Computer Graphics

# Rasterization

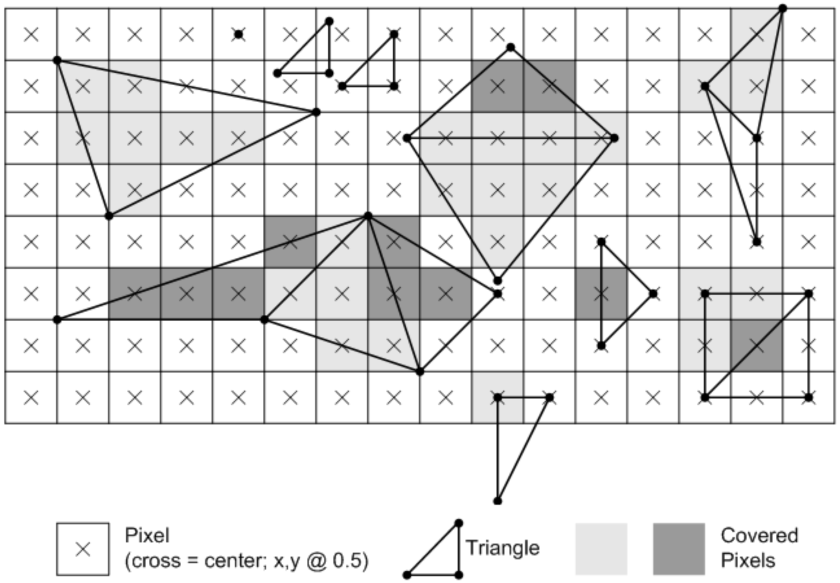

Teacher:  A.prof. Chengying Gao(高成英)

E-mail: mcsgcy@mail.sysu.edu.cn

School of Data and Computer Science

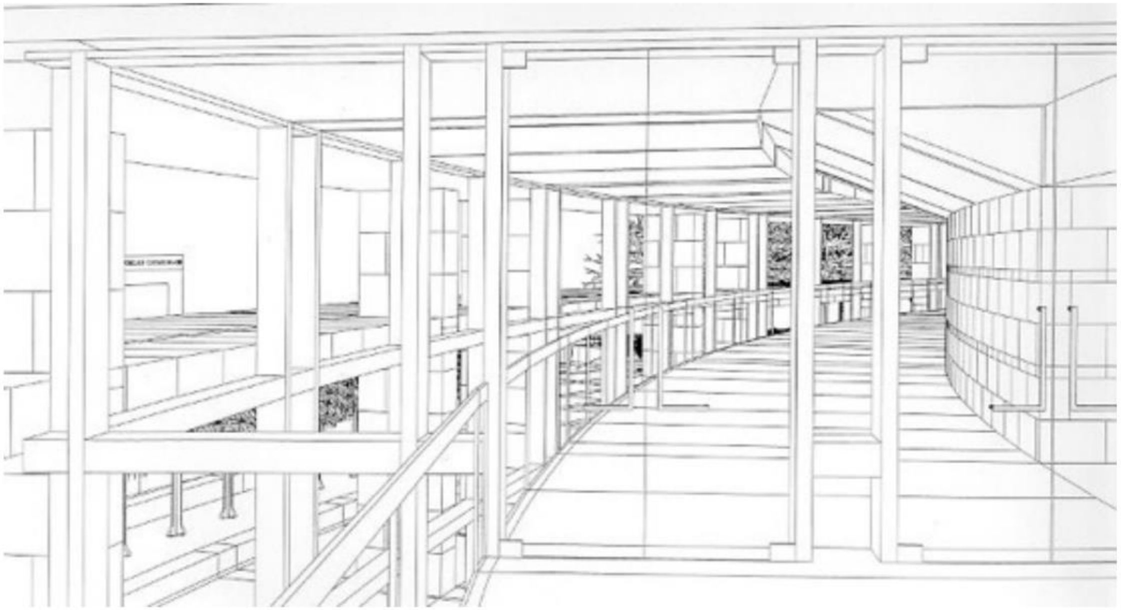

# Rasterization

- The task of displaying a world modeled using primitives like lines, polygons, filled/patterned area, etc. can be carried out in two steps:
  - determine the pixels through which the primitive is visible, a process called rasterization or scan conversion
  - determine the color value to be assigned to each such pixel



# Scan converting a line segment

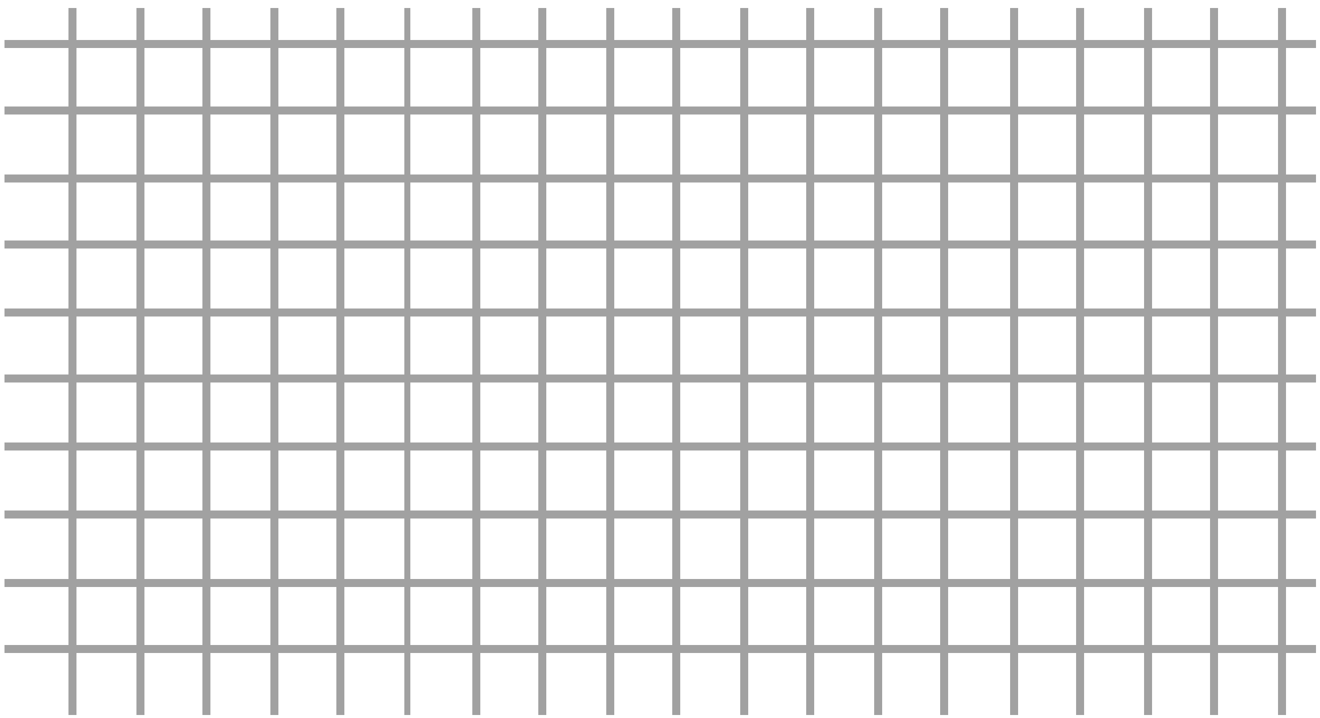
- The line is a powerful element used since the days of Euclid to model the edges in the world.



- Given a line segment defined by its endpoints determine the pixels and color which best model the line segment.

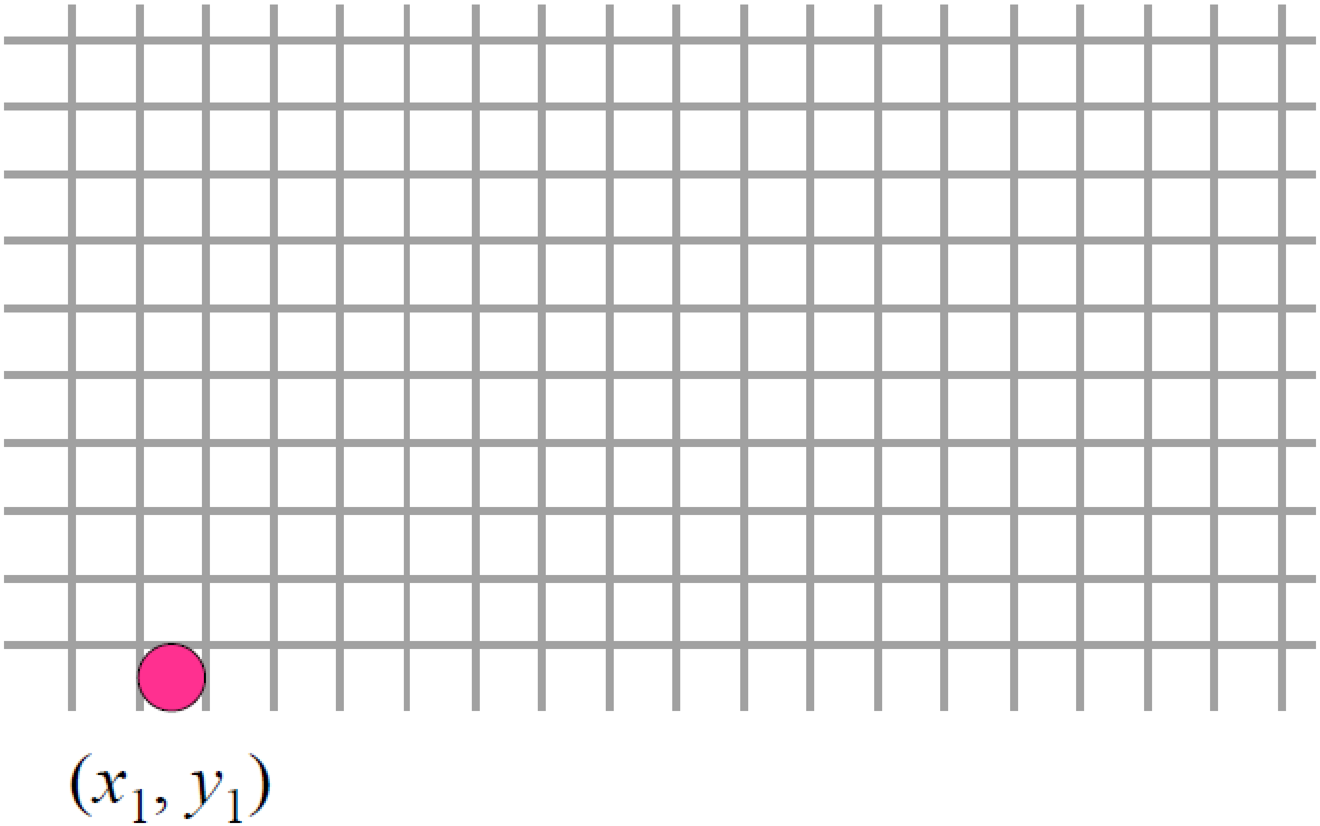
# Scan converting a line segment

start from $(x_1, y_1)$ end at $(x_2, y_2)$



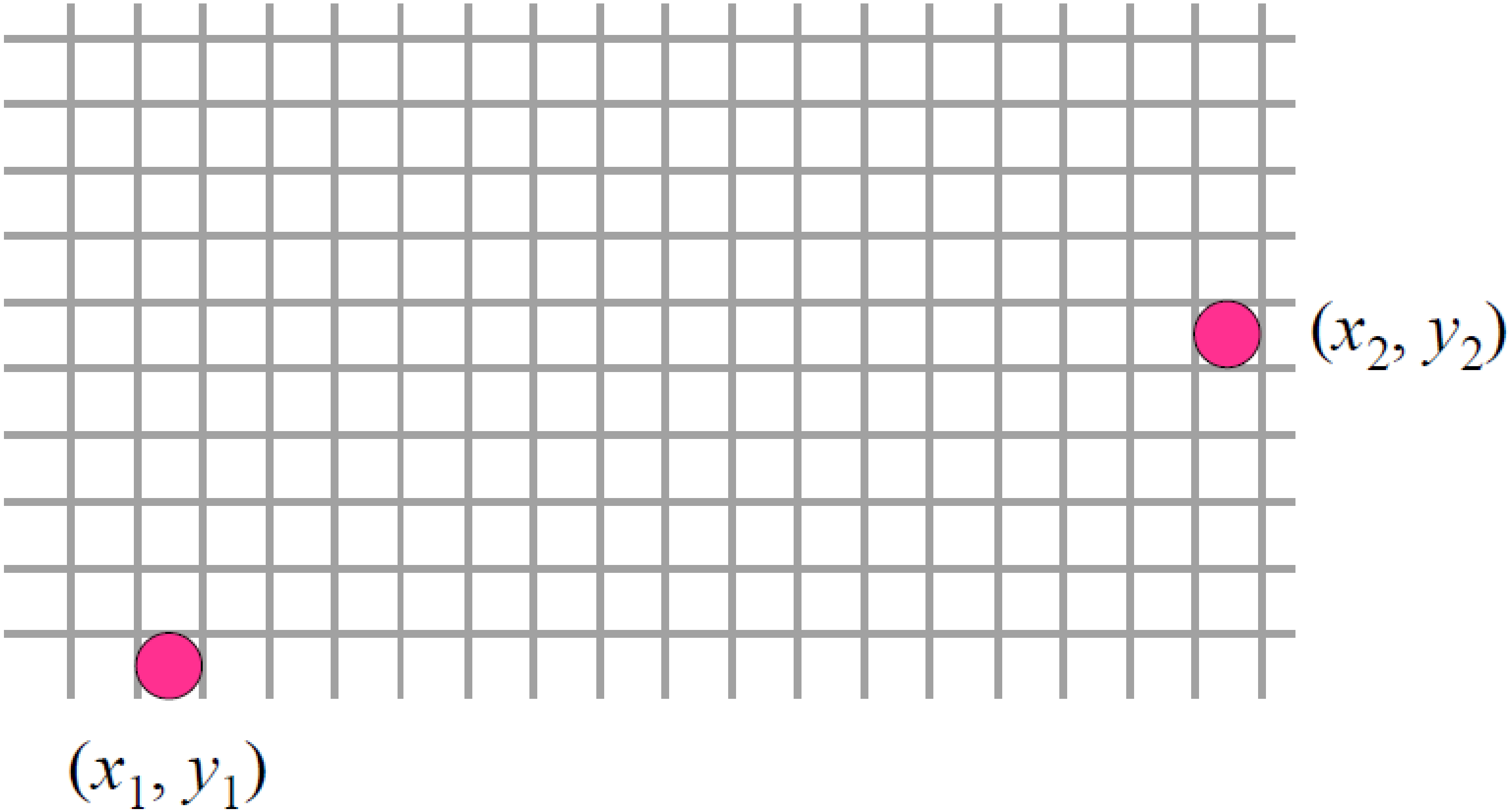
# Scan converting a line segment

start from $(x_1, y_1)$ end at $(x_2, y_2)$



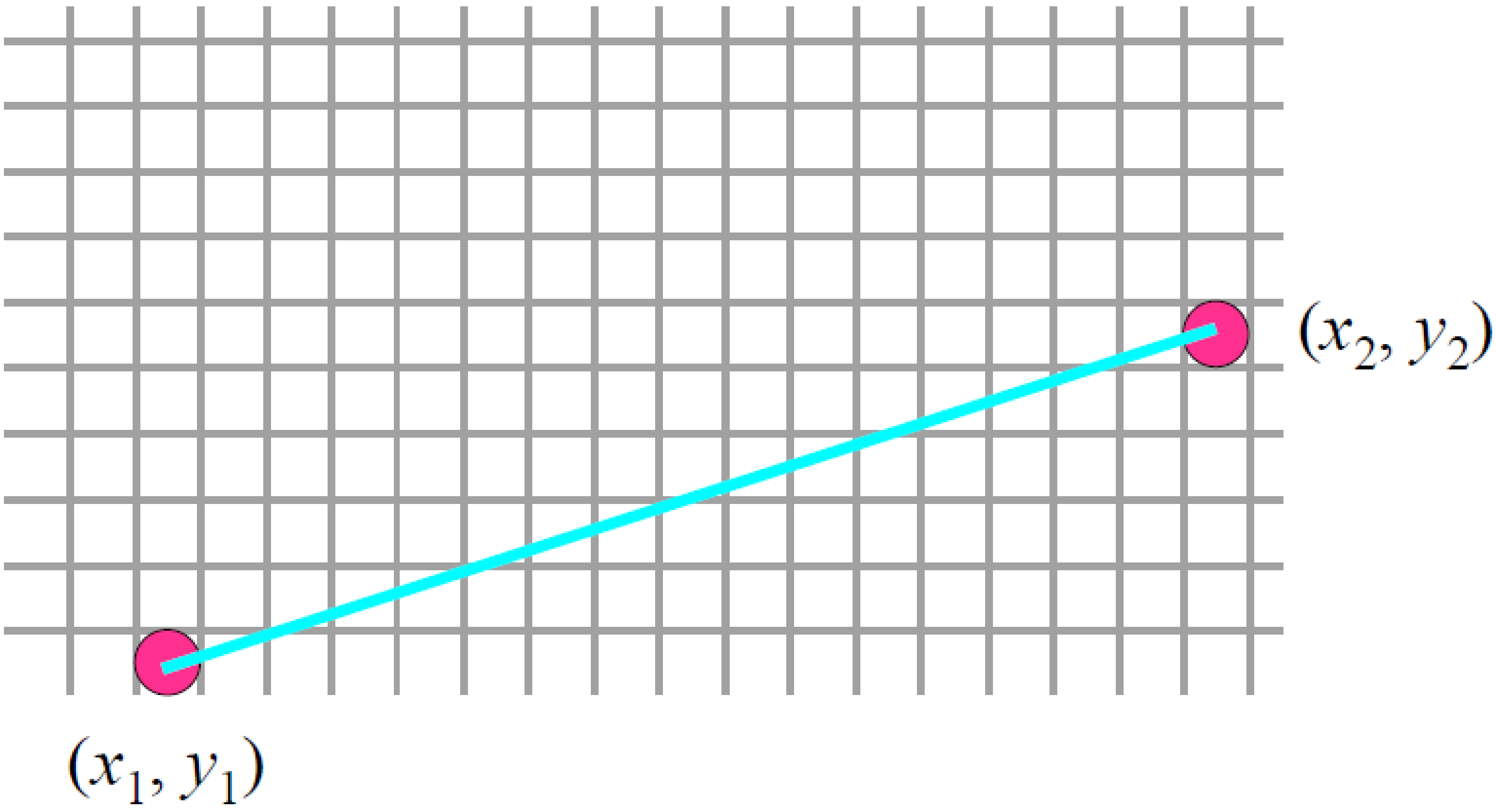
# Scan converting a line segment

start from $(x_1, y_1)$ end at $(x_2, y_2)$



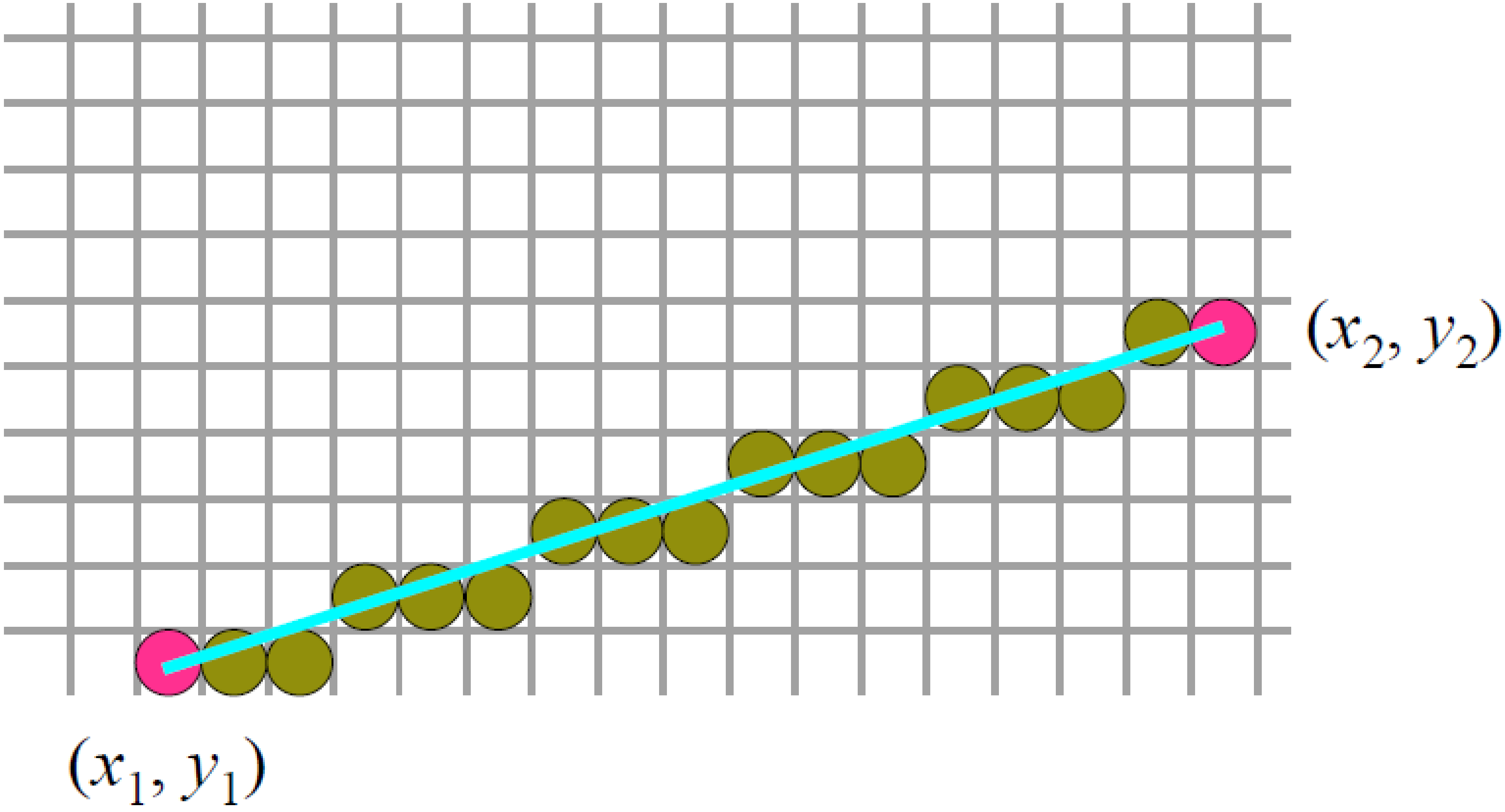
# Scan converting a line segment

start from $(x_1, y_1)$ end at $(x_2, y_2)$



# Scan converting a line segment

start from $(x_1, y_1)$ end at $(x_2, y_2)$



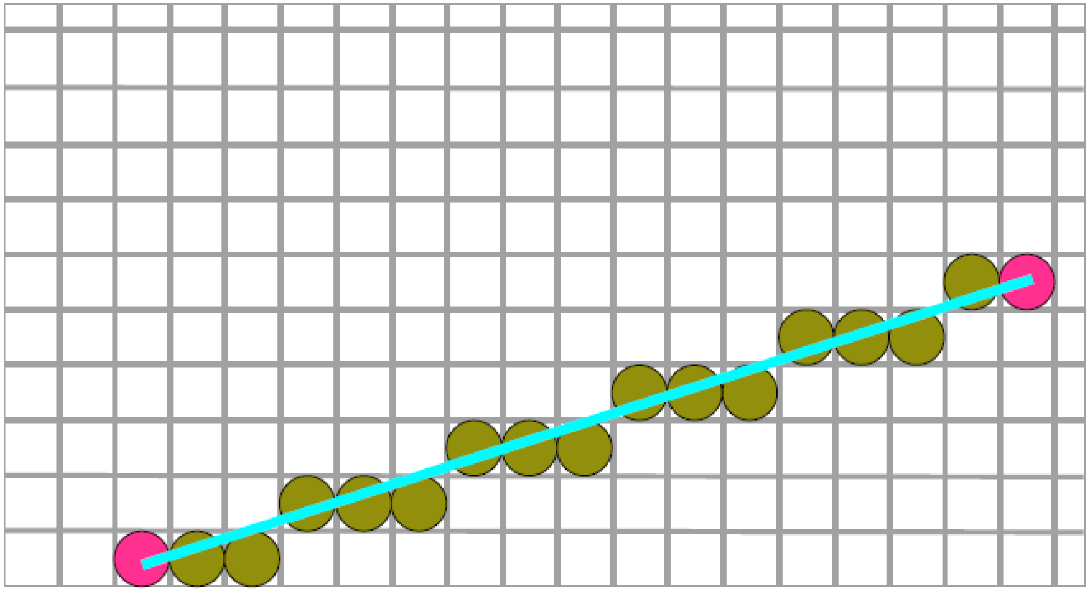
# Scan converting a line segment

- Requirements
  - chosen pixels should lie as close to the ideal line as possible
  - the sequence of pixels should be as straight as possible
  - all lines should appear to be of constant brightness independent of their length and orientation
  - should start and end accurately
  - should be drawn as rapidly as possible
  - should be possible to draw lines with different width and line styles

# Scan converting a line segment

Question 1: How ?

$(x_1, y_1), (x_2, y_2)$



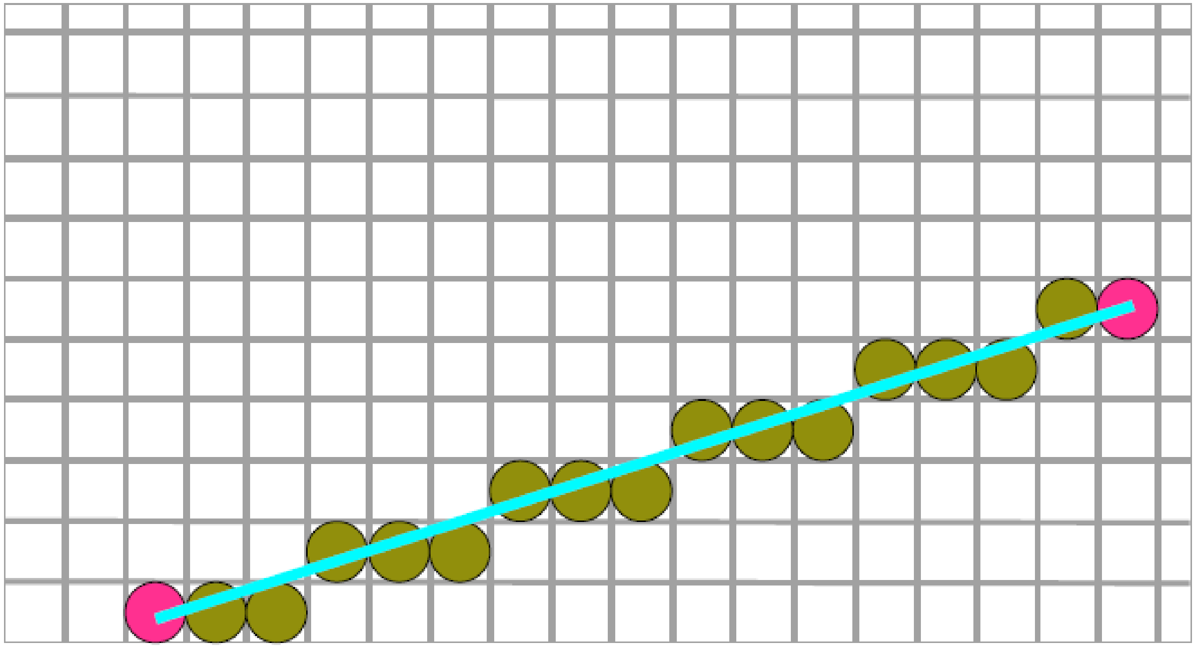
# Scan converting a line segment

Question 1: How ?

$(x_1, y_1), (x_2, y_2)$

$y=mx+b$

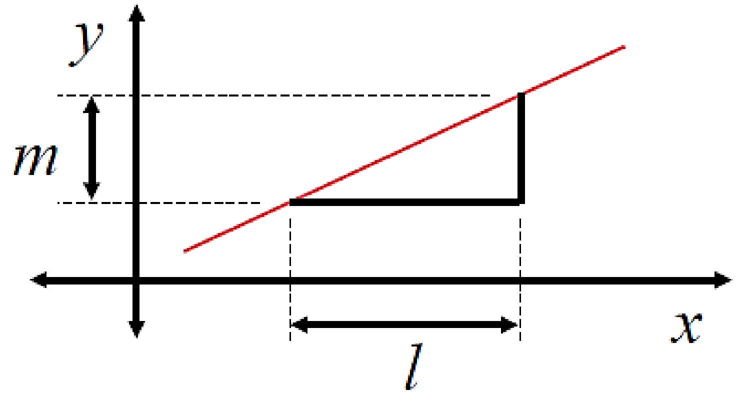
$x_1+1 \Rightarrow y=?$, rounding



$x_1+2 \Rightarrow y=?$, rounding → $x_1+i \Rightarrow y=?$, rounding

# Equation of Line

- For a line segment joining points
- $P(x_1, y_1)$ and $Q(x_2, y_2)$ *slope* $m = \dfrac{y2 - y1}{x2 - x1} = \dfrac{\Delta y}{\Delta x}$
- Slope $m$ means that for every unit increment in $x$ the increment in $y$ is $m$ units



# Digital Differential Analyzer (DDA, 数值微分法)

- We consider the line in the first octant. Other cases can be easily derived.
- Uses differential equation of the line

$$y_i = mx_i + c$$

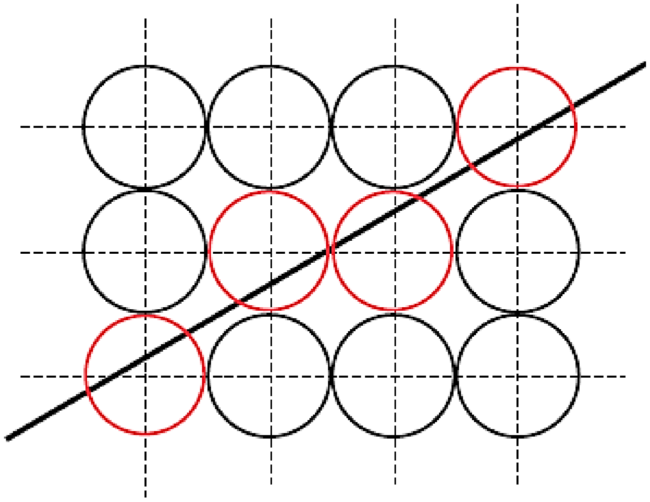
$$where, \quad m = \frac{y2 - y1}{x2 - x1}$$

- Incrementing X-coordinate by 1

$$x_i = x_{i\_prev} + 1$$

$$y_i = y_{i\_prev} + \mathrm{m}$$

- Illuminate the pixel $[x_i, round(y_i)]$

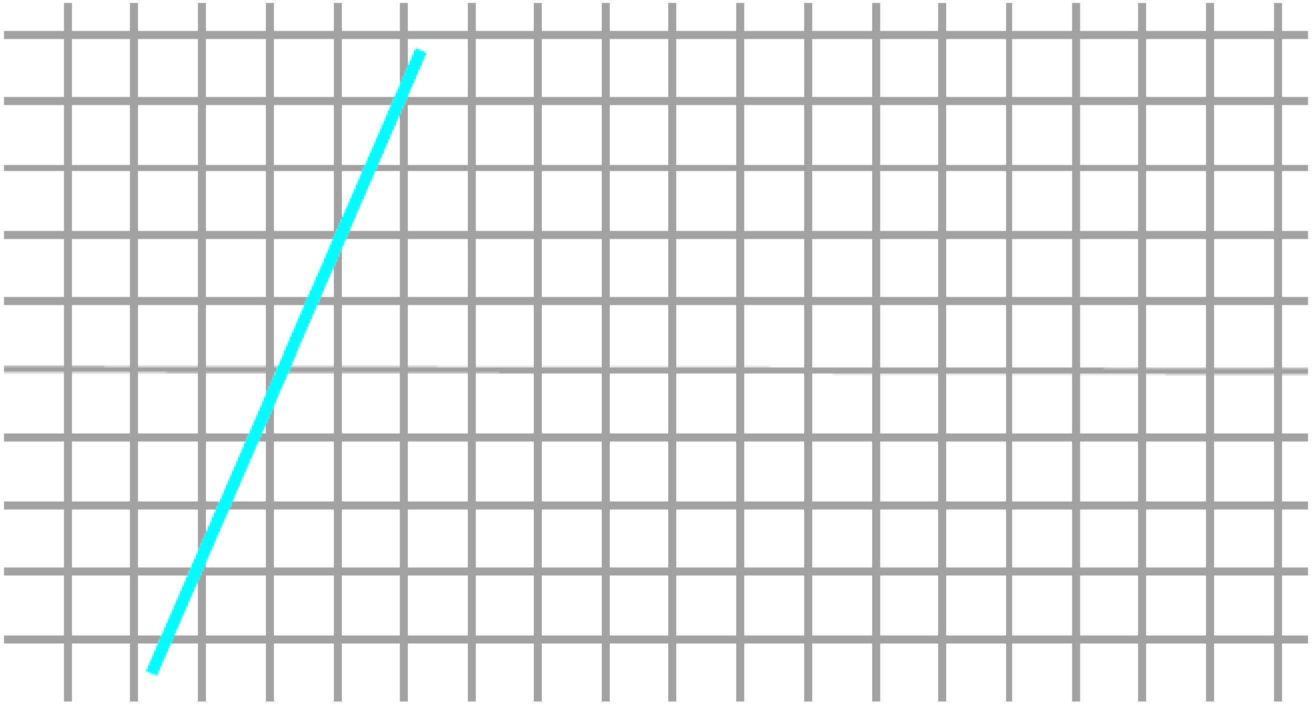


# Digital Differential Analyzer (DDA, 数值微分法)

If $\triangle x < \triangle y$

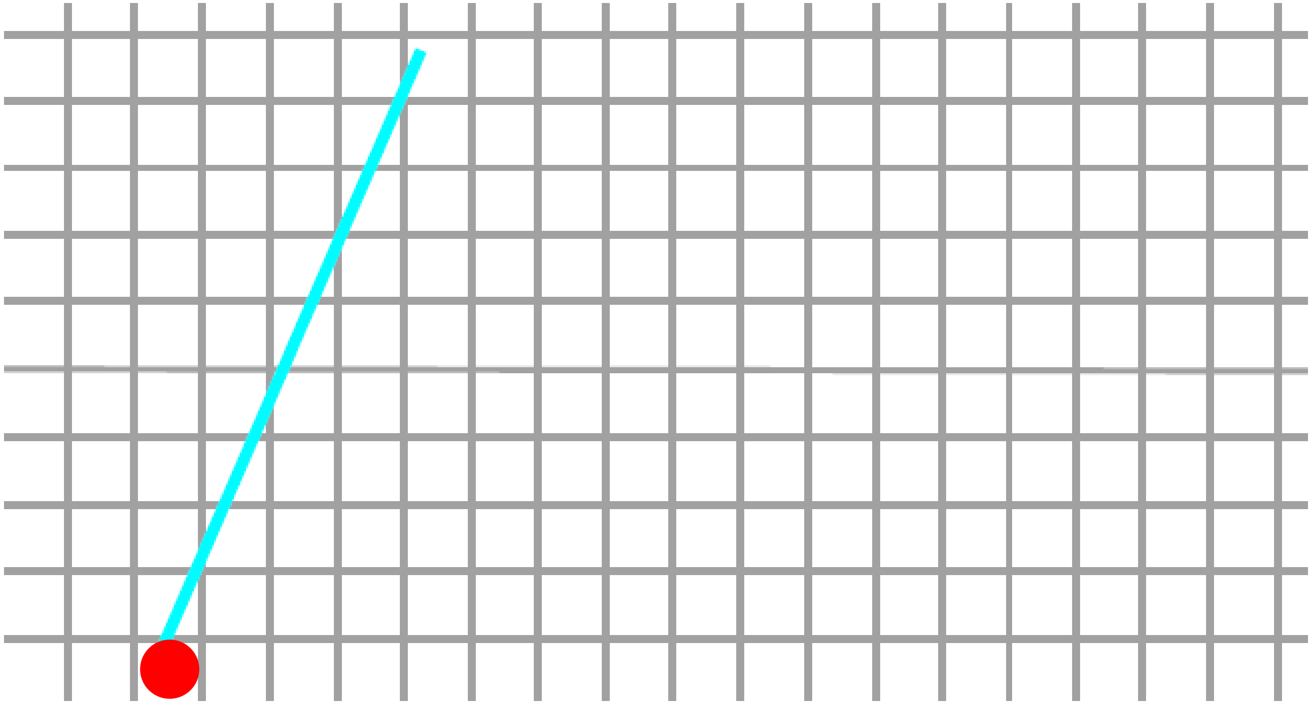
# Digital Differential Analyzer (DDA, 数值微分法)

If $\triangle x < \triangle y$



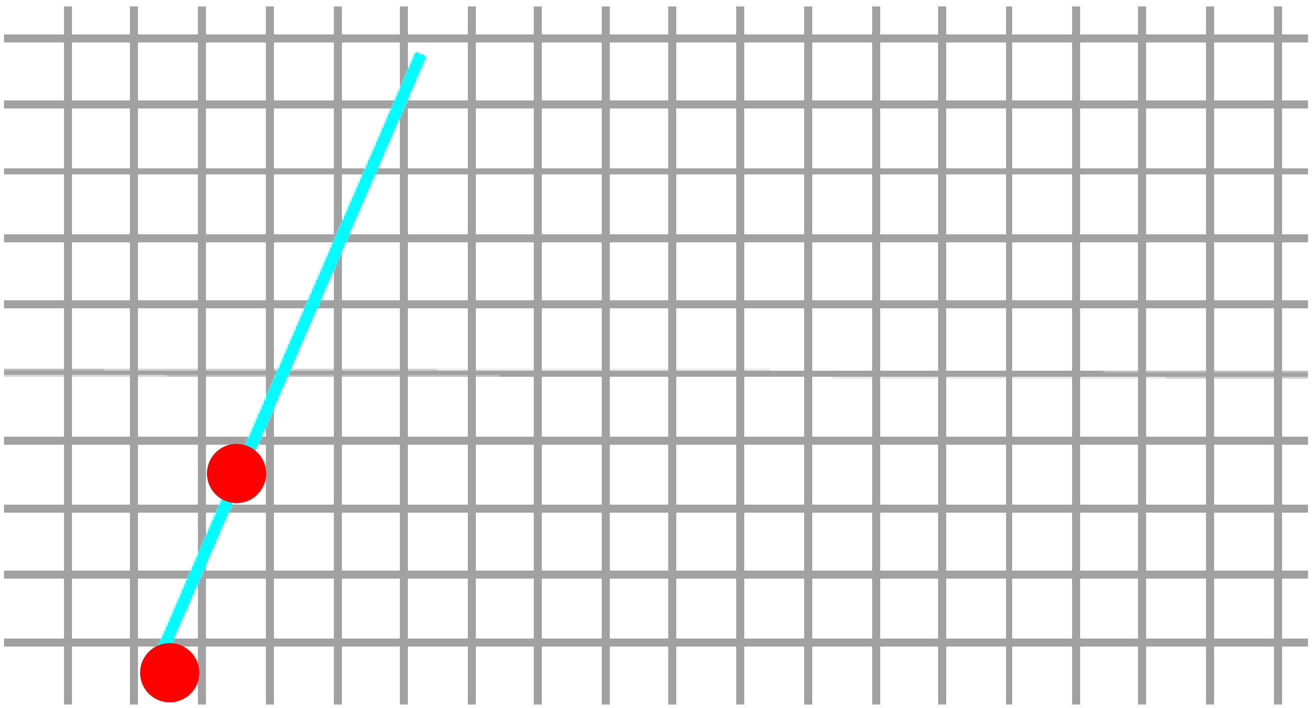
# Digital Differential Analyzer (DDA)

If $\triangle x < \triangle y$



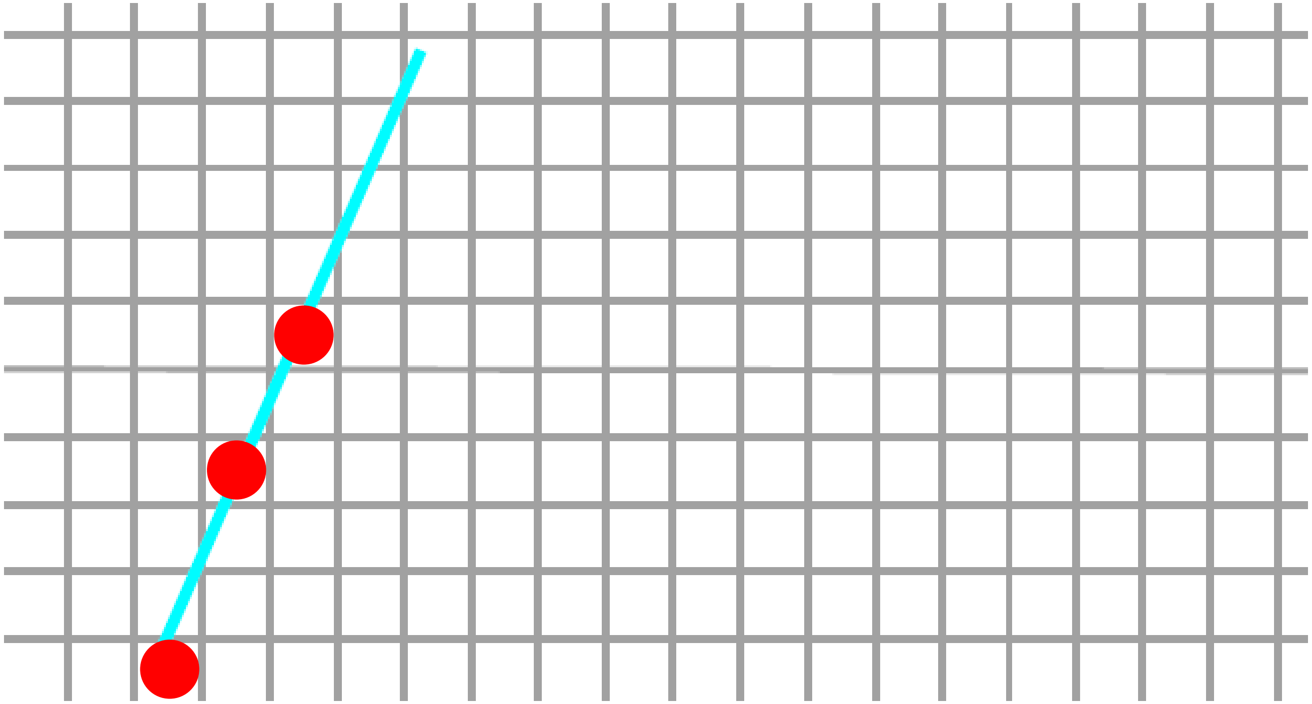
# Digital Differential Analyzer (DDA)

If $\triangle x < \triangle y$



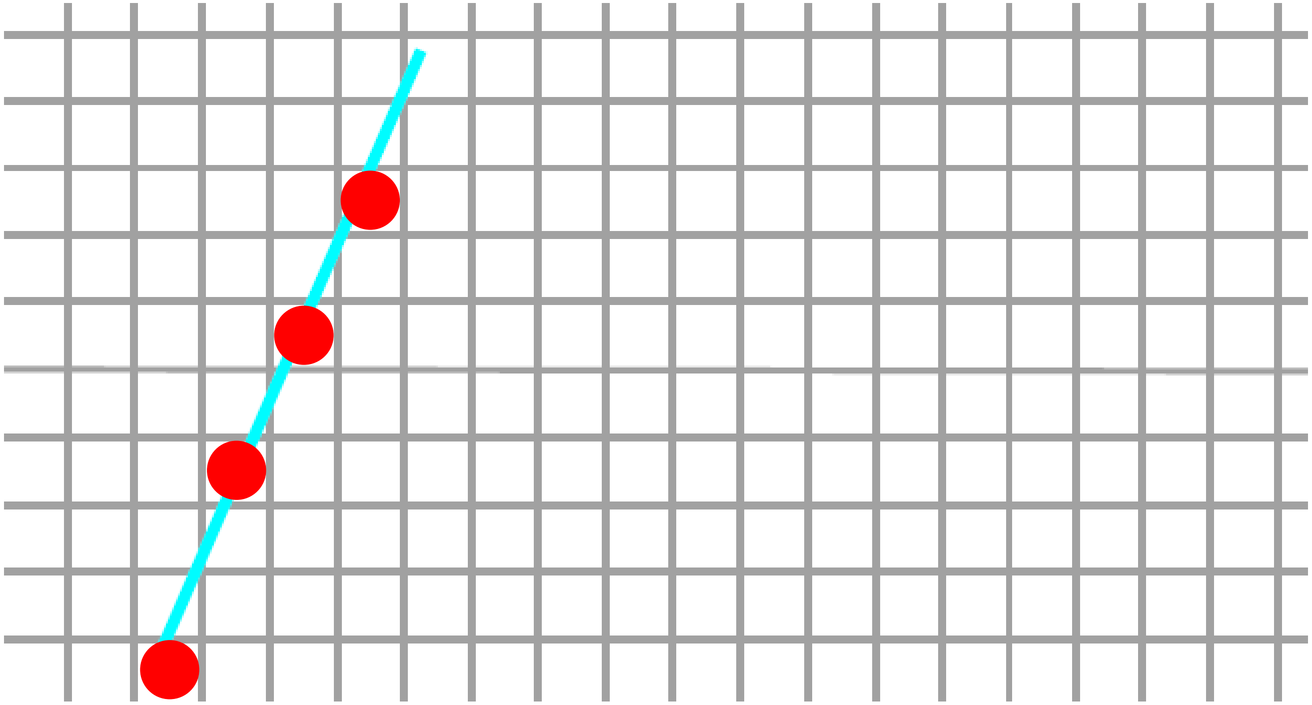
# Digital Differential Analyzer (DDA)

If $\triangle x < \triangle y$



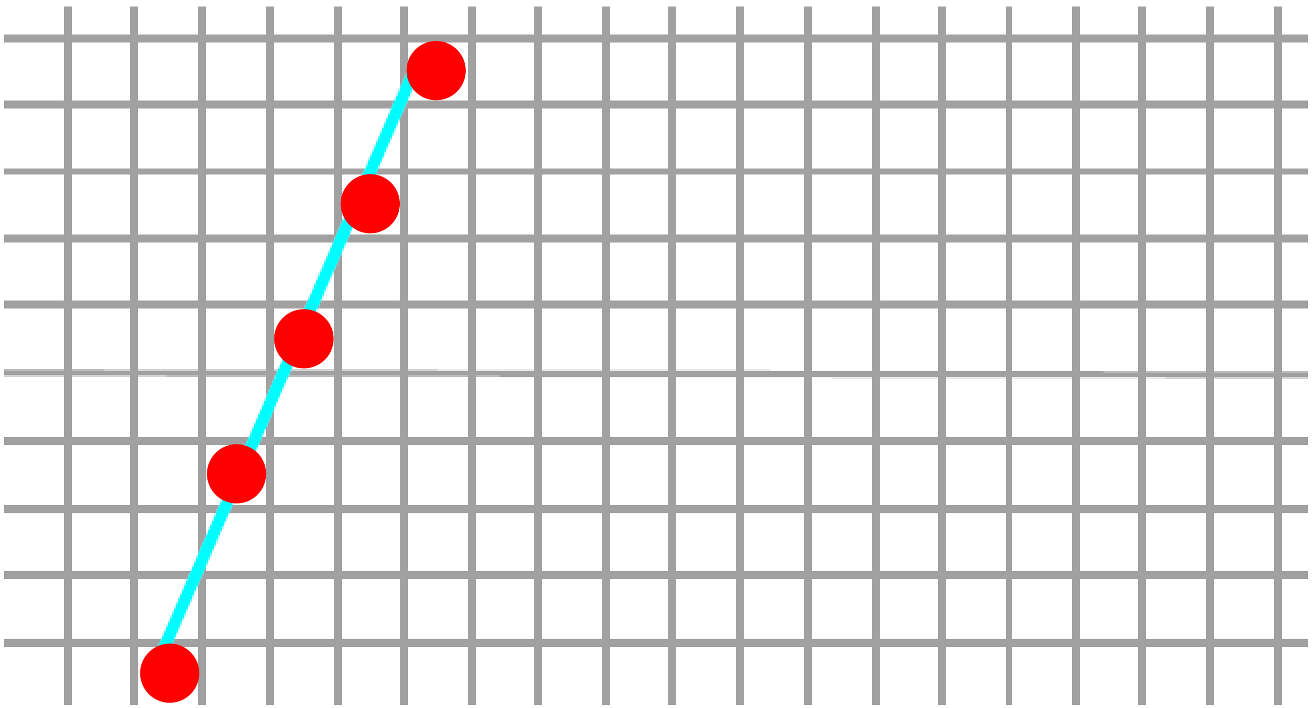
# Digital Differential Analyzer (DDA)

If $\triangle x < \triangle y$



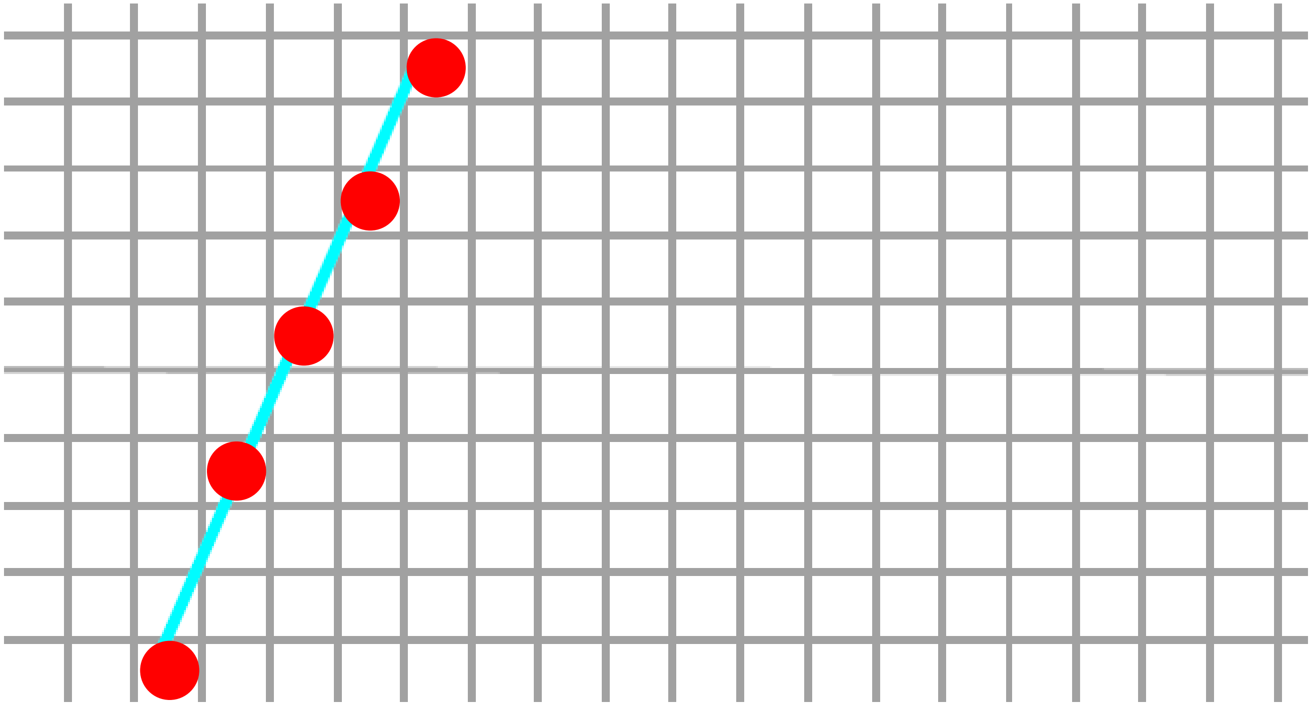
# Digital Differential Analyzer (DDA)

If $\triangle x < \triangle y$



# Digital Differential Analyzer (DDA)

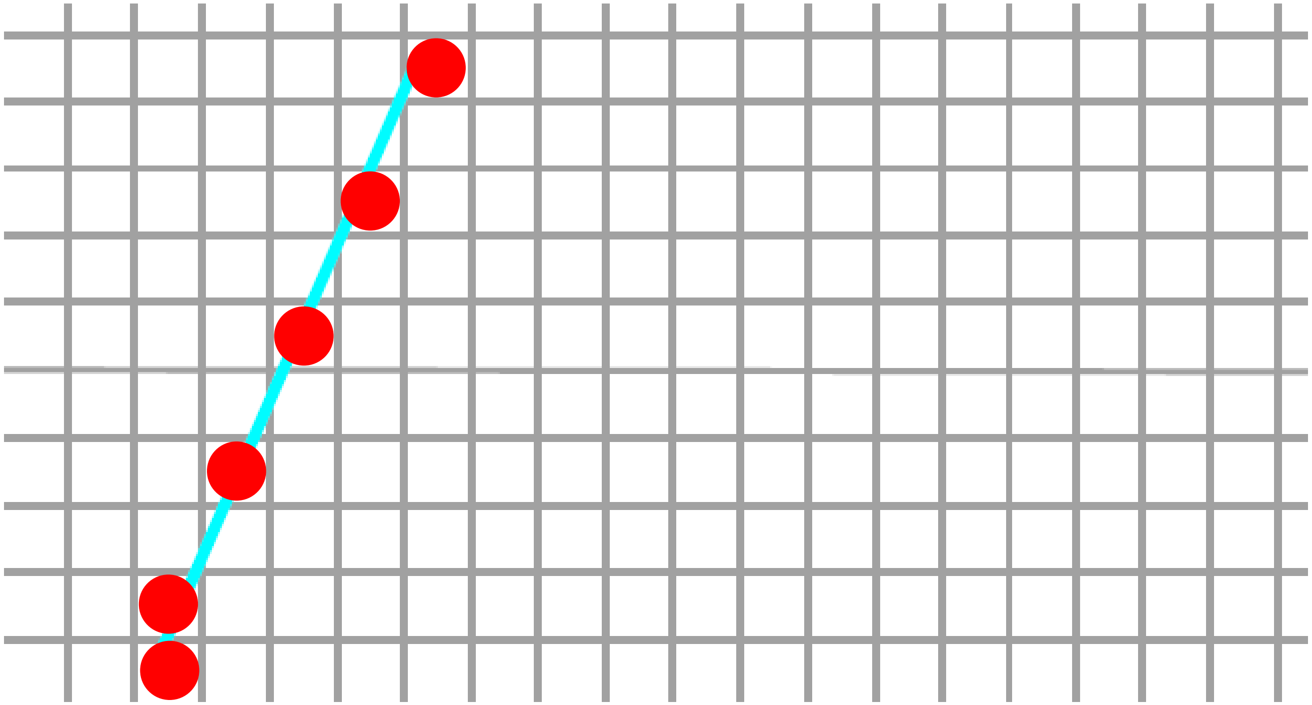
If $\triangle x < \triangle y$



y += 1,  x += 1/m

# Digital Differential Analyzer (DDA)

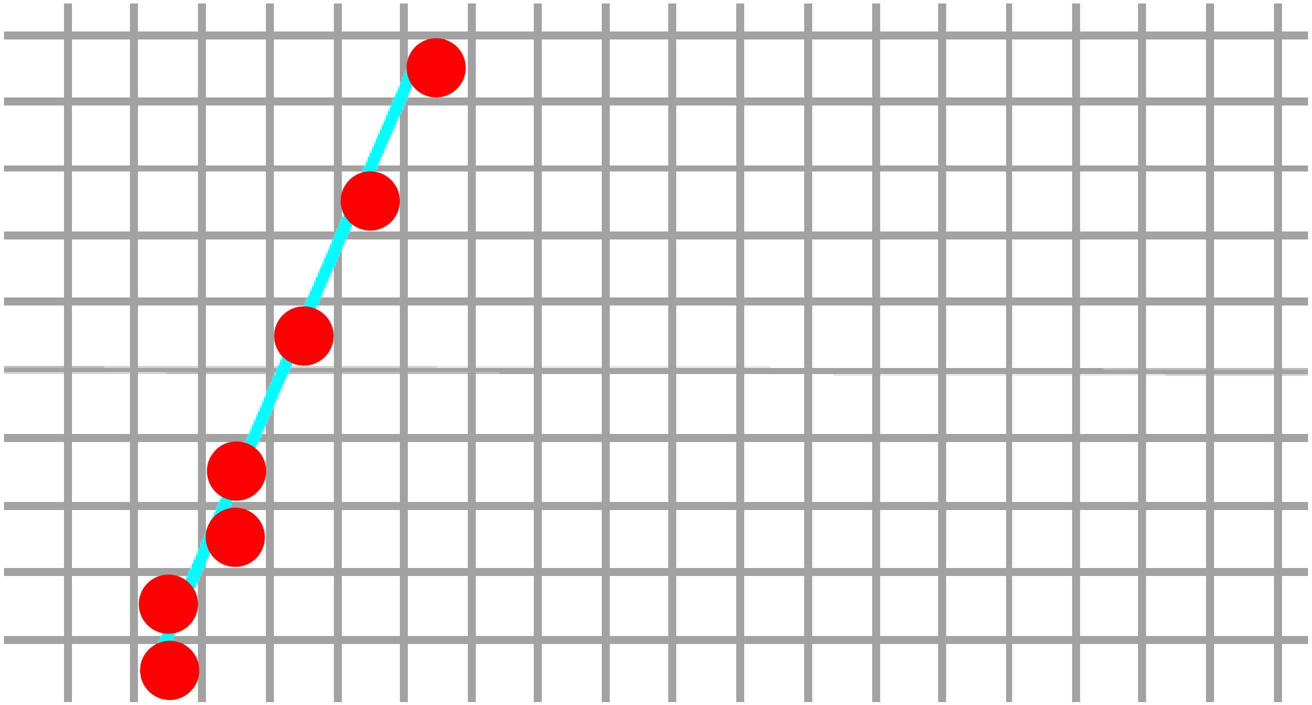
If $\triangle x < \triangle y$



y += 1,  x += 1/m

# Digital Differential Analyzer (DDA)

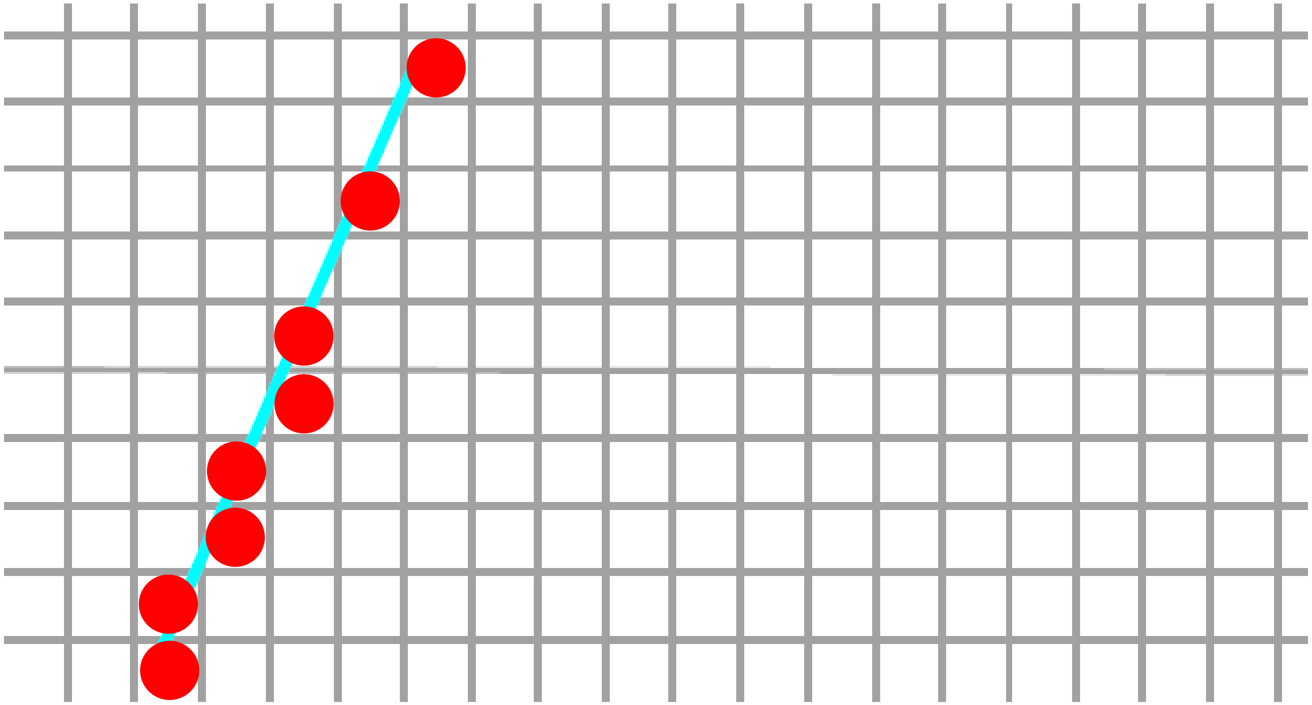
If $\triangle x < \triangle y$



y += 1,  x += 1/m

# Digital Differential Analyzer (DDA)

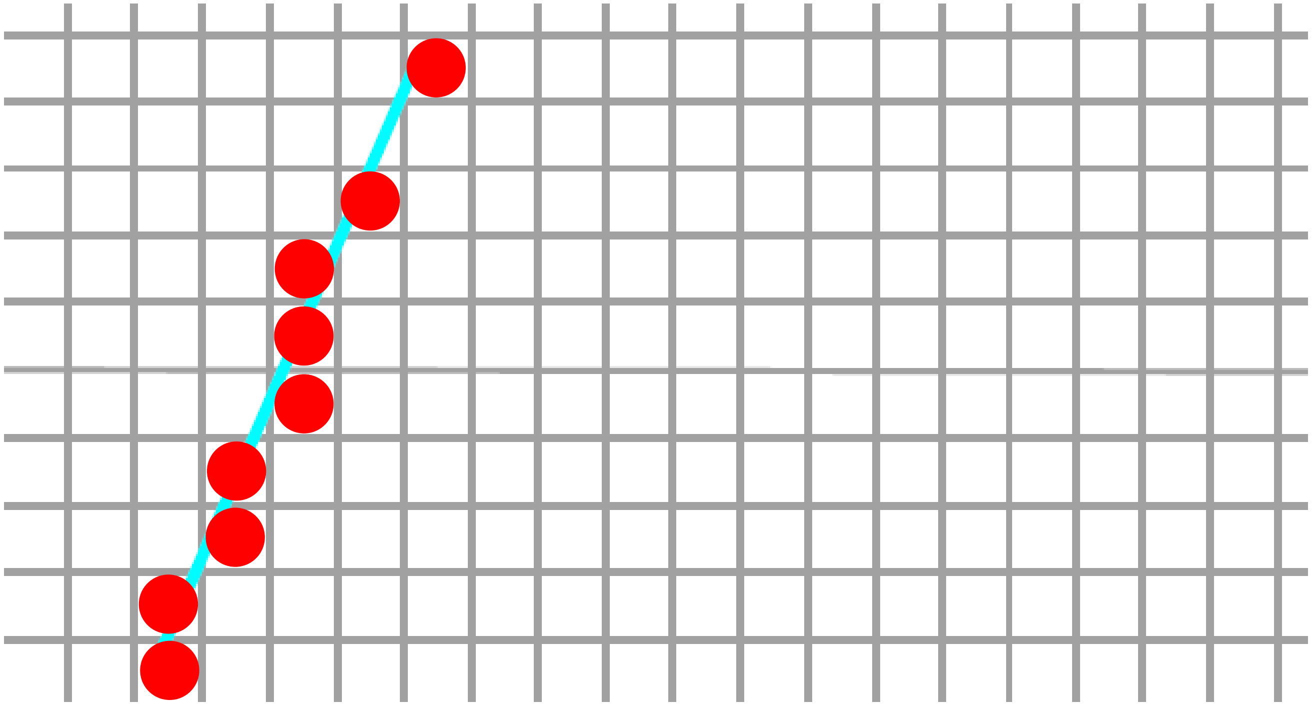
If $\triangle x < \triangle y$



y += 1,  x += 1/m

# Digital Differential Analyzer (DDA)

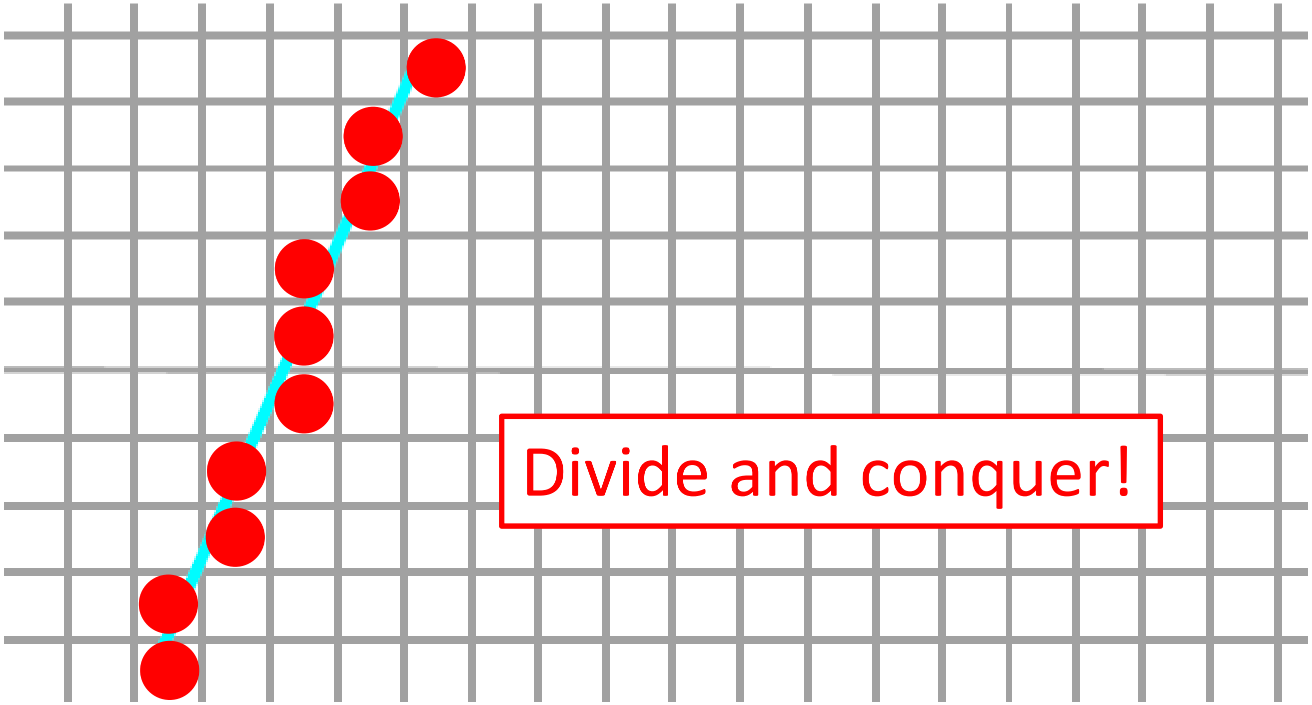
If $\triangle x < \triangle y$



y += 1, x += 1/m

# Digital Differential Analyzer (DDA)

If $\triangle x < \triangle y$




y += 1, x += 1/m

# DDA Algorithm

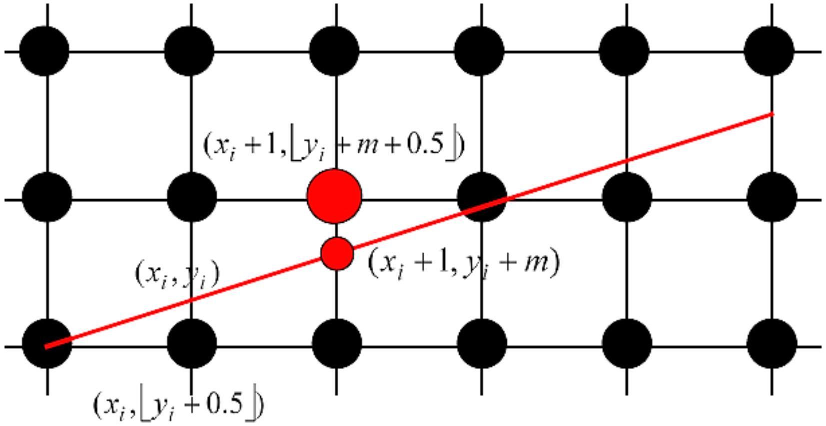
```
#include "device.h"
#include ROUND(a) ((int) (a+0.5))
Void LineDDA( int xa, int ya, int xb, int yb)
{
  int dx =xb-xa, dy=yb-ya, steps, k;
  float xIncrement, yIncrement, x=xa, y=ya;

  if (abs(dx)>abs(dy)) steps=abs(dx);
    else steps=abs(dy);
  xIncrement=dx/(float) steps;
  yIncrement=dx/(float) steps;

  setPixel (ROUND(x), ROUND(y));
  for (k=0;k<steps; k++)
  {  x+=xIncrement; y+=Yincrement; SetPixel (ROUND(x), ROUND(y)); }
}
```

# Bresenham's algorithm (布兰森汉姆算法)

- Introduced in 1967 by **J. Bresenham** of IBM
- Best-fit approximation under some conditions
- In DDA, only $y_i$ is used to compute $y_i$+1, the information for selecting the pixel is neglected
- Bresenham algorithm employs the information to constrain the position of the next pixel



# Notations

- The line segment is from $(x_0, y_0)$ to $(x_1, y_1)$
- Denote $\Delta x = x_1 - x_0 > 0, \Delta y = y_1 - y_0 > 0 \quad m = \Delta y / \Delta x$
- Assume that slope $|m| \leq 1$
- Like DDA algorithm, Bresenham Algorithm also starts from $x = x_0$ and increases x coordinate by 1 each time
- Suppose the i-th point is $(x_i, y_i)$
- Then the next point can only be one of the following two

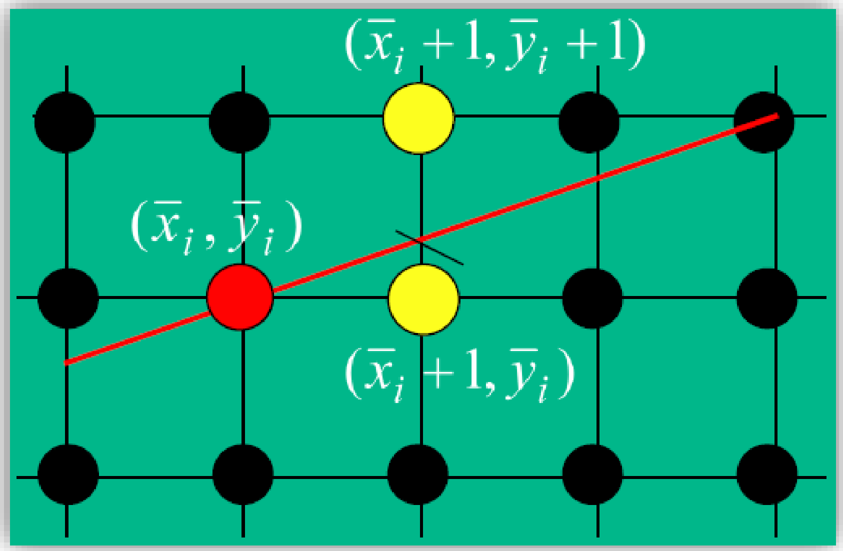
$$(\overline{x}_i + 1, \overline{y}_i) \quad (\overline{x}_i + 1, \overline{y}_i + 1)$$

# Criteria(判别标准)

- We will choose one which distance to the following intersection is shorter

$$x_{i+1} = x_i + 1$$

$$y_{i+1} = mx_{i+1} + B$$
$$= m(x_i + 1) + B.$$

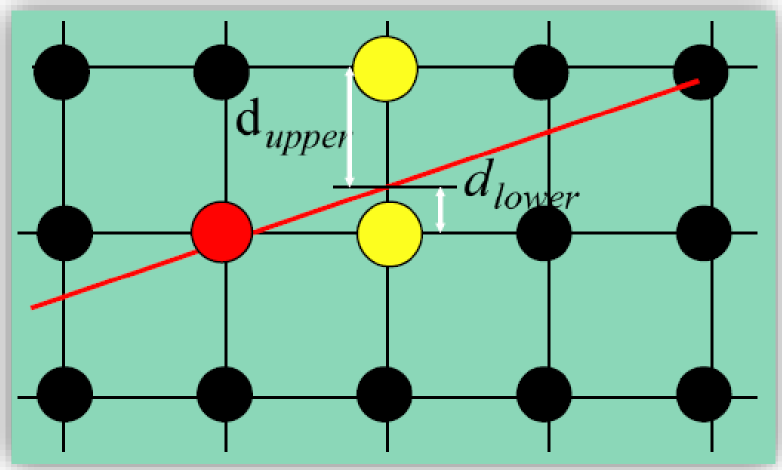


# Computation of Criteria

- The distances are respectively

$$d_{upper} = \bar{y}_i + 1 - y_{i+1}$$
$$= \bar{y}_i + 1 - mx_{i+1} - B$$

$$d_{lower} = y_{i+1} - \bar{y}_i$$
$$= mx_{i+1} + B - \bar{y}_i$$




显然：如果 $d_{lower} - d_{upper} > 0$ 则应取右上方的点；如果 $d_{lower} - d_{upper} < 0$ 则应取右边的点； $d_{lower} - d_{upper} = 0$ 可任取，如取右边点。

# Computation of Criteria

$$d_{lower} - d_{upper} = m(x_i + 1) + B - \bar{y}_i - (\bar{y}_i + 1 - m(x_i + 1) - B)$$

$$= 2\boxed{m}(x_i + 1) - 2\bar{y}_i + 2B - 1$$

division operation

- **It has the same sign with**

$$p_{\mathrm{i}} = \Delta x \bullet (d_{lower} - d_{upper}) = 2\Delta \mathrm{y} \bullet (x_i + 1) - 2\Delta x \bullet \bar{y}_i + (2B - 1)\Delta x$$

$$= 2\Delta y \bullet x_i - 2\Delta x \bullet \bar{y}_i + \boxed{(2B - 1)\Delta x + 2\Delta y}$$

$$= 2\Delta y \bullet x_i - 2\Delta x \bullet \bar{y}_i + c$$

**where**

$$\Delta x = x_1 - x_0, \Delta y = \mathrm{y}_1 - y_0, \quad m = \Delta y / \Delta x$$

$$\mathrm{c} = (2B - 1)\Delta x + 2\Delta y$$

# Restatement of the Criteria

- If $p_i \succ 0$, then $(\bar{x}_i + 1, \bar{y}_i + 1)$ is selected

  If $p_i \prec 0$, then $(\bar{x}_i + 1, \bar{y}_i)$ is selected

  If $p_i = 0$, *arbitrary one*

- **Can we simplify the computation of $p_i$ ?**

$$
\begin{aligned}
p_0 &= 2\Delta y \bullet x_0 - 2\boxed{\Delta x \bullet \bar{y}_0} + (2B-1)\Delta x + 2\Delta y \\
&= 2\Delta y \bullet x_0 - 2\boxed{(\Delta y \bullet x_0 + B \bullet \Delta x)} + (2B-1)\Delta x + 2\Delta y \\
&= 2\Delta y - \Delta x
\end{aligned}
$$

$$y_{i+1} = mx_{i+1} + B$$

# Recursive for computation of $p_i$

- As

$$p_{i+1} - p_i = (2\Delta y \bullet x_{i+1} - 2\Delta x \bullet \bar{y}_{i+1} + c) - (2\Delta y \bullet x_i - 2\Delta x \bullet \bar{y}_i + c)$$
$$= 2\Delta y - 2\Delta x(\bar{y}_{i+1} - \bar{y}_i)$$

- **If** $p_i \le 0$ **then** $\bar{y}_{i+1} - \bar{y}_i = 0$ **therefore**

$$p_{i+1} = p_i + 2\Delta y$$

- **If** $p_i > 0$ **then** $\bar{y}_{i+1} - \bar{y}_i = 1$ **therefore**

$$p_{i+1} = p_i + 2\Delta y - 2\Delta x$$

# Summary of Bresenham Algorithm

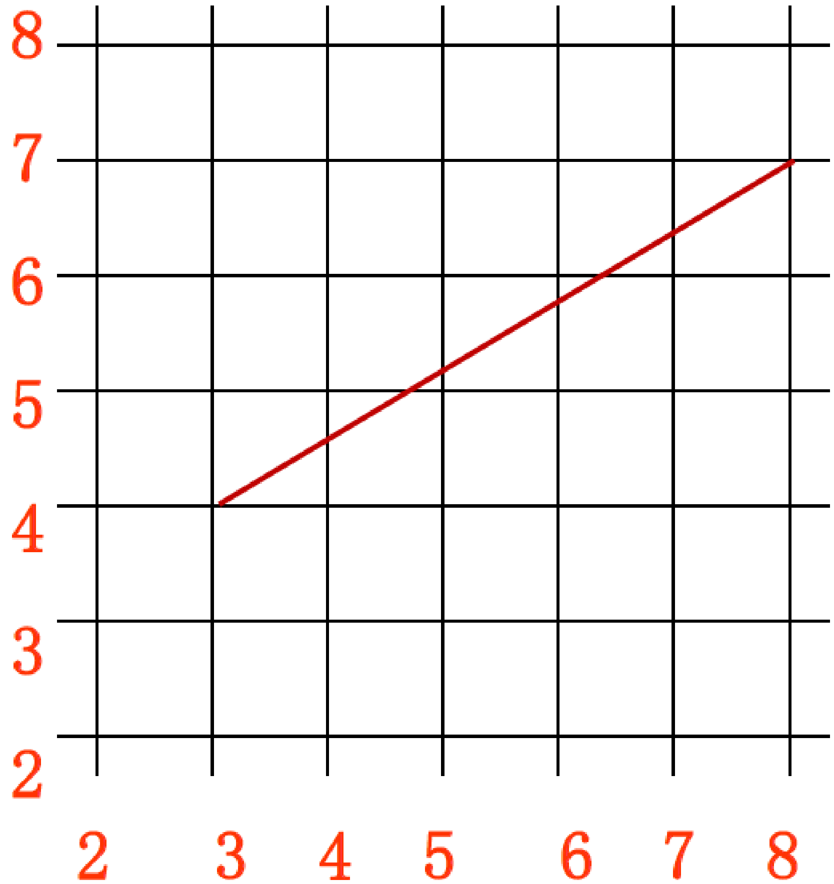
- **draw** $(x_0, y_0)$
- **Calculate** $\Delta x, \Delta \mathrm{y}, 2\Delta \mathrm{y}, 2\Delta \mathrm{y} - 2\Delta x$, $\mathrm{p}_0 = 2\Delta \mathrm{y} - \Delta x$
- **If** $\mathrm{p_i} \le 0$ **draw** $(\mathrm{x}_{i+1}, \bar{y}_{i+1}) = (\mathrm{x}_i + 1, \bar{y}_i)$

  **and compute** $\mathrm{p_{i+1}} = \mathrm{p_i} + 2\Delta y$
- **If** $\mathrm{p_i} > 0$ **draw** $(\mathrm{x}_{i+1}, \bar{y}_{i+1}) = (\mathrm{x}_i + 1, \bar{y}_i + 1)$

  **and compute** $\mathrm{p_{i+1}} = \mathrm{p_i} + 2\Delta y - 2\Delta x$
- **Repeat the last two steps**

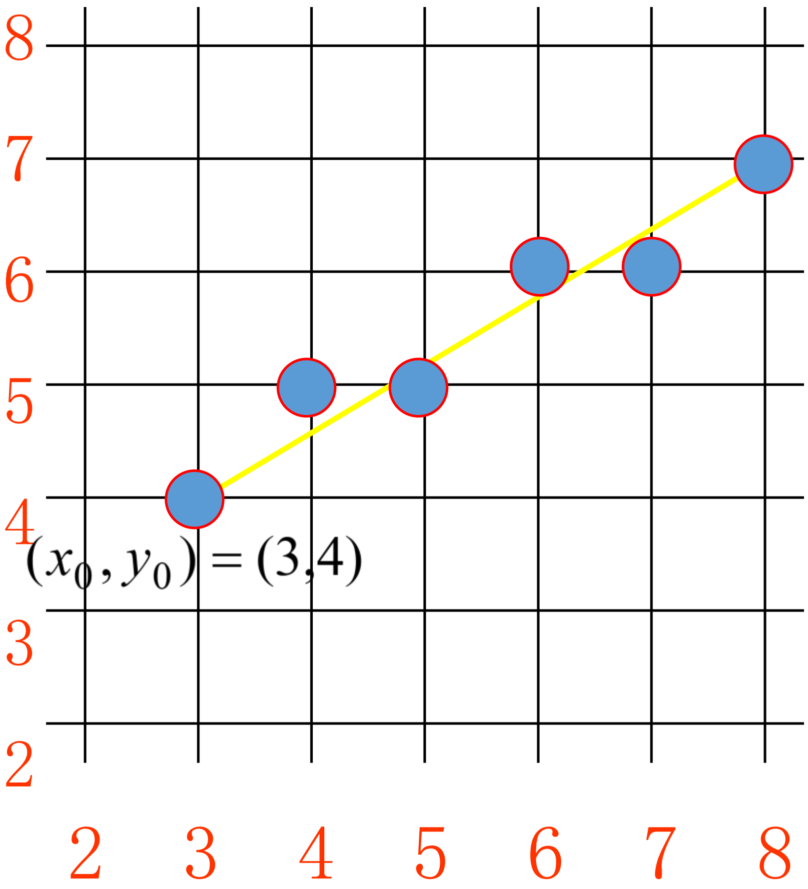
# Example

- Draw line segment (3,4)-(8,7)



# (Continued)： Draw line segment (3,4)-(8,7)

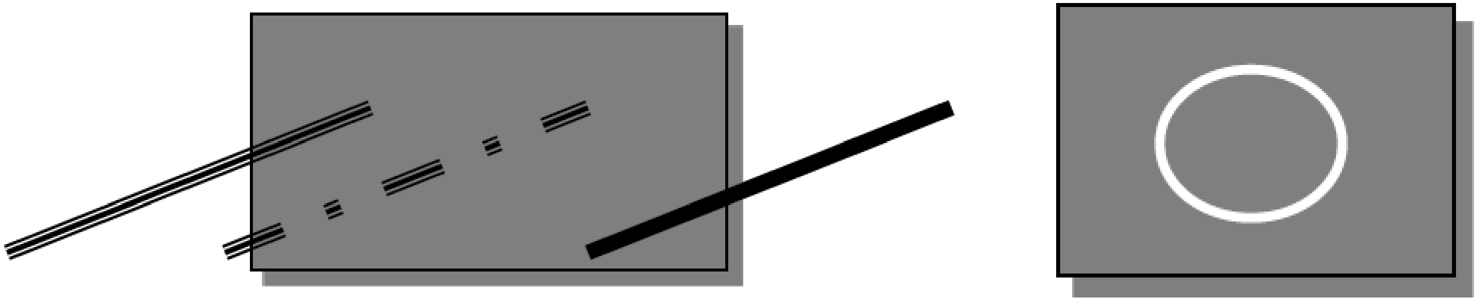
| k | $p_k$ | $(x_{k+1}, y_{k+1})$ |
|---|---|---|
| **0** | **1** | **(4,5)** |
| **1** | **-3** | **(5,5)** |
| **2** | **3** | **(6,6)** |
| **3** | **-1** | **(7,6)** |
| **4** | **5** | **(8,7)** |



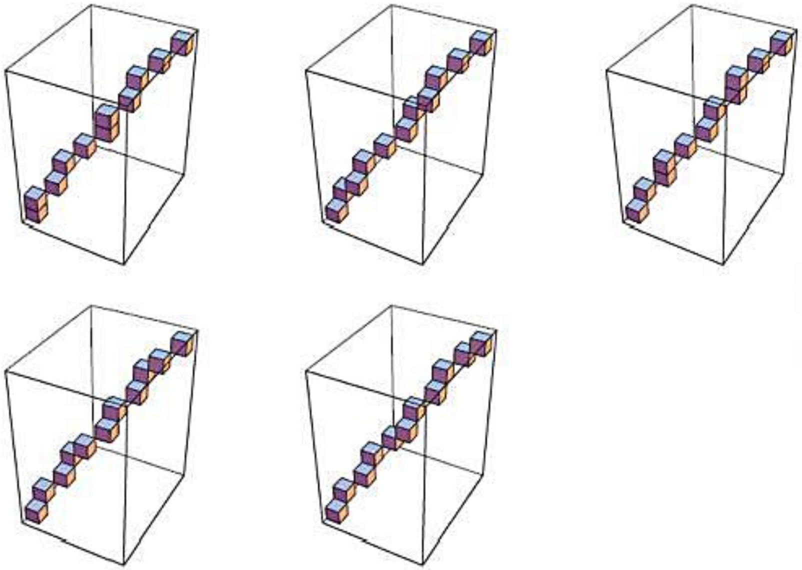

注： $p_0 = 2\Delta y - \Delta x \quad p_{i+1} = p_i + 2\Delta y \quad p_{i+1} = p_i + 2\Delta y - 2\Delta x$

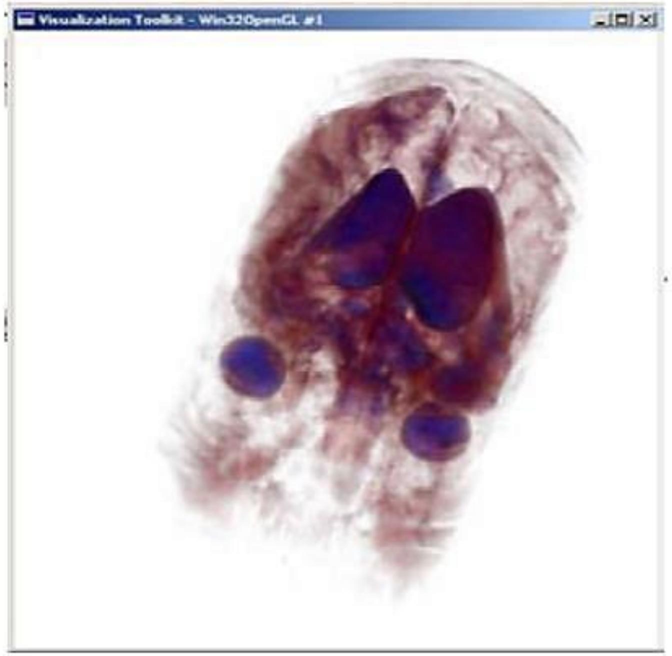
# More Raster Line Issues

- The coordinates of endpoints are not integer
- Generalize to draw other primitives: circles, ellipsoids
- Line pattern and thickness?



# 3D Bresenham algorithm






Computer Graphics @ ZJU

Hongxin Zhang, 2014

# What Makes a Good Line?

- Not too jaggy
- Uniform thickness of lines at different angles
- Symmetry, Line(P,Q) = Line(Q,P)

- A good line algorithm should be fast.

# Line Attributes

- line width
- dash patterns
- end caps: butt, round, square

# Line Attributes

- Joins: round, bevel, miter

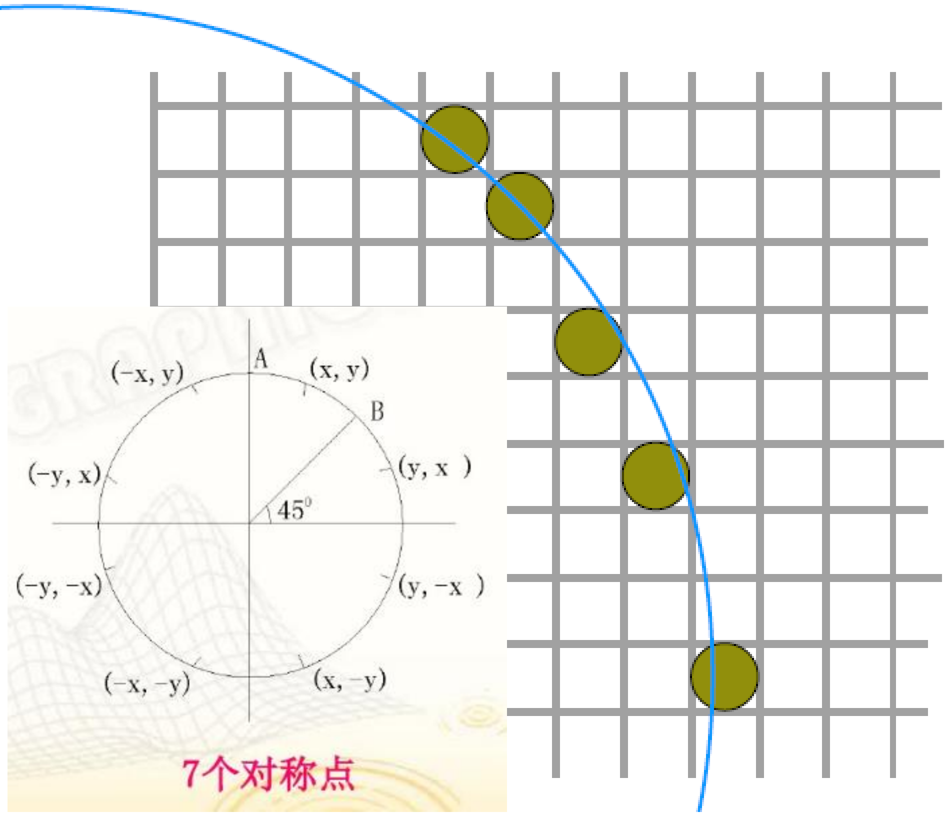
# Scan conversion of circles

A circle with center $(x_c, y_c)$ and radius $r$:

$$(x-x_c)^2 + (y-y_c)^2 = r^2$$

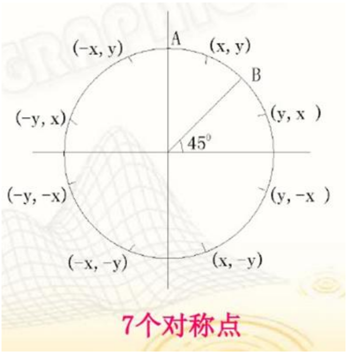
orthogonal coordinate

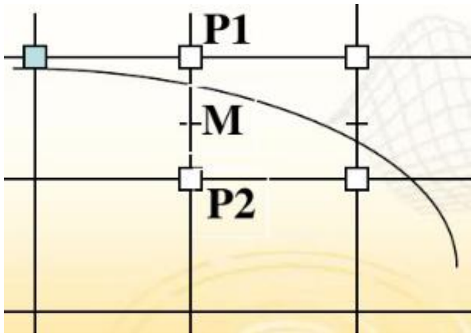
$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$



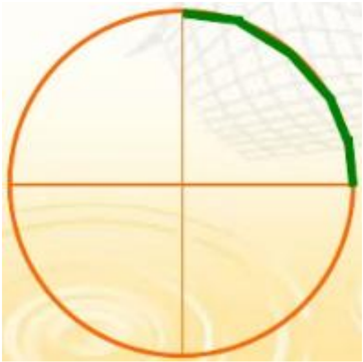
# Scan conversion of circles

- 中点画圆法
- 圆弧的多边形逼近法
- 正负法画圆
- Bresenham画圆

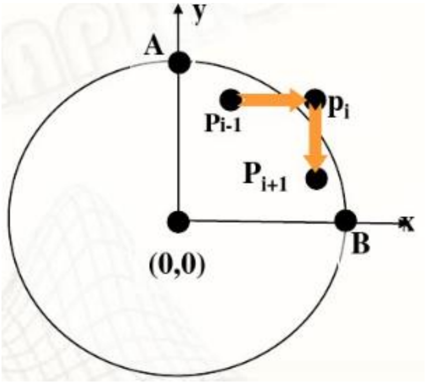


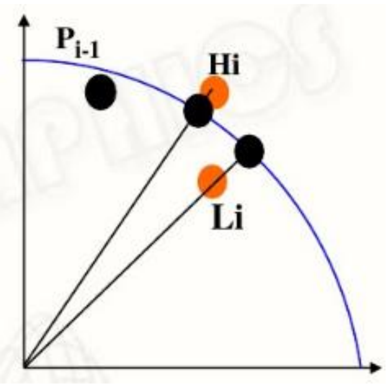

中点画圆法



圆弧的多边形逼近法



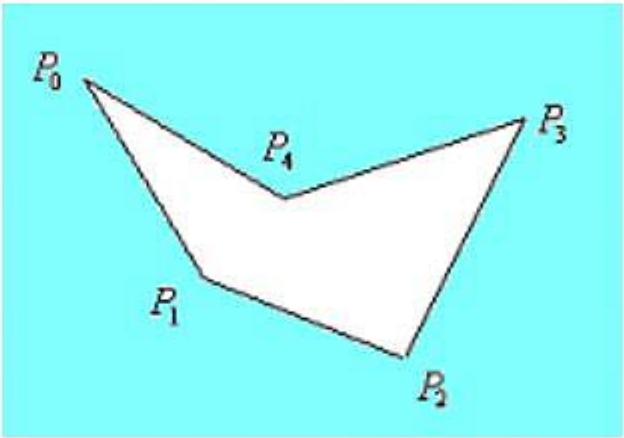

正负法画圆



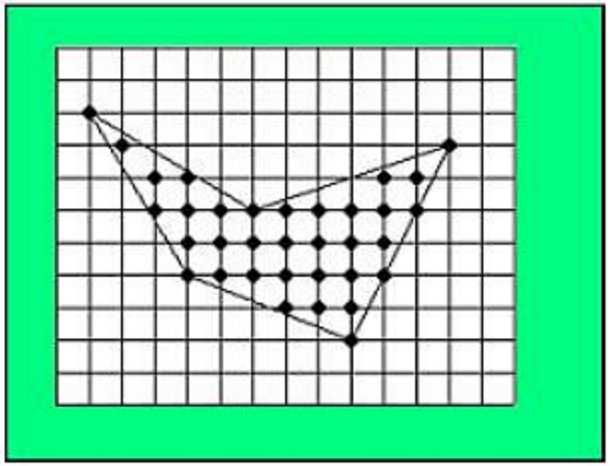

Bresenham画圆

# Scan conversion of polygon

- Polygon representation




By vertex



By lattice

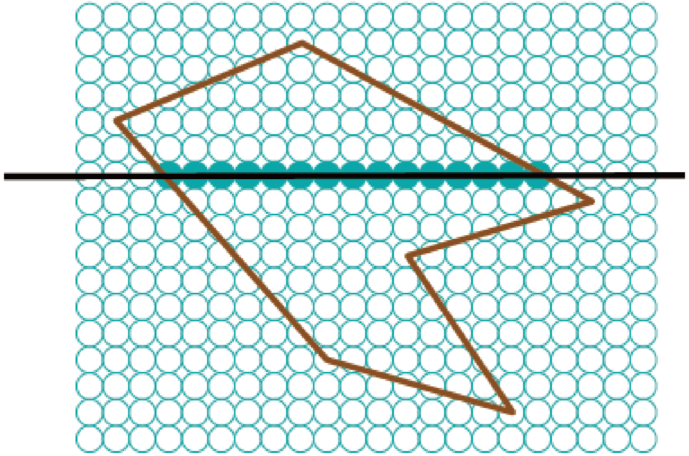
- Polygon filling:

  vertex representation → lattice representation

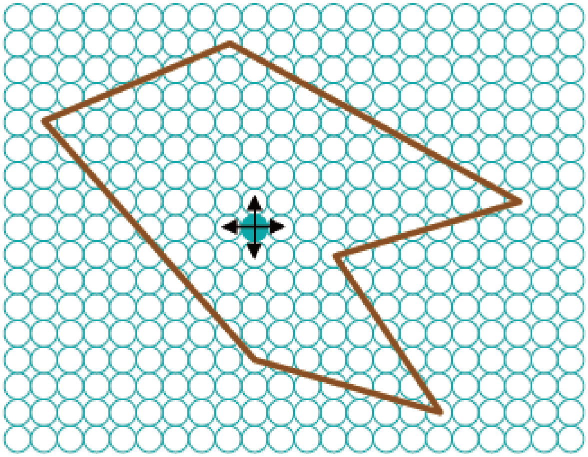
# Polygon Rasterization

Takes shapes like triangles and determines which pixels to set

1. Polygon scan-conversion
   - sweep the polygon by scan line, set the pixels whose center is inside the polygon for each scan line

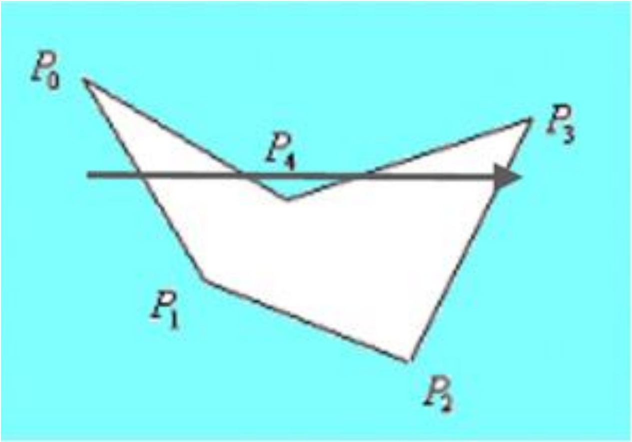


2. Polygon fill
   - select a pixel inside the polygon
   - grow outward until the whole polygon is filled



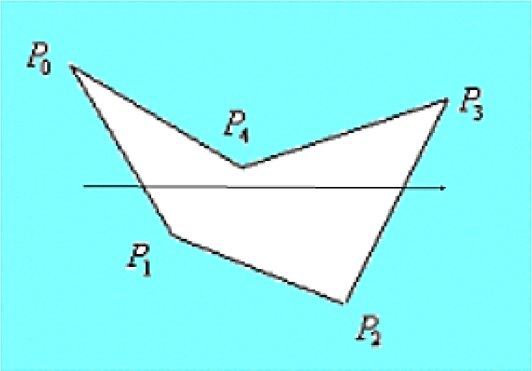
# Polygon filling

- fill a polygonal area --> test every pixel in the raster to see if it lies inside the polygon.

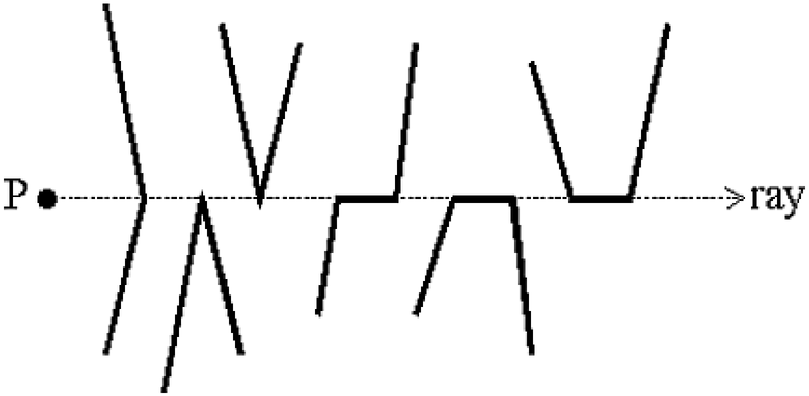


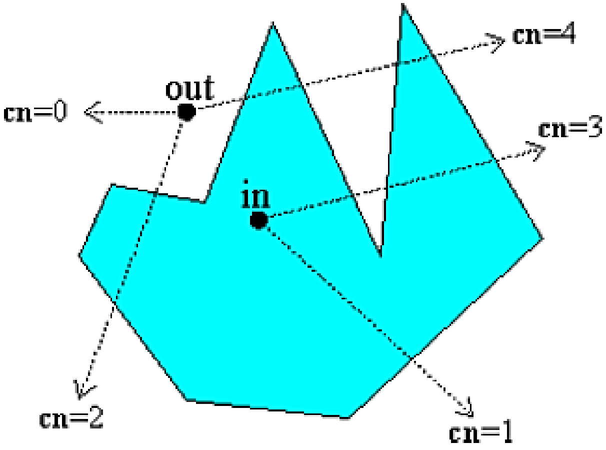

even-odd test

# Inside Check




even-odd test



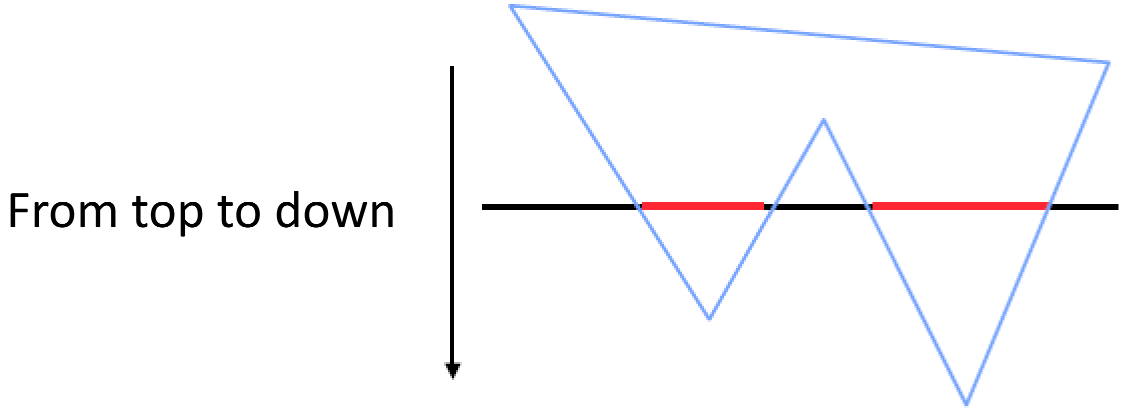

Computer Graphics 2014, ZJU

# Scan-line Methods

- Makes use of the coherence properties
  - Spatial coherence : Except at the boundary edges, adjacent pixels are likely to have the same characteristics
  - Scan line coherence : Pixels in the adjacent scan lines are likely to have the same characteristics
- Uses intersections between area boundaries and scan lines to identify pixels that are inside the area

# Scan Line Method

- Proceeding from left to right the intersections are paired and intervening pixels are set to the specified intensity
- Algorithm
  - Find the intersections of the scan line with all the edges in the polygon
  - Sort the intersections by increasing X-coordinates
  - Fill the pixels between pair of intersections

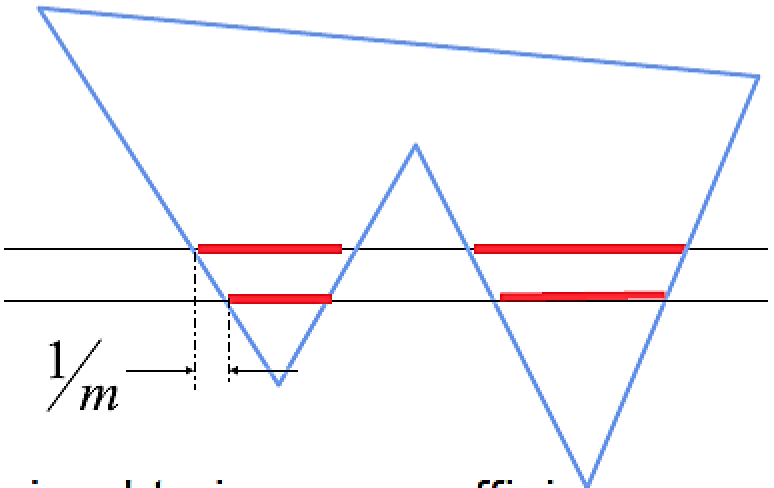



**Discussion : How to speed up, or how to avoid calculating intersection**

# Efficiency Issues Scan-line Methods

- Intersections could be found using edge coherence
  the X-intersection value $x_{i+1}$ of the lower scan line can be computed from the X-intersection value $x_i$ of the preceeding scanline as

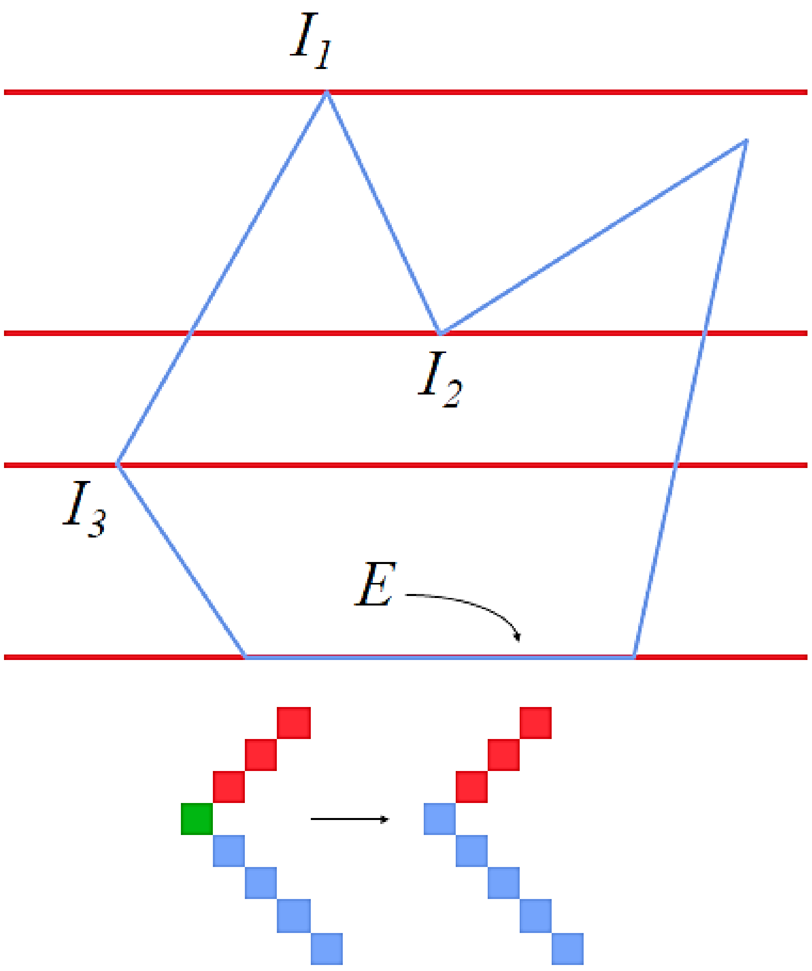
$$x_{i+1} = x_i + \frac{1}{m}$$




- List of active edges could be maintained to increase efficiency

# Special cases for Scan-line Methods

- Overall topology should be considered for intersection at the vertices
- Intersections like $I_1$ and $I_2$ should be considered as two intersections
- Intersections like $I_3$ should be considered as one intersection
- Horizontal edges like $E$ need not be considered

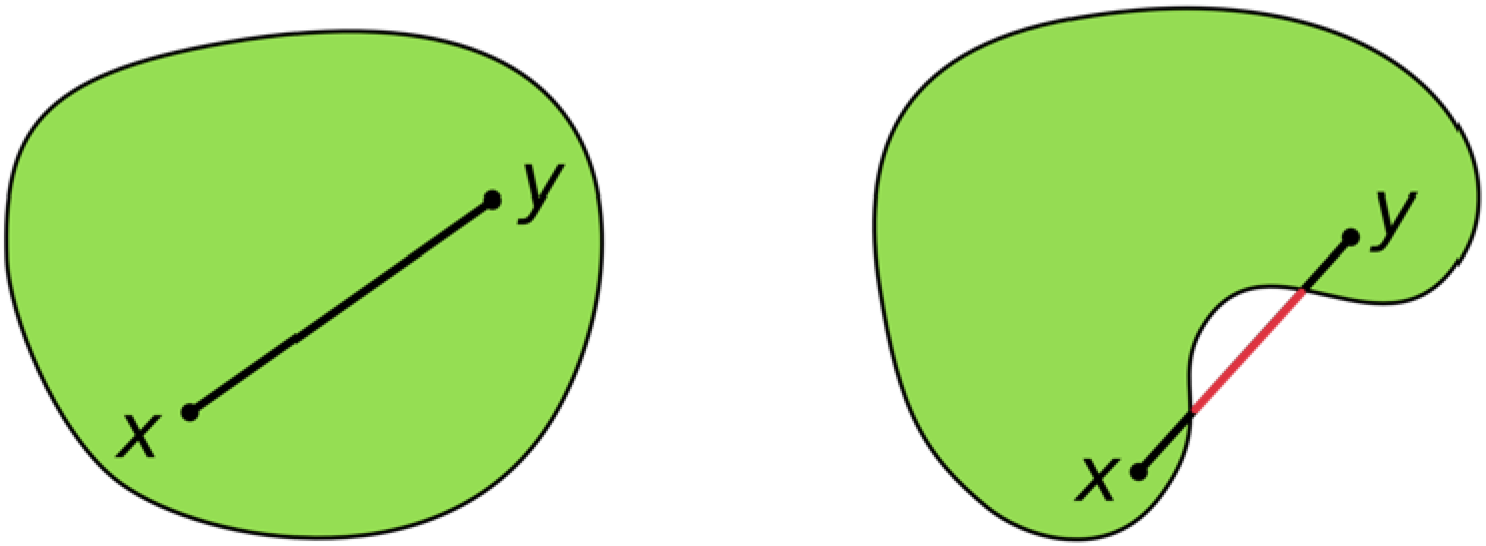


# Advantages of Scan Line method

- The algorithm is efficient
- Each pixel is visited only once
- Shading algorithms could be easily integrated with this method to obtain shaded area

- Efficient could be further improved if polygons are convex
- Much better if they are only triangles

# What is Convex?

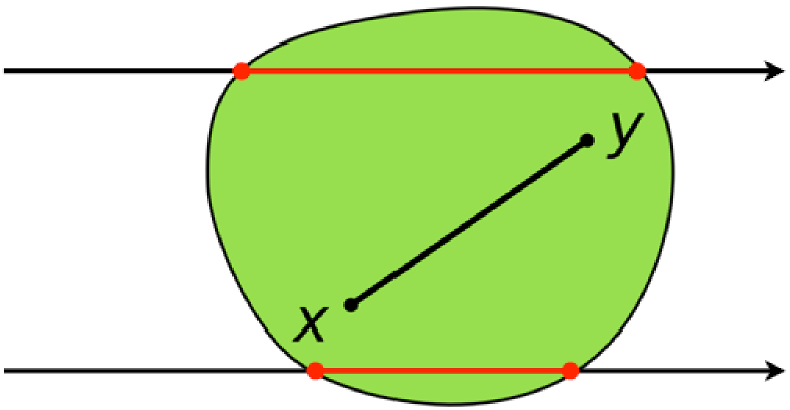



A set C in S is said to be **convex** if, for all x and y in C and all $t$ in the interval $[0,1]$, the point

$$(1 - t)\, x + t\, y$$

is in C.

# Convex Polygon Rasterization




One in and one out

Computer Graphics 2014, ZJU

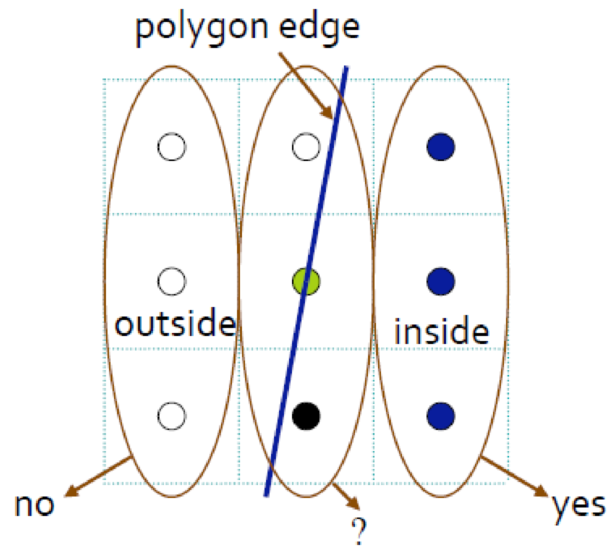
# Triangle Rasterization

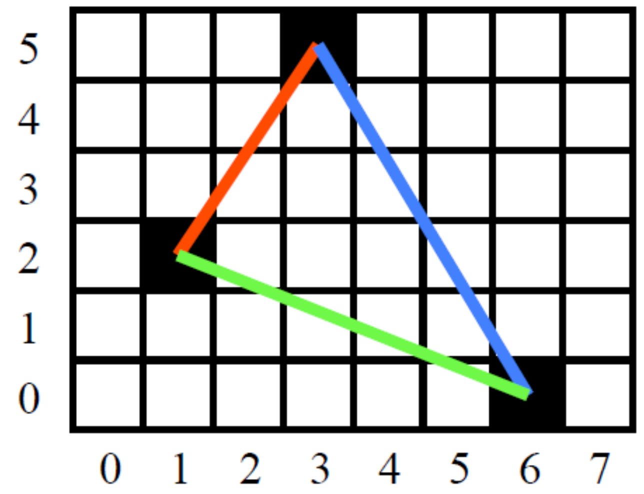
Two questions:

- which pixel to set?
- what color to set each pixel to?

How would you rasterize a triangle?

1. Edge-walking
2. Edge-equation
3. Barycentric-coordinate based

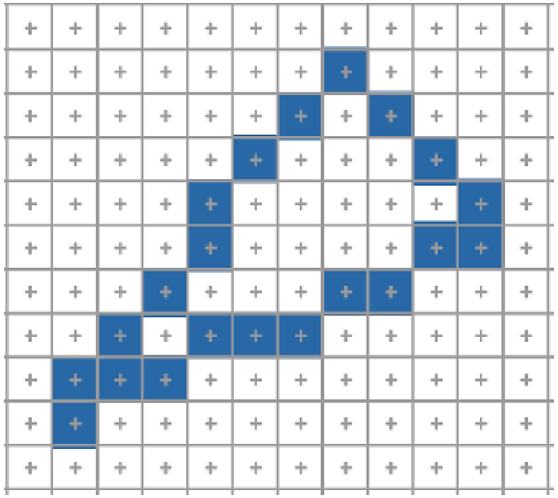


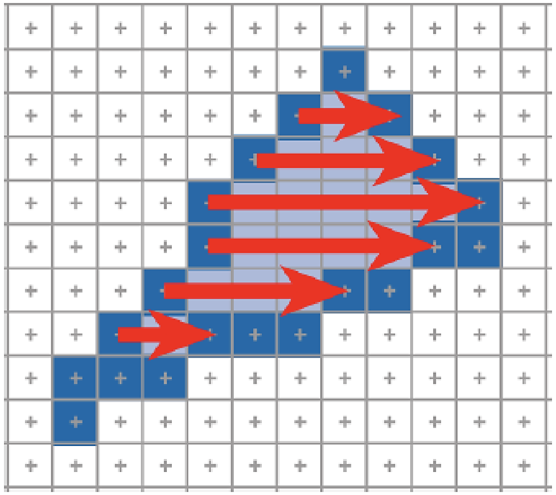



# Edge Walking

Idea:

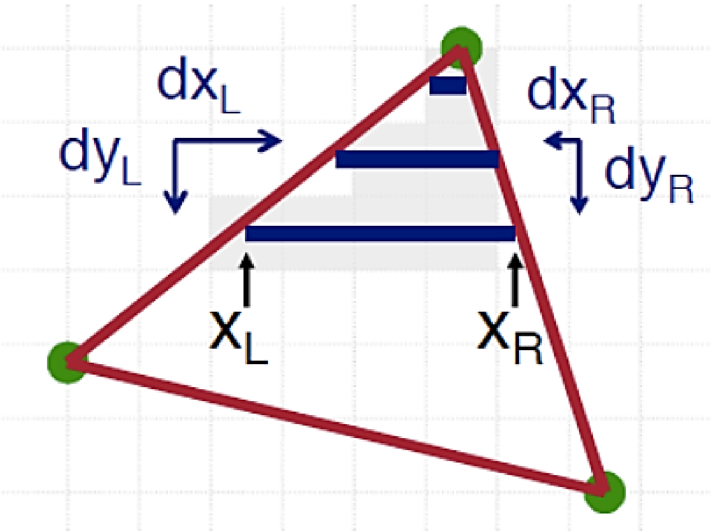
- scan top to bottom in scan-line order
- "walk" edges: use edge slope to update coordinates incrementally
- on each scan-line, scan left to right (horizontal span), setting pixels
- stop when bottom vertex or edge is reached





# Edge Walking

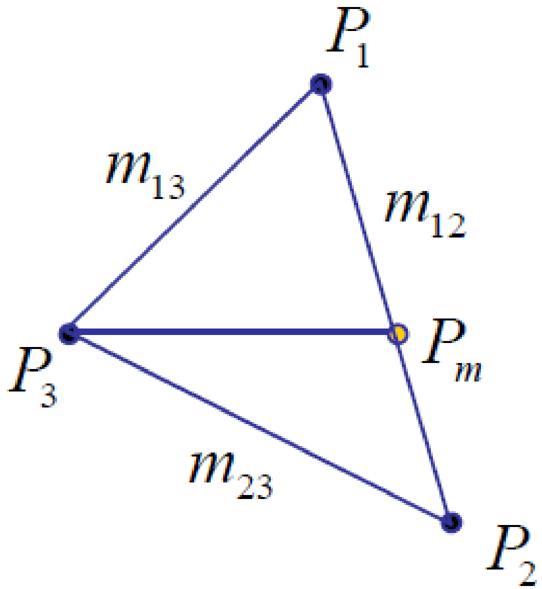
```
void edge_walking(vertices T[3])
{
  for each edge pair of T {
    initialize xL, xR;
    compute dxL/dyL and dxR/dyR;
    for scanline at y {
      for (int x = xL; x <= xR; x++) {
        set_pixel(x, y);
      }
    }
    xL += dxL/dyL;
    xR += dxR/dyR;
  }
}
```




Funkhouser09

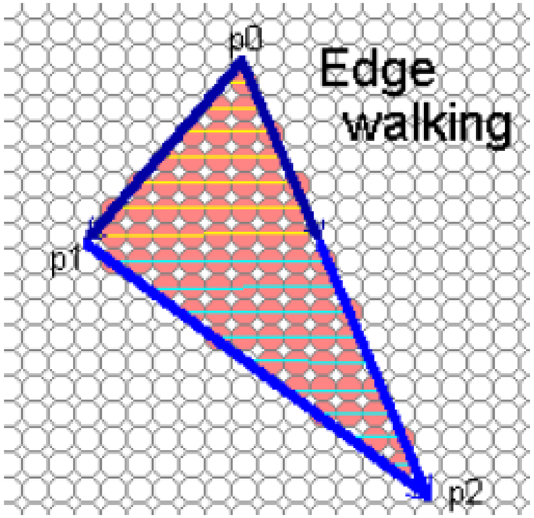
# Edge Walking Triangle

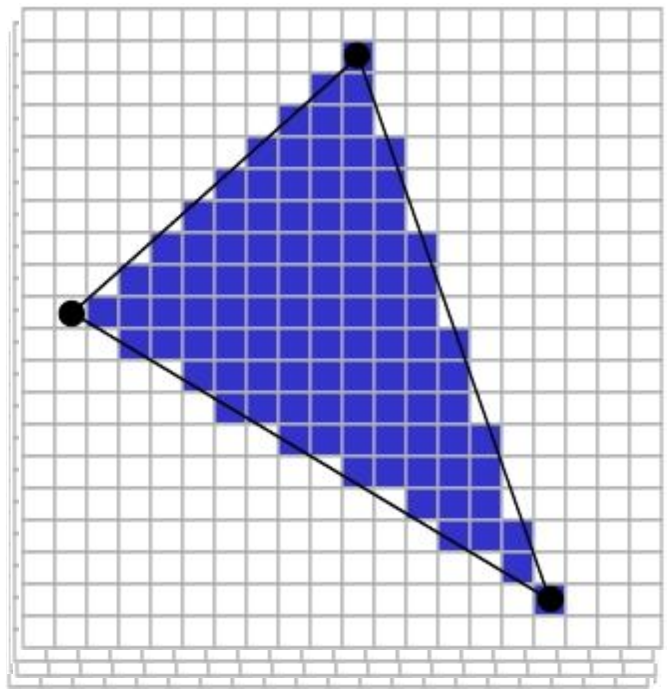
- Split triangles into two "trapezoids" with continuous left and right edges




$$\text{scanTrapezoid}(x_3, x_m, y_3, y_1, \frac{1}{m_{13}}, \frac{1}{m_{12}})$$

$$\text{scanTrapezoid}(x_2, x_2, y_2, y_3, \frac{1}{m_{23}}, \frac{1}{m_{12}})$$



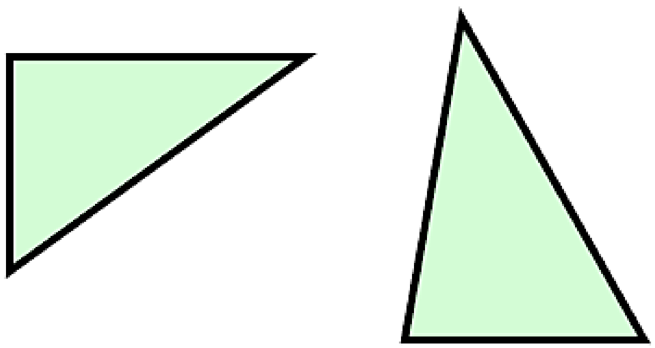


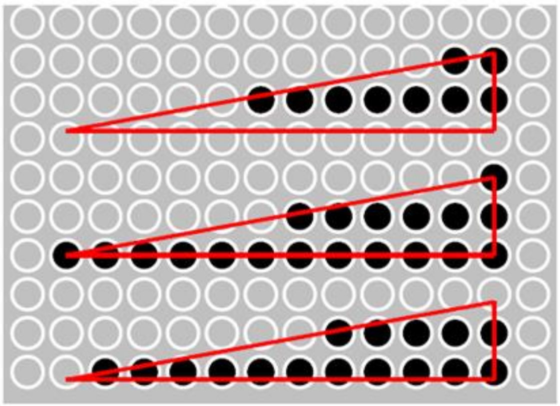
# Edge Walking

Advantage: very simple

Disadvantages:

- very serial (one pixel at a time) ⇒ can't parallelize
- inner loop bottleneck if lots of computation per pixel
- special cases will make your life miserable
  - horizontal edges: computing intersection causes divide by 0!

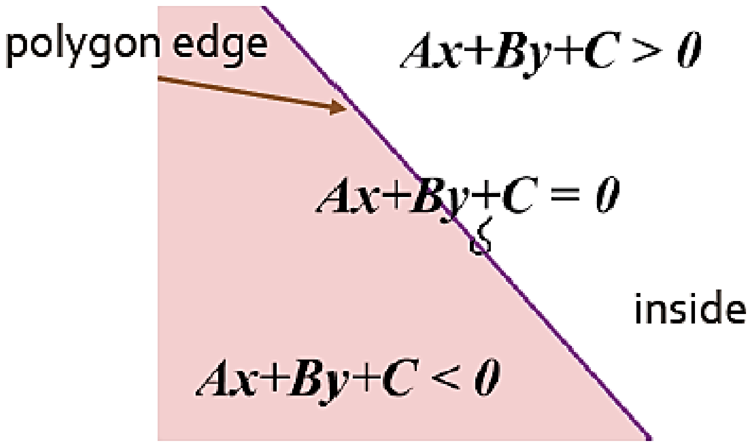




  - sliver: not even a single pixel wide

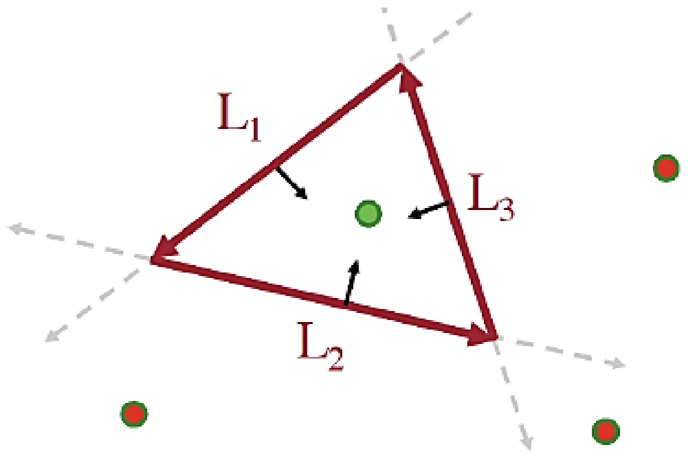
# Edge Equations

1. compute edge equations from vertices
   - orient edge equations: let negative halfspaces be on the triangle's exterior (multiply by -1 if necessary)
2. scan through each pixel and evaluate against all edge equations
3. set pixel if all three edge equations > 0



# Edge Equations

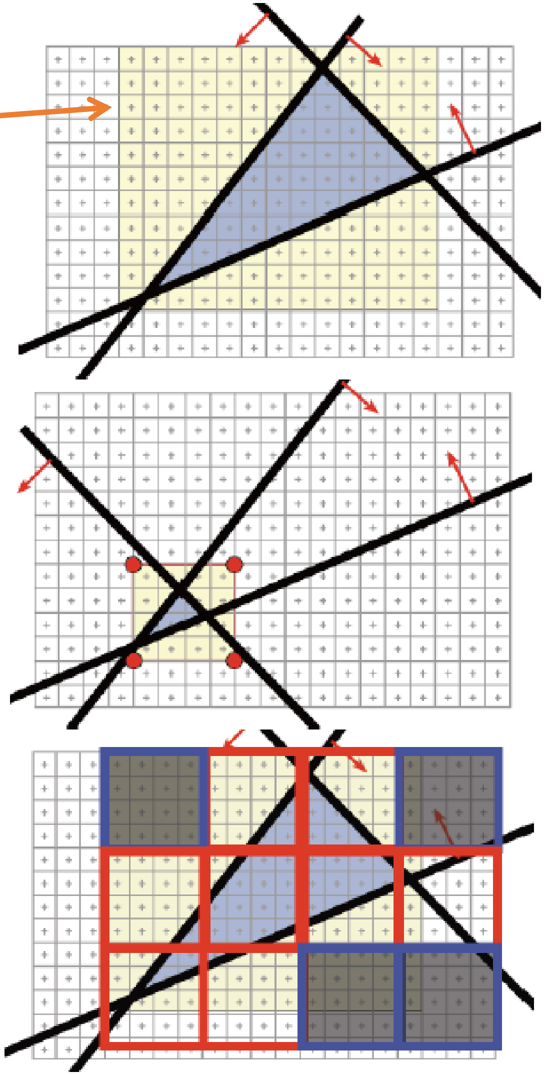
```
void edge_equations(vertices T[3])
{
  bbox b = bound(T);
  foreach pixel(x, y) in b {
    inside = true;
    foreach edge line Li of Tri {
      if (Li.A*x+Li.B*y+Li.C < 0) {
        inside = false;
      }
    }
    if (inside) {
      set_pixel(x, y);
    }
  }
}
```



# Edge Equations

## Can we reduce #pixels tested?

1. compute a bounding box:
   $x_{min}, y_{min}, x_{max}, y_{max}$ of triangle
2. compute edge equations from vertices
   - orient edge equations: let negative halfspaces be on the triangle's exterior (multiply by -1 if necessary)
   - can be done incrementally per scan line
3. scan through *each* pixel in bounding box and evaluate against all edge equations
4. set pixel if all three edge equations $> 0$

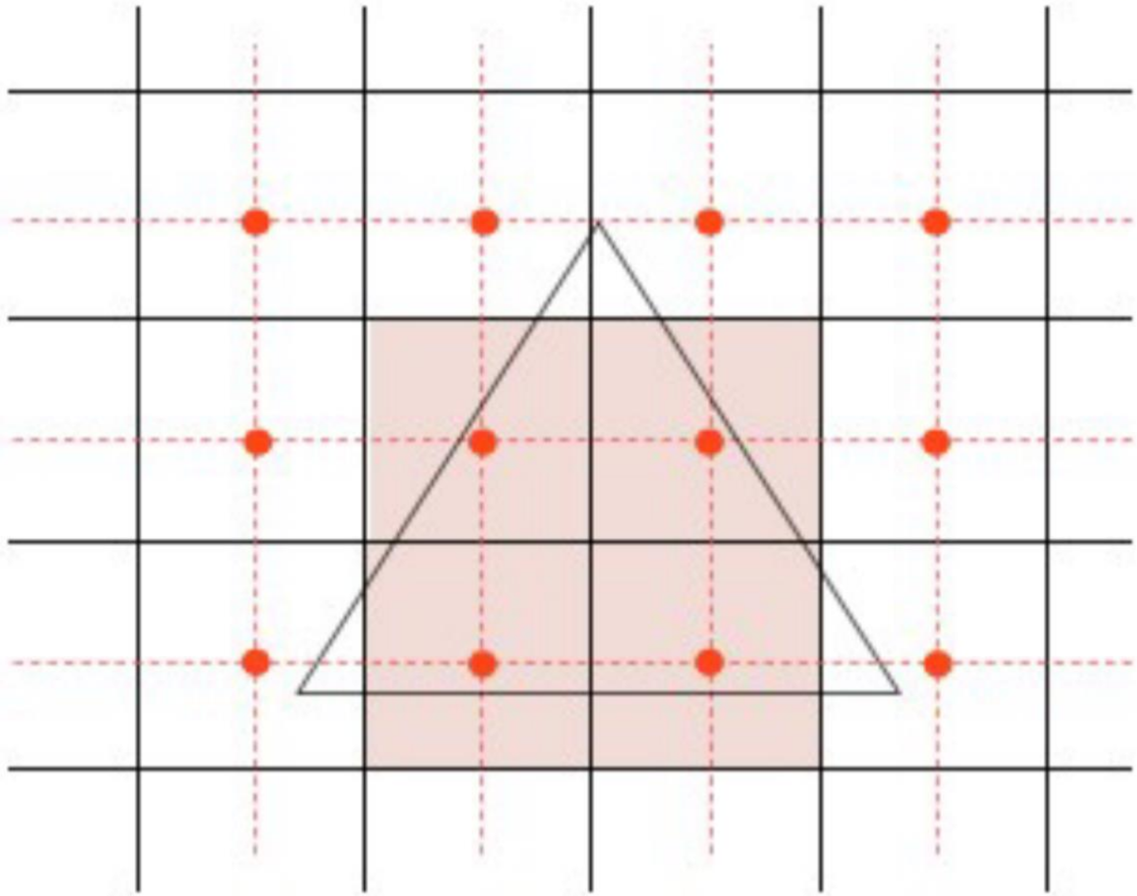


## Hierarchical bounding boxes

- how to quickly exclude a bounding box?

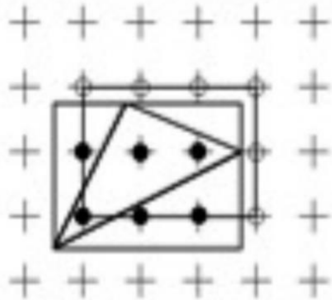
# Triangle Rasterization

**Output fragment if pixel center is inside the triangle**



# Triangle Rasterization

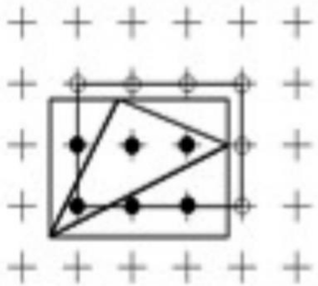
```
rasterize( vert v[3] )
{
  bbox b; bound3(v,b);
  for( int y=b.ymin; y<b.ymax, y++ )
    for( int x=b.xmin; x<b.xmax, x++ )
      if( inside3(v,x,y) )
        fragment(x,y);
}
```



**GPUs contain triangle rasterization hardware**
**Can output billions of fragments per second**

Computer Graphics 2014, ZJU

# Compute Bound Box



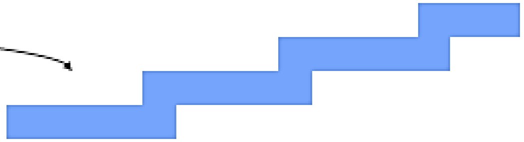
```
bound3( vert v[3], bbox& b )
{
  b.xmin = ceil(min(v[0].x, v[1].x, v[2].x));
  b.xmax = ceil(max(v[0].x, v[1].x, v[2].x));
  b.ymin = ceil(min(v[0].y, v[1].y, v[2].y));
  b.ymax = ceil(max(v[0].y, v[1].y, v[2].y));
}
```

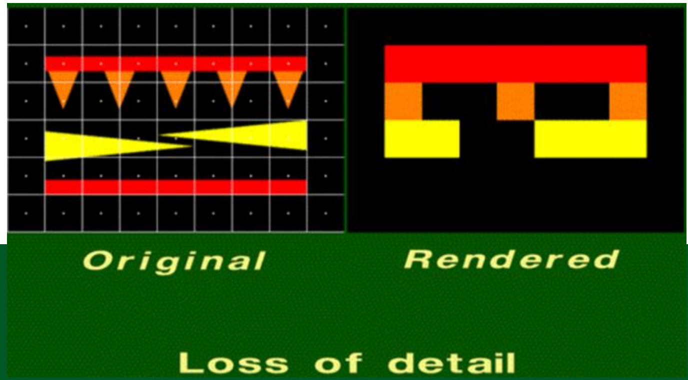
**Calculate tight bound around the triangle**
**Round coordinates upward (ceil) to the nearest integer**

Computer Graphics 2014, ZJU

# Aliasing

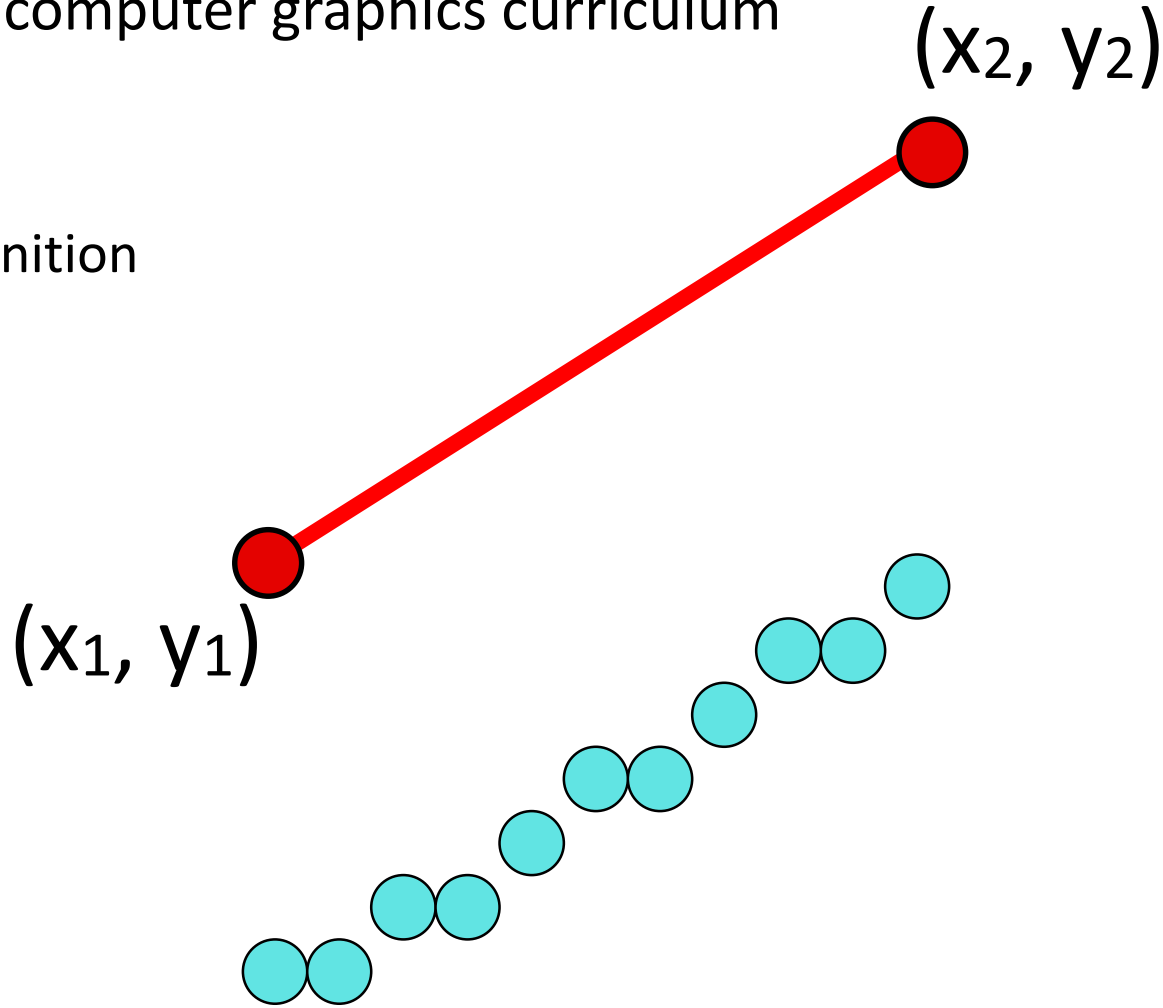
- Aliasing is caused due to the discrete nature of the display device
- Rasterizing primitives is like sampling a continuous signal by a finite set of values (point sampling)
- Information is lost if the rate of sampling is not sufficient. This sampling error is called ***aliasing***.
- Effects of aliasing are
  - Jagged edges
  - Incorrectly rendered fine details
  - Small objects might miss





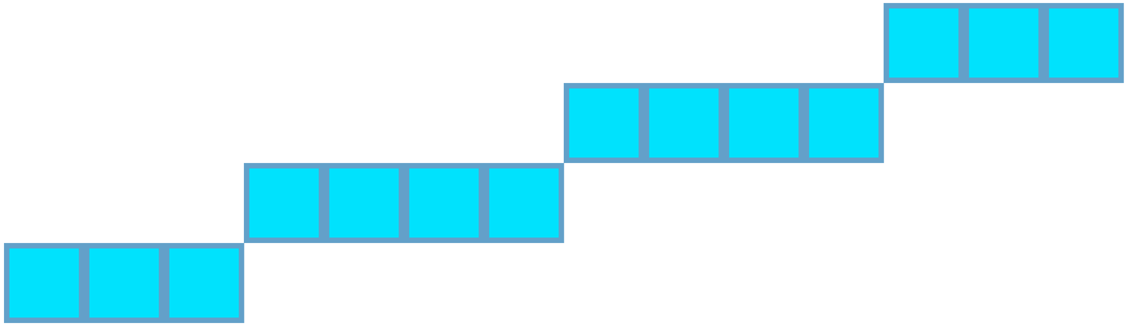
# Aliasing

- A classic part of the computer graphics curriculum
- Input:
  - Line segment definition
  - (x1, y1), (x2, y2)
- Output:
  - List of pixels

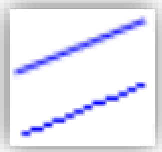


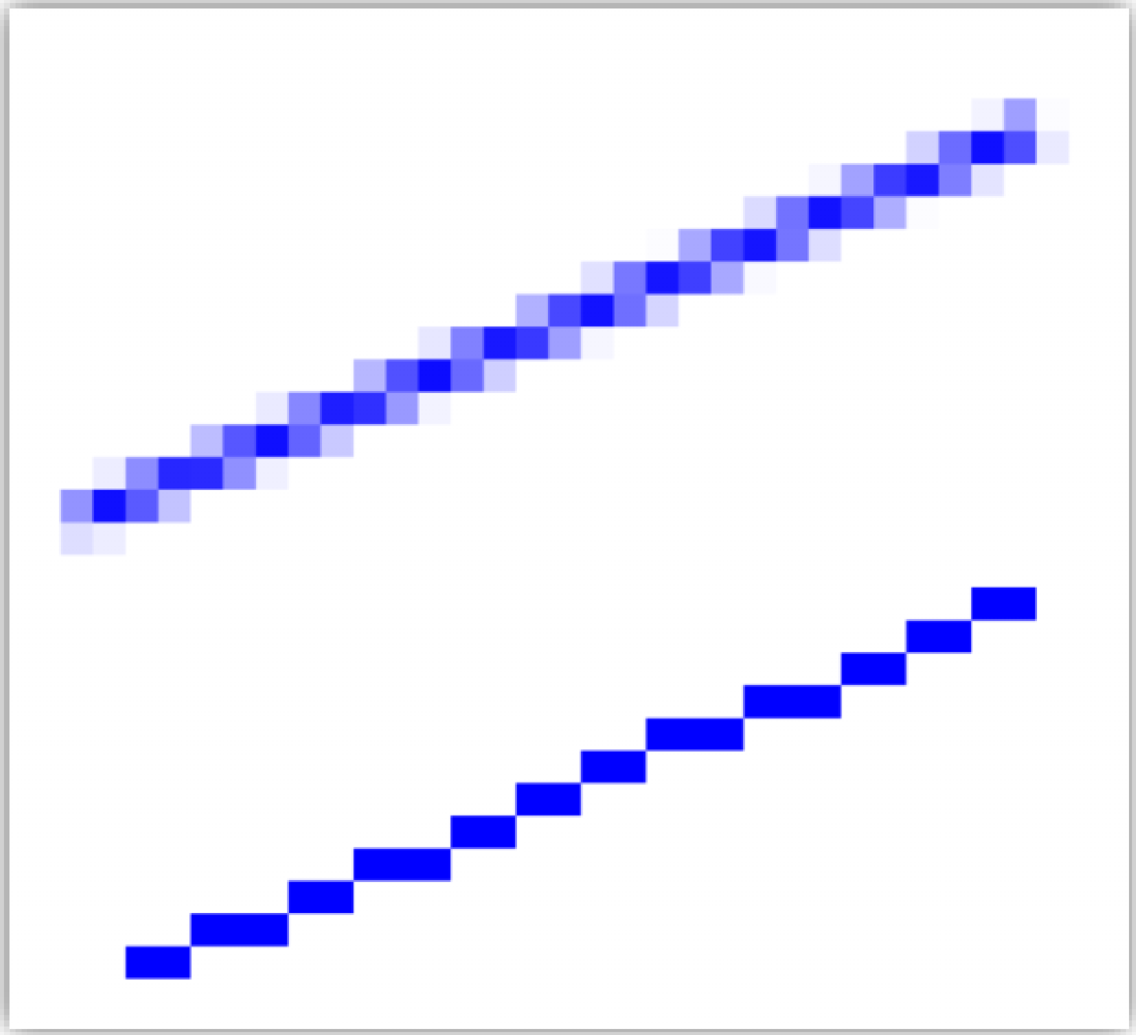
# Aliasing

- How Do They Look?
- So now we know how to draw lines
- But they don't look very good:



# Aliasing & Antialiasing





# Aliasing & Antialiasing

© Adobe, inc., https://helpx.adobe.com/photoshop/key-concepts/aliasing-anti-aliasing.html

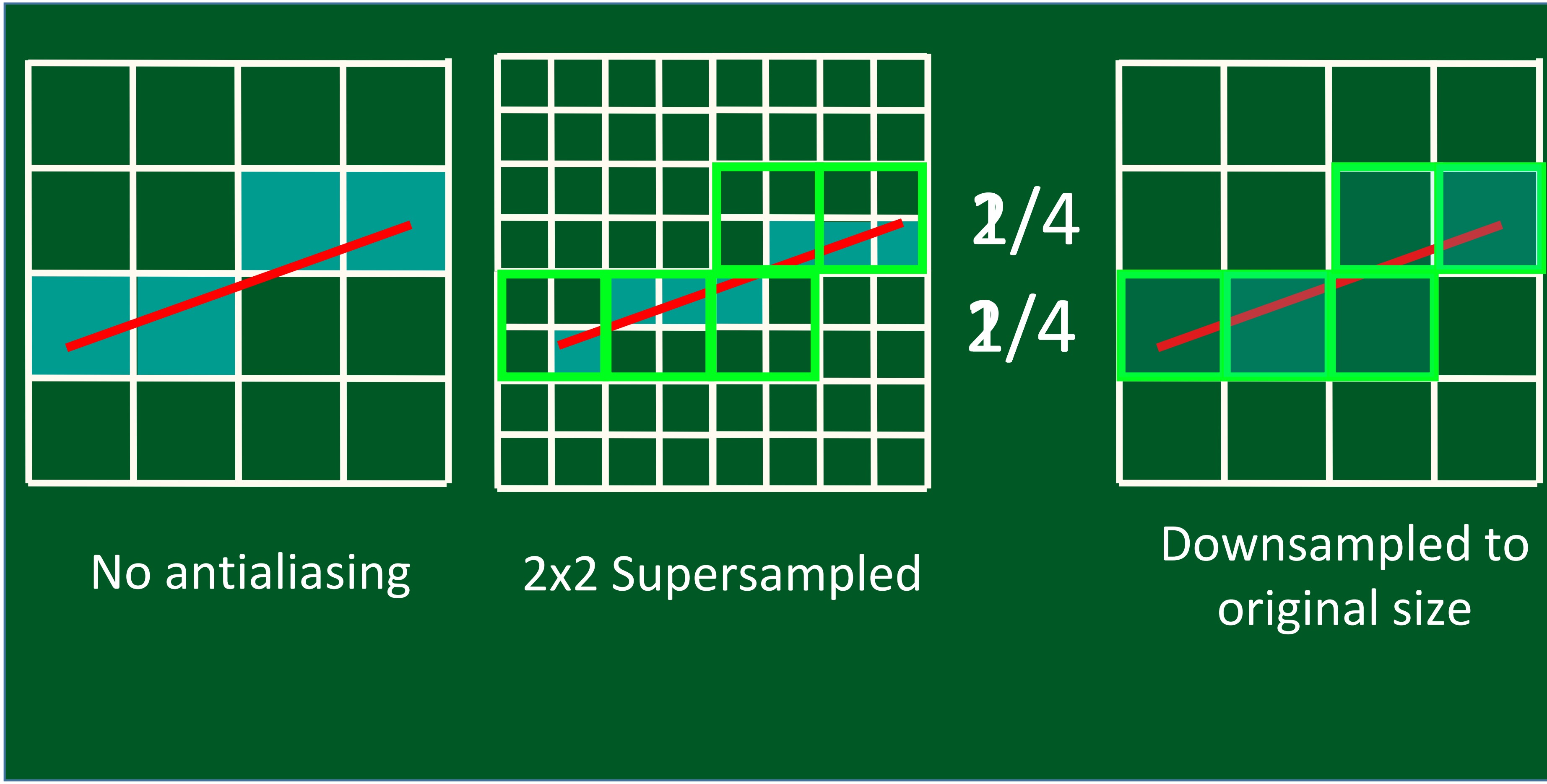
# Anti-aliasing

- Application of techniques to reduce/eliminate aliasing artifacts.
- Some of methods are:
  - Increasing sampling rate by increasing the resolution.
  - Averaging methods(post processing). Intensity of a pixel is set as the weighted average of its own intensity and the intensity of the surrounding pixels
  - Area Sampling, more popular

# Antialiasing: Super-sampling(postfiltering)

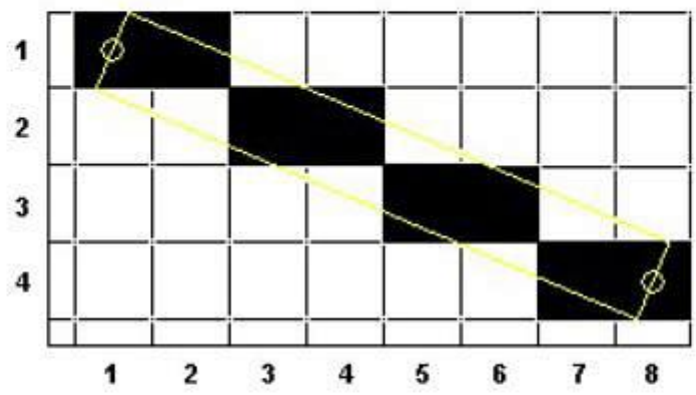
- Technique:
  1. Create an image 2x (or 4x, or 8x) bigger than the real image
  2. Scale the line endpoints accordingly
  3. Draw the line as before
     - No change to line drawing algorithm
  4. Average each 2x2 (or 4x4, or 8x8) block into a single pixel

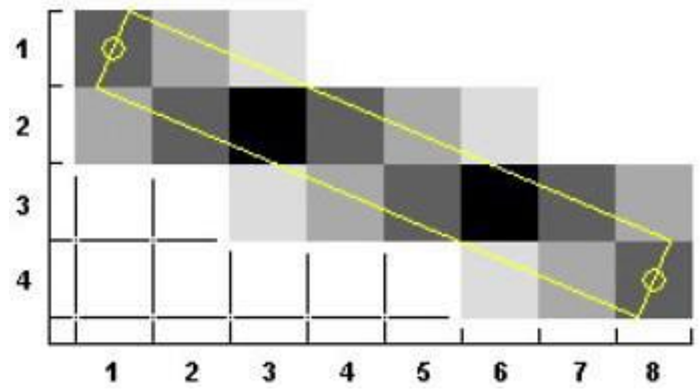
# Antialiasing: Super-sampling(postfiltering)




高分辨率计算，低分辨率显示

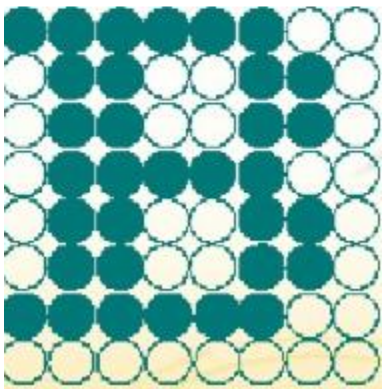
# Antialiasing (Area Sampling)



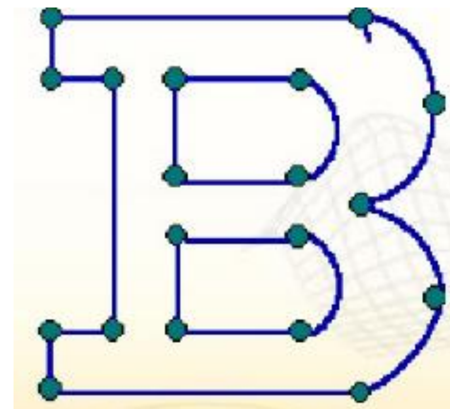

- A scan converted primitive occupies finite area on the screen
- Intensity of the boundary pixels is adjusted depending on the percent of the pixel area covered by the primitive. This is called weighted area sampling

# 字符的生成



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

点阵式字符



向量式字符