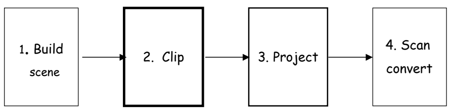
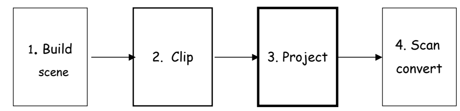
graphics pipeline



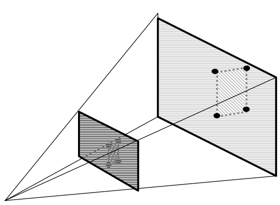
Done

graphics pipeline

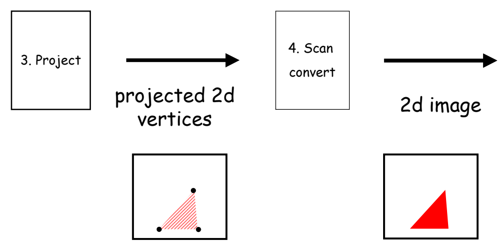


graphics pipeline 3

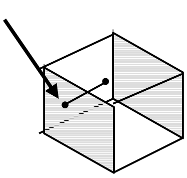
3. project



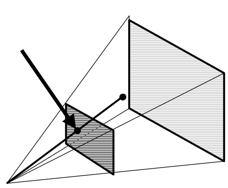
graphics pipeline



projected 2d vertex

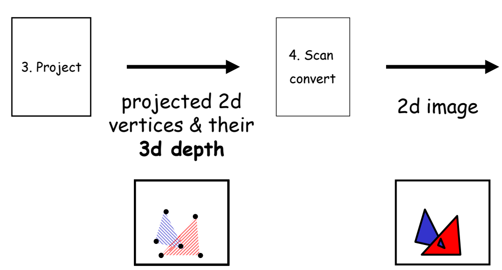


orthographic

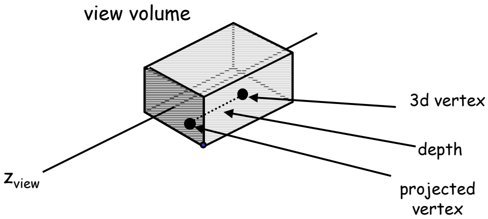


perspective

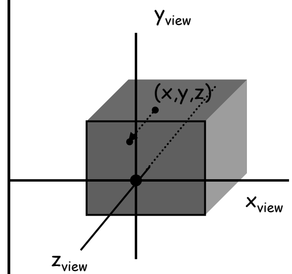
graphics pipeline



orthographic projection

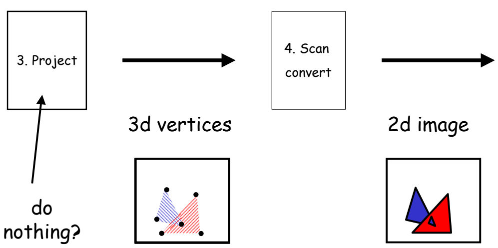


orthographic projection

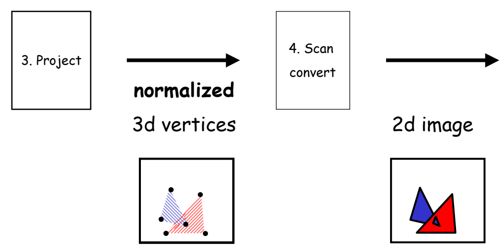


- 3d coordinates (x,y,z) ________
- 3d projected coordinates ________
- 2d projected coordinates ________
- depth ________

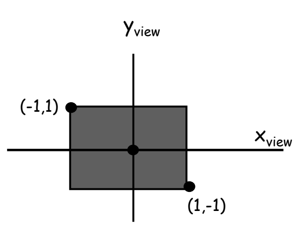
orthographic projection



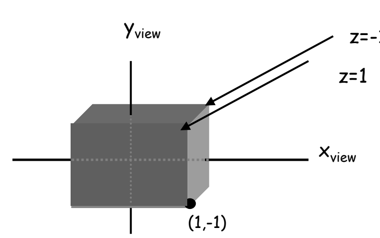
orthographic projection



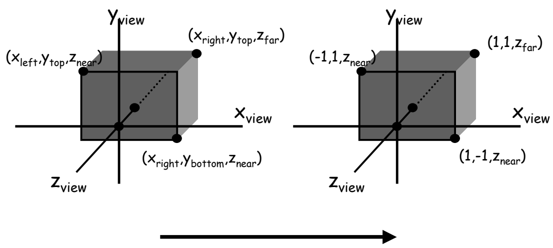
normalized device coordinates



normalized device coordinates

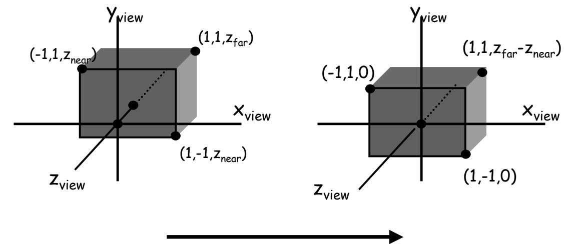


normalize – step 1



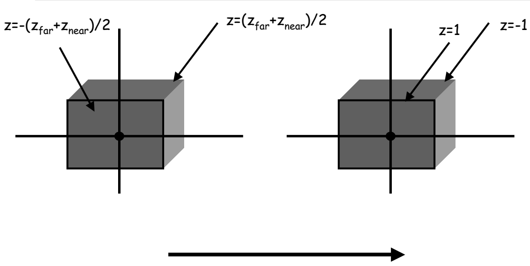
scale by ___________

normalize – step 2



translate by ___________

normalize – step 3



scale by ___________

orthographic projection matrix

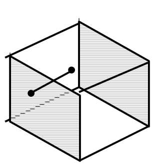
view coordinates

normalized device coordinates

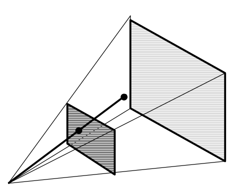
$$\begin{pmatrix} 2/(r-l) & 0 & 0 & 0 \\ 0 & 2/(t-b) & 0 & 0 \\ 0 & 0 & 2/(f-n) & -(f+n)/(f-n) \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 2x/(r-l) \\ 2y/(t-b) \\ 2(z-(f+n)/2)/(f-n) \\ 1 \end{pmatrix}$$

shorthand: $r=x_{right}$, $l=x_{left}$, $t=y_{top}$, $b=y_{bottom}$, $n=z_{near}$, $f=z_{far}$
warning: Woo generalized and uses slightly different notation

now for perspective projection…

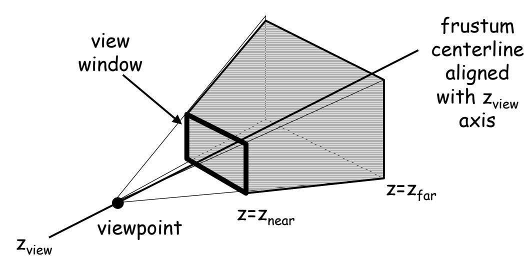


orthographic

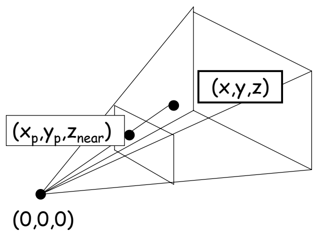


perspective

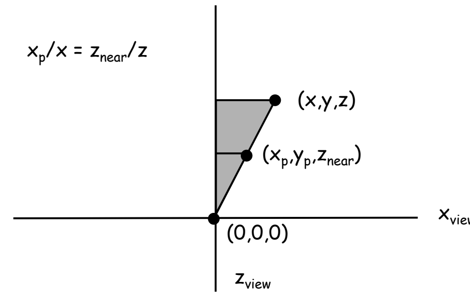
perspective view volume



perspective projection

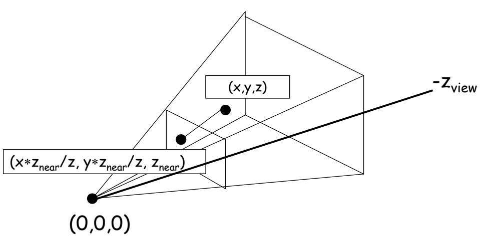


x-projection



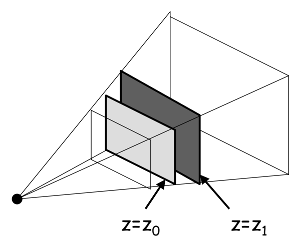
perspective projection

$(x,y,z) \rightarrow (x*z_{near}/z,\ \ y*z_{near}/z,\ \ z_{near})$



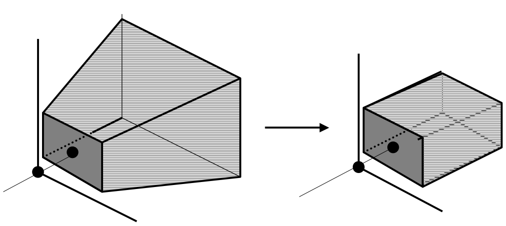
perspective projection

$(x,y,z) \rightarrow (x*z_{near}/z,\ \ y*z_{near}/z,\ \ z_{near})$



depth-dependent scale

depth dependent x,y scale



perspective projection: 2 step

1. depth dependent scale
   $(x,y,z) \rightarrow (x*z_{near}/z,\ \ y*z_{near}/z,\ \ z)$

2. orthographic projection

perspective projection matrix

depth-dependent scale is not a 3d linear transform

BUT we can fake it

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} nx \\ ny \\ z \\ z \end{pmatrix} \longrightarrow \begin{pmatrix} nx/z \\ ny/z \\ 1 \\ 1 \end{pmatrix}$$

view coordinates

normalize by w component

perspective projection matrix

depth-dependent scale is not a 3d linear transform

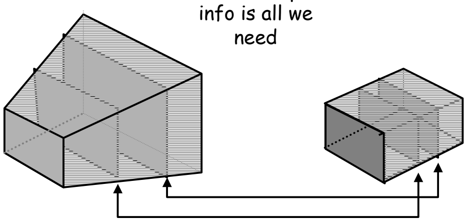
BUT we can fake it

problem – we've lost depth information $\longrightarrow \begin{pmatrix} nx/z \\ ny/z \\ 1 \\ 1 \end{pmatrix}$

there are many hacks to fix this … next we'll show the one we will use!!

depth dependent x,y scale

relative depth info is all we need



perspective projection

$$\begin{pmatrix} -2n/(r-l) & 0 & 0 & 0 \\ 0 & -2n/(t-b) & 0 & 0 \\ 0 & 0 & -(f+n)/(f-n) & 2fn/(f-n) \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} \longrightarrow \begin{pmatrix} x'/w' \\ y'/w' \\ z'/w' \\ 1 \end{pmatrix}$$

view coordinates

normalization

normalized device coordinates

projection

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \longrightarrow M_P \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} \longrightarrow \begin{pmatrix} x'/w' \\ y'/w' \\ z'/w' \\ 1 \end{pmatrix}$$

view coordinates

normalized device coordinates

2d projection (x'/w',y',w')

relative depth z'/w'

projection matrix: M_P

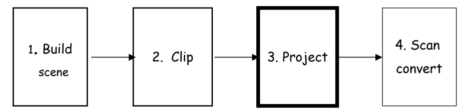
$$\begin{pmatrix} -2n/(r-l) & 0 & 0 & 0 \\ 0 & -2n/(t-b) & 0 & 0 \\ 0 & 0 & -(f+n)/(f-n) & 2fn/(f-n) \\ 0 & 0 & -1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 2/(r-l) & 0 & 0 & 0 \\ 0 & 2/(t-b) & 0 & 0 \\ 0 & 0 & 2/(f-n) & -(f+n)/(f-n) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

perspective orthographic

geometric primitives

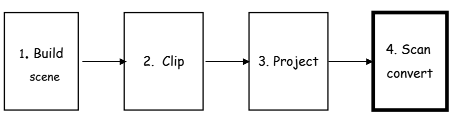
object coordinates: v	description of vertex
world coordinates: $M_W v$	description of vertex situated in world
view coordinates: $M_V M_W v$	description of vertex in world as seen from a particular viewpoint
normalized device coordinates: $M_P M_V M_W v$	description of vertex seen from viewpoint in a normalized world

graphics pipeline



Done

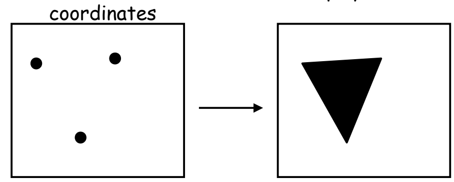
graphics pipeline



scan conversion

2d normalized device coordinates

display window



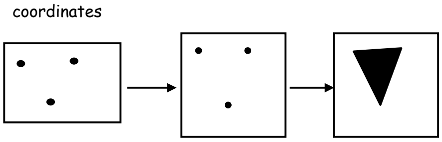
display window can be re-sized by end user

scan conversion

2d normalized device coordinates

display coordinates

display coordinates



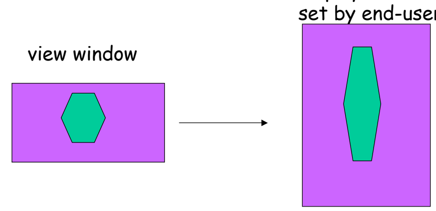
viewport transformation

scan conversion

viewport transformation

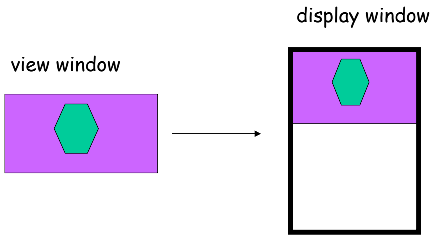
display window: set by end-user

view window

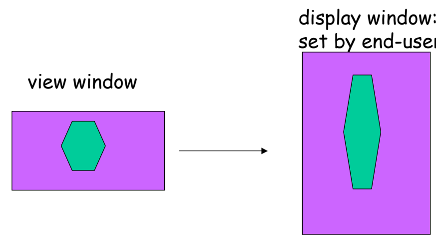


aspect ratio may be different

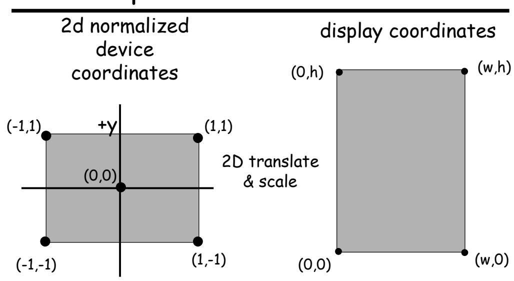
viewport transformation



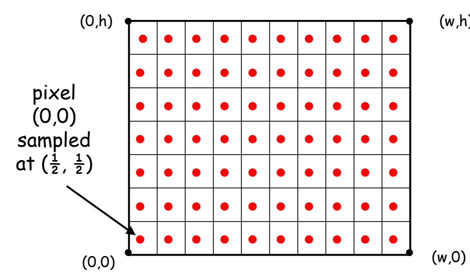
our default transformation



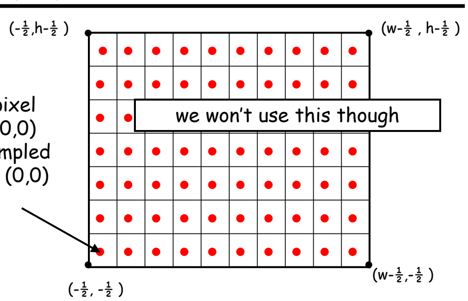
our viewport transform



display coordinates

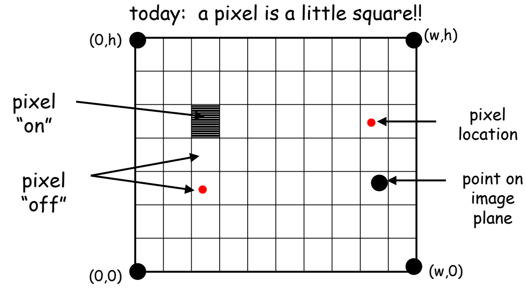


display coordinates: alternative



display coordinates

today: a pixel is a little square!!

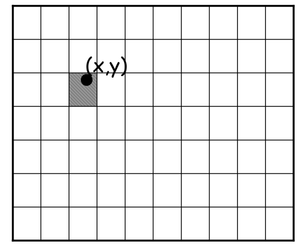


scan conversion

- points
- line segments
- polygons

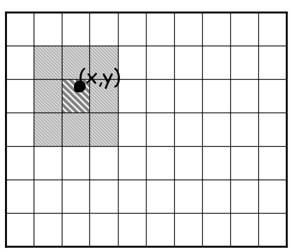
scan conversion: point

closest pixel



scan conversion: point

nearest neighbors

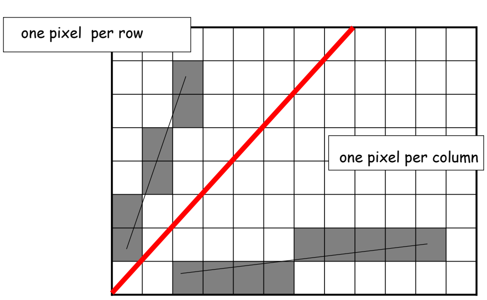


line segments

scan converting line segments

- naïve algorithm
- midpoint algorithm
- bresenham's algorithm

1-pixel wide lines

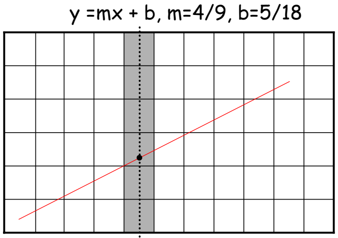


scan conversion

- 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 ($9\frac{1}{2}$,$4\frac{1}{2}$)

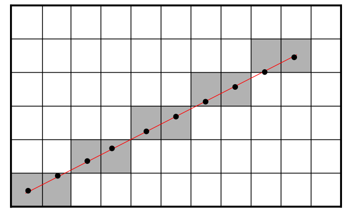
y =mx + b, m=4/9, b=5/18



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
$(i, \lfloor m*(i+\frac{1}{2})+b \rfloor)$

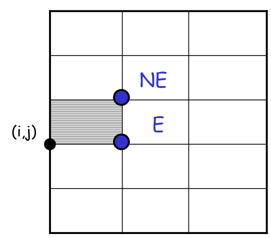
naïve algorithm

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

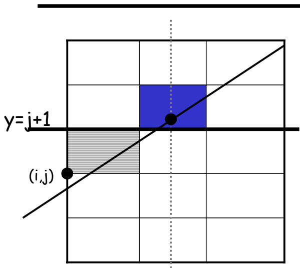
is there a better/faster algorithm?
yes, we can avoid (almost all) multiplication!

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)

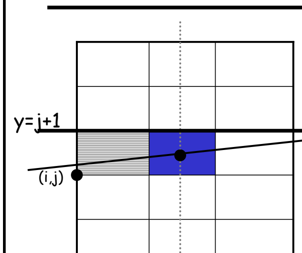
midpoint algorithm: $0 \le m \le 1$



x = i + 3/2

- 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**

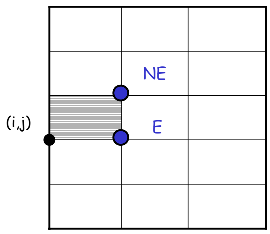
midpoint algorithm: $0 \le m \le 1$



x = i + 3/2

- 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: $0 \le m \le 1$

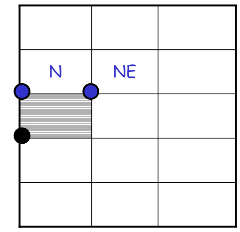


- suppose we've just turned on pixel (i,j)
- next we'll turn on
  - NE: if $j+1 \le m(i+3/2) + b$
  - E: otherwise

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 $\Delta y = y_1 - y_0$

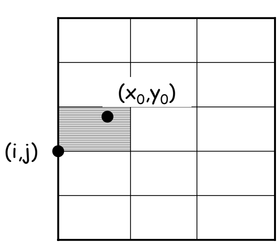
$\leftrightarrow \quad 2\Delta_x(j+1) \le \Delta_y(2i+3) + 2\Delta_x b$

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

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 $i \le x_1$ {
   - if $d \le 0$ // go NE
     - {
     - }
   - else // go E
     - {
     - }
   - }

bresenham's algorithm (0≤m≤1)

3. while $i \le x_1$ {
   - if $d \le 0$ // go NE
     - {
       - i++, j++
       - turn on pixel (i,j)
       - update d
     - }
   - else // go E
     - {
       - i++
       - turn on pixel (i,j)
       - update d
     - }
   - }

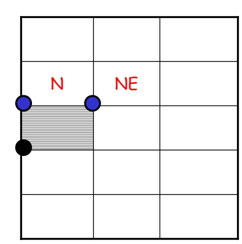
bresenham's algorithm (0≤m≤1)

3. while $i \le x_1$ {
   - if $d \le 0$ // go NE
     - {
       - i++, j++
       - turn on pixel (i,j)
       - $d = d + 2\Delta_x - 2\Delta_y$
     - }
   - else // go E
     - {
       - i++
       - turn on pixel (i,j)
       - $d = d - 2\Delta_y$
     - }
   - }

bresenham's algorithm: other cases

- similar rules

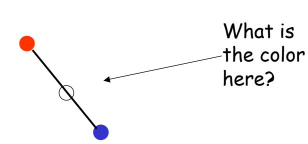
e.g. m>1



scan conversion

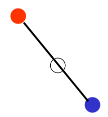
- input: endpoint coordinates
- output: pixels to turn on for a 1-pixel wide line segment + **pixel color**

shading models



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

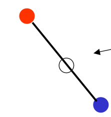
flat shading



What is the color here?

use color of first vertex

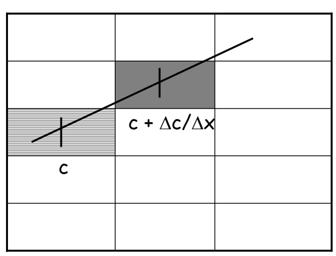
smooth (gouraud) shading



What is the color here?

interpolate

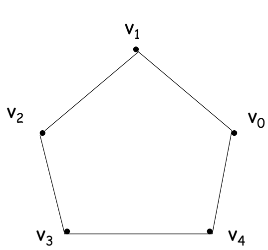
interpolation computation



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,$p_{i+1 \bmod n}$)

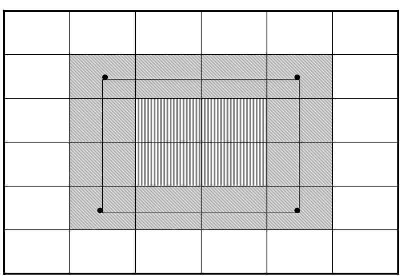
scan conversion

- points
- line segments
- polygons
  - filled

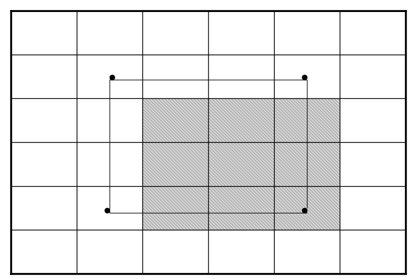
scan conversion

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

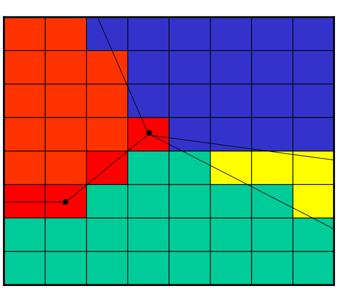
which pixels should be on?



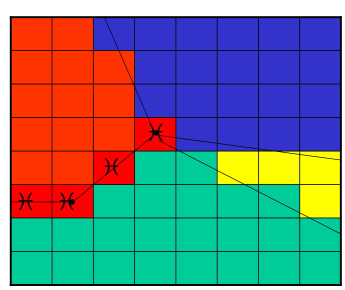
here we get the same size!



tesselating polygons

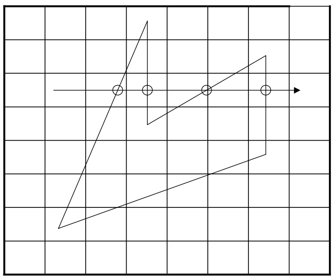


tesselating polygons with edges



exercise: mark remaining edge pixels

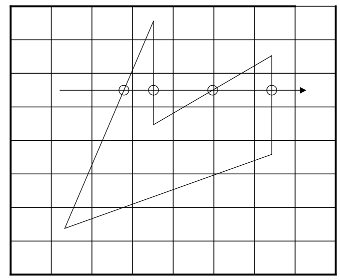
scan line algorithm



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

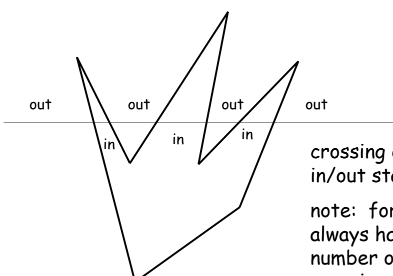
scan line algorithm



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**

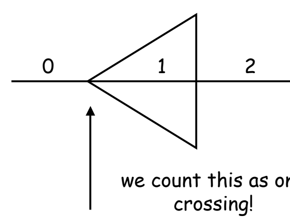
odd-even test



crossing edge changes in/out state

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

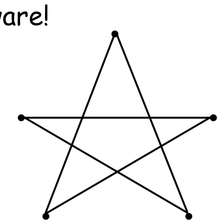
odd-even test



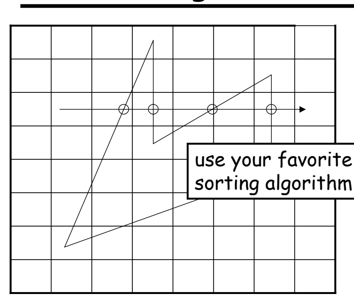
we count this as one crossing!

odd-even test

buyer beware!



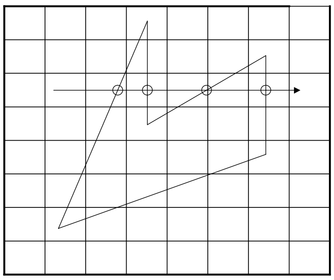
scan line algorithm



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

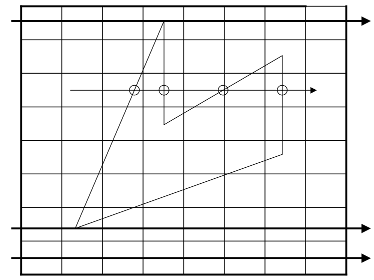
scan line algorithm



for each scan line

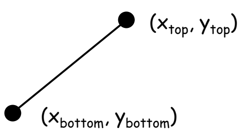
1. **find edge/scan line intersection points**
2. order by x-coordinate
3. use odd-even test to turn on pixels

scan line notation



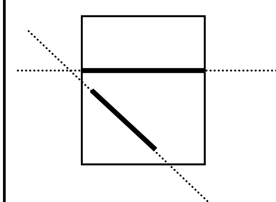
scan line i is defined as the line $y=i+\frac{1}{2}$

edge notation



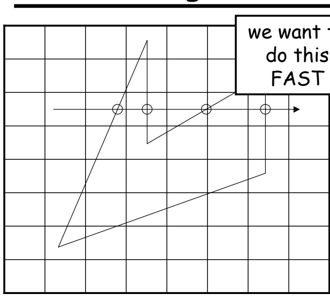
$y_{bottom} \leq y_{top}$

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

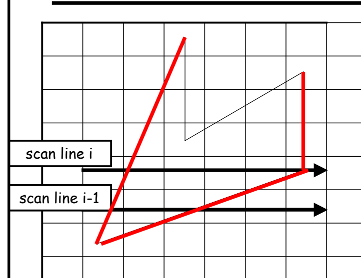


we want to do this FAST

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 $i - \frac{1}{2} < y_{bottom} \leq i + \frac{1}{2}$

- edges with $y_{top} \leq i + \frac{1}{2}$

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)

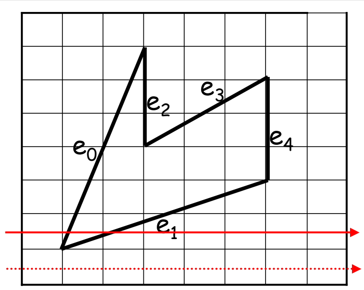
scan line	edge list
5	
4	
3	
2	
1	
0	

list of edges with with:

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

example edge table

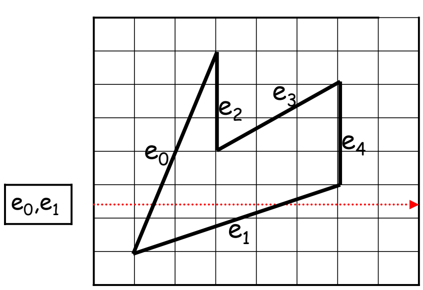
scan line	edge list
7	-
6	-
5	-
4	e_2,e_3
3	e_4
2	-
1	e_0,e_1
0	0



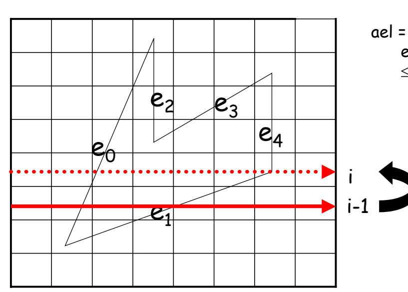
data structures

- 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

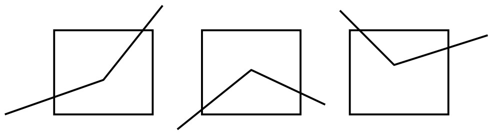


ael update



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

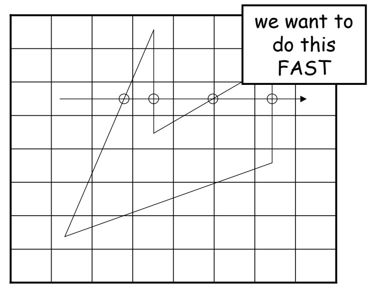
ael changes



end/begin　　two end　　two begin

ael always contains an even number of edges!

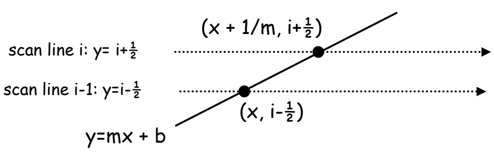
scan line algorithm



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

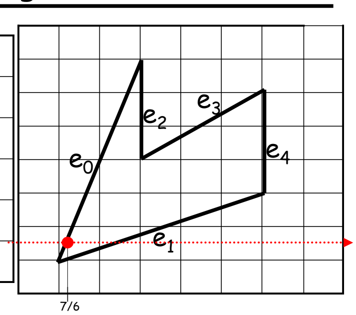
edge record at scan line i

- 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

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 $y_{top} \le$ scanLine + $\frac{1}{2}$ }
　sort edges in ael by x_{int}
　compute "on" pixels by odd-even rule

scan line algorithm

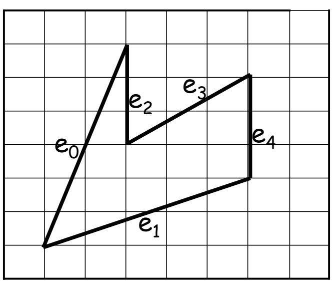
build et
scanLine=-1
ael=φ
while scanLine < h
    scanLine ++
    for each edge in ael: x_{int} += 1/m
    ael += et[scanLine]
    ael -= {edges with $y_{top} \leq$ scanLine + $\frac{1}{2}$ }
    sort edges in ael by x_{int}

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 + $\frac{1}{2} \leq$ aelEdges[2k+1].x_{int}
            turn on pixels (j, scanLine)

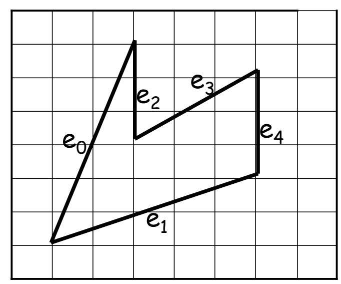
initialize edge records

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



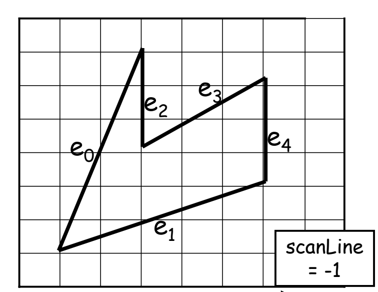
initialize edge table

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



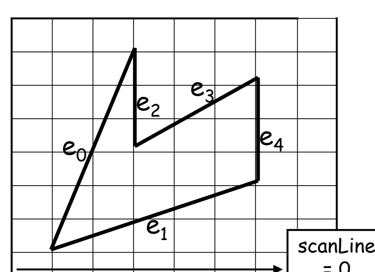
example

ael = ϕ



example

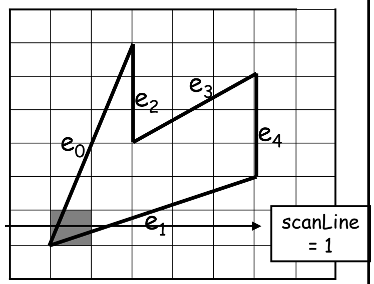
ael = ϕ



example

ael sorted by x_{int}

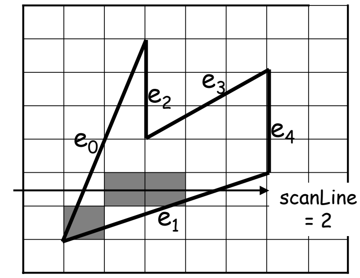
edge	x_{int}	...
e_0	7/6	...
e_1	9/4	...



example

ael sorted by x_{int}

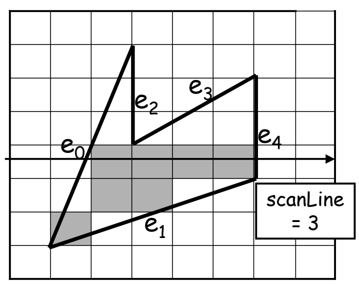
edge	x_{int}	...
e_0	9/6	...
e_1	19/4	...



example

ael sorted by x_{int}

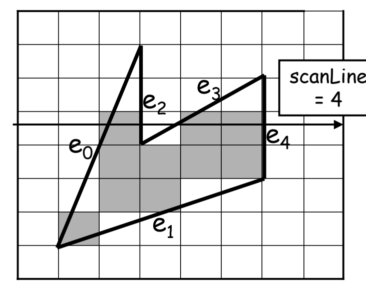
edge	x_{int}	...
e_0	11/6	...
e_4	6	...



example

ael sorted by x_{int}

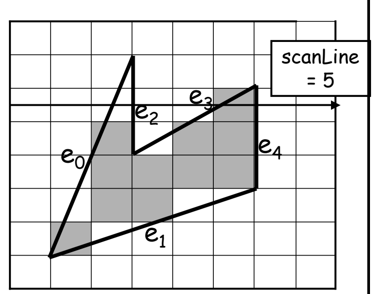
edge	x_{int}	...
e_0	13/6	...
e_2	3	...
e_3	15/4	...
e_4	6	...



example

ael sorted by x_{int}

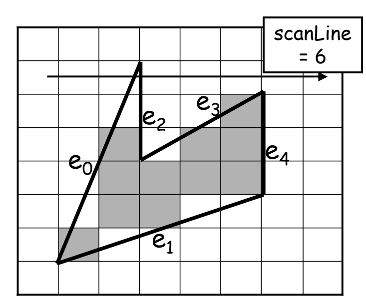
edge	x_{int}	...
e_0	15/6	...
e_2	3	...
e_3	21/4	...
e_4	6	...



example

ael sorted by x_{int}

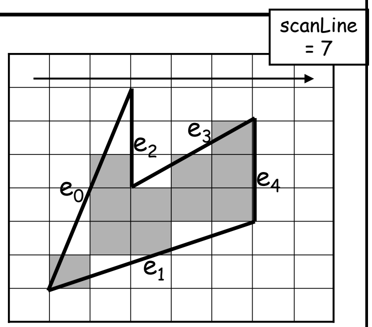
edge	x_{int}	...
e_0	17/6	...
e_2	3	...



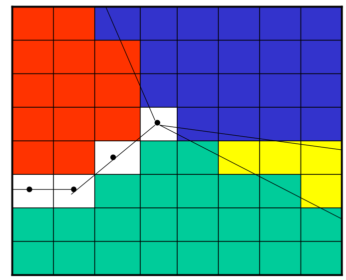
example

ael sorted by x_{int}

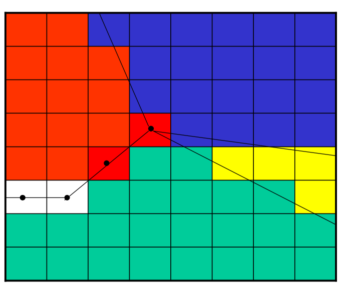
edge	x_{int}	...



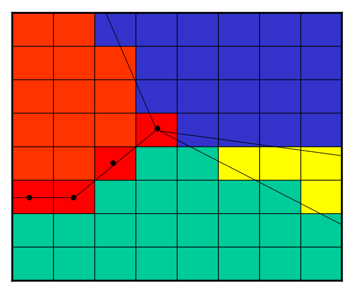
tesselation: center claims



tie breaker 1: left owns



tie breaker 2: above owns

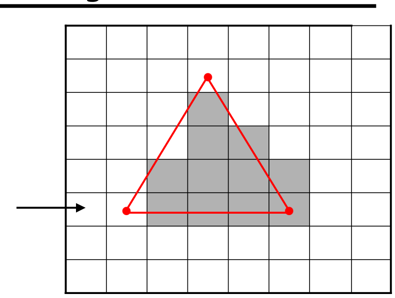


exercise

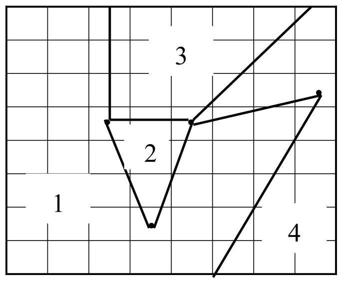
- where in the algorithm are these tie-breaking rules specified?

horizontal edges

what is in aet when scanLine = 2



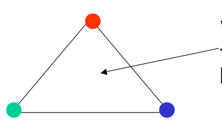
Exercise



scan conversion

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

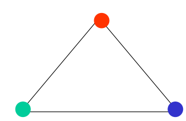
shading models



What is the color here?

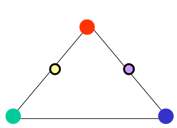
Color is defined at vertices!!!!!!

flat shading



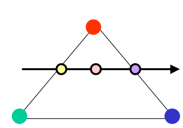
Color entire polygon the color of first vertex.

smooth shading



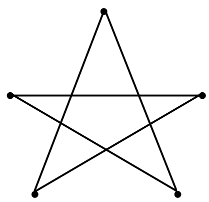
1. Interpolate along edges.

smooth shading



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

what happens here?



edge record at scan line i

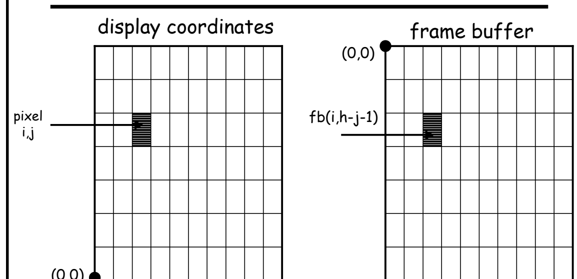
- y_{bottom}
- y_{top}
- $1/m$
- x_{int}
- c_{int} ← color on edge at (x_{int}, i)
- Δ_c/Δ_x ← color increment

scan conversion

- points
- lines segments
- polygons
  - filled

the frame and z buffers and hidden surface removal

display coordinates vs. frame buffer

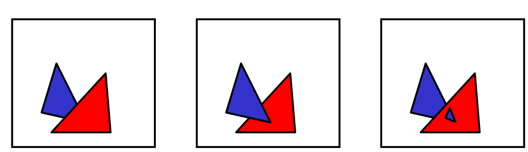


scan conversion

- points
- lines segments
- polygons
  - filled

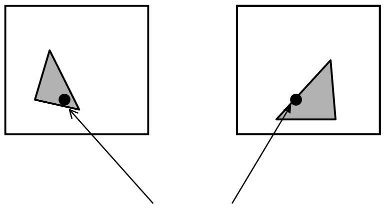
the frame & z buffers and hidden surface removal

hidden surface removal



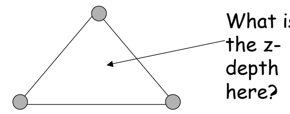
which is right?

hidden surface removal



compare z-depth of corresponding
points on 3d surfaces

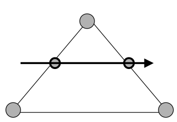
z-depth



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

so interpolate

z-depth

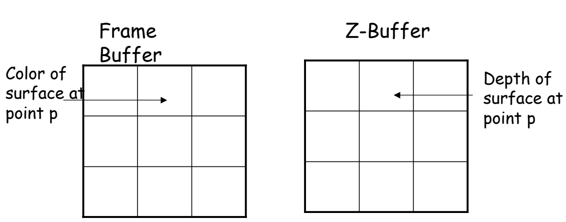


1. Interpolate along edges.
2. Interpolate along scan line.

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



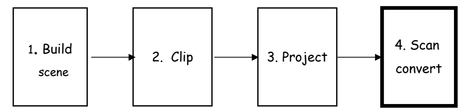
initialize the z-buffer

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

scan conversion

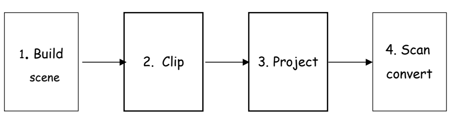
- Without z-buffering:
  fb(i,h-j)=currcolor:
- With z-buffering
  // z val is the normalized depth of the point on the polygon
     //that projects to point (i,j)
  Compute zval
   If zval > zb(i,h-j) {
            fb(i,h-j) = currcolor
            zb(i,h-j) = zval
            }

graphics pipeline



Done

graphics pipeline



Done