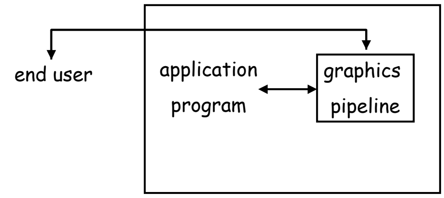
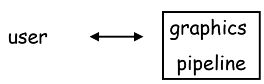
cs155 – z sweedyk

graphics pipeline systems

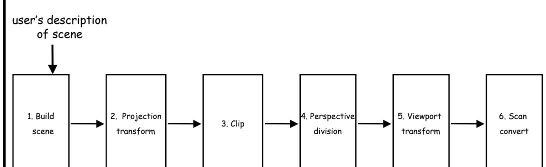
who's who



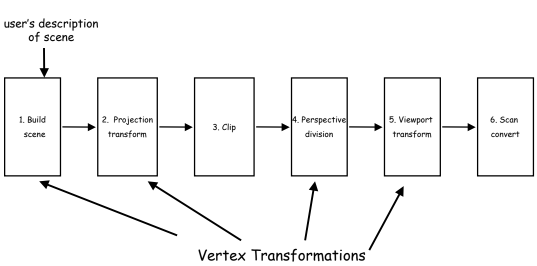
who's who for today



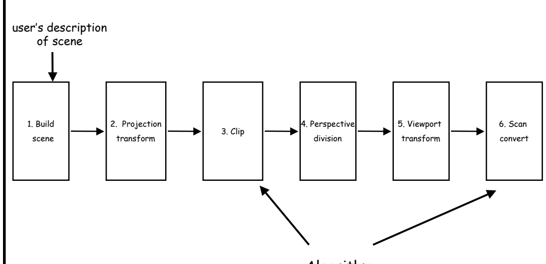
graphics pipeline



graphics pipeline



graphics pipeline



graphics pipeline overview

- simplified pipeline
- general pipeline

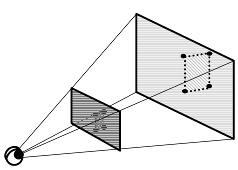
graphics pipeline overview

- simplified pipeline
- general pipeline

- geometric primitives defined in standardized, homogenous world coordinates
- orthographic projection
- standardized view volume
- standard viewport transformation

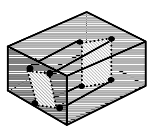
projection

perspective projection



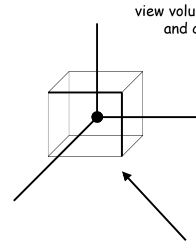
orthographic projection

(viewer at infinity)



standard orthographic view volume

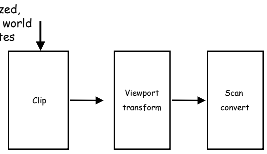
view volume is 2x2x2, axis-aligned, and centered at the origin



rendered image is the parallel projection of scene onto front plane

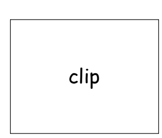
simplified graphics pipeline

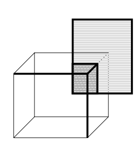
user's description of scene in standardized, homogenous world coordinates



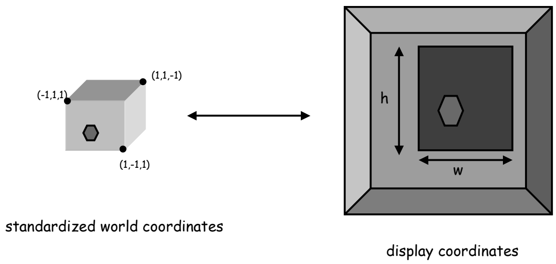
graphics pipeline: clip

eliminate "outside" primitive

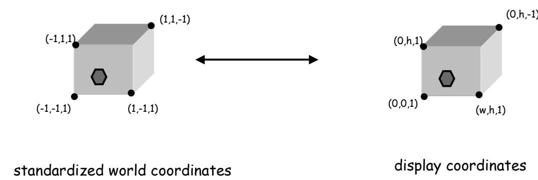




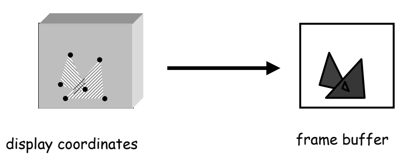
viewport transformation



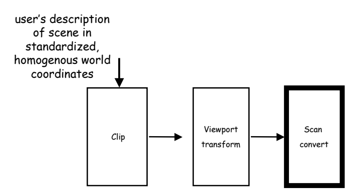
viewport transformation



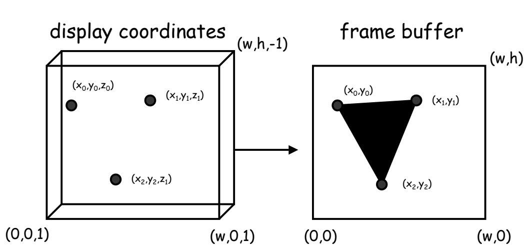
scan conversion



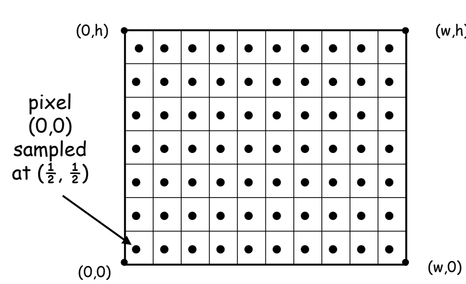
simplified graphics pipeline



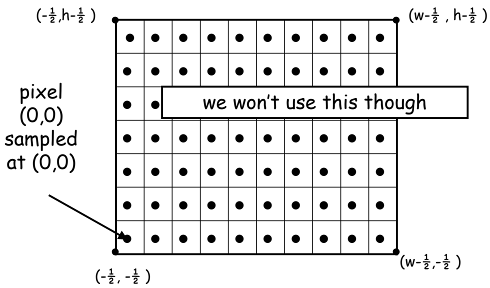
scan conversion



frame buffer coordinates

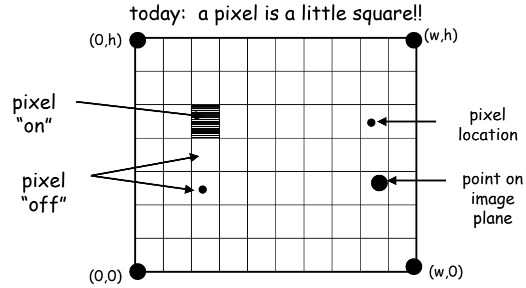


frame buffer coordinates: alternative



frame buffer coordinates

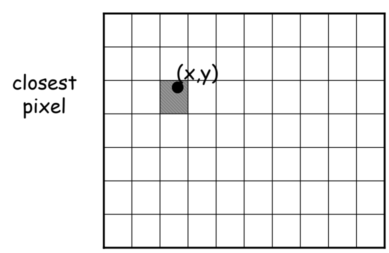
today: a pixel is a little square!!



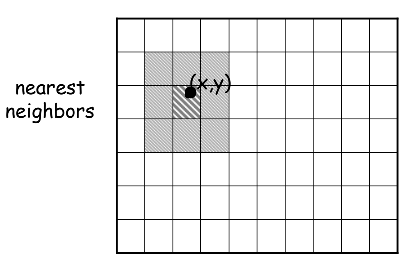
scan conversion

- points
- line segments
- polygons

scan conversion: point



scan conversion: point

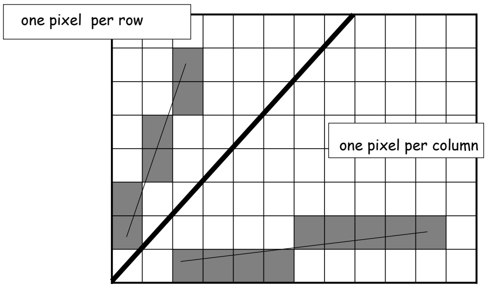


line segments

scan converting line segments

- naïve algorithm
- midpoint algorithm
- bresenham's algorithm

1-pixel wide lines

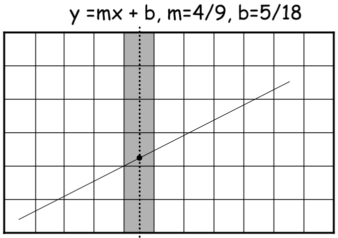


scan conversion

- input: endpoint coordinates
- output: pixels to turn on (and their color) for a 1-pixel wide line segment

endpoints: ($\frac{1}{2}$, $\frac{1}{2}$) and ($9\frac{1}{2}$,$4\frac{1}{2}$)

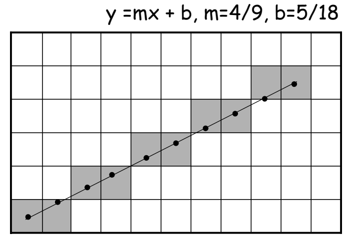
y =mx + b, m=4/9, b=5/18



for each column i
compute the y-intercept at $x=i+\frac{1}{2}$

endpoints: ($\frac{1}{2}$, $\frac{1}{2}$) and ($9\frac{1}{2}$,$4\frac{1}{2}$)

y =mx + b, m=4/9, b=5/18



for (i=0…9)
turn on pixel
$(i, \lfloor m*(i+\frac{1}{2})+b \rfloor)$

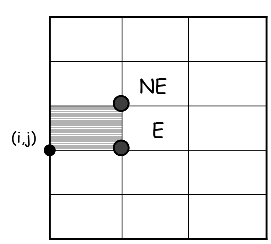
naïve algorithm

- input: endpoints (x_0, y_0) to (x_1, y_1)
  for now we'll assume $x_0<x_1$!
- line: $y = mx + b$ where $m = (y_1 - y_0)/(x_1 - x_0)$, $b=y_0-mx_0$

for $i=\lfloor x_0 \rfloor \ldots \lfloor x_1 \rfloor$
  turn on pixel $(i, \lfloor m(i+\frac{1}{2}) + b \rfloor)$

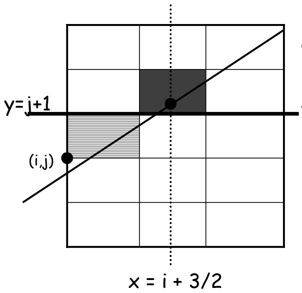
is there a better/faster algorithm?
yes, we can avoid (almost all) multiplication!

midpoint algorithm: $0 \le m \le 1$



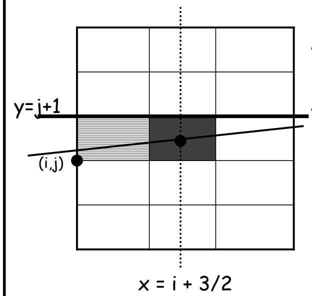
- suppose we've just turned on pixel (i,j)
- next we'll turn on
  NE: (i+1,j+1) or
  E: (i+1, j)

midpoint algorithm: $0 \le m \le 1$



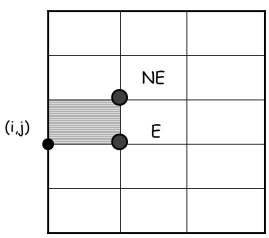
- suppose we've just turned on pixel (i,j)
- next we'll turn on
  - **NE: if the y intercept at x=i+3/2 is at least j+1**

midpoint algorithm: $0 \le m \le 1$



- suppose we've just turned on pixel (i,j)
- next we'll turn on
  - NE: if the y intercept at x=i+3/2 is at least j+1
  - **E: otherwise**

midpoint algorithm: $0 \le m \le 1$

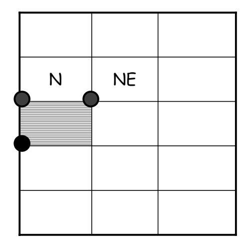


- suppose we've just turned on pixel (i,j)
- next we'll turn on
  - NE: if $j+1 \le m(i+3/2) + b$
  - E: otherwise

midpoint algorithm: other cases

- similar rules

e.g. m>1



advantage of midpoint algorithm

if the endpoints of the line segment have integer coordinates we can avoid floating point operations

this was a big deal in the dark ages!

avoiding (almost all) multiplication (0≤m≤1)

- we just turned on pixel (i,j)
- next we'll turn on
  NE: if $j+1 \le m(i+3/2) + b$
  E: otherwise

$$j+1 \le m(i+3/2) + b \quad \leftrightarrow \quad \Delta_x(j+1) \le \Delta_y (i+3/2) + \Delta_x b$$

where $\Delta x = x_1 - x_0$ and $\Delta y = y_1 - y_0$

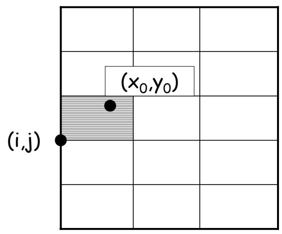
$$\leftrightarrow \quad 2\Delta_x(j+1) \le \Delta_y (2i+3) + 2\Delta_x b$$

$$\leftrightarrow \quad \underbrace{2\Delta_x(j+1) - \Delta_y (2i+3) - 2\Delta_x b}_{d} \le 0$$

we can compute d incrementally without multiplication

bresenham's algorithm (0≤m≤1)

1. turn on (i,j) where $i = \lfloor x_0 \rfloor$ $j = \lfloor y_0 \rfloor$



//find first pixel

bresenham's algorithm (0≤m≤1)

2. $d = 2\Delta_x(j+1) - \Delta_y (2i+3) - 2\Delta_x b$

// initialize d

bresenham's algorithm (0≤m≤1)

```
3. while i ≤ x1 {
       if d ≤ 0     // go NE
       {
       
       }
       else     // go E
       {
       
       }
   }
```

bresenham's algorithm (0≤m≤1)

```
3. while i ≤ x1 {
       if d ≤ 0     // go NE
       {
           i++, j++
           turn on pixel (i,j)
           update d
       }
       else     // go E
       {
           i++
           turn on pixel (i,j)
           update d
       }
   }
```

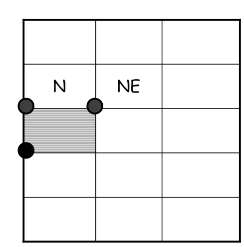
bresenham's algorithm (0≤m≤1)

```
3. while i ≤ x1 {
       if d ≤ 0     // go NE
       {
           i++, j++
           turn on pixel (i,j)
           d = d + 2Δx - 2Δy
       }
       else     // go E
       {
           i++
           turn on pixel (i,j)
           d = d - 2Δy
       }
   }
```

bresenham's algorithm: other cases

- similar rules

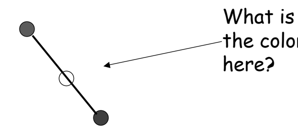
e.g. m>1



scan conversion

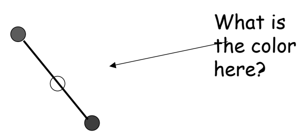
- input: endpoint coordinates
- output: pixels to turn on for a 1-pixel wide line segment **+ pixel color**

shading models



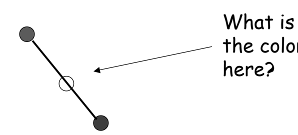
Color is defined at vertices!!!!!!

flat shading



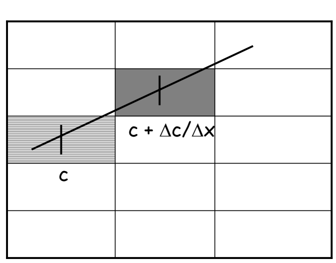
use color of first vertex

smooth (gouraud) shading



interpolate

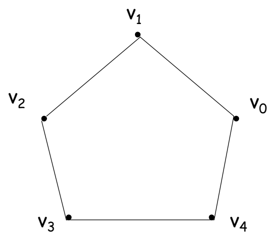
interpolation computation



scan conversion

- points
- line segments
- **polygons**

polygon: v_0,v_1,v_2,v_3,v_4



polygon: scan conversion

polygon(v_0, …, v_{n-1})
  for i=0 to n-1
    draw-line-segment(p_i,$p_{i+1 \text{ mod } n}$)

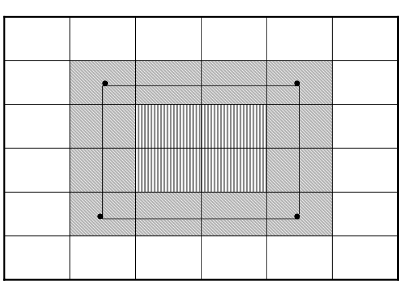
scan conversion

- points
- line segments
- polygons
  - filled

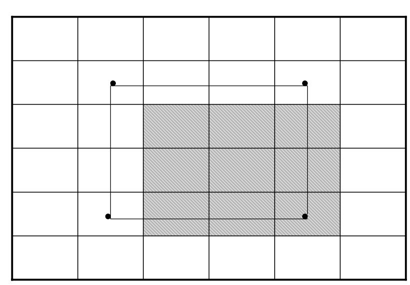
scan conversion

- input: vertex coordinates
- output: **pixels to turn on** (and their color) **for filled polygon**

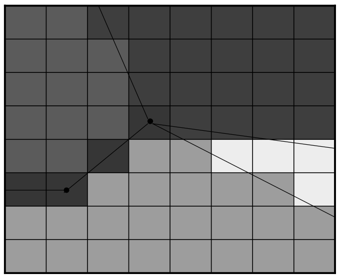
which pixels should be on?



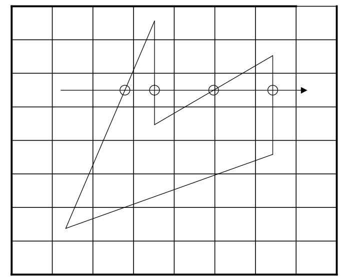
here we get the same size!



tesselating polygons



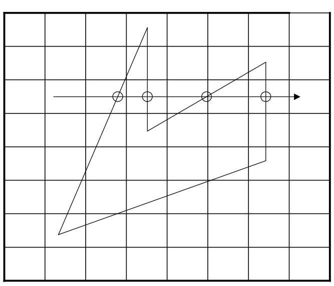
scan line algorithm



for each scan line

1. find edge/scan line intersection points
2. order by x-coordinate
3. use odd-even test to turn on pixels

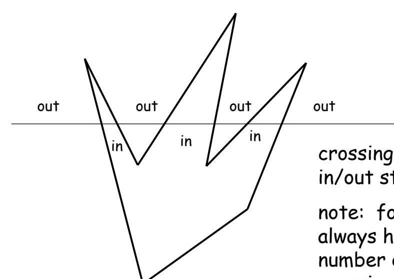
scan line algorithm



for each scan line

1. find edge/scan line intersection points
2. order by x-coordinate
3. **use odd-even test to turn on pixels**

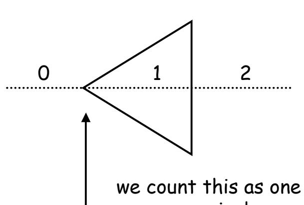
odd-even test



crossing edge changes in/out state

note: for polygon we'll always have an even number of edge crossings

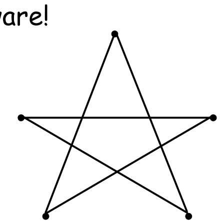
odd-even test



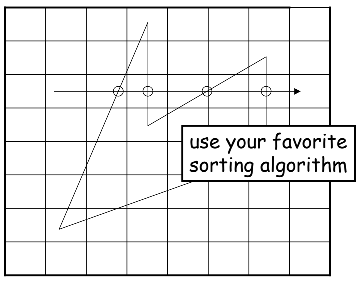
we count this as one crossing!

odd-even test

buyer beware!



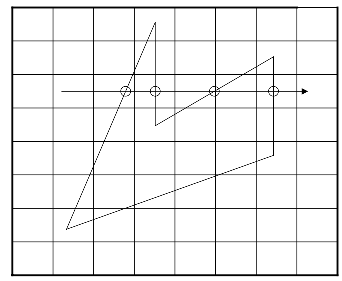
scan line algorithm



for each scan line

1. find edge/scan line intersection points
2. **order by x-coordinate**
3. use odd-even test to turn on pixels

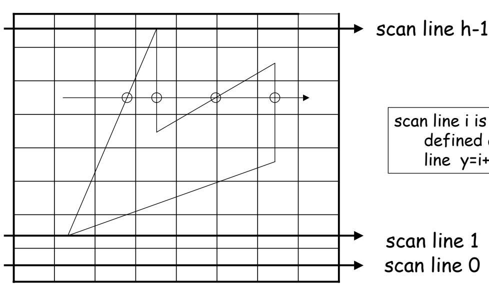
scan line algorithm



for each scan line

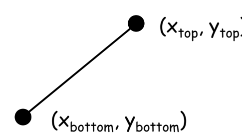
1. **find edge/scan line intersection points**
2. order by x-coordinate
3. use odd-even test to turn on pixels

scan line notation



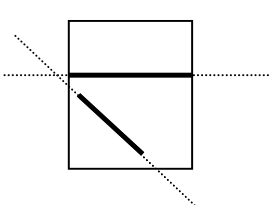
scan line i is defined as the line $y=i+\frac{1}{2}$

edge notation



$y_{bottom} \leq y_{top}$

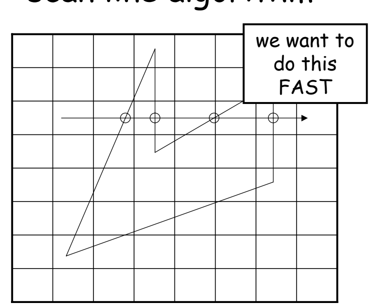
naïve algorithm



for each edge of polygon

- compute intersection of current scan line & edge line
- check if intersection is on edge segment

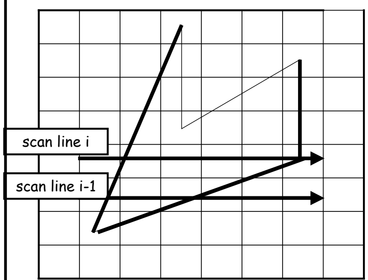
scan line algorithm



for each scan line

1. **find edge/scan line intersection points**
2. order by x-coordinate
3. use odd-even test to turn on pixels

exploit coherence



let E_i be the edges that intersect scan line i

then E_i =

E_{i-1}

+ edges with i- ½ < y_{bottom} ≤ i + ½

- edges with y_{top} ≤ i + ½

data structures

- **edge table**
  - for each scan line i a list of edges with $i - \frac{1}{2} < y_{bottom} \le i + \frac{1}{2}$
- active edge list
  - edges intersecting current scan line

edge table (et)

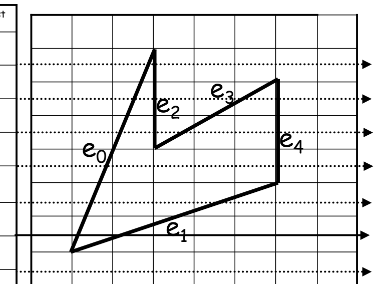
scan line	edge list
5	
4	
3	
2	
1	
0	

list of edges with with:

$2\frac{1}{2} < y_{bottom} \le 3\frac{1}{2}$

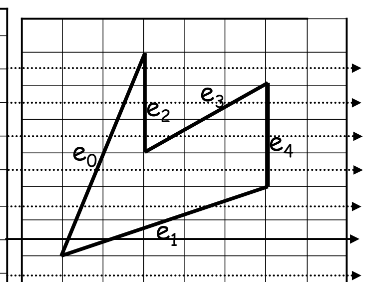
exercixe: edge table

scan line	edge list
7	
6	
5	
4	
3	
2	
1	
0	



example edge table

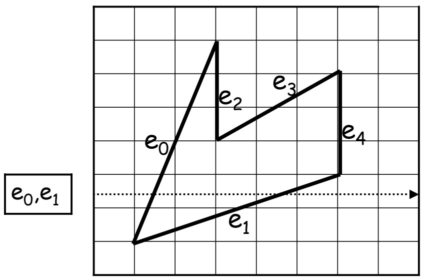
scan line	edge list
7	-
6	-
5	-
4	e_2,e_3
3	e_4
2	-
1	e_0,e_1
0	-



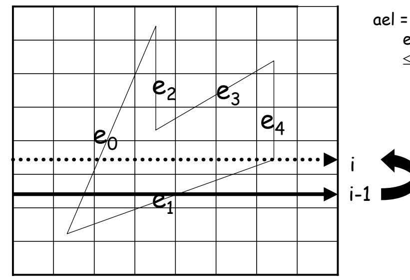
data structures

- edge table
  - for each scan line *i* a list of edges with $i - \frac{1}{2} < y_{bottom} \le i + \frac{1}{2}$
- **active edge list**
  - edges intersecting current scan line

example active edge list

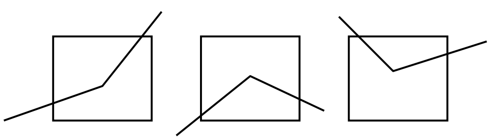


ael update



ael = ael + et[i] – edges with $y_{top} \le i+\frac{1}{2}$

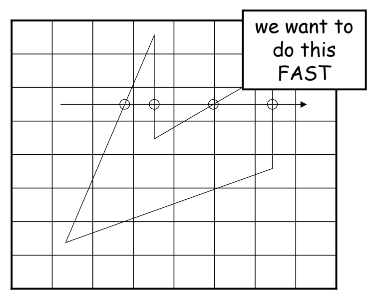
ael changes



end/begin two end two begin

ael always contains an even number of edges!

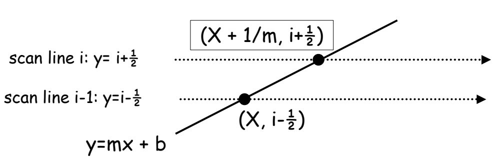
scan line algorithm



for each scan line

1. **find edge/scan line intersection points**
2. order by x-coordinate
3. use odd-even test to turn on pixels

intersection point calculation



all we need to store is x-intercept, X, and inverse slope

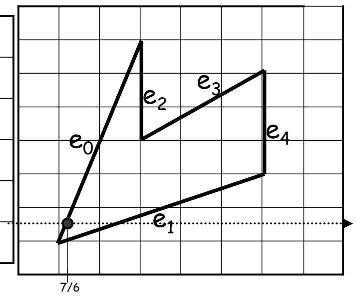
edge record at scan line i

- y_{bottom}
- y_{top}
- $1/m$
- x_{int} : x-intercept at scan line i

initialize to x-intercept at first scan line the edge intersects

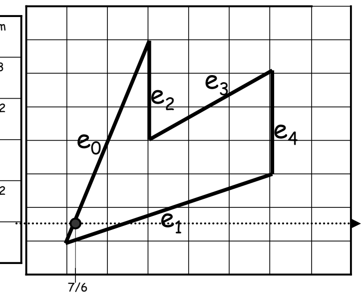
exercise: initialize edge records

edge	y_{bottom}	y_{top}	x_{int}	1/m
e_0	1	7		
e_1	1	3		
e_2	4	7		
e_3	4	6		
e_4	3	6		



initialize edge records

edge	y_{bottom}	y_{top}	x_{int}	1/m
e_0	1	7	7/6	1/3
e_1	1	3	9/4	5/2
e_2	4	7	3	0
e_3	4	6	15/4	3/2
e_4	3	6	6	0



scan line algorithm

build et
scanLine=-1
ael=φ
while scanLine < h
    scanLine ++
    for each edge in ael: x_{int} += 1/m
    ael += et[scanLine]
    ael -= {edges with $y_{top} \leq$ scanLine + $\frac{1}{2}$ }
    sort edges in ael by x_{int}
    compute "on" pixels by odd-even rule

scan line algorithm

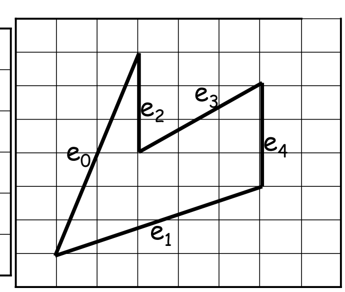
build et
scanLine=-1
ael=φ
while scanLine < h
    scanLine ++
    for each edge in ael: x_{int} += 1/m
    ael += et[scanLine]
    ael -= {edges with $y_{top} \leq$ scanLine + $\frac{1}{2}$ }
    sort edges in ael by x_{int}

scan line algorithm

//compute "on" pixels by odd-even rule
for k=0 … aelNumEdges/2
    for j + $\frac{1}{2}$ > aelEdges[2k].x_{int} and
        j + $\frac{1}{2}$ ≤ aelEdges[2k+1].x_{int}
            turn on pixels (j, scanLine)

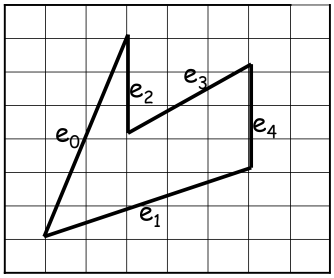
initialize edge records

edge	y_{bottom}	y_{top}	x_{int}	1/m
e_0	1	7	7/6	1/3
e_1	1	3	9/4	5/2
e_2	4	7	3	0
e_3	4	6	15/4	3/2
e_4	3	6	6	0



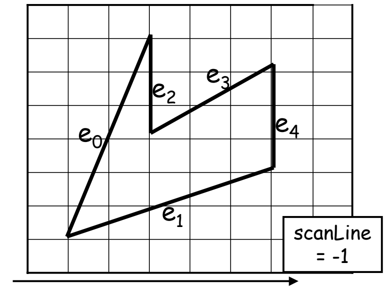
initialize edge table

scan line	Edges
6	-
5	-
4	e_2, e_3
3	e_4
2	-
1	e_0, e_1
0	-



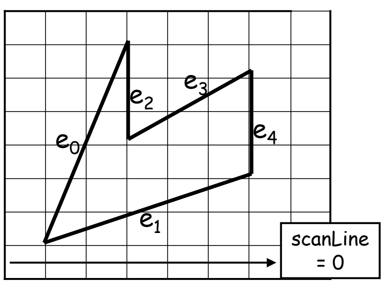
example

ael = ϕ



example

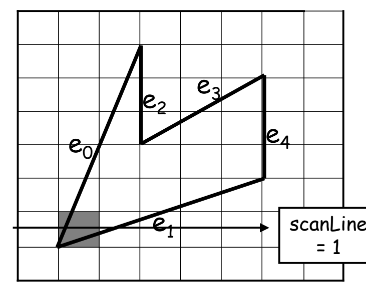
ael = ϕ



example

ael sorted by x_{int}

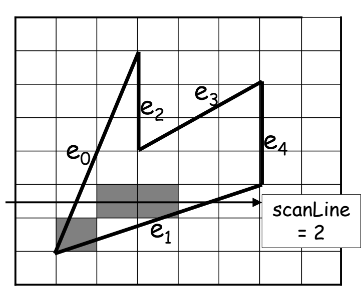
edge	x_{int}	...
e_0	7/6	...
e_1	9/4	...



example

ael sorted by x_{int}

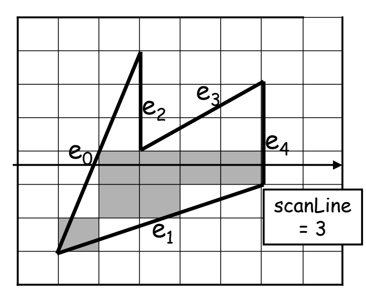
edge	x_{int}	...
e_0	9/6	...
e_1	19/4	...



example

ael sorted by x_{int}

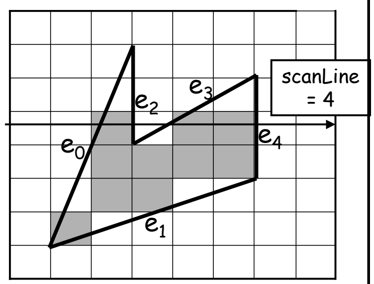
edge	x_{int}	...
e_0	11/6	...
e_4	6	...



example

ael sorted by x_{int}

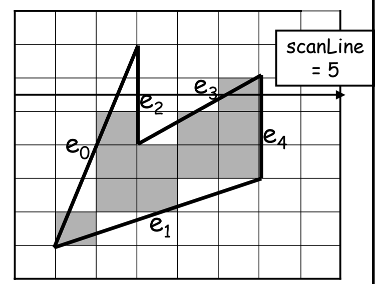
edge	x_{int}	...
e_0	13/6	...
e_2	3	...
e_3	15/4	...
e_4	6	...



example

ael sorted by x_{int}

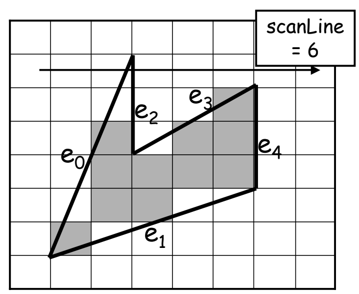
edge	x_{int}	...
e_0	15/6	...
e_2	3	...
e_3	21/4	...
e_4	6	...



example

ael sorted by x_{int}

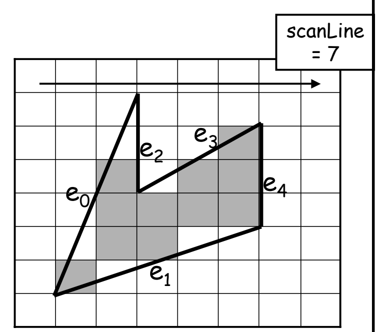
edge	x_{int}	...
e_0	17/6	...
e_2	3	...



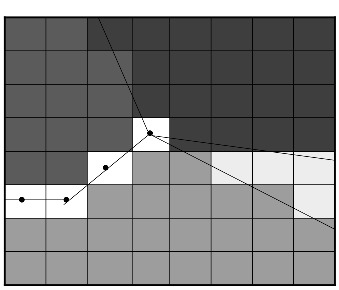
example

ael sorted by x_{int}

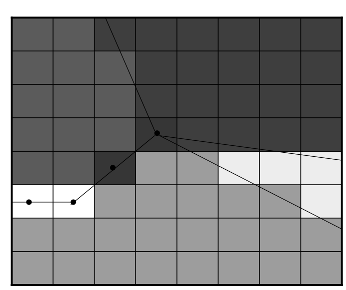
edge	x_{int}	...



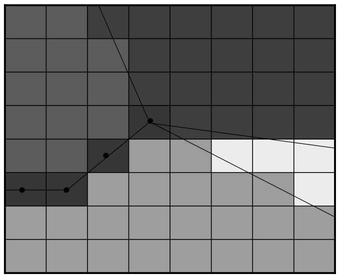
tesselation: center claims



tie breaker 1: left owns



tie breaker 2: above owns

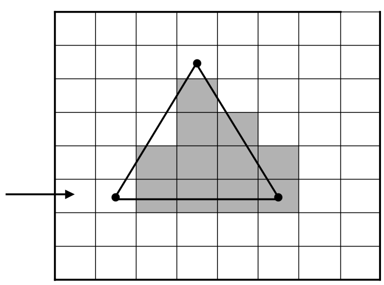


exercise

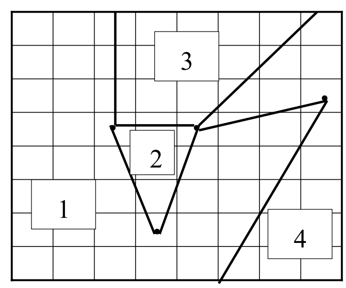
- where in the algorithm are these tie-breaking rules specified?

horizontal edges

what is in aet when scanLine = 2



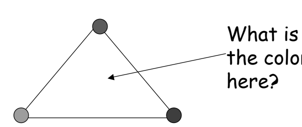
Exercise



scan conversion

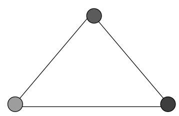
- input: vertex coordinates
- output: pixels to turn on for filled polygon + **pixel color**

shading models



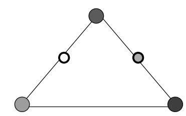
Color is defined at vertices!!!!!!!

flat shading



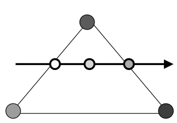
Color entire polygon the color of first vertex.

smooth shading



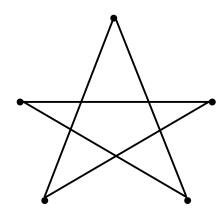
1. Interpolate along edges.

smooth shading



1. Interpolate along edges.
2. Interpolate along scan line between edges.

what happens here?



edge record at scan line i

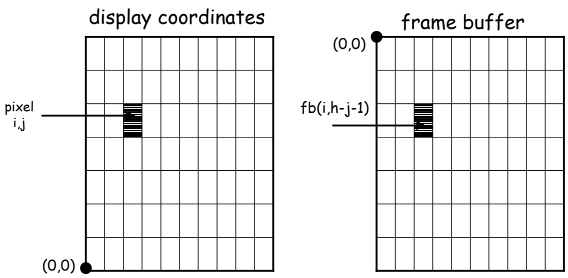
- y_{bottom}
- y_{top}
- 1/m
- x_{int}
- c_{int} — color on edge at (x_{int}, i)
- Δ_c/Δ_x — color increment

scan conversion

- points
- lines segments
- polygons
  - filled

the frame and z buffers and hidden surface removal

display coordinates vs. frame buffer

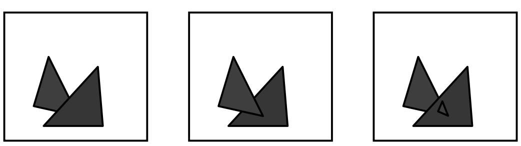


scan conversion

- points
- lines segments
- polygons
  - filled

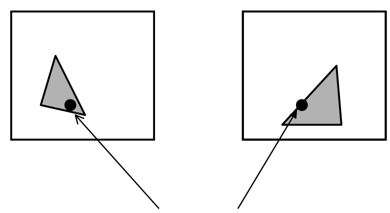
the frame & z buffers and hidden surface removal

hidden surface removal



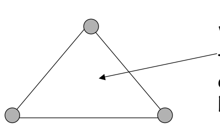
which is right?

hidden surface removal



compare z-depth of corresponding points on 3d surfaces

z-depth

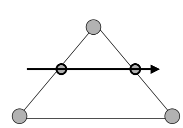


What is the z-depth here?

z-depth is defined at vertices!!!!!!!

so interpolate

z-depth

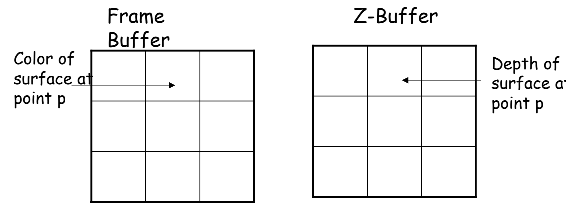


1. Interpolate along edges.
2. Interpolate along scan line.

edge record at scan line i

- y_{top}
- x_i
- $1/m$
- c_i
- Δ_i
- z_i ← depth on edge at (i,x_i)
- δ_i ← depth increment: to compute z_i incrementally

z-buffering



initialize the z-buffer

-1	-1	-1
-1	-1	-1
-1	-1	-1

scan conversion

- Without z-buffering:
  fb(i,h-j)=currcolor:

- With z-buffering
  // z val is the normalized depth of the point on the polygon
  //that projects to point (i,j)
  Compute zval
  If zval > zb(i,h-j) {
  fb(i,h-j) = currcolor
  zb(i,h-j) = zval
  }

graphics pipeline overview

- simplified pipeline ←
- general pipeline

- geometric primitives defined in standardized, homogenous world coordinates
- orthographic projection
- standardized view volume
- standard viewport transformation