bullet \mathbf{n} = \mathbf{0}$$

Rearrange

$$(\mathbf{P}_e - \mathbf{X}_1) \bullet \mathbf{n} = t\mathbf{D} \bullet \mathbf{n}$$

And solve

$$t = \frac{(\mathbf{P}_e - \mathbf{X}_1) \bullet \mathbf{n}}{\mathbf{D} \bullet \mathbf{n}}$$

Computing t for intersection point

From previous slide $t = \frac{(\mathbf{P}_e - \mathbf{X}_1) \bullet \mathbf{n}}{\mathbf{D} \bullet \mathbf{n}}$

This simplifies greatly for axis aligned rectangles

Consider left edge. Now $\mathbf{n}$=? and $\mathbf{P}_e$=?

And t = **?**

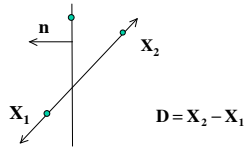
- All four special cases can expressed by: $t = \frac{q_k}{p_k}$
- Where

$$p_1 = -\Delta x \quad q_1 = x_1 - x_{\min}$$
$$p_2 = \Delta x \quad q_2 = x_{\max} - x_1$$
$$p_3 = -\Delta y \quad q_3 = y_1 - y_{\min}$$
$$p_4 = \Delta y \quad q_4 = y_{\max} - y_1$$

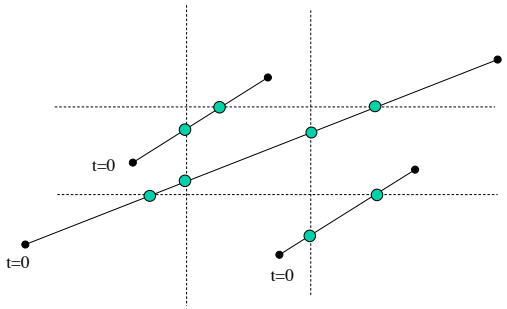
- Faster derivation for this special case?

Cyrus-Beck/Liang-Barsky (cont)

- Next step: Use the t's to determine the clip points
- Recall that only t in (0,1) is relevant, but we need additional logic to determine clip endpoints from multiple t's inside (0,1).
- We imagine going from X1 to X2 and classify intersections as either potentially entering (PE) or potentially leaving (PL) if they go across a clip edge from outside in or inside out.
- This is easily determined from the sign of **D•n** which we have already computed.



PE vs PL example



Cyrus-Beck/Liang-Barsky--Algorithm

- Compute incoming (PE) *t* values, which are q_k/p_k for each $p_k<0$
- Compute outgoing (PL) *t* values, which are q_k/p_k for each $p_k>0$
- Parameter value for small *t* end of the segment is:

  t_{small}= max(0, incoming values)
- Parameter value for large *t* end of the segment is:

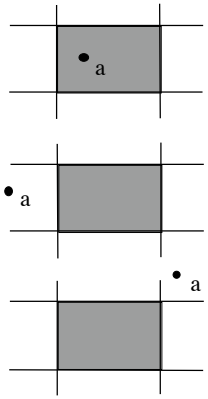
  t_{large}=min(1, outgoing values)
- If $t_{small}<t_{large}$, there is a segment portion in the clip window - compute endpoints by substituting these two *t* values (how)?
- Otherwise reject because it is outside.

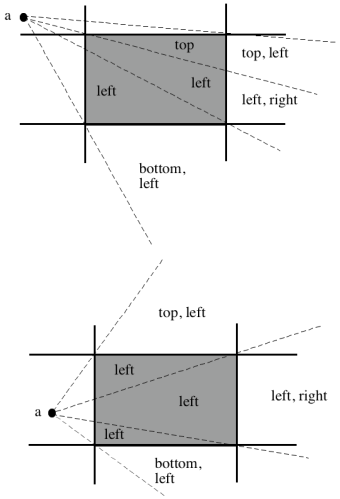
Cyrus-Beck/Liang-Barsky--Notes

- Works fine if clipping window is not an axis-aligned rectangle. Computing the *t* values is just more expensive.
- **Bibliographic note**: Original algorithm was Cyrus-Beck (close to what we have done here). A very similar algorithm was independently developed later by Liang-Barsky with some additional improvements for identifying early rejects as the *t* values are computed.

Nicholl-Lee-Nicholl clipping

- Fast specialized method
- We will just outline the basic idea
- Consider segment with endpoints: a, b
- Cases:
  - a inside
  - a in edge region
  - a in corner region
- For each case, we generate specialized test regions for b
- Which region b is in is determined by simple"which-side" tests.
- The region b is in determines which edges need to be clipped against.
- Speed is enhanced by good ordering of tests, and caching intermediate results





NLN clipping: Compute the area that b is in, and clip the segment ab against the edges specified.