

# 计算机图形学

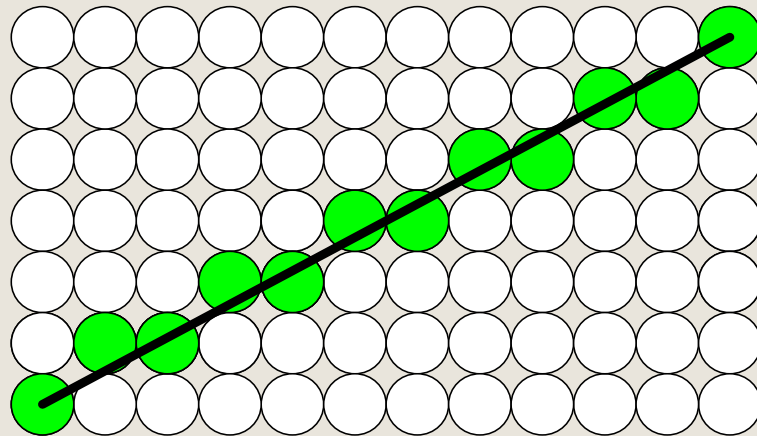
哈尔滨工业大学（威海）

计算机科学与技术学院

伯彭波

# 第3章 基本图形生成算法

如何在光栅扫描显示器上绘制一条直线？



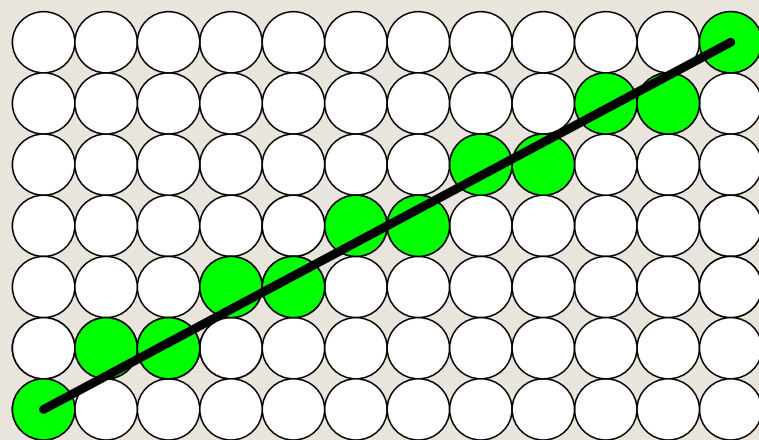
用像素点集逼近直线

# 第3章 基本图形生成算法

## 光栅图形生成的概念

- 图形的生成：在指定的输出设备上，根据坐标描述构造二维几何图形。
- 图形的扫描转换：在光栅显示器等数字设备上确定一个图形的最佳逼近像素集。

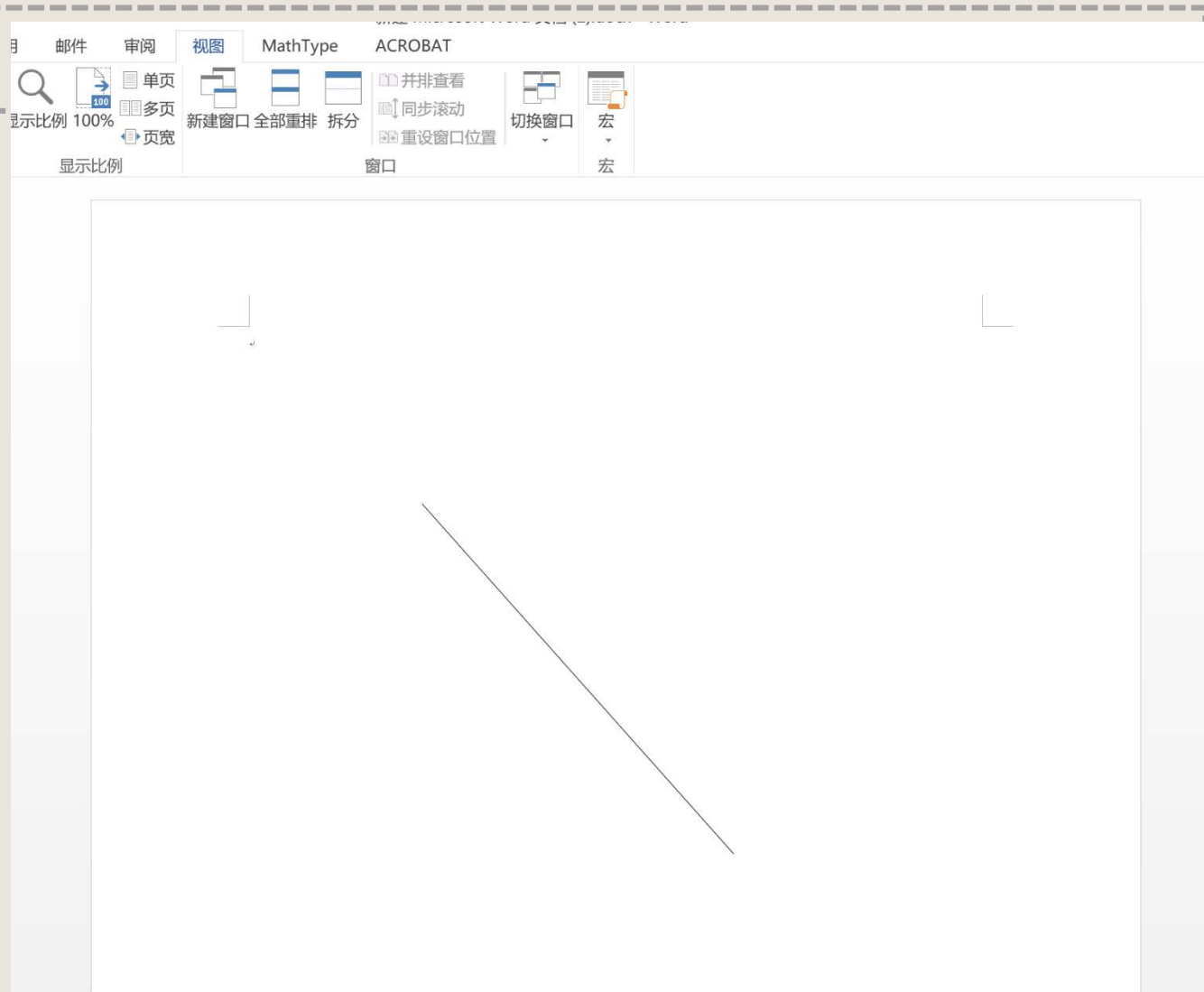
绘制端点坐标是(0,0)和  
(11, 6) 的直线段



用像素点集逼近直线

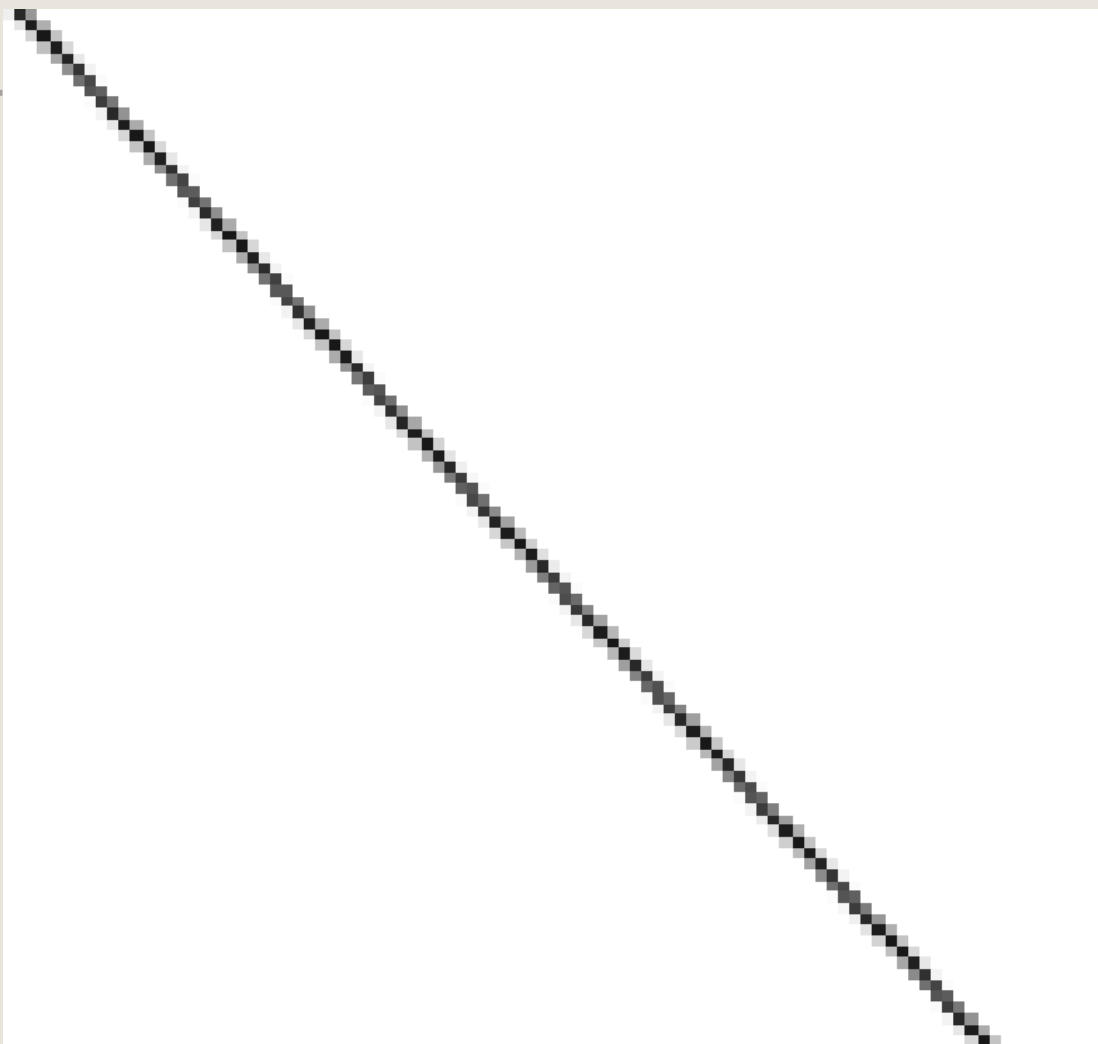
# 第3章 基本图形生成算法

## 图形生成



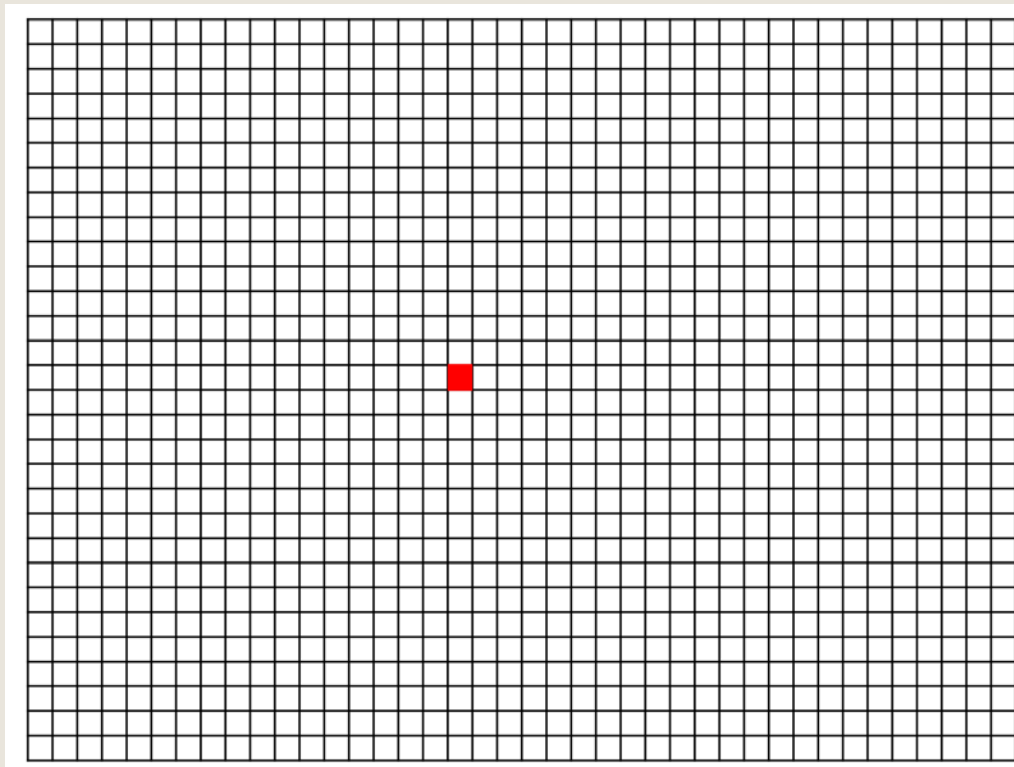
# 第3章 基本图形生成算法

图形生成



# 第3章 基本图形生成算法

- 光栅扫描算法采用的坐标系：  
每一个像素占据一个小的区域

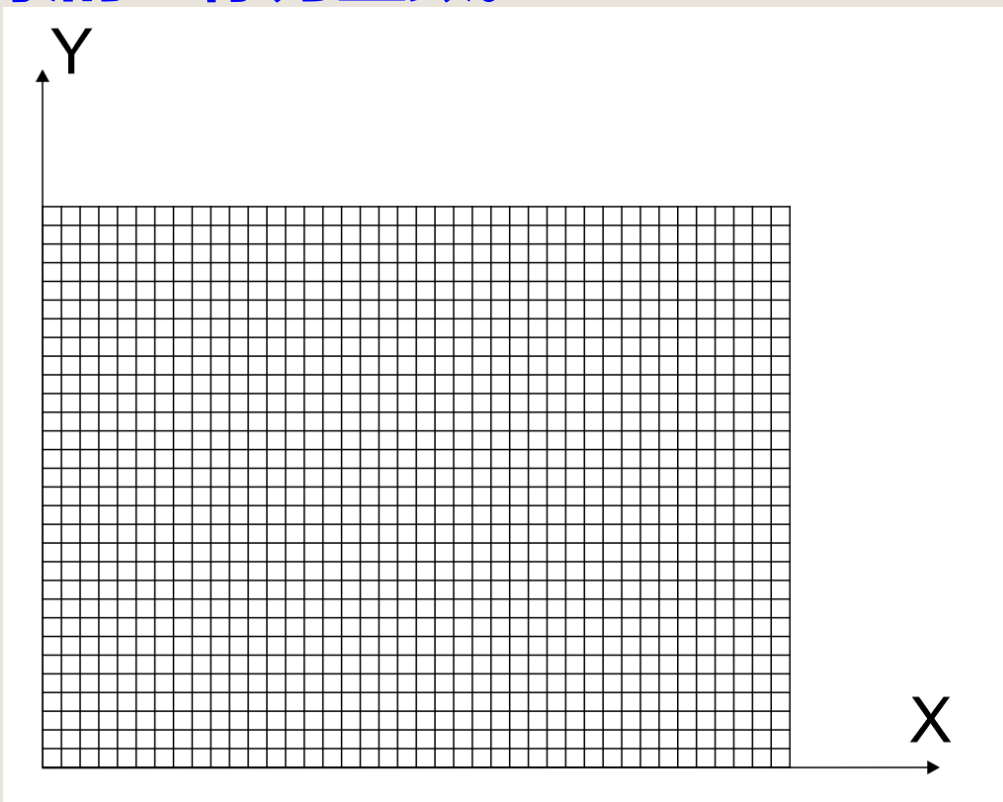


# 第3章 基本图形生成算法

## ➤ 光栅扫描算法采用的坐标系

在图形生成算法讨论中：

- 把一个交点看做一个像素。
- 像素的坐标为整数。



# 第3章 基本图形生成算法

## 主要内容

- 直线段的生成算法
- 圆弧的生成算法
- 多边形扫描转换
- 区域填充算法
- 反走样技术



# 第3章 基本图形生成算法

## 主要内容

- 直线段的生成算法
- 圆弧的生成算法
- 多边形扫描转换
- 区域填充算法
- 反走样技术

## 要求:

- 直线要直;
- 直线的端点要准确, 无定向性无断裂;
- 直线的亮度、色泽要均匀;
- 画线的速度要快;
- 具有不同的色泽、亮度、线型等。

# 第3章 基本图形生成算法

## 主要内容

- 直线段的生成算法
- 圆弧的生成算法
- 多边形扫描转换
- 区域填充算法
- 反走样技术

**要求：**给定直线两端点 $P_0(x_0, y_0)$ 和 $P_1(x_1, y_1)$ ，画出该直线。

- **数值微分法 (DDA算法)**
- **中点Bresenham算法**
- **改进Bresenham算法**

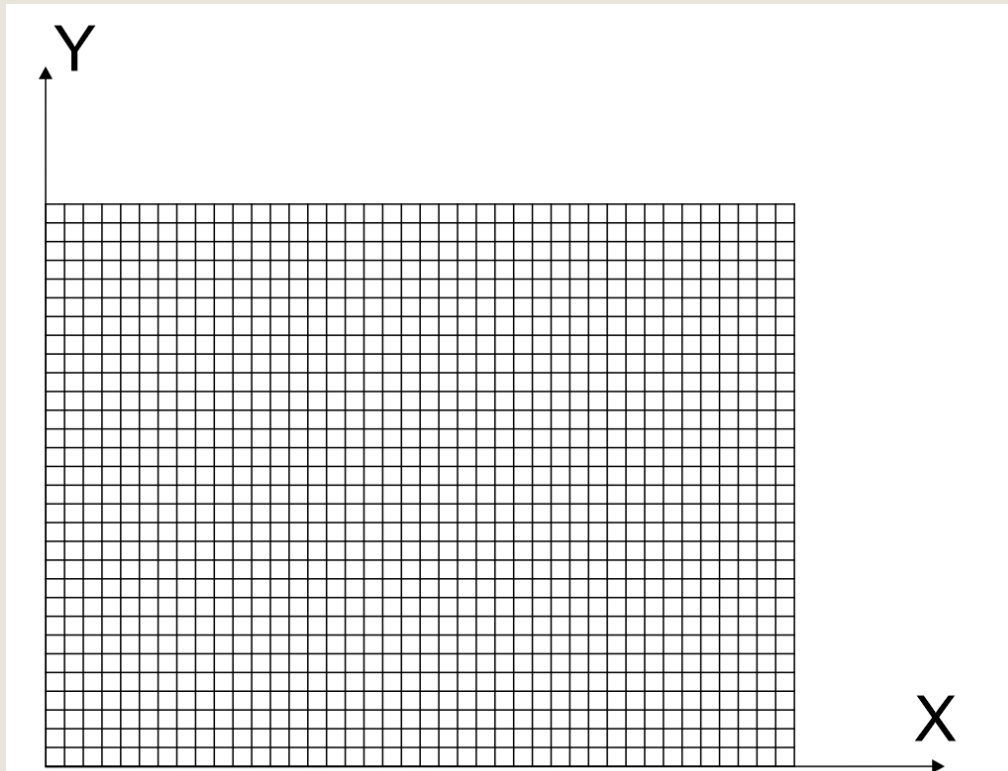
# 直线段的生成算法

- 数值微分法（DDA算法）
- 中点Bresenham算法
- 改进Bresenham算法

## ➤ 利用直线的方程

**注意:** (1) 网格的交点表示像素

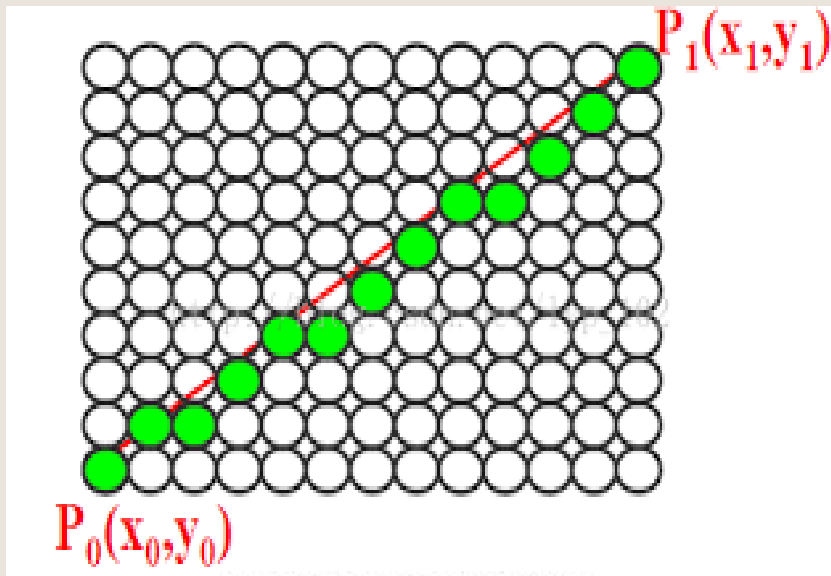
(2) 像素的坐标值是整数，像素的边长为1



# 直线段的生成算法

$x_0, y_0$  是线段起点坐标  
 $x_n, y_n$  是线段终点坐标

- 数值微分法（DDA算法）
- 中点Bresenham算法
- 改进Bresenham算法



**DDA算法原理：** 从初始点出发，  
不断寻找位于直线上的下一个  
点，直到线段的终点

# 直线段的生成算法

- 数值微分法（DDA算法）
- 中点Bresenham算法
- 改进Bresenham算法

**x0,y0是线段起点坐标**  
**xn,yn是线段终点坐标**

✓ 直线的斜率:

$$k = \frac{Y_n - Y_0}{X_n - X_0}$$

✓ 直线的方程:

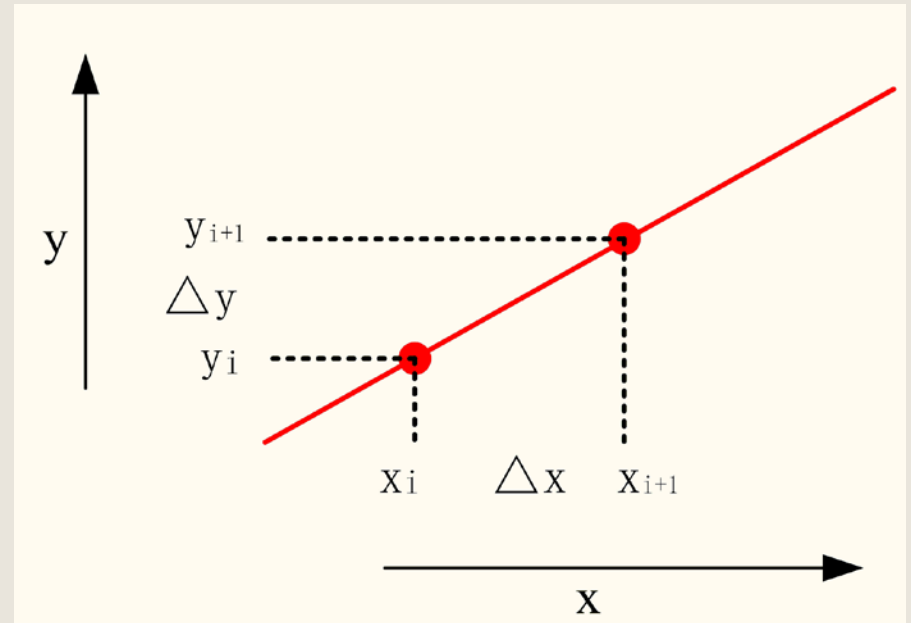
$$y = kx + b$$

# 直线段的生成算法

假设直线上两个点:  
( $x_i, y_i$ )和( $x_{i+1}, y_{i+1}$ )

$$\begin{aligned}x_{i+1} &= x_i + \Delta x \\y_{i+1} &= kx_{i+1} + b \\&= k(x_i + \Delta x) + b \\&= kx_i + b + k\Delta x \\&= y_i + k\Delta x\end{aligned}$$

- 数值微分法 (DDA算法)
- 中点Bresenham算法
- 改进Bresenham算法



DDA算法原理

# 直线段的生成算法

➤ 数值微分法（DDA算法）

➤ 中点Bresenham算法

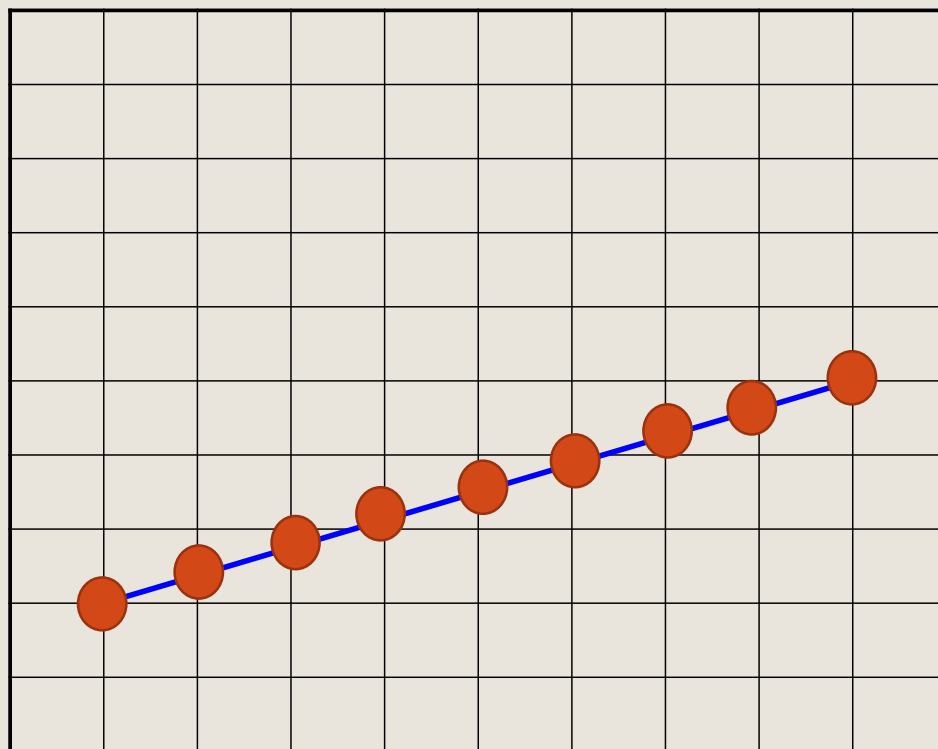
➤ 改进Bresenham算法

假设直线上两个点：  
( $x_i, y_i$ )和( $x_{i+1}, y_{i+1}$ )

令  $\Delta x = 1$  ,

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

$$y_{i+1} = y_i + k$$



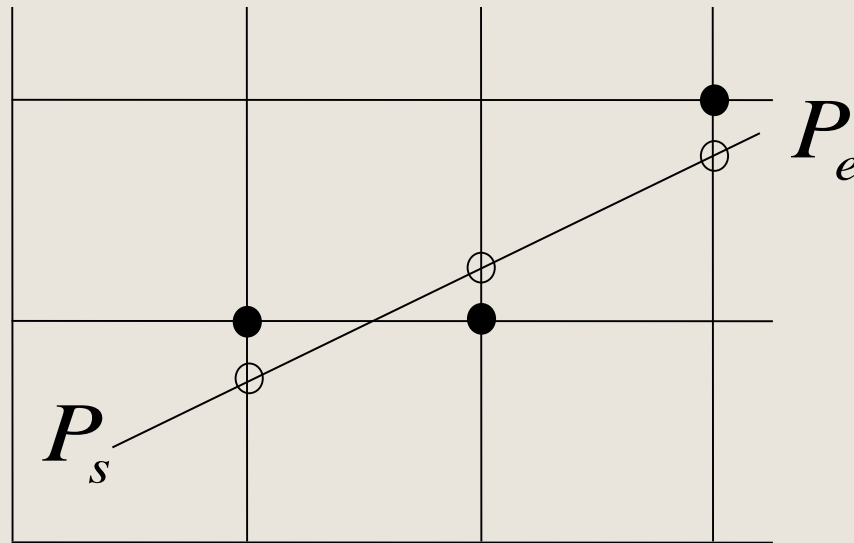
# 直线段的生成算法

➤ 数值微分法

➤ 中点Bresenham算法

➤ 改进Bresenham算法

- 在光栅化过程中，只能绘制整数位置的像素点，因此需要对求出的 $x_{i+1}, y_{i+1}$ 进行四舍五入，即加0.5再取整。

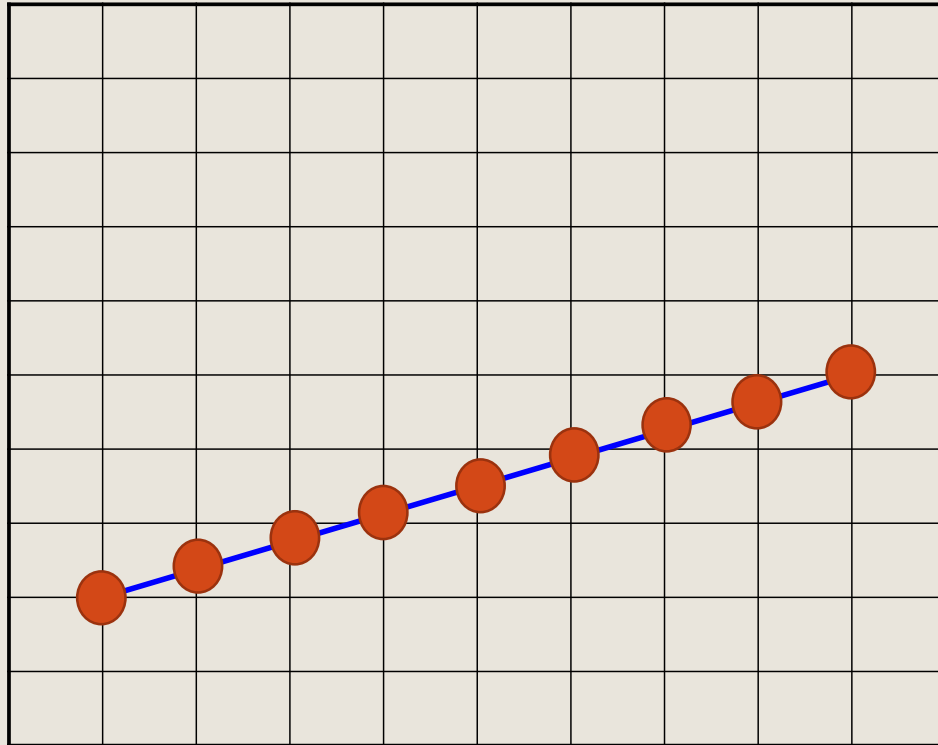




# 直线段的生成算法

- 数值微分法
- 中点Bresenham算法
- 改进Bresenham算法

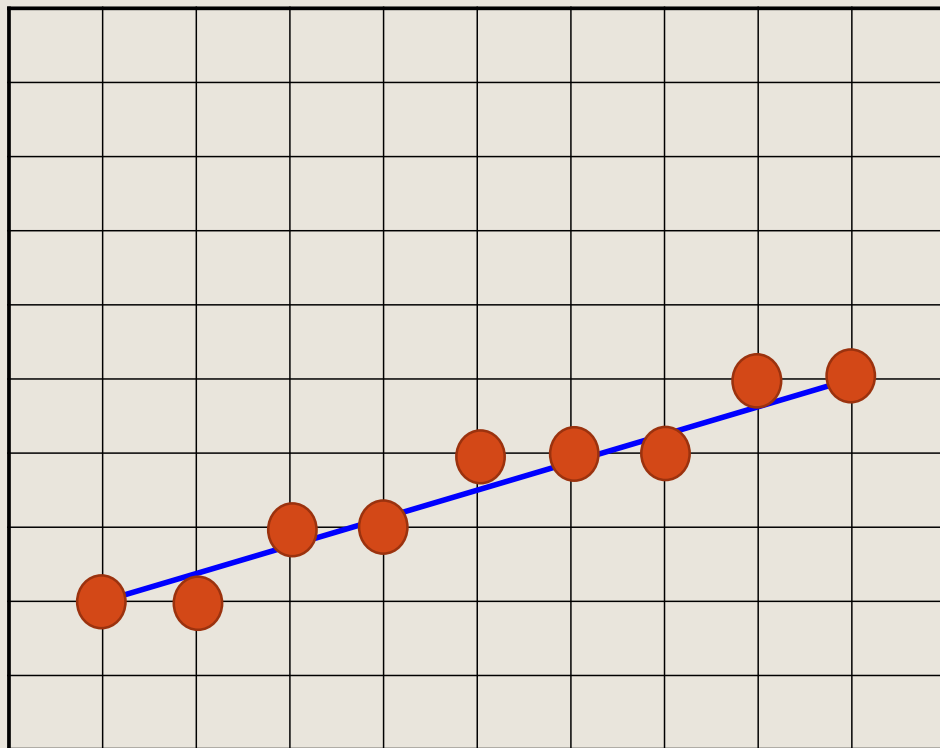
- 在光栅化过程中，只能绘制整数位置的像素点，因此需要对求出的 $x_{i+1}, y_{i+1}$ 进行四舍五入，即加0.5再取整。



# 直线段的生成算法

- 数值微分法
- 中点Bresenham算法
- 改进Bresenham算法

- 在光栅化过程中，只能绘制整数位置的像素点，因此需要对求出的 $x_{i+1}, y_{i+1}$ 进行四舍五入，即加0.5再取整。



# 直线段的生成算法

➤ 数值微分法（DDA算法）

➤ 中点Bresenham算法

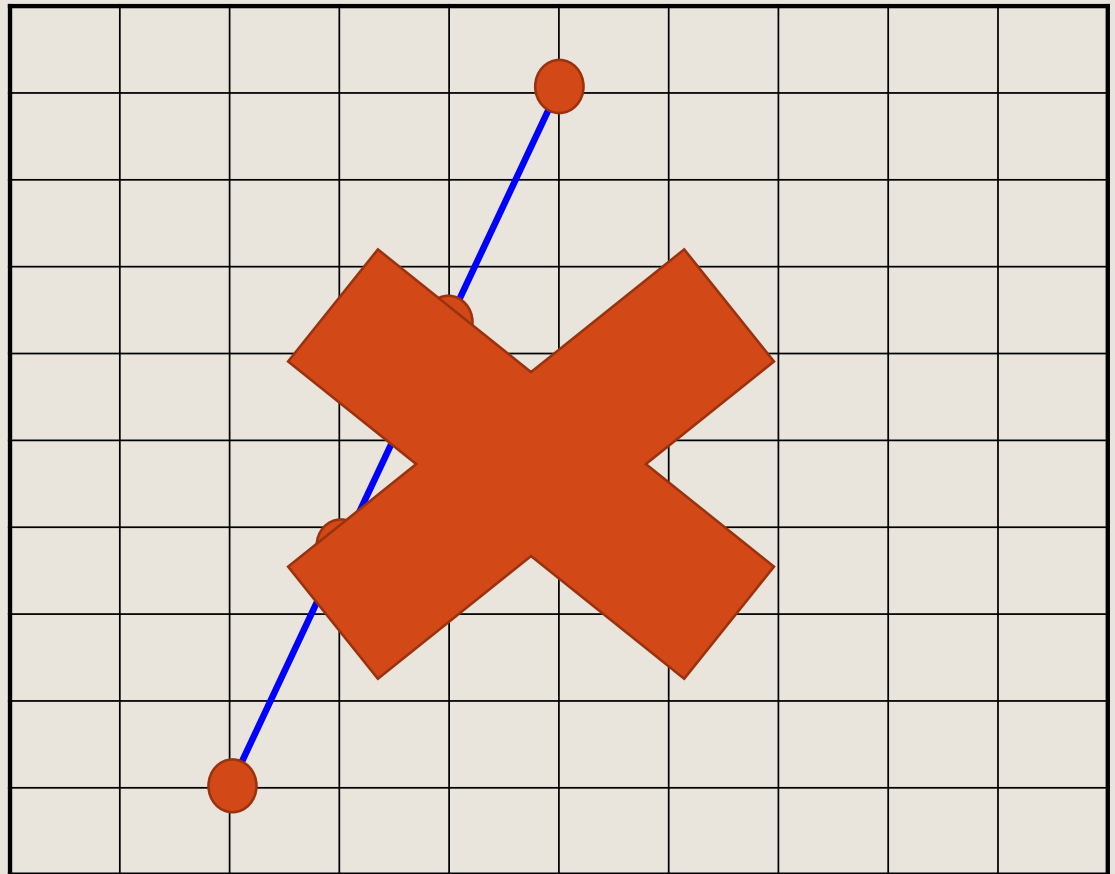
➤ 改进Bresenham算法

假设直线上两个点：  
( $x_i, y_i$ )和( $x_{i+1}, y_{i+1}$ )

令  $\Delta x = 1$  ,

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

$$y_{i+1} = y_i + k$$



# 直线段的生成算法

➤ 数值微分法（DDA算法）

➤ 中点Bresenham算法

➤ 改进Bresenham算法

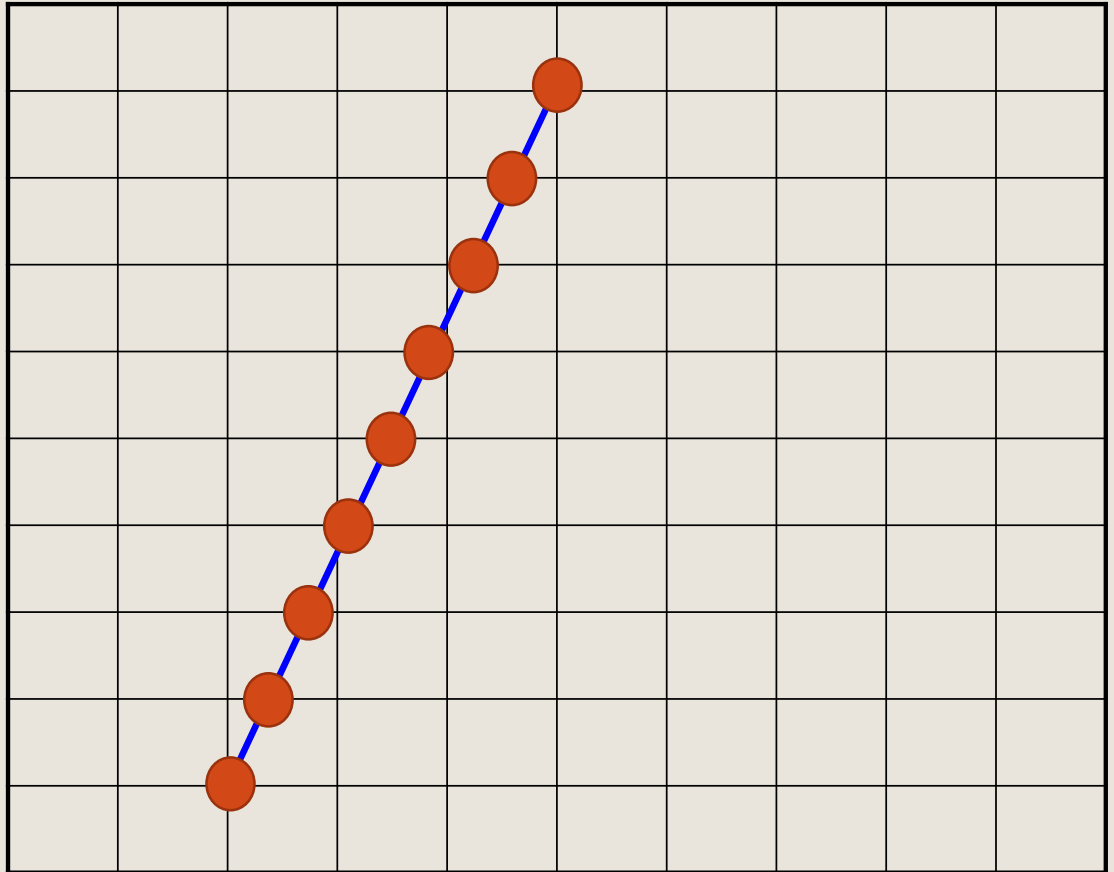
假设直线上两个点:

$(x_i, y_i)$  和  $(x_{i+1}, y_{i+1})$

令  $\Delta y = 1$  ,

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

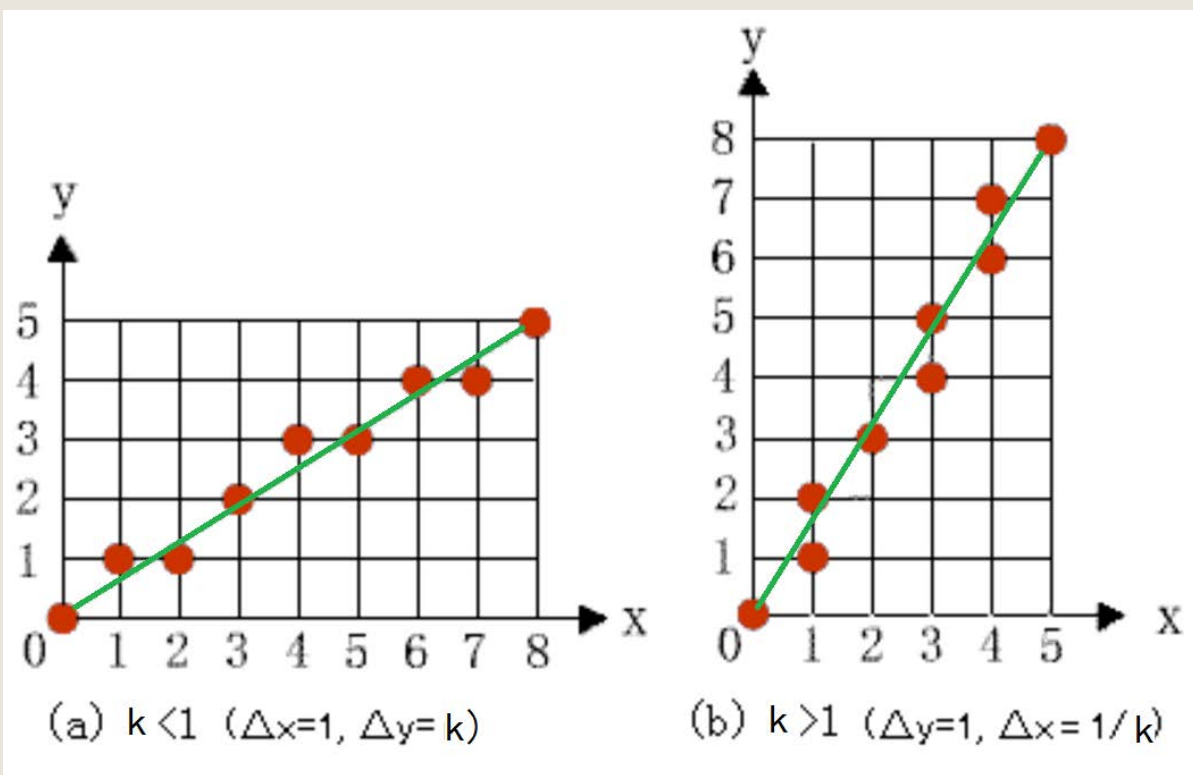
$$y_{i+1} = y_i + 1$$



# 直线段的生成算法

- 数值微分法
- 中点Bresenham算法
- 改进Bresenham算法

**问题：令  $\Delta x = 1$  或  $\Delta y = 1$ ?**



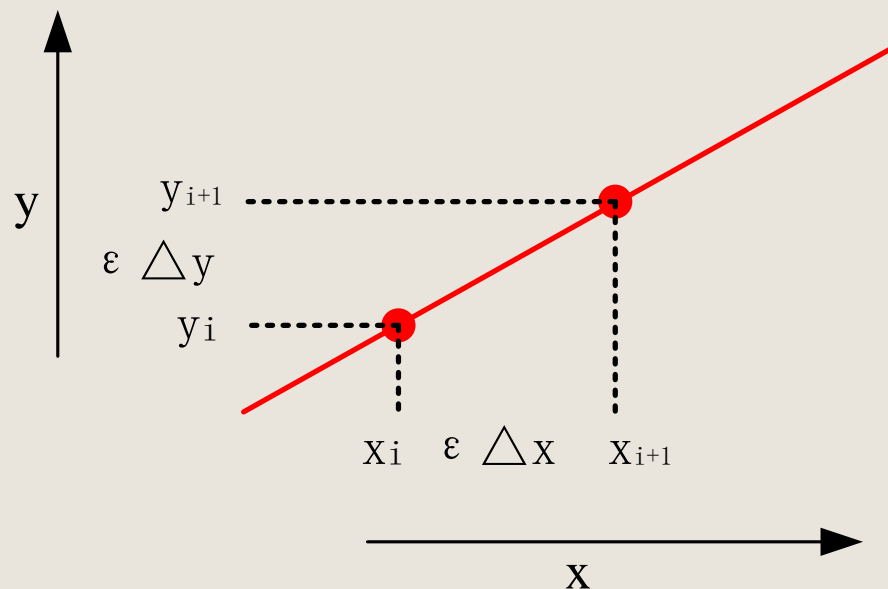
# 直线段的生成算法

- 数值微分法
- 中点Bresenham算法
- 改进Bresenham算法

$$x_{i+1} = x_i + \varepsilon \cdot \Delta x$$

$$y_{i+1} = y_i + \varepsilon \cdot \Delta y$$

$$\varepsilon = 1 / \max(|\Delta x|, |\Delta y|)$$



DDA算法原理

# 直线段的生成算法

➤ 数值微分法

➤ 中点Bresenham算法

➤ 改进Bresenham算法

- $\max(|\Delta x|, |\Delta y|) = |\Delta x|$ , 即  $|k| \leq 1$  的情况:

$$x_{i+1} = x_i + \varepsilon \cdot \Delta x = x_i + \frac{1}{|\Delta x|} \cdot \Delta x = x_i \pm 1$$

$$y_{i+1} = y_i + \varepsilon \cdot \Delta y = y_i + \frac{1}{|\Delta x|} \cdot \Delta y = y_i \pm k$$

- $\max(|\Delta x|, |\Delta y|) = |\Delta y|$ , 此时  $|k| \geq 1$ :

$$x_{i+1} = x_i + \varepsilon \cdot \Delta x = x_i + \frac{1}{|\Delta y|} \cdot \Delta x = x_i \pm \frac{1}{k}$$

$$y_{i+1} = y_i + \varepsilon \cdot \Delta y = y_i + \frac{1}{|\Delta y|} \cdot \Delta y = y_i \pm 1$$

# 直线段的生成算法

- 数值微分法
- 中点Bresenham算法
- 改进Bresenham算法

## 特点:

- 增量算法（每一步都是上一步的x,y值增加一个量）
- 直观、易实现

## 缺点:

- 需要进行浮点数运算;
- 产生一个像素要做两次加法和两次取整运算;
- 运行效率低;
- 取整运算不利于硬件实现。



# 直线段的生成算法

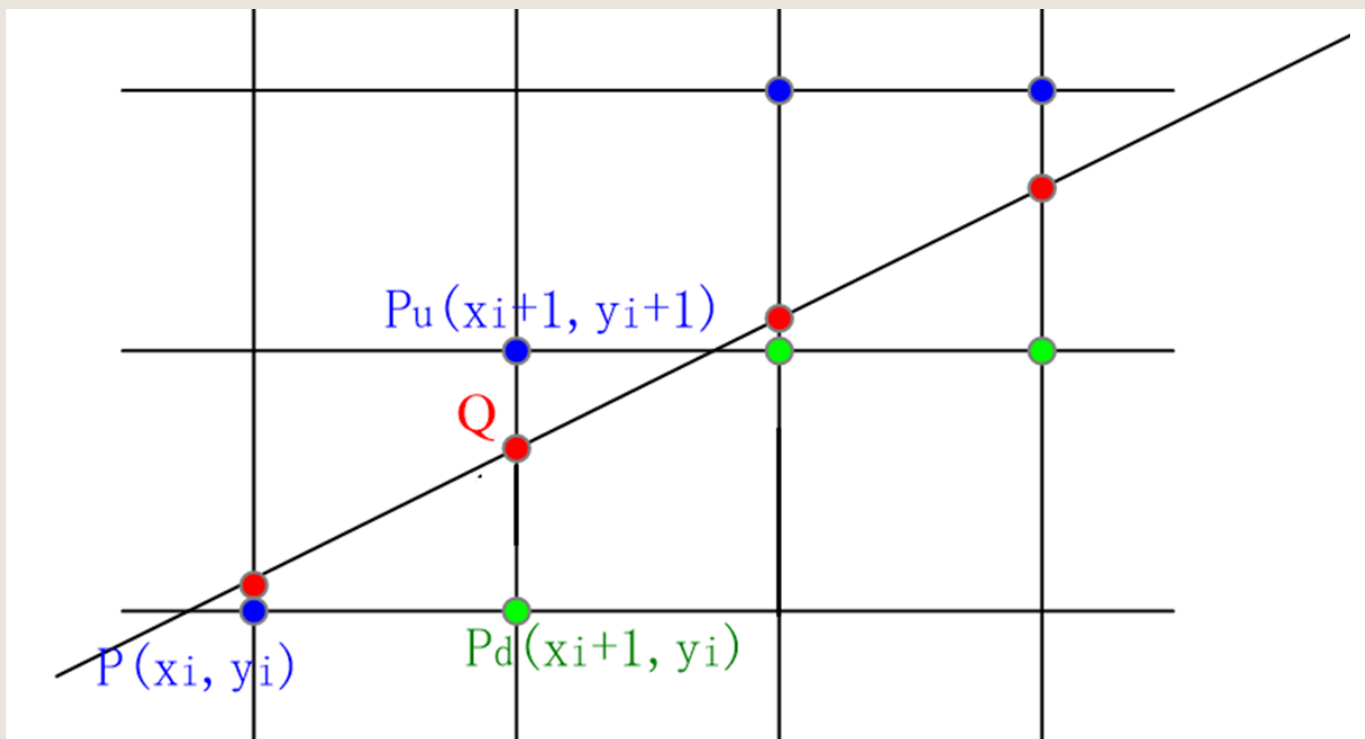
- 数值微分法
- 中点Bresenham算法
- 改进Bresenham算法

假定  $0 \leq k \leq 1$ ，即  $0 \leq \Delta y / \Delta x \leq 1$ ， $x$  是最大位移方向

假设当前像素是  $P(x_i, y_i)$ ，下一个像素点是  $P_u$  还是  $P_d$ ？



回答：下一个像素取  $P_u$ ， $P_d$  中距离直线最近的那个。



# 直线段的生成算法

- 数值微分法
- 中点Bresenham算法
- 改进Bresenham算法

## 算法原理

- ◆ 根据直线的斜率确定最大位移方向是x方向或者y方向，将该方向坐标值增加1；
- ◆ 另一方向的坐标值增量为1或0，取决于直线与相邻像素点的距离——**误差项**。

# 直线段的生成算法

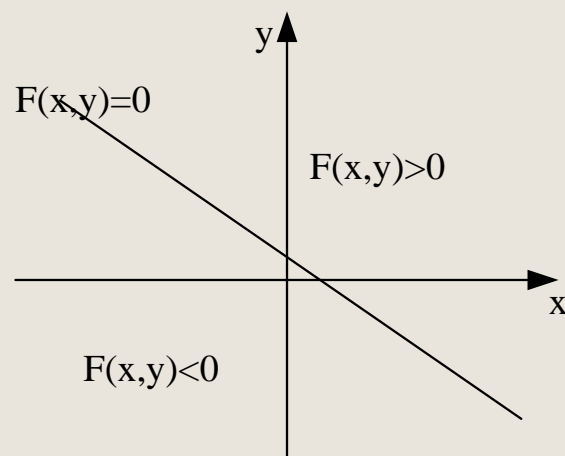
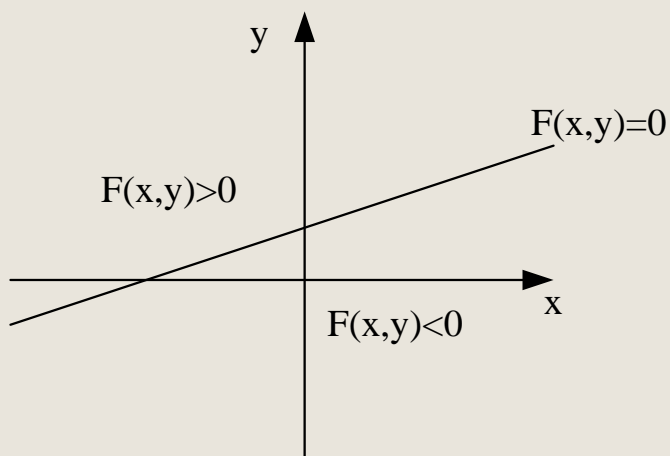
➤ 数值微分法

➤ 中点Bresenham算法

➤ 改进Bresenham算法

## ● 直线的方程

$$F(x, y) = y - kx - b = 0, \text{ 其中 } k = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$$



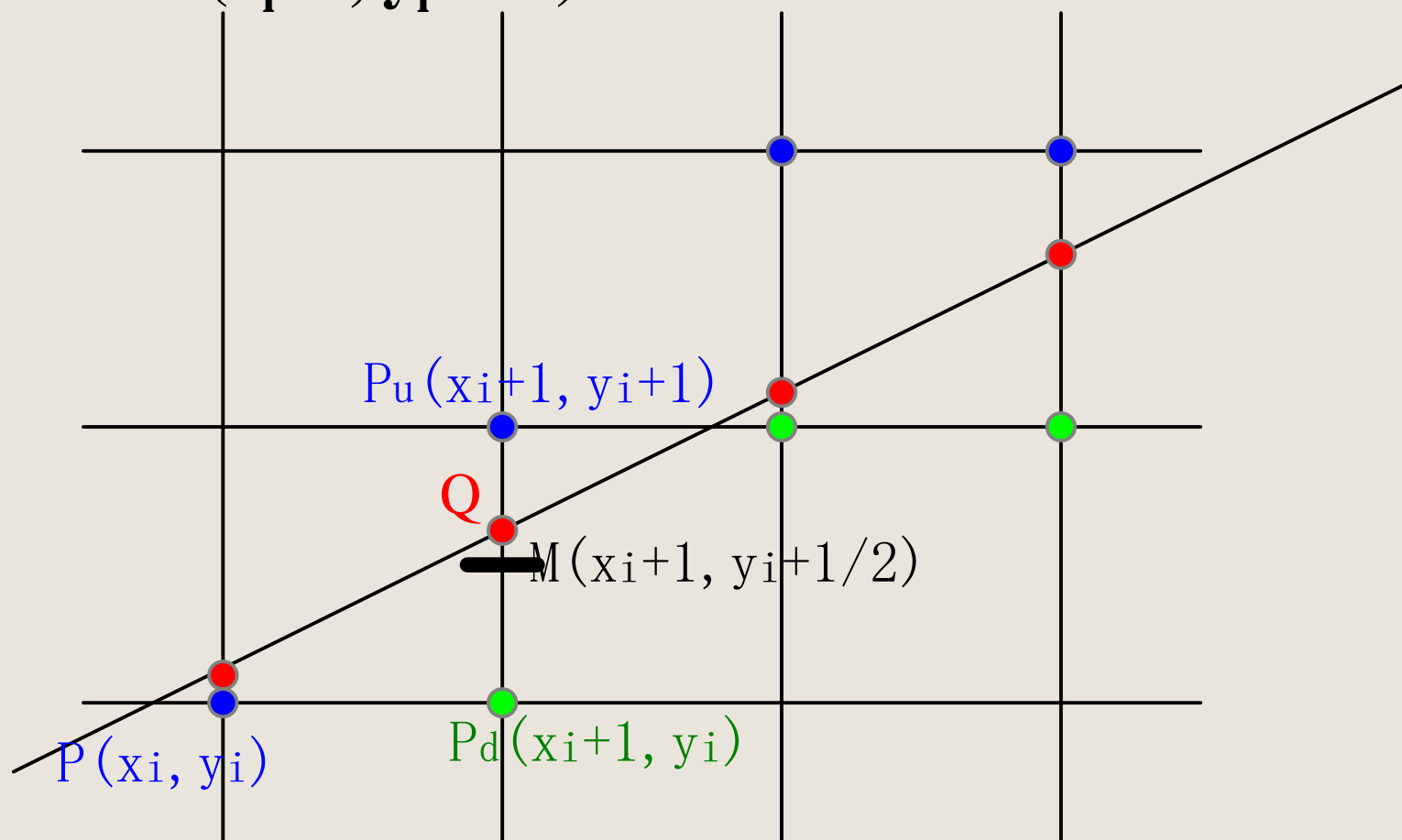
直线将平面分为三个区域

# 直线段的生成算法

- 数值微分法
- 中点Bresenham算法
- 改进Bresenham算法

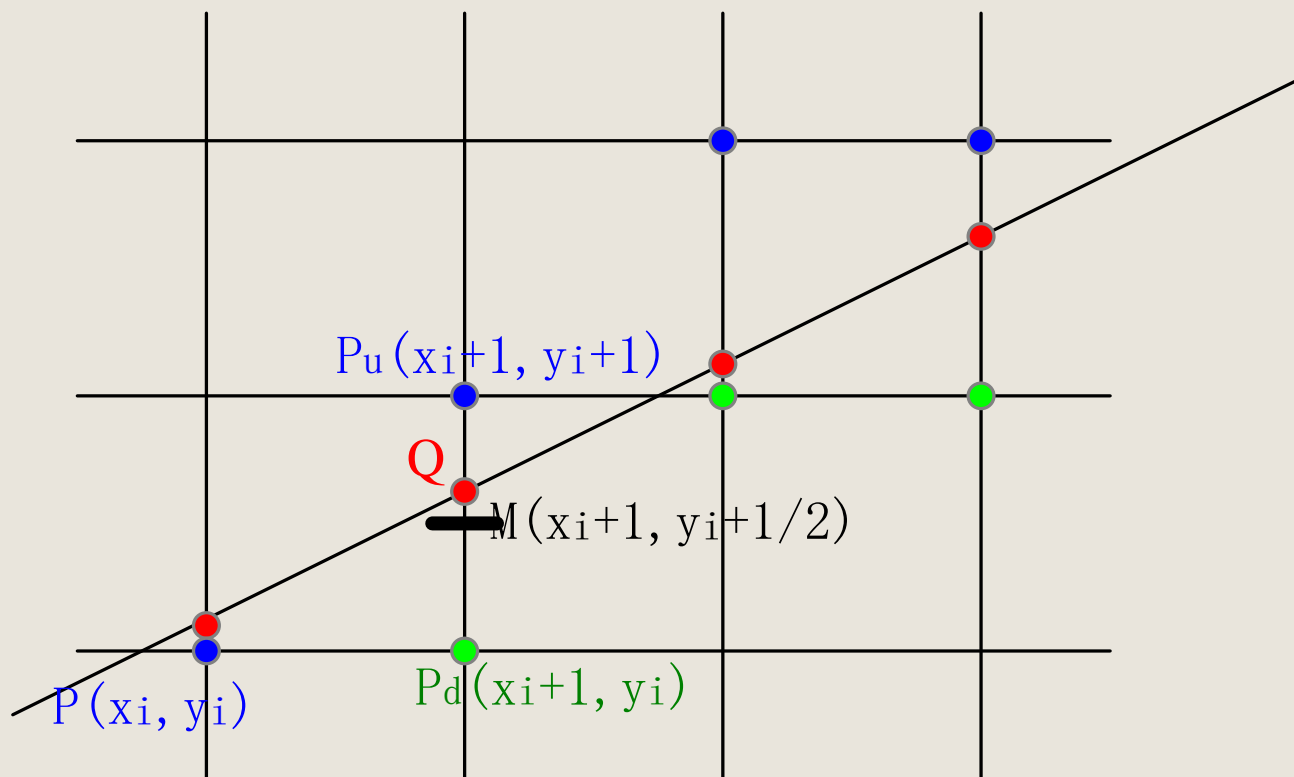
定义距离误差：中点在直线上方还是下方？

$$d = F(x_i+1, y_i+1/2)$$



则有:

$$\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} = \begin{cases} y_i + 1 & (d < 0) \\ y_i & (d \geq 0) \end{cases} \end{cases}$$

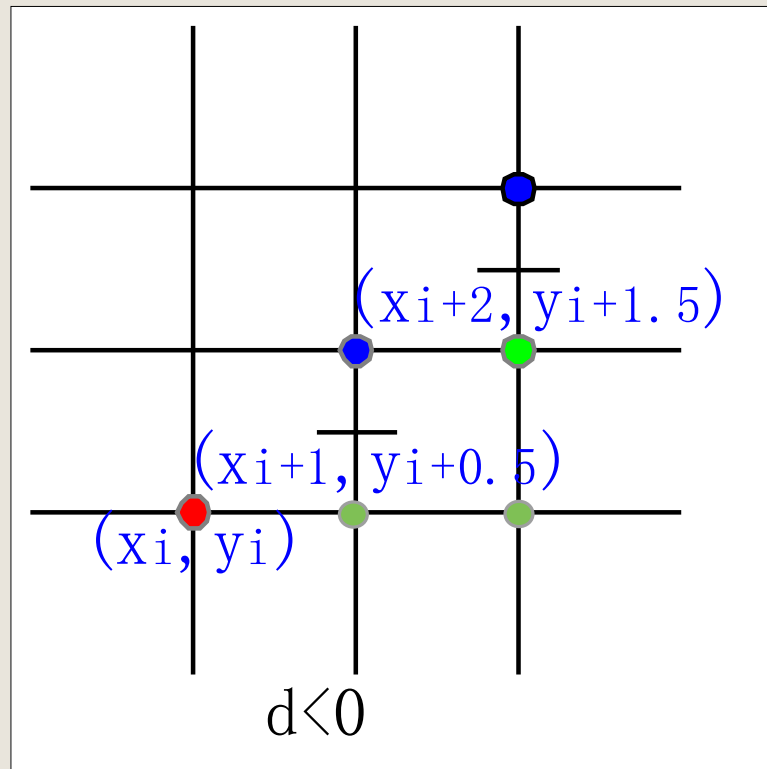


# 如何减小d的计算量?

## 误差项的递推 ( $d < 0$ )

$$\begin{aligned}d_1 &= F(x_i + 1, y_i + 0.5) \\ &= y_i + 0.5 - k(x_i + 1) - b\end{aligned}$$

$$\begin{aligned}d_2 &= F(x_i + 2, y_i + 1.5) \\ &= y_i + 1.5 - k(x_i + 2) - b \\ &= y_i + 0.5 - k(x_i + 1) - b + 1 - k \\ &= d_1 - k + 1\end{aligned}$$

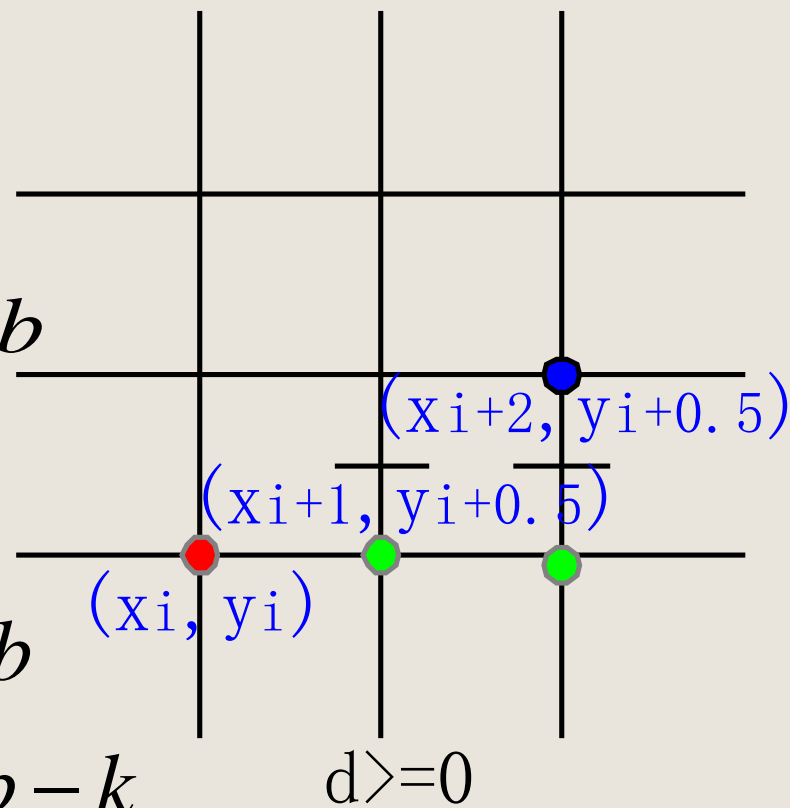


误差项递推

## 误差项的递推 ( $d \geq 0$ )

$$\begin{aligned}d_1 &= F(x_i + 1, y_i + 0.5) \\&= y_i + 0.5 - k(x_i + 1) - b\end{aligned}$$

$$\begin{aligned}d_2 &= F(x_i + 2, y_i + 0.5) \\&= y_i + 0.5 - k(x_i + 2) - b \\&= y_i + 0.5 - k(x_i + 1) - b - k \\&= d_1 - k\end{aligned}$$



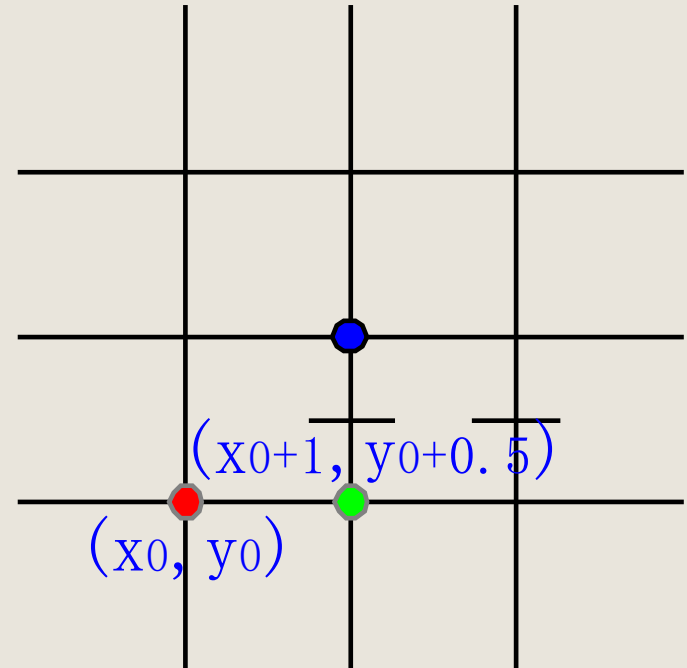
误差项递推

# 直线段的生成算法

- 数值微分法
- 中点Bresenham算法
- 改进Bresenham算法

初始值 $d$ 的计算：假设起始点位于一个像素上

$$\begin{aligned}d_0 &= F(x_0 + 1, y_0 + 0.5) \\&= y_0 + 0.5 - k(x_0 + 1) - b \\&= y_0 - kx_0 - b - k + 0.5 \\&= 0.5 - k\end{aligned}$$



计算初值



# 直线段的生成算法

## 如何减小d的计算量?

➤ 数值微分法

➤ 中点Bresenham算法

➤ 改进Bresenham算法

$K = \Delta y / \Delta x$ , 涉及到除法, 改进:

用  $D = 2d\Delta x$  作为误差项

$$D_0 = 2\Delta x d_0 = 2\Delta x (0.5 - k) = \Delta x - 2\Delta y$$

$$(1) \quad d < 0 \quad D < 0$$

$$D_2 = 2\Delta x d_2 = 2\Delta x (d_1 + 1 - k)$$

$$= 2\Delta x d_1 + 2\Delta x - 2\Delta y$$

$$= D_1 + 2\Delta x - 2\Delta y$$

$$(2) \quad d \geq 0 \quad D \geq 0$$

$$D_2 = 2\Delta x d_2 = 2\Delta x (d_1 - k) = D_1 - 2\Delta y$$

# 直线的生成算法

➤ 数值微分法

➤ 中点Bresenham算法

➤ 改进Bresenham算法

## 算法步骤

- 输入直线的两 endpoint  $P_0(x_0, y_0)$  和  $P_1(x_1, y_1)$ 。
- 计算初始值  $\Delta x$ 、 $\Delta y$ 、 $D = \Delta x - 2\Delta y$ 、 $x = x_0$ 、 $y = y_0$ 。
- 绘制点  $(x, y)$ 。判断  $D$  的符号。若  $D < 0$ ，则  $(x, y)$  更新为  $(x+1, y+1)$ ， $D$  更新为  $D + 2\Delta x - 2\Delta y$ ；否则  $(x, y)$  更新为  $(x+1, y)$ ， $D$  更新为  $D - 2\Delta y$ 。
- 当直线没有画完时，重复上一步骤，否则结束。

# 直线段的生成算法

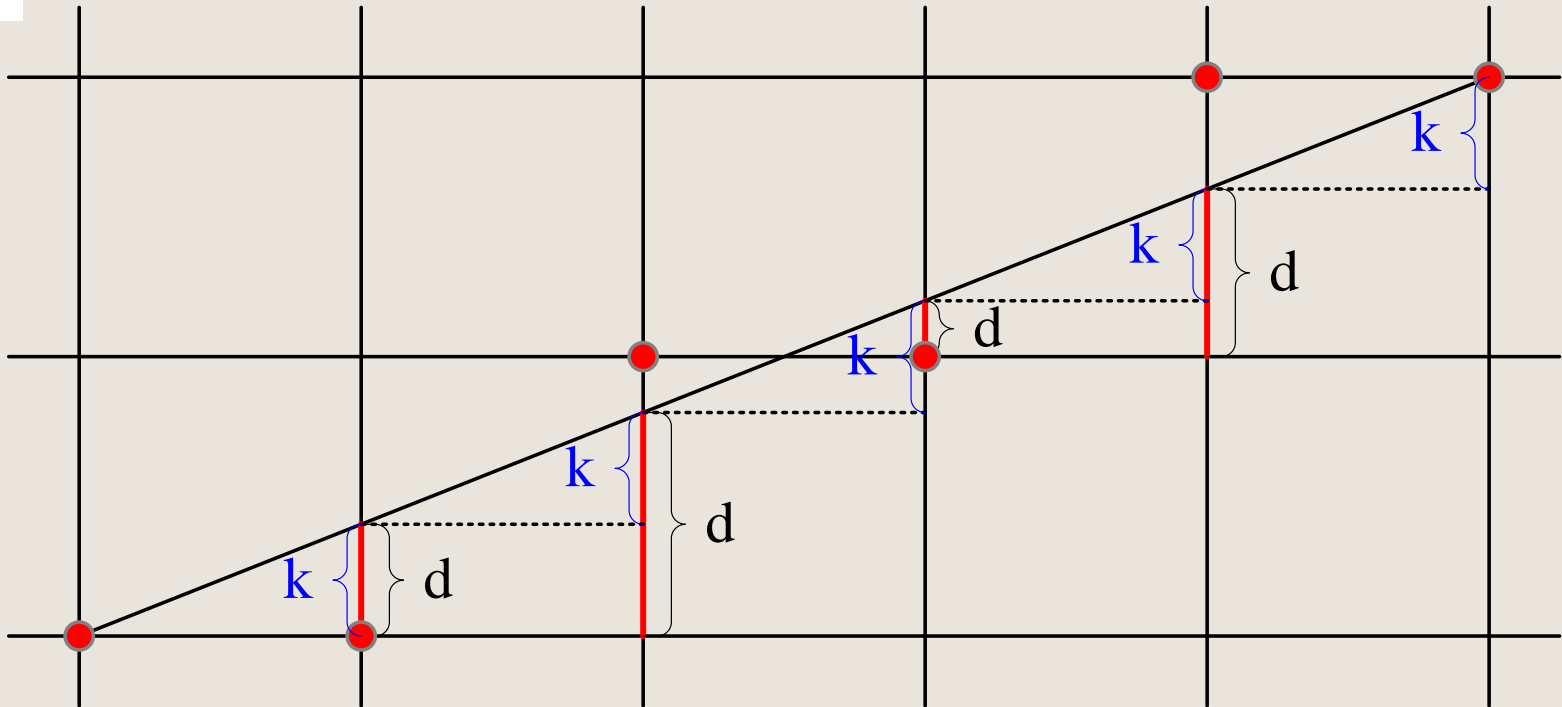
➤ 数值微分法

➤ 中点Bresenham算法

➤ 改进Bresenham算法



判别式  $d$ : 直线与网格线的交点与网格点的误差.  
 $d$  大于还是小于0.5?



$k$ : 直线的斜率

改进的Bresenham算法绘制直线的原理

# 直线段的生成算法

➤ 数值微分法

➤ 中点Bresenham算法

➤ 改进Bresenham算法

- 假定直线段的  $0 \leq k \leq 1$

直线方程  $y = kx + b$

$$\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} = y_i + k \end{cases}$$

# 直线段的生成算法

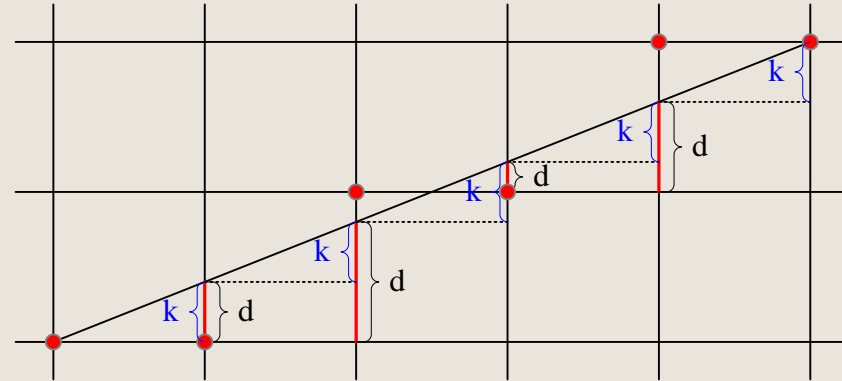
➤ 数值微分法

➤ 中点Bresenham算法

➤ 改进Bresenham算法

## ● 误差项的计算

- $d_{\text{初}}=0$ ,
- 每一步:  $x=x+1$ ,  $d=d+k$
- 如果  $d>1$ ,  $d=d-1$



## ● 像素坐标的计算

$$\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} = \begin{cases} y_i + 1 & (d > 0.5) \\ y_i & (d \leq 0.5) \end{cases} \end{cases}$$

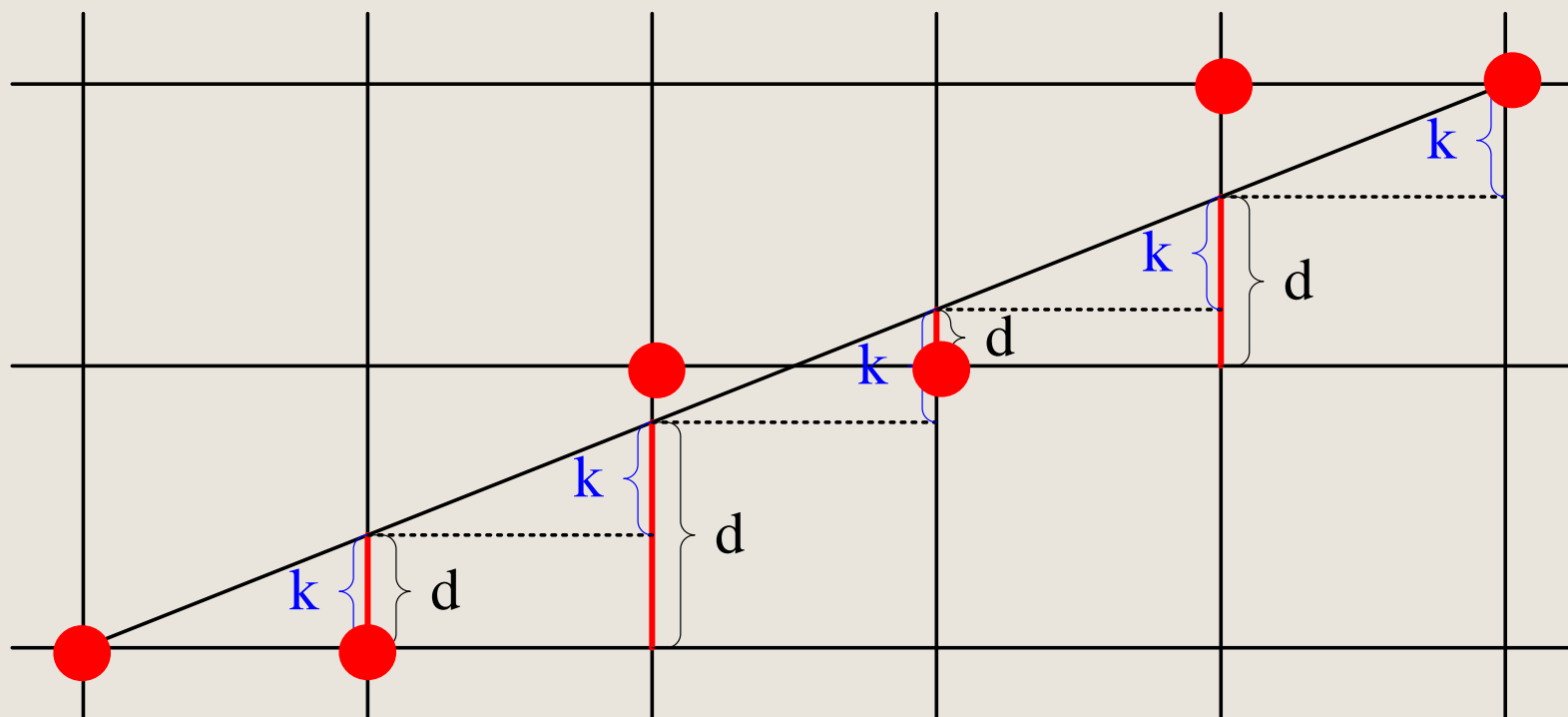
# 直线段的生成算法

➤ 数值微分法

➤ 中点Bresenham算法

➤ 改进Bresenham算法

## ● 改进的Bresenham算法绘制直线



$k$ : 直线的斜率

**缺点1：与0.5做比较，涉及浮点数运算**

**改进：引入判别式 $e$ 替换 $d$ ： $e=d-0.5$**

$$\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} = \begin{cases} y_i + 1 & (d > 0.5) \\ y_i & (d \leq 0.5) \end{cases} \end{cases} \longrightarrow \begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} = \begin{cases} y_i + 1 & (e > 0) \\ y_i & (e \leq 0) \end{cases} \end{cases}$$

## 误差项的计算

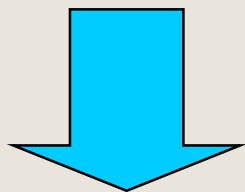
- $d_{\text{初}}=0,$
- $e_{\text{初}}=-0.5,$
- 每走一步： $d=d+k$
- 每走一步有 $e=e+k。$
- if  $(d>1)$  then  $d=d-1$
- if  $(e>0.5)$  then  $e=e-1$

**含有0.5和k**

缺点2:  $k$ 涉及浮点数的运算

改进: 引入判别式 $E=2e\Delta x$ 来替换 $e$

- $e_{\text{初}} = -0.5$
- 每走一步有 $e = e + k$
- if ( $e > 0.5$ ) then  $e = e - 1$



- $E_{\text{初}} = -0.5 * 2\Delta x = -\Delta x$
- 每走一步有 $E = (e + k) * 2\Delta x = E + 2\Delta y$
- if ( $E > \Delta x$ ) then  $E = (e - 1) * 2\Delta x = E - 2\Delta x$



# 直线的生成算法

## 算法步骤

➤ 数值微分法

➤ 中点Bresenham算法

➤ 改进Bresenham算法

1. 输入直线的两端点  $P_0(x_0, y_0)$  和  $P_1(x_1, y_1)$ 。
2. 计算初始值  $\Delta x$ 、 $\Delta y$ 、 $e = -\Delta x$ 、 $x = x_0$ 、 $y = y_0$ 。
3. 绘制点  $(x, y)$ 。
4.  $e$  更新为  $e + 2\Delta y$ ，判断  $e$  的符号。若  $e > 0$ ，则  $(x, y)$  更新为  $(x + 1, y + 1)$ ，同时将  $e$  更新为  $e - 2\Delta x$ ；否则  $(x, y)$  更新为  $(x + 1, y)$ 。
5. 当直线没有画完时，重复步骤3和4。否则结束。

# 第3章 基本图形生成算法

## 主要内容

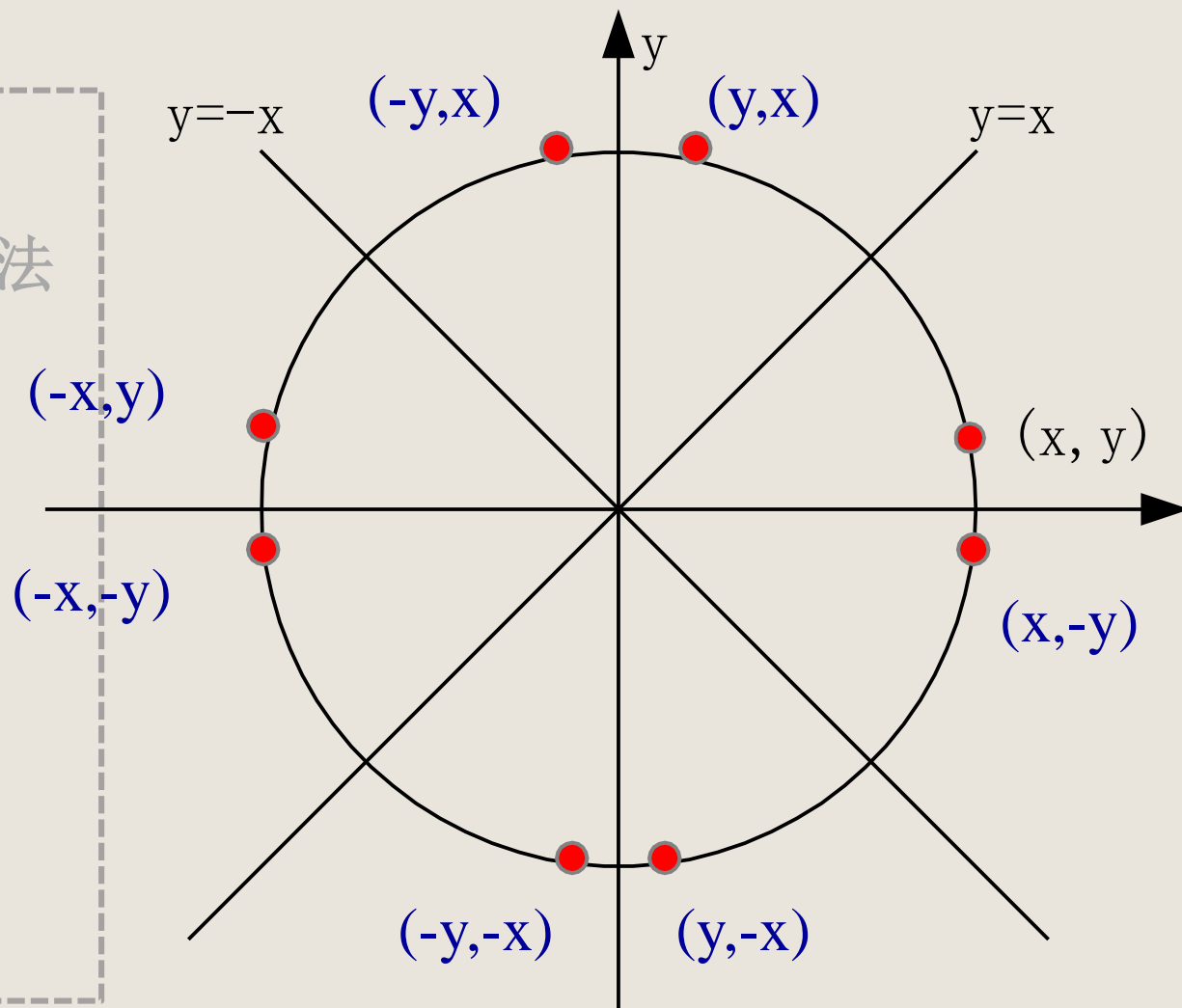
- 直线段的生成算法
- 圆弧的生成算法
- 多边形扫描转换
- 区域填充算法
- 反走样技术

**解决的问题：**绘出圆心在原点，  
半径为整数R的圆 $x^2+y^2=R^2$ 。

# 第3章 基本图形生成算法

## 主要内容

- 直线段的生成算法
- 圆弧的生成算法
- 多边形扫描转换
- 区域填充算法
- 反走样技术

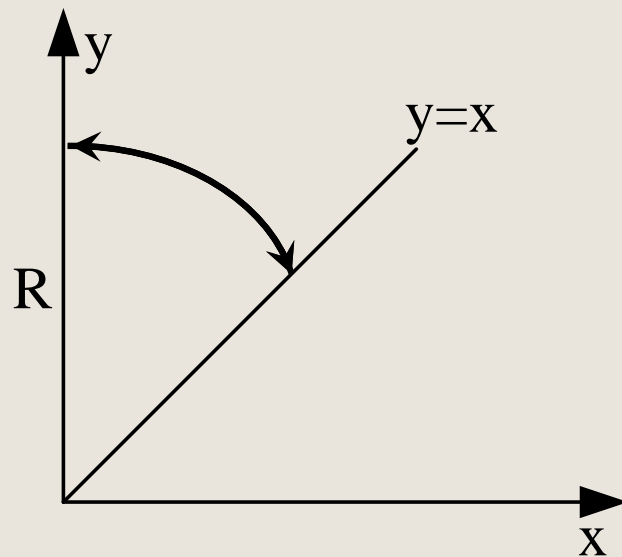


八分法画圆

# 第3章 基本图形生成算法

## 主要内容

- 直线段的生成算法
- 圆弧的生成算法
- 多边形扫描转换
- 区域填充算法
- 反走样技术



1/8圆弧

# 第3章 基本图形生成算法

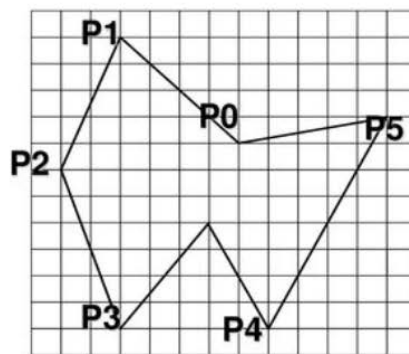
## 主要内容

- 直线段的生成算法
- 圆弧的生成算法
- 多边形扫描转换
- 区域填充算法
- 反走样技术

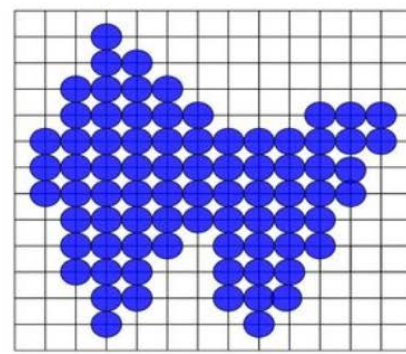
**概念：**多边形的顶点表示转换为多边形的点阵表示

**1) 顶点表示：**用多边形的顶点的序列来描述多边形，几何意义强、占内存少。

**2) 点阵表示：**用位于多边形内的像素的集合来描述多边形。



多边形的顶点表示法



多边形的点阵表示法

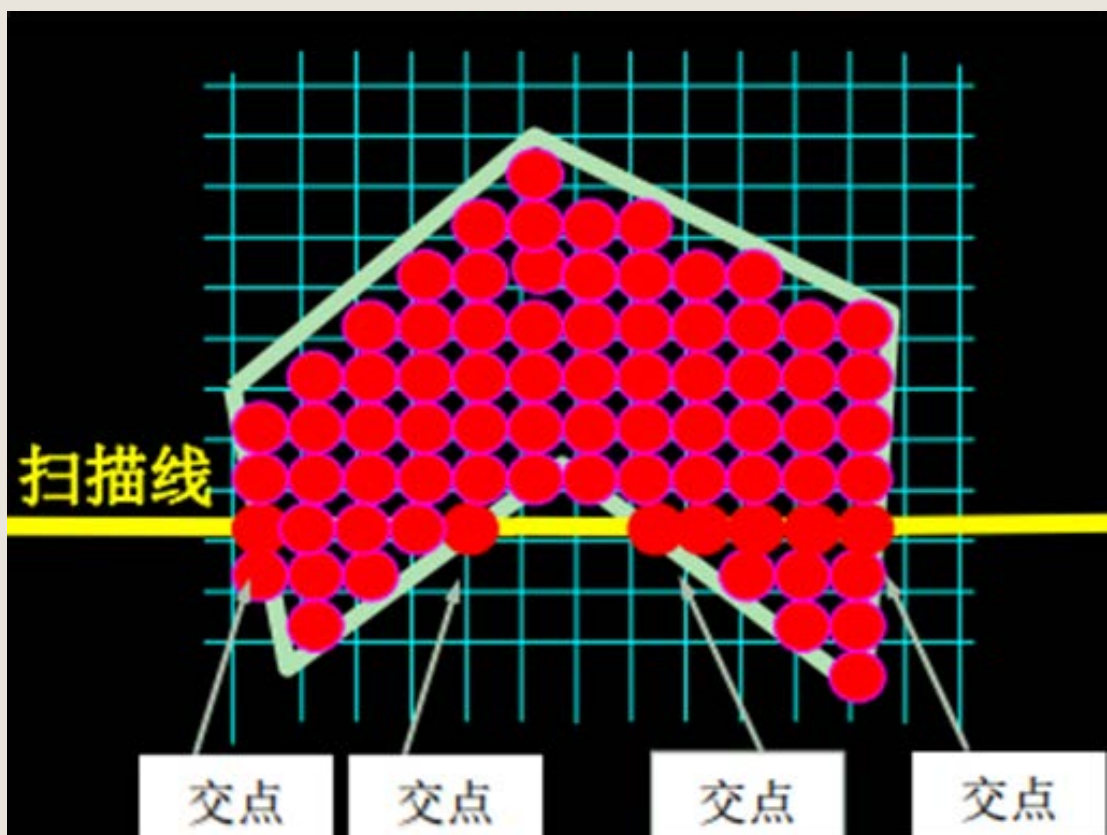
# 第3章 基本图形生成算法

## 主要内容

- 直线段的生成算法
- 圆弧的生成算法
- 多边形扫描转换
- 区域填充算法
- 反走样技术



**基本思想：**按扫描线顺序，计算扫描线与多边形的相交区间，再用要求的颜色显示这些区间的所有像素。



# 多边形扫描转换

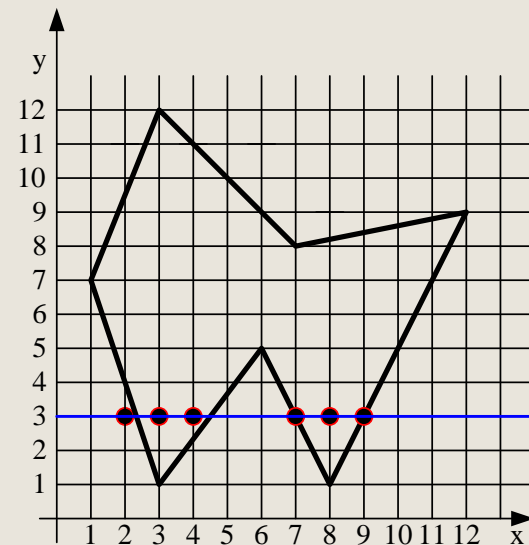
## X 扫描线算法

# 多边形扫描转换

## 算法步骤:

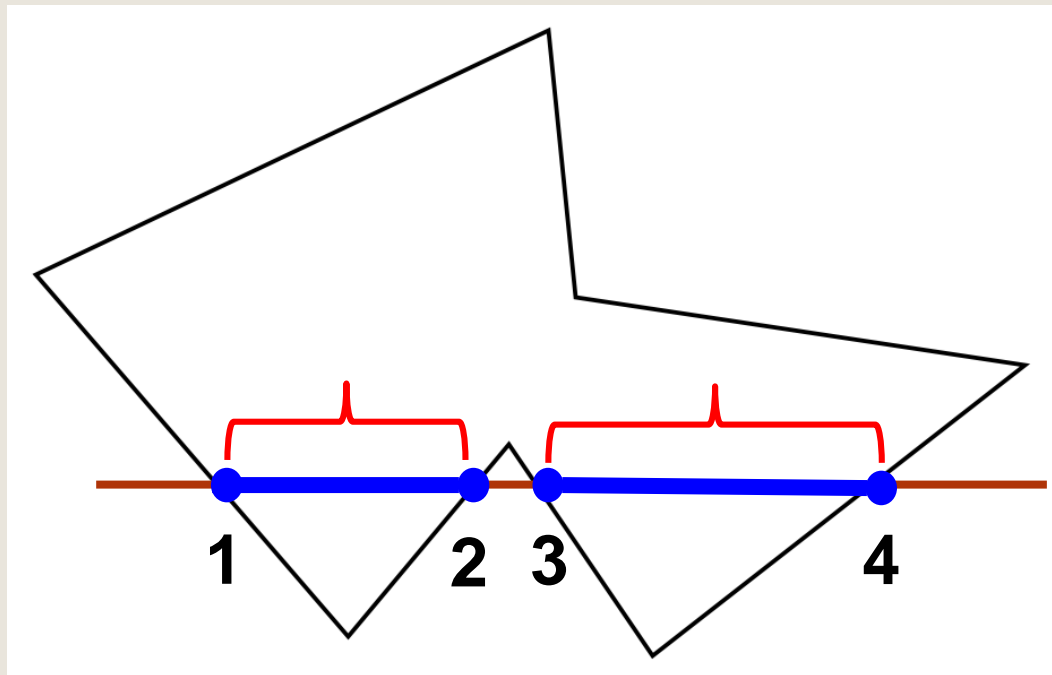
1. 得到多边形顶点的最小和最大y值 ( $y_{\min}$  和  $y_{\max}$ )。
2. 从  $y=y_{\min}$  到  $y=y_{\max}$ ，每次用一条扫描线进行填充。
3. 对一条扫描线填充的过程可分为四个步骤:

求交→排序→交点配对→区间填色





# 多边形扫描转换

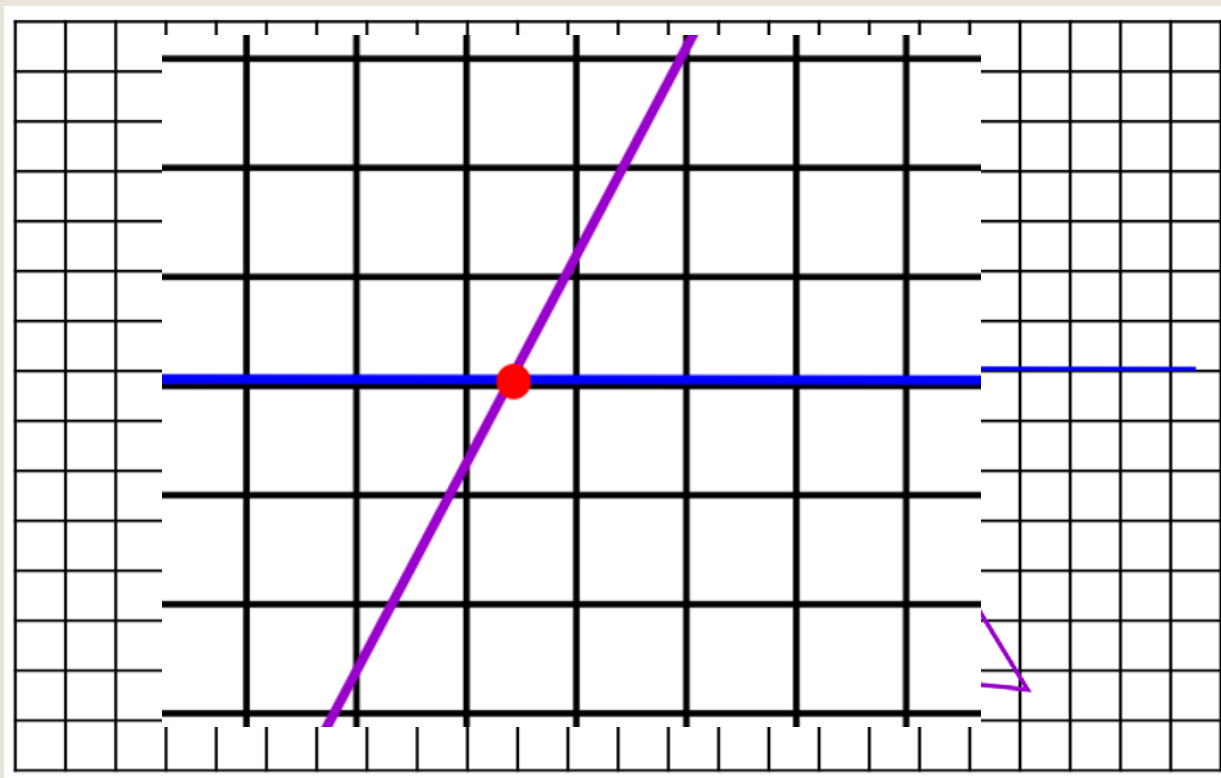


- **求交**: 计算扫描线与多边形所有边的交点;
- **排序**: 把所有交点按 $x$ 坐标递增进行排序;
- **交点配对**: 配对第1与第2个、第3与第4个交点等, 每对交点代表扫描线与多边形的一个相交区间;
- **区间填色**: 把相交区间内的像素设置成填充色。

# 多边形扫描转换

**问题1：** 为避免多边形区域扩大化，交点坐标的取整需要特殊处理。

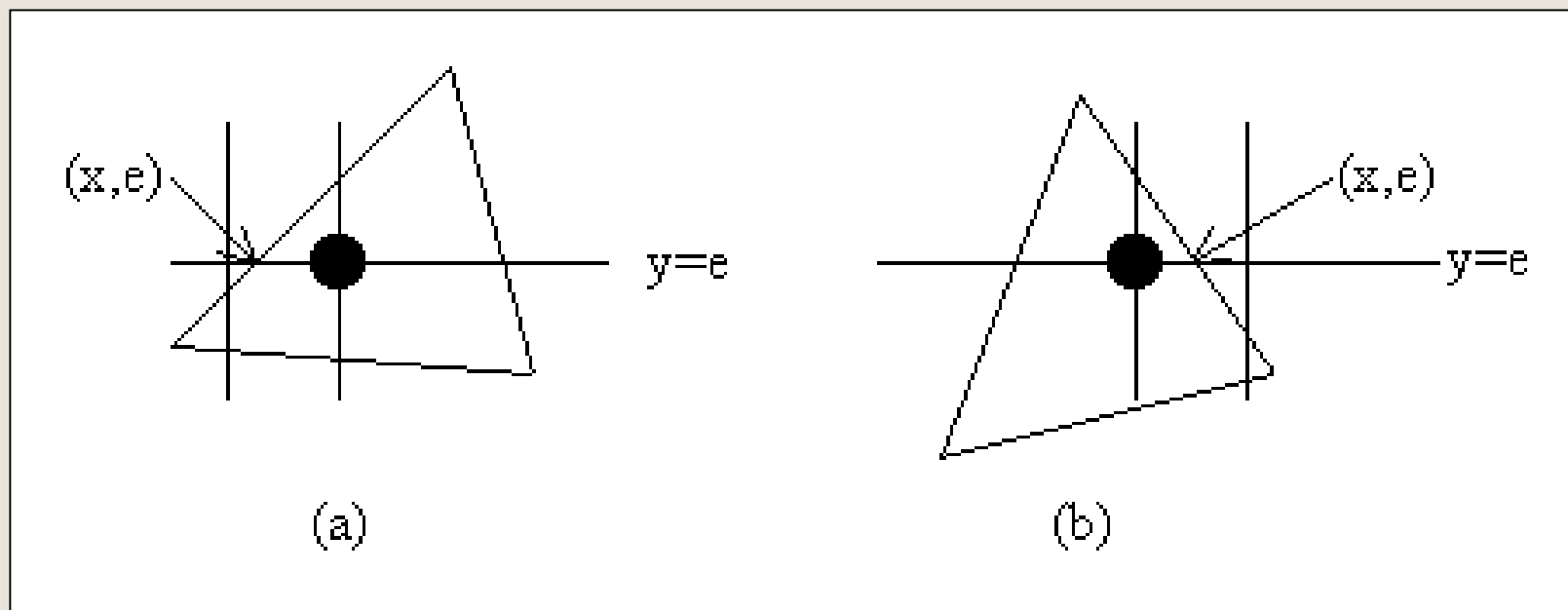
- **处理原则：** 使生成的像素全部位于多边形内部。



**问题1:** 为避免多边形区域扩大化, 交点坐标的取整需要特殊处理。

● **规则1:** 交点落于扫描线上两个相邻像素之间时( $x$ 为小数):

- 交点位于左边界之上, 向右取整;
- 交点位于右边界之上, 向左取整;

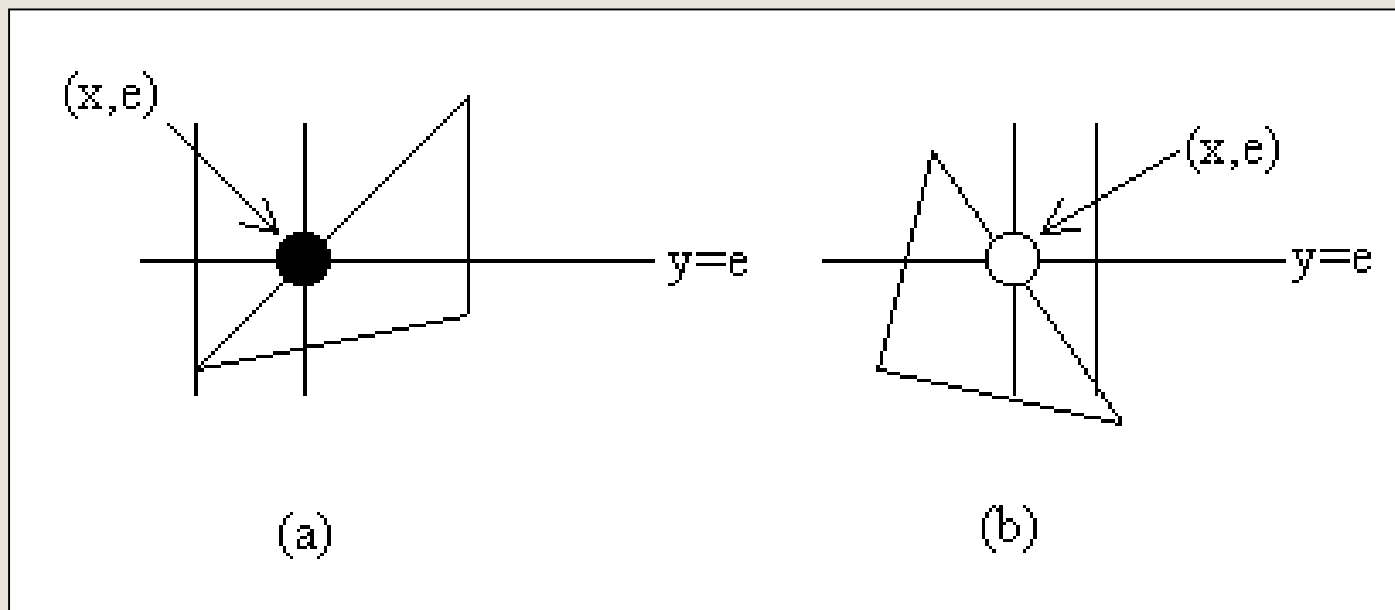


取整规则1

**问题1:** 为避免多边形区域扩大化, 交点坐标的取整需要特殊处理。

● **规则2: 交点位于边界像素上时( $x$ 为整数)**

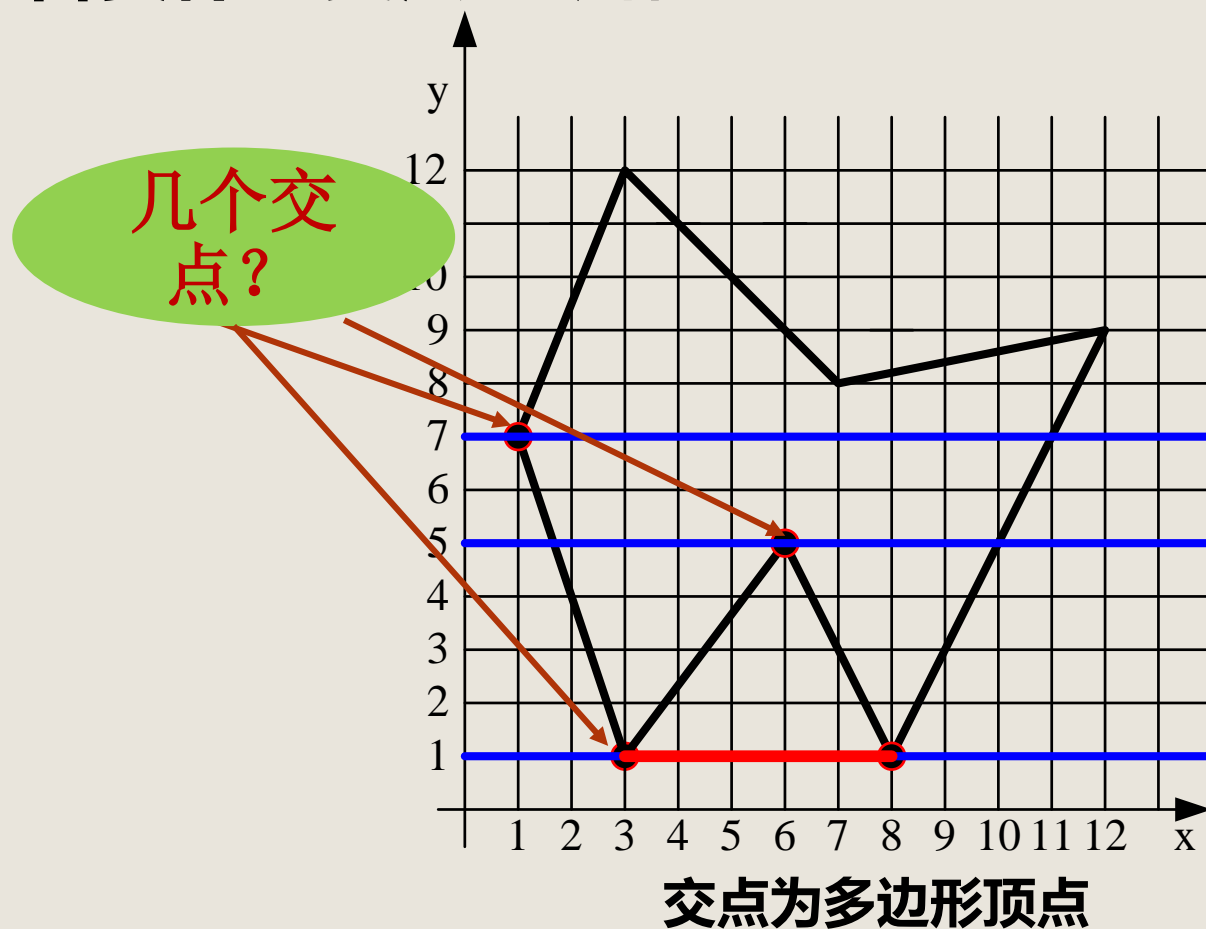
- 落在右边边界上的像素不予填充, 扫描线与多边形的相交区间左闭右开。避免两个多边形的共享边重复绘制。



取整规则2

# 多边形扫描转换

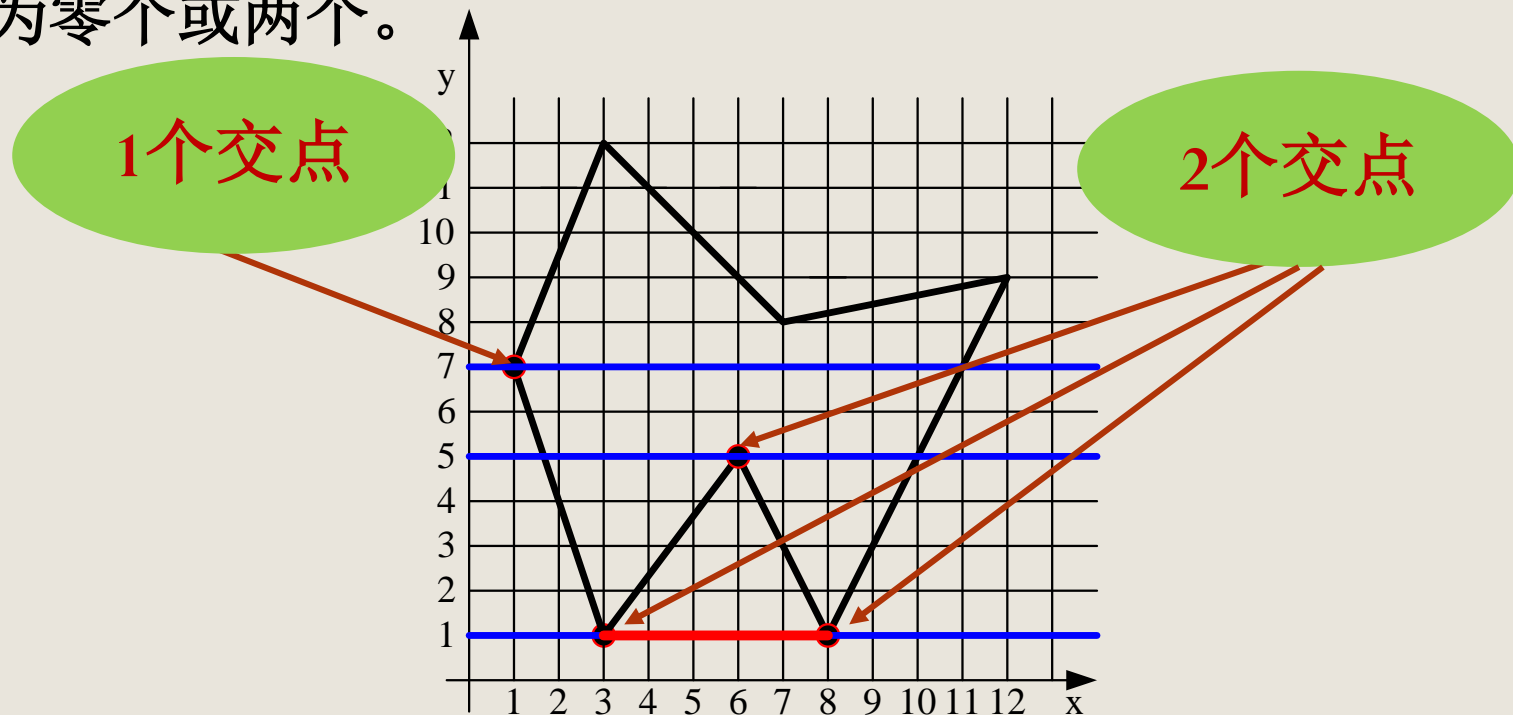
- **问题2：**当扫描线与多边形顶点相交时，交点的取舍要保证交点正确配对。



- **问题2:** 当扫描线与多边形顶点相交时, 交点的取舍要保证交点正确配对.

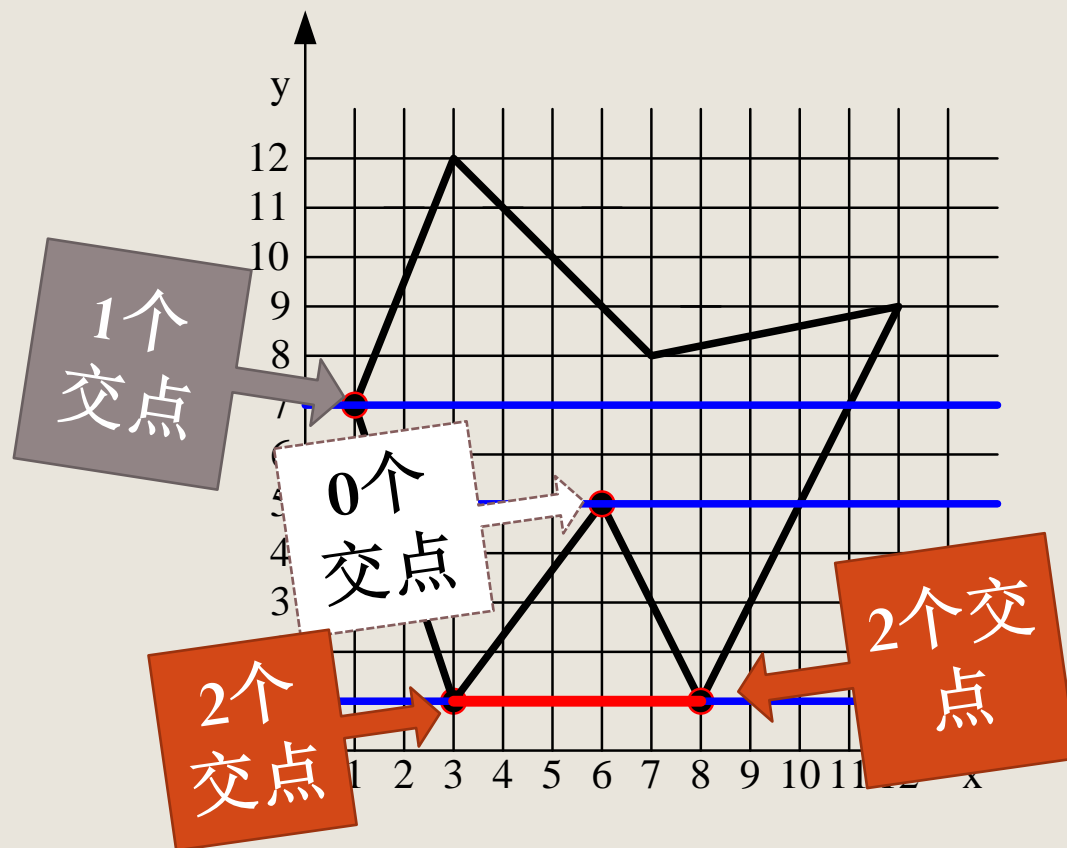
## 保证交点正确配对的方法:

- 若共享顶点的两条边分别落在扫描线的两边, 交点只算一个;
- 若共享顶点的两条边在扫描线的同一边, 这时交点作为零个或两个。



- **问题2:** 当扫描线与多边形顶点相交时，交点的取舍要保证交点正确配对.

**实际处理:** 只要检查多边形顶点的两条边的另外两个端点的Y值，两个Y值中大于交点Y值的个数是0, 1, 2, 来决定取0, 1, 2个交点。



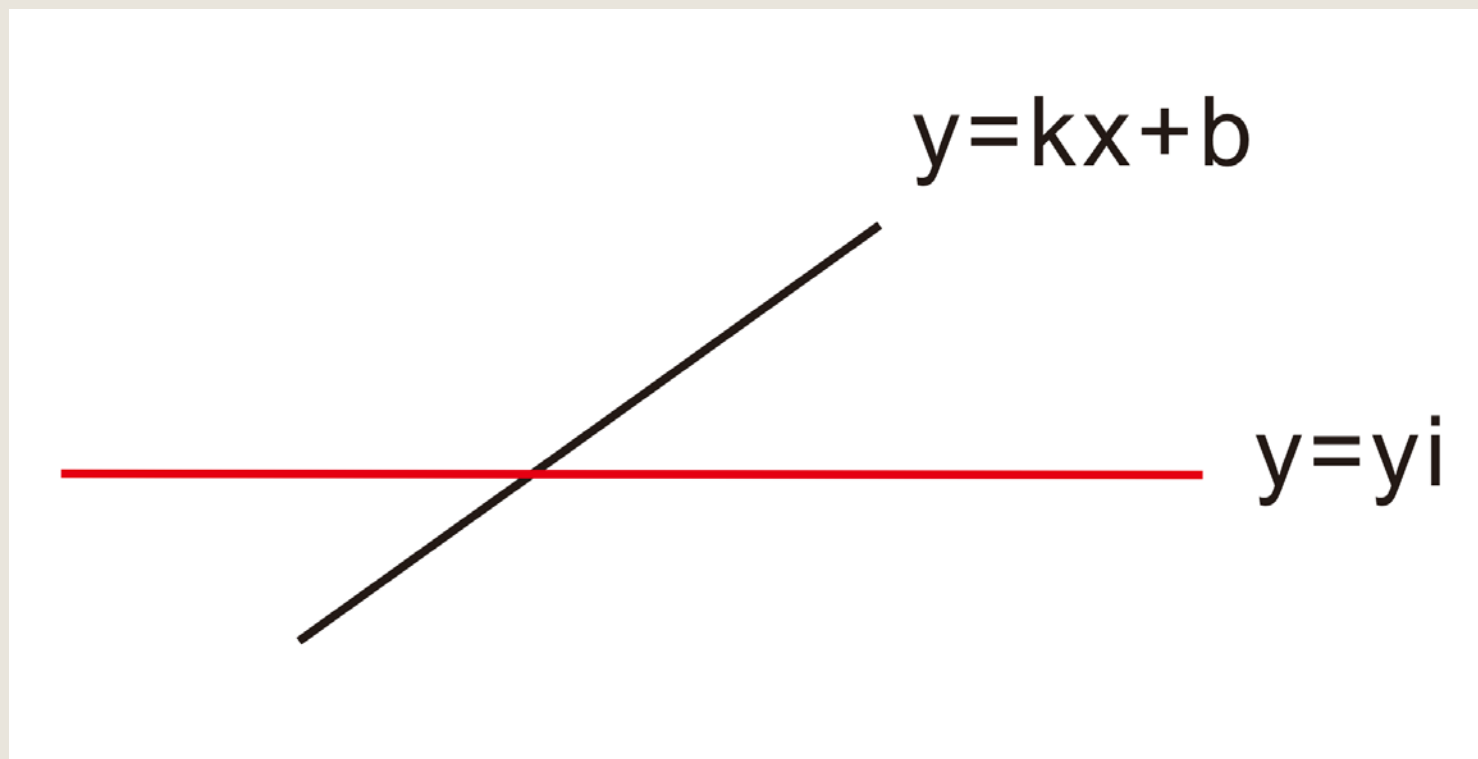
# 多边形扫描转换

**有效边表算法(Y连贯性算法)**



# 多边形扫描转换

- 多边形扫描转换算法大量涉及扫描线和多边形边界线段的交点计算



# 多边形扫描转换

- 多边形扫描转换算法大量涉及 扫描线和多边形边界线段的交点计算
- 为了简化交点的计算，利用

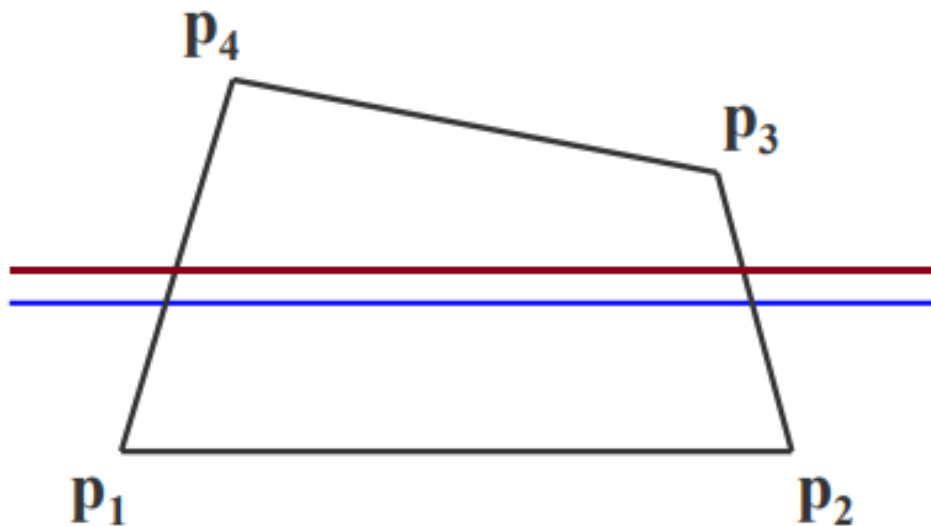
✓ 多边形**边**的连贯性

✓ 扫描线的连贯性

# 多边形扫描转换

- **多边形边的连贯性**：多边形的一条边是连续的

✓ 某边与当前扫描线相交，则它很可能与下一条扫描线相交。

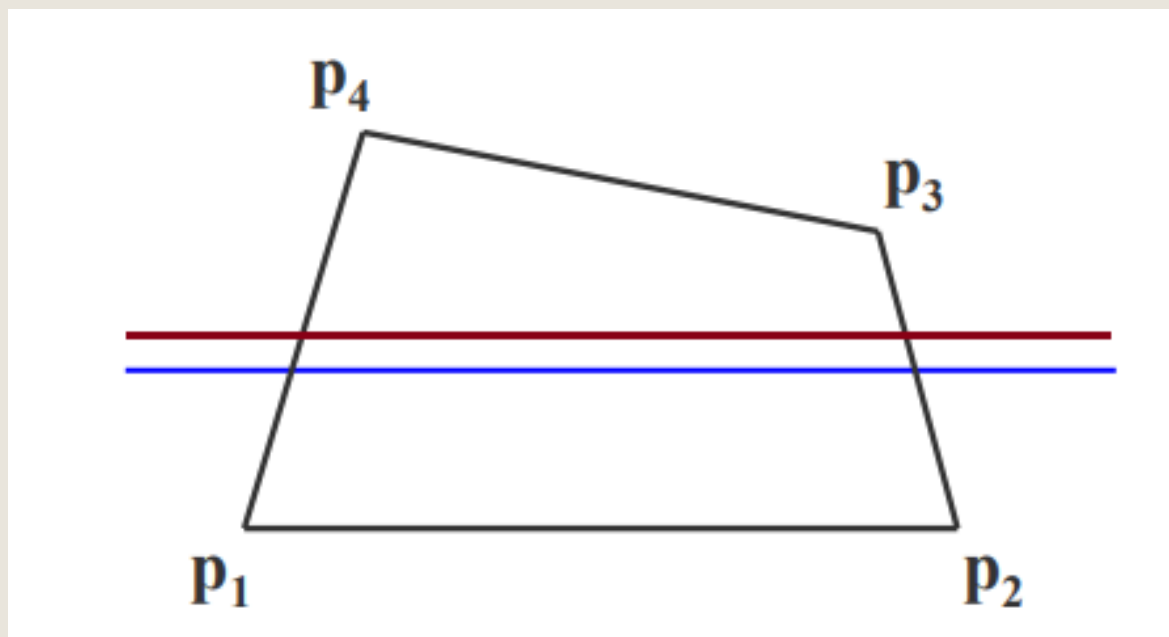


**注意：**自下而上依次处理扫描线 ( $y=y+1$ )

# 多边形扫描转换

- **扫描线的连贯性：** 两条相邻扫描线离得很近

✓ 当前扫描线与各边交点顺序与下一条扫描线与各边的交点顺序可能相同或相近。



**注意：** 自下而上依次处理扫描线 ( $y=y+1$ )

# 多边形扫描转换

□ 如何利用扫描线连贯性和边连贯性加速多边形扫描转换？ 有效边表算法

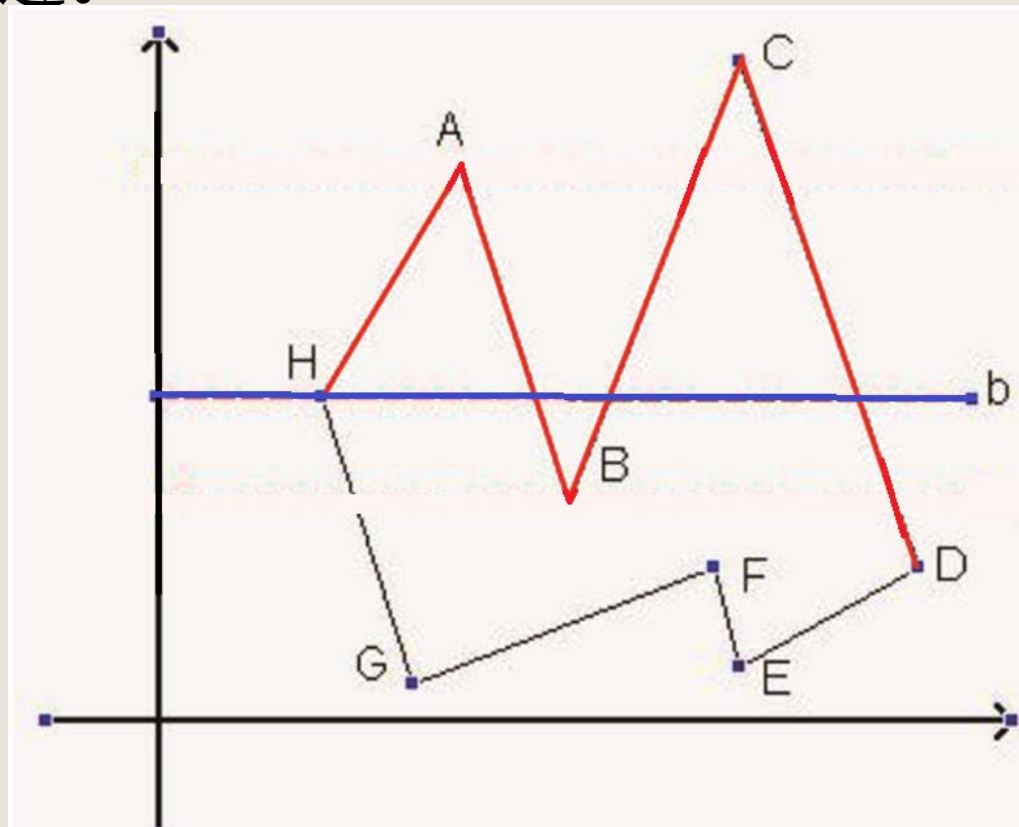
两个数据结构：

- 有效边表 (Active Edge Table, AET)
- 边表 (Edge Table, ET)

# 多边形扫描转换

## □有效边表算法（Y连贯性算法）

- 有效边（**Active Edge**）：指与当前扫描线相交的多边形的边。



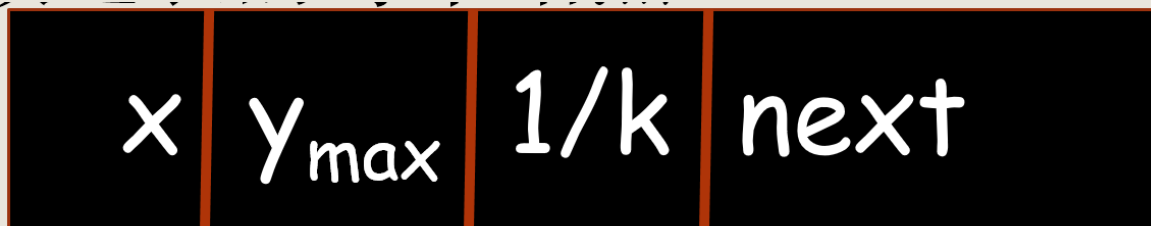
# 多边形扫描转换

## □有效边表算法（Y连贯性算法）

**数据结构 1：**有效边表(Active Edge Table, AET):

把有效边按与扫描线交点 $x$ 坐标递增的顺序存放在一个链表中，此链表称为有效边表。

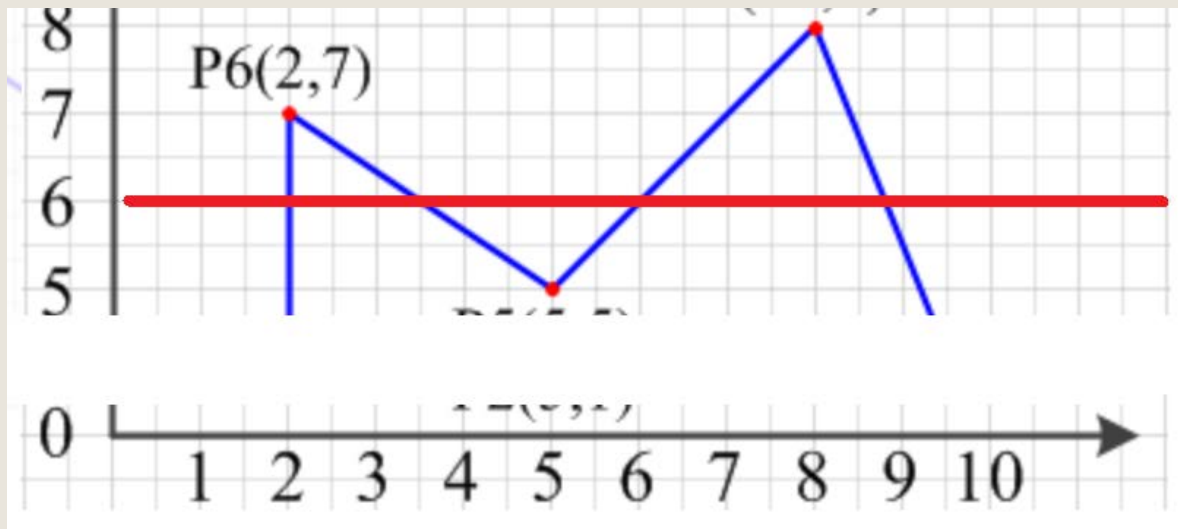
- 有效边表中，多边形一个边用如下结点表示



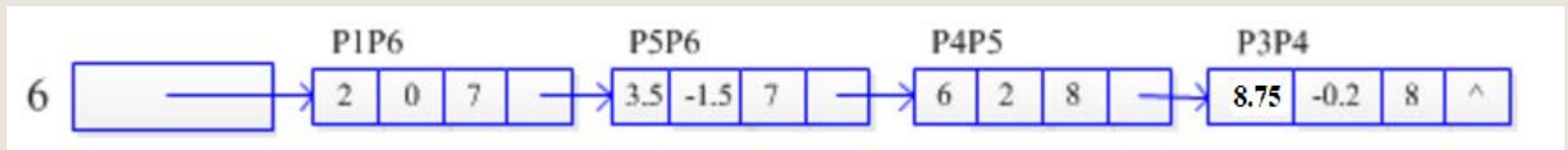
# 多边形扫描转换

## 有效边表算法 (Y连贯性算法)

数据结构 1: 有效边表(Active Edge Table, AET):



第6条扫描线的活动边表是:





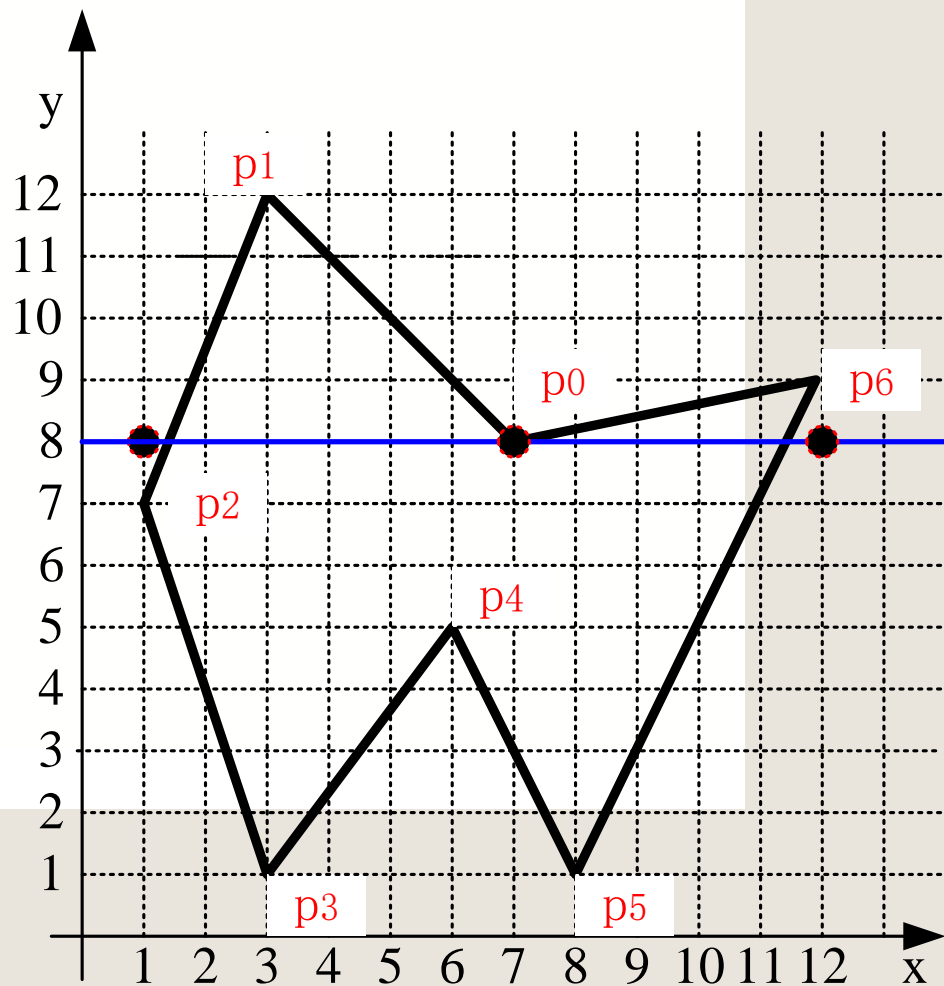
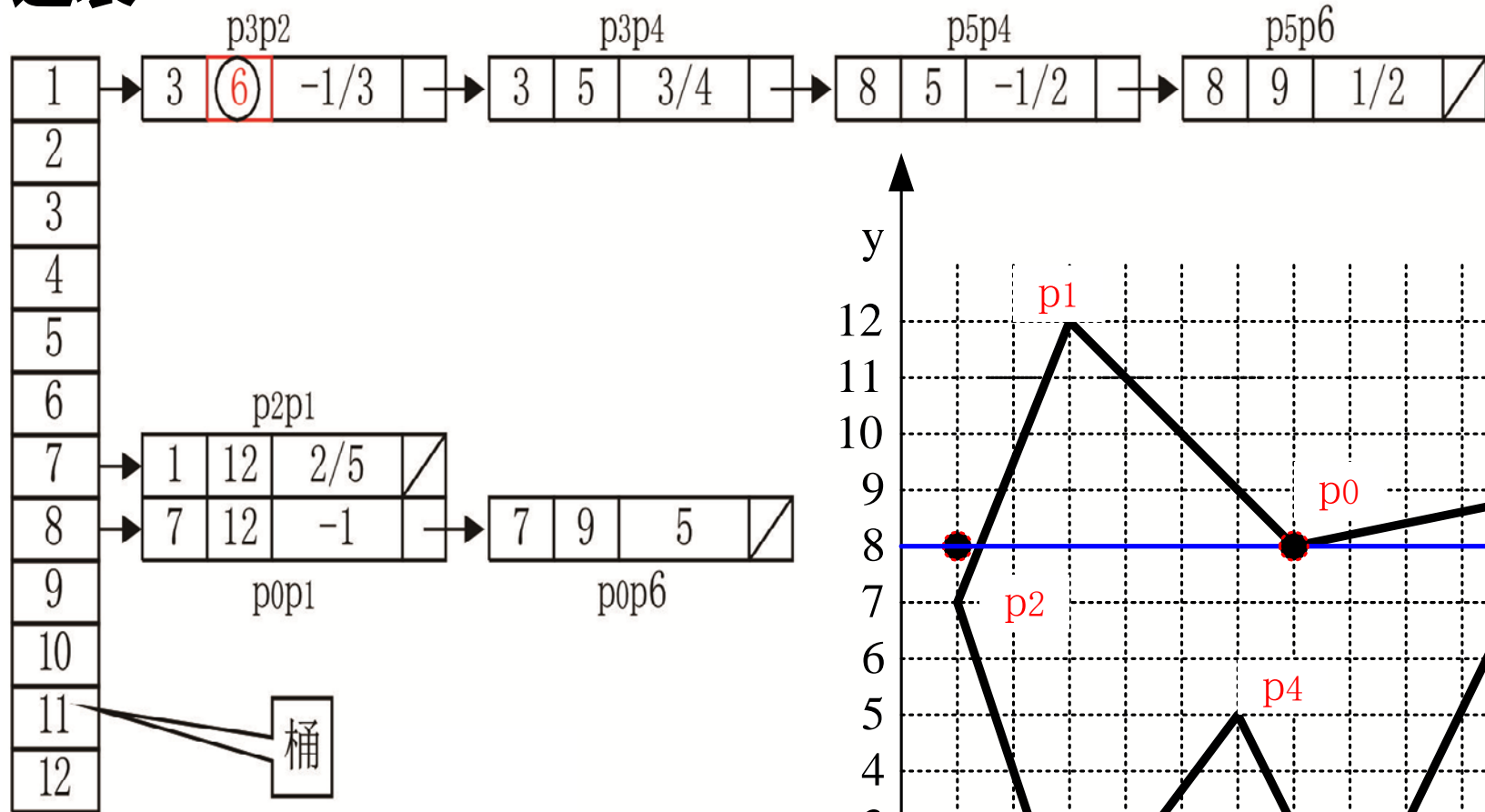
# 多边形扫描转换

## □有效边表算法（Y连贯性算法）

数据结构 2：边表（Edge Table, ET）：

- ① 一个纵向链表，链表的长度为多边形所占有的扫描线数，链表的一个结点(桶)对应多边形覆盖的二条扫描线。
- ② 将每条边的信息放入与该边最小 $y$ 坐标（ $y_{\min}$ ）相对应的桶处： 若某边的较低端点为 $y_{\min}$ ，则该边就放在相应的扫描线桶中。

# 边表



多边形 $P_0P_1P_2P_3P_4P_5P_6$

## 边表(ET) (续)

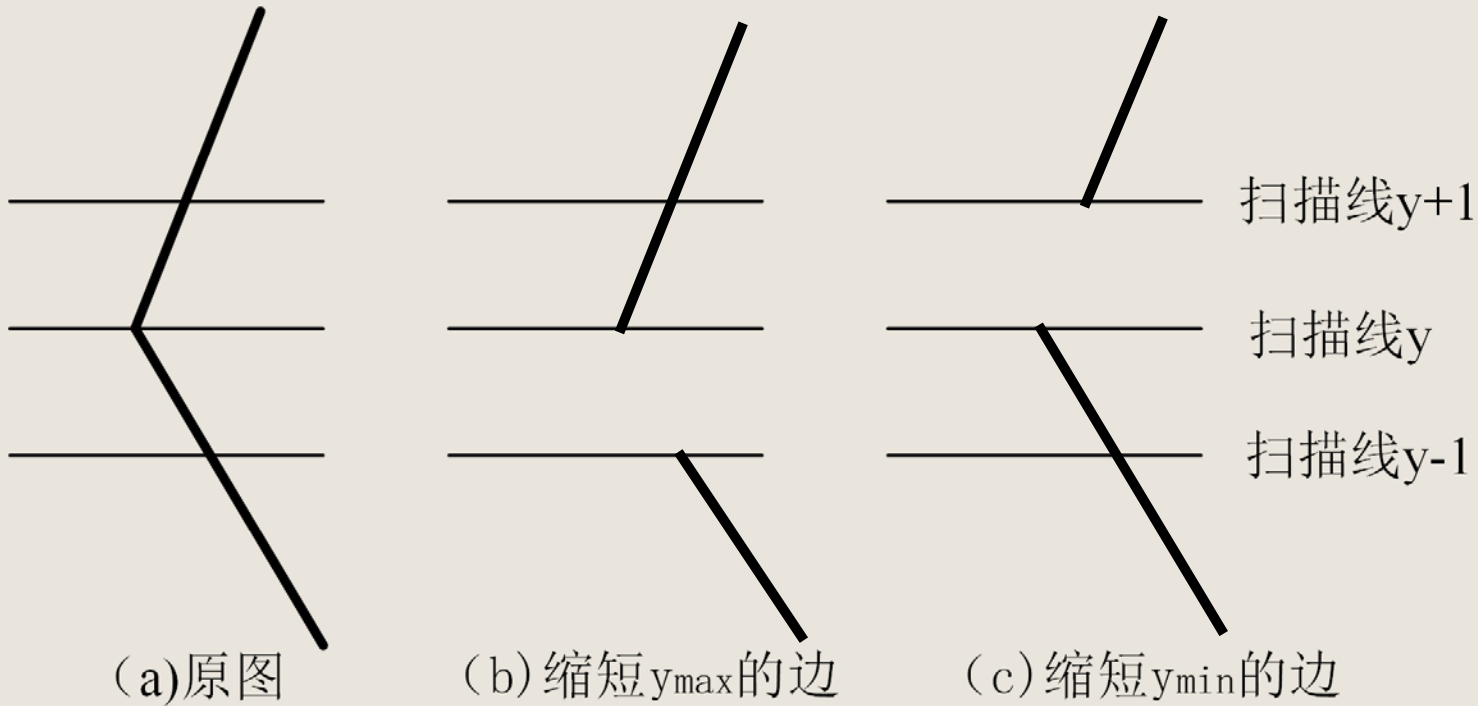
- 每条边结点内容包括：该边较低端点的 $x$ 值： $X|_{ymin}$ ， $1/k$ ，该边的最大 $y$ 值 $y_{max}$ 。



- 同一桶中若干条边按 $X|_{ymin}$ 由小到大排序，若 $X|_{ymin}$ 相等，则按照 $1/k$ 由小到大排序。

## 边表(ET) (续)

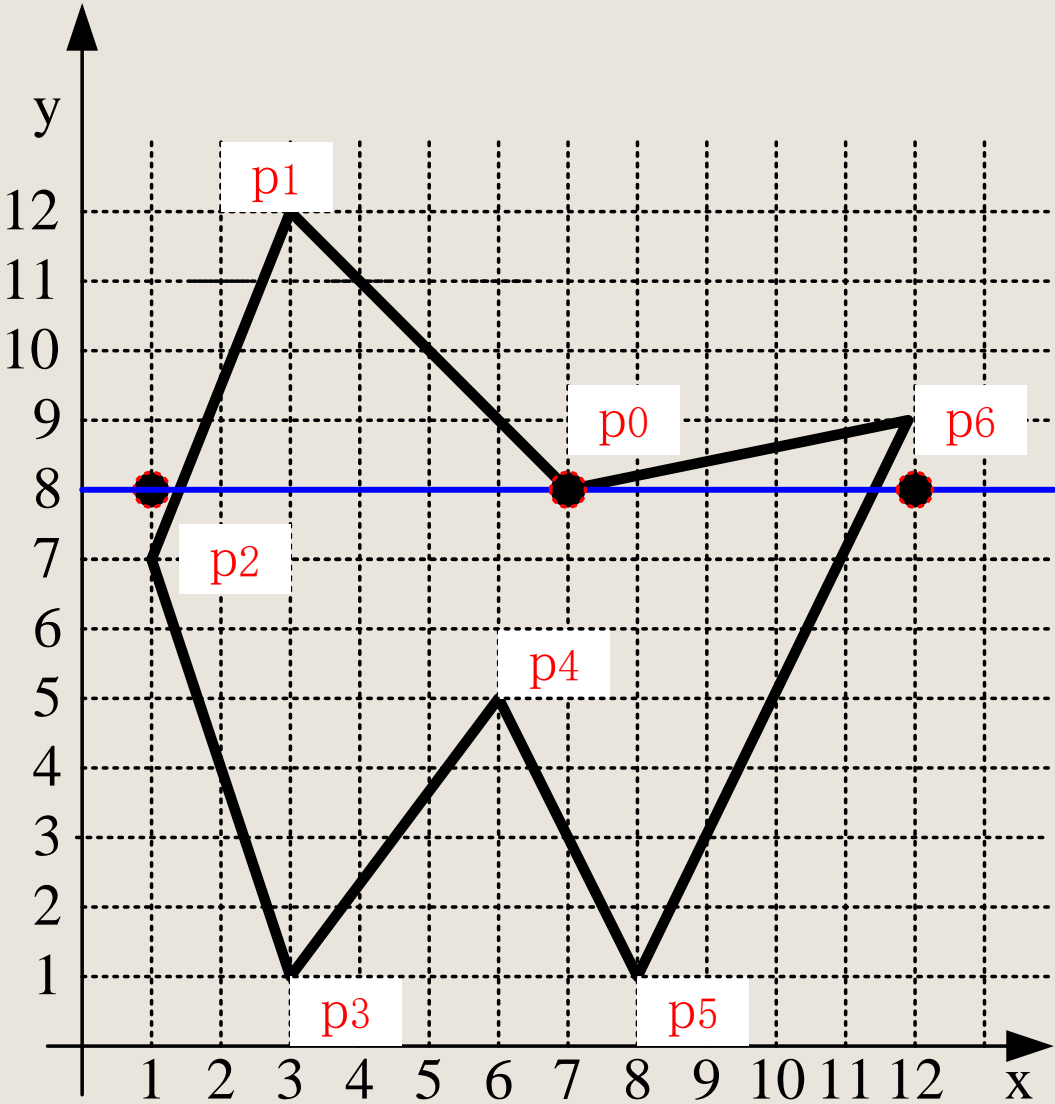
顶点与扫描线交点计为**1**时(希望只有一条边与扫描线相交), 边表的处理:



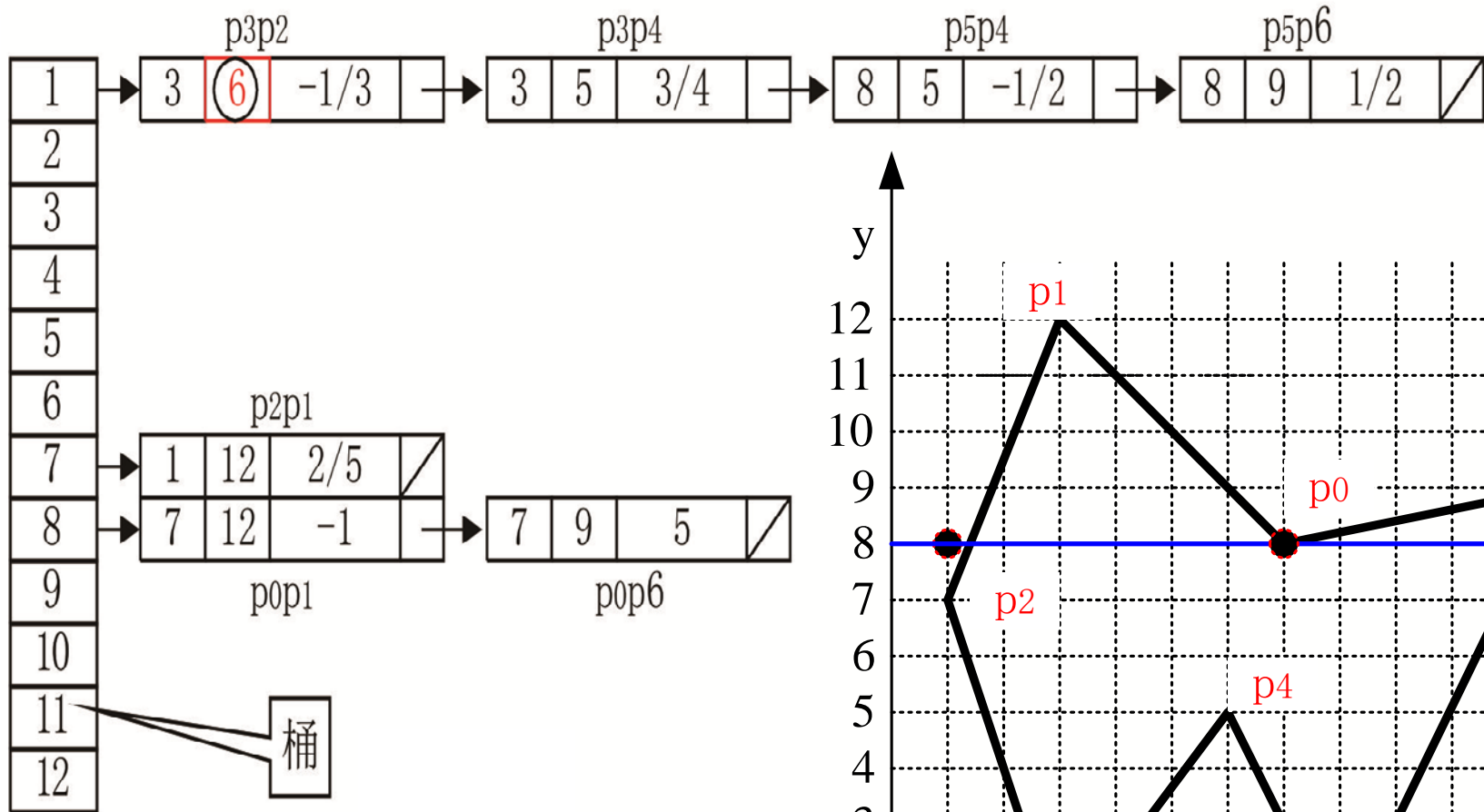
**将多边形的某些边缩短以分离那些应计为1个交点的顶点**

多边形所有的边

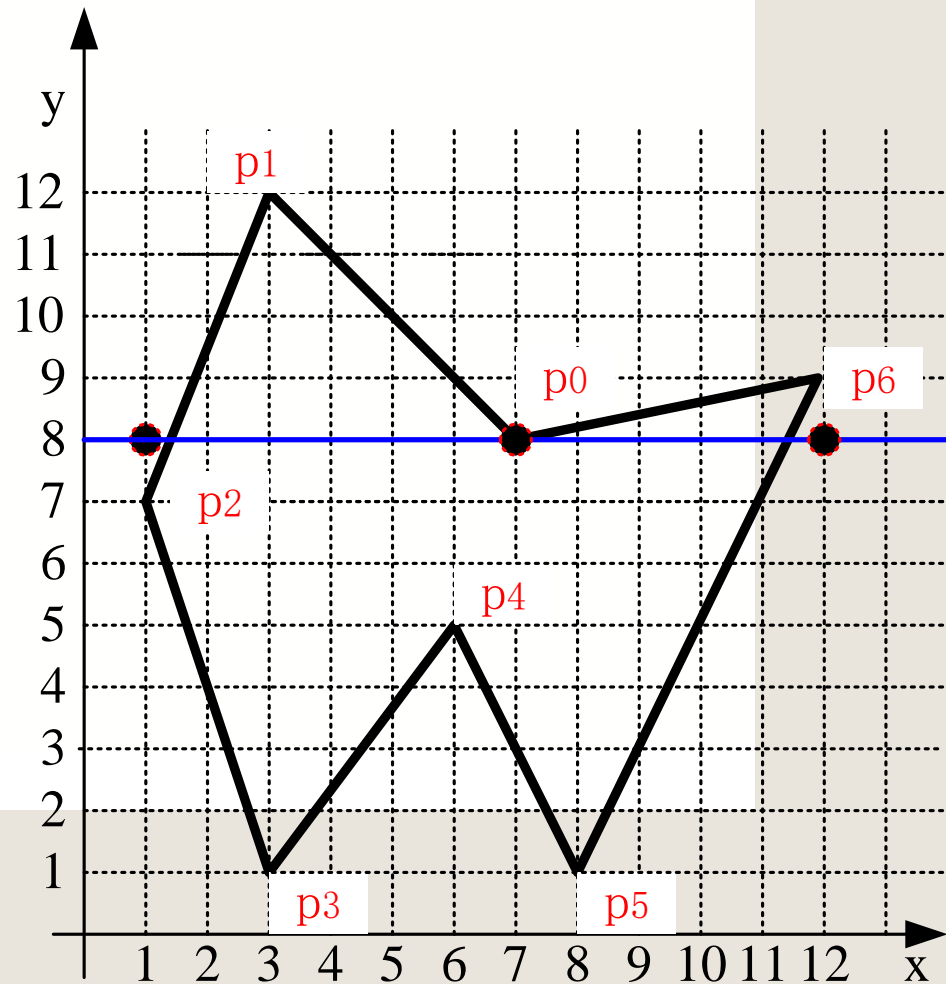
p3p2	3	6	-1/3	
p3p4	3	5	3/4	
p5p4	8	5	-1/2	
p5p6	8	9	1/2	
p2p1	1	12	2/5	
p0p1	7	12	-1	
p0p6	7	9	5	



多边形 $P_0P_1P_2P_3P_4P_5P_6$



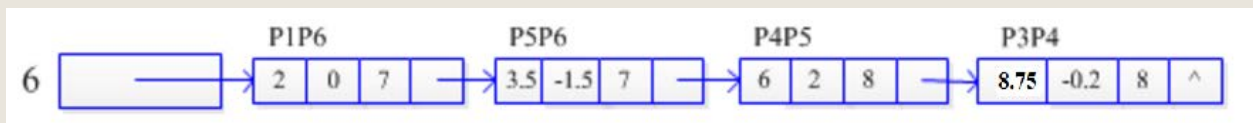
边表



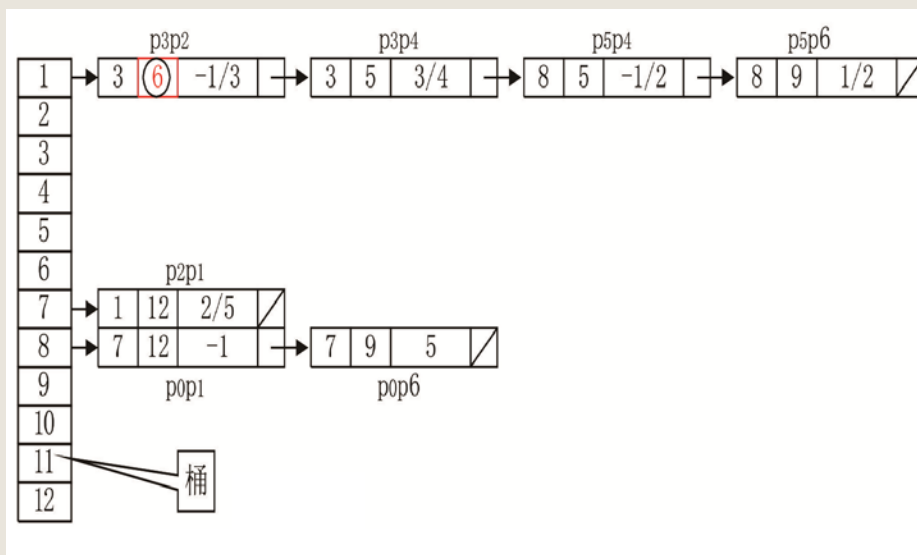
多边形 $P_0P_1P_2P_3P_4P_5P_6$

## ■ 有效边表算法——主要思想

(1) 活动边表**AET**的作用？ 得到当前扫描线和多边形边的交点， 然后进行配对和填充。



(2) 边表**ET**的作用？ 整个多边形边的一种组织， 用来对**AET**进行更新。



## ■ 有效边表算法——算法步骤

- (1)初始化：构造边表，**AET**表置空；
- (2)将**ET**表中以一个非空的桶的边列表与**AET**合并；
- (3)由**AET** 中取出交点对进行填充。填充之后删除 $y=y_{\max}$ 的边；
- (4) $y_{i+1}=y_i+1$ ，根据 $x_{i+1}=x_i+1/k$ 计算并修改**AET**表，同时合并**ET**表中 $y=y_{i+1}$ 桶中的边，按次序插入到**AET**表中，形成新的**AET**表；
- (5)**AET**表不为空则转(3)，否则结束。

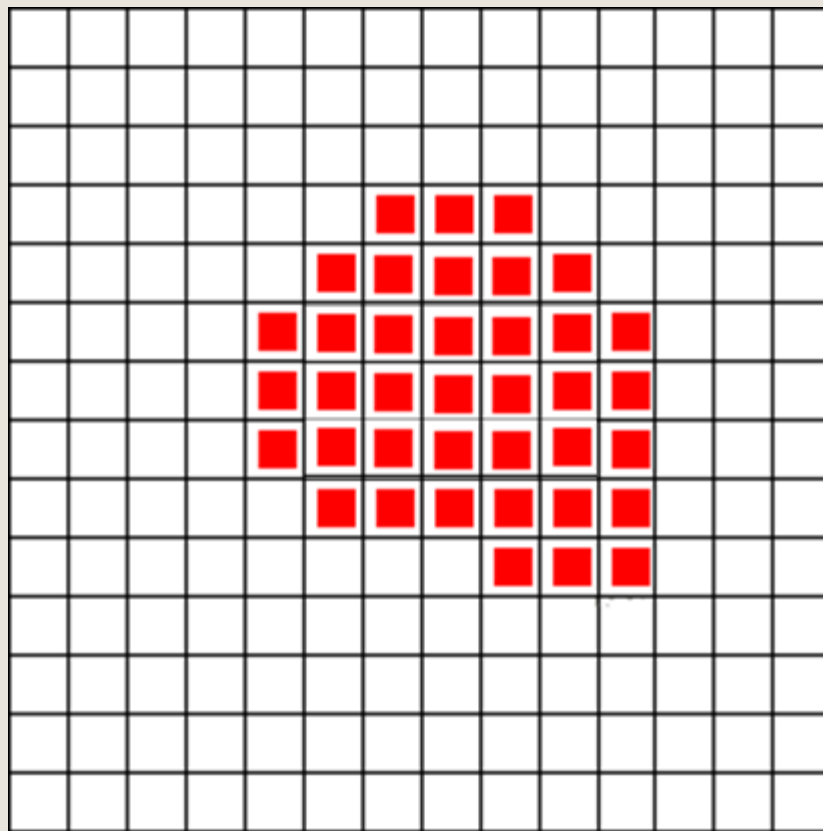


# 第3章 基本图形生成算法

## 主要内容

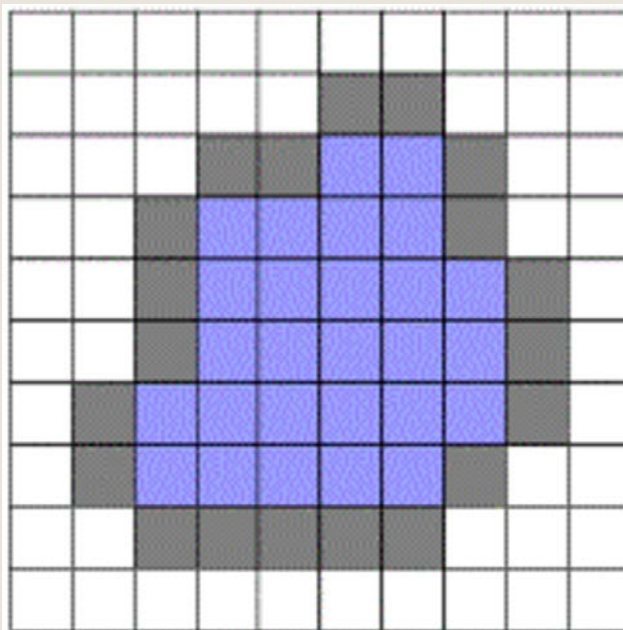
- 直线段的生成算法
- 圆弧的生成算法
- 多边形扫描转换
- 区域填充算法
- 反走样技术

**概念：** **区域**是指已经表示成点阵形式的图形，它是相互**连通**的一组像素的集合。



# 区域填充

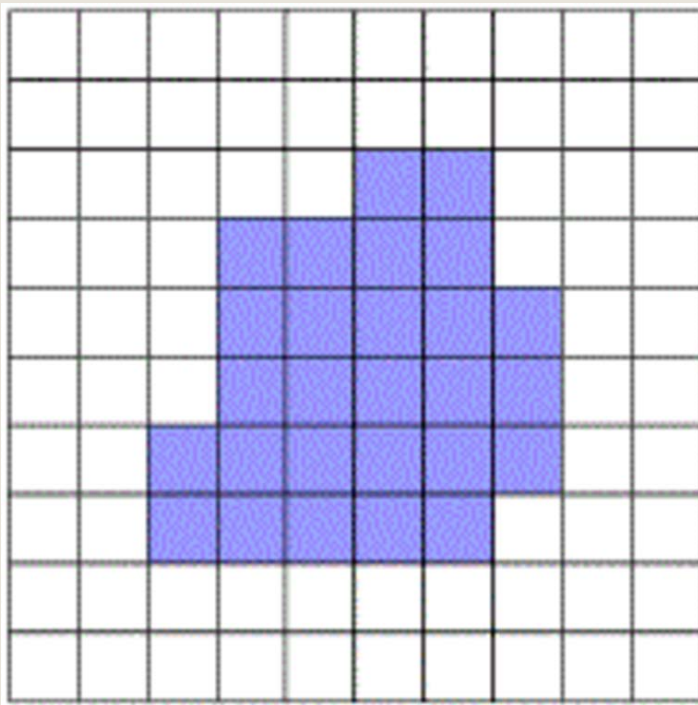
- 区域的表示方法
- 区域的分类
- 区域填充算法



- **边界表示法**：把位于给定区域的边界上的像素一一列举出来的方法。
- 边界上的像素着一种颜色，区域内和区域外的像素着不同于边界的颜色。
- 有显式边界。

# 区域填充

- 区域的表示方法
- 区域的分类
- 区域填充算法



- **内点表示法**：枚举出给定区域内所有像素的表示方法。
- 区域内的所有像素着一种颜色，区域外的像素着另一种颜色。
- 没有显式边界。

# 区域填充

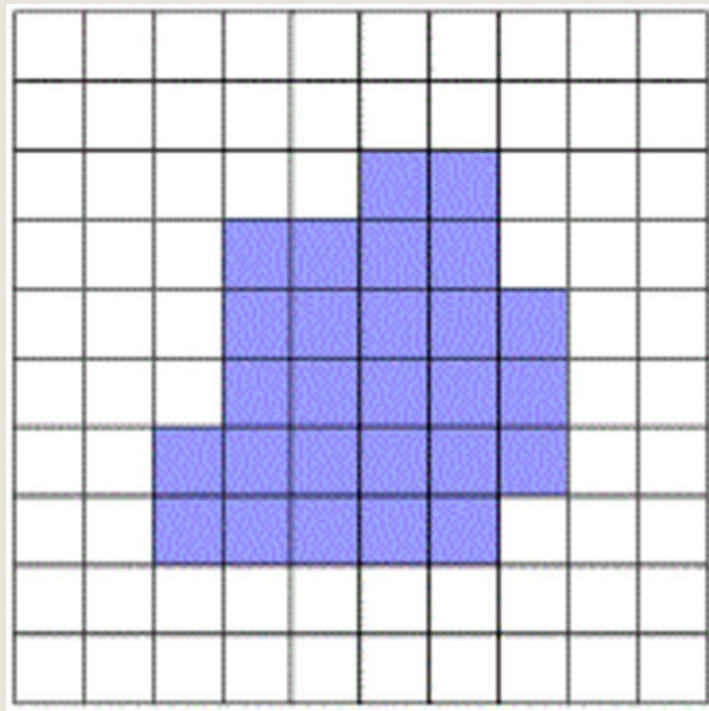
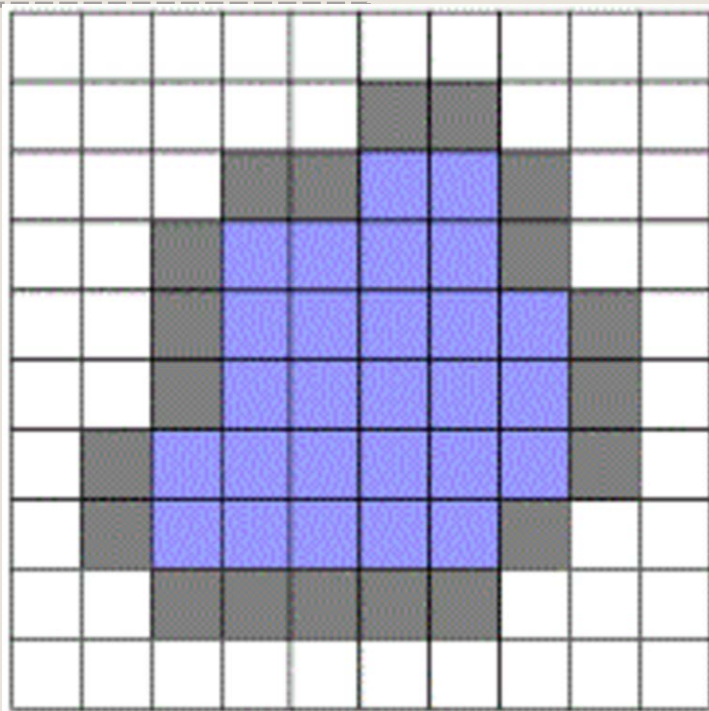
- 区域的表示方法

- 区域的分类

- 区域填充算法

- 边界表示法

- 内点表示法

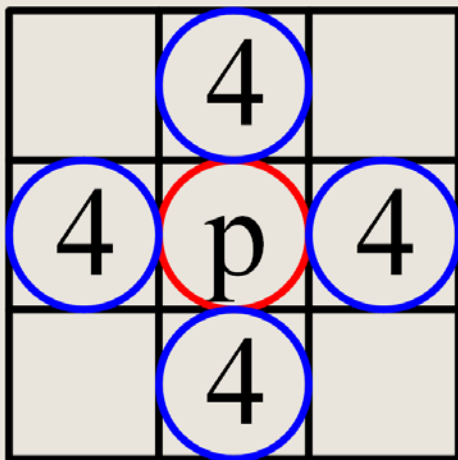


# 区域填充

- 区域的表示方法
- 区域的分类
- 区域填充算法

- 4连通区域
- 8连通区域

➤ **4连通区域**：从区域内一像素出发，通过4个方向，即上、下、左、右移动的组合，在不越出区域的前提下，可到达区域内任何像素。



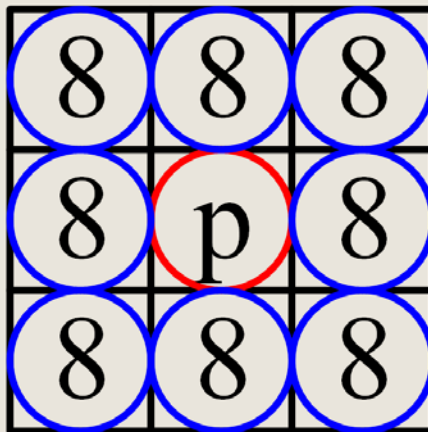
# 区域填充

- 区域的表示方法
- 区域的分类
- 区域填充算法

➤ 4连通区域

➤ 8连通区域

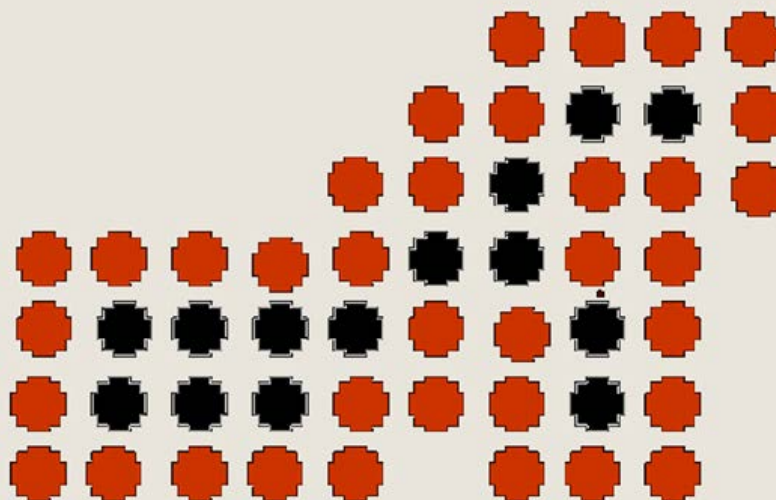
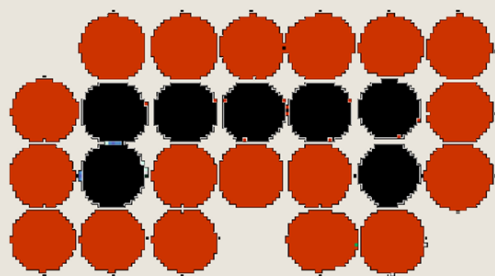
➤ **8连通区域**：从区域内一像素出发，通过上、下、左、右、左上、左下、右上、右下8种移动的组合，在不越出区域的前提下，可到达区域内任何像素。



# 区域填充

- 区域的表示方法
- 区域的分类
- 区域填充算法

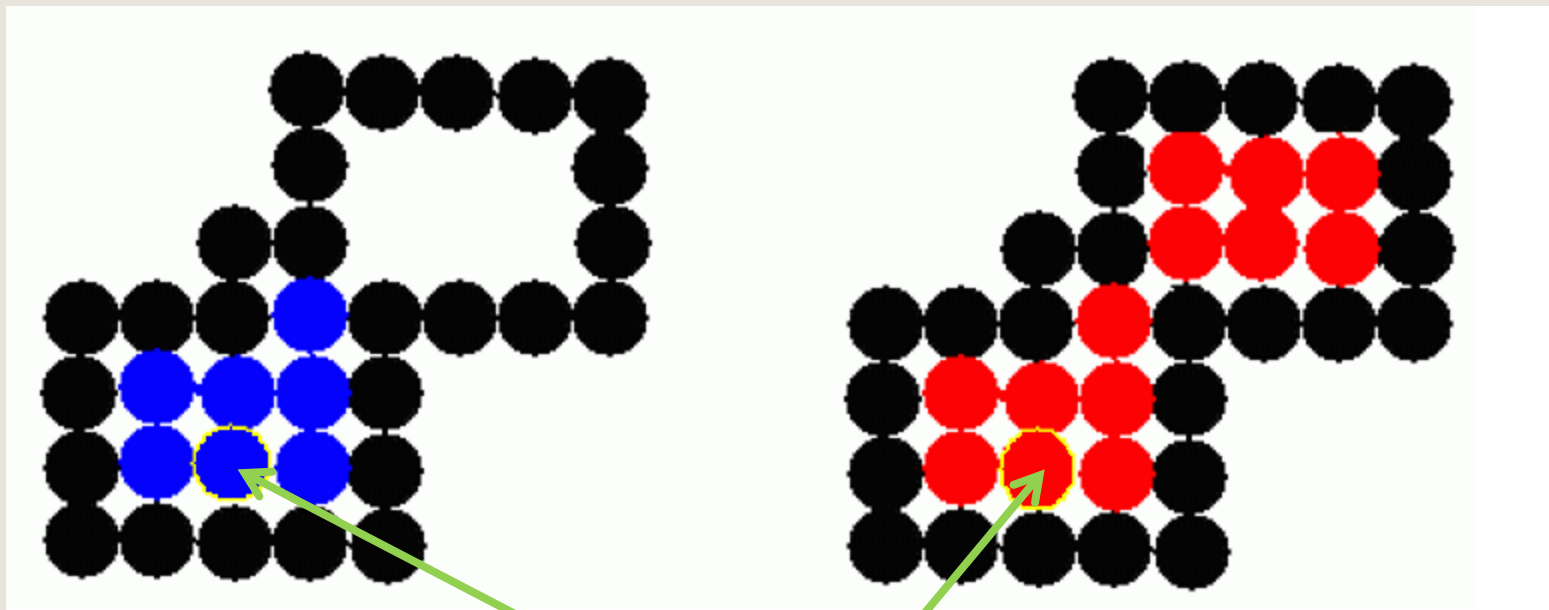
- 4-连通区域
- 8-连通区域



# 区域填充

- 区域的表示方法
- 区域的分类
- 区域填充算法

**概念：**区域填充是先将区域内的一个象素点（种子点）赋予指定颜色，然后将该颜色扩展到整个区域的过程。





# 区域填充

- 区域的表示方法
- 区域的分类
- 区域填充算法

## ➤ 边界填充算法

□ 边界表示法中，由于边界由特殊颜色指定，填充算法从种子点出发，逐个像素地向外处理，直到遇到边界颜色为止，这种方法称为边界填充算法（Boundary-fill Algorithm）。

## 区域填充算法——边界填充算法

栈结构实现4-连通边界填充算法：

种子像素入栈；当栈非空时重复执行如下三步操作：

(a)栈顶像素出栈；

(b)将出栈像素置成填充色；

(c)检查出栈像素的4-邻接点，若其中某个像素点不是边界色且未置成填充色，则把该像素入栈。

## 区域填充算法——边界填充算法

栈结构实现8-连通边界填充算法：

种子像素入栈；当栈非空时重复执行如下三步操作：

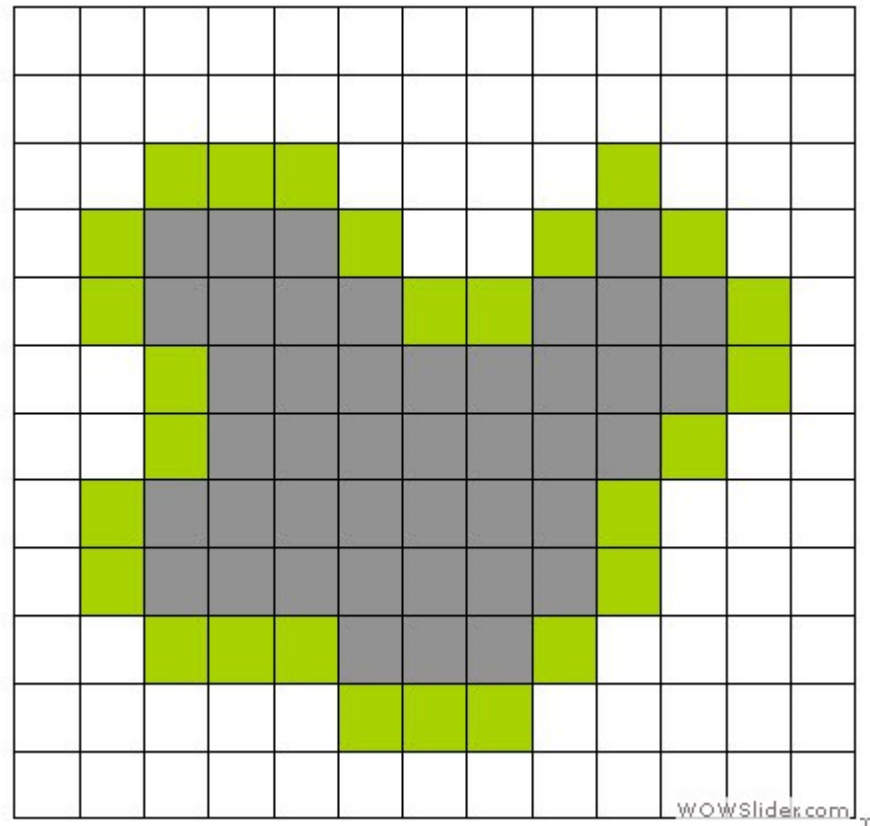
(a)栈顶像素出栈；

(b)将出栈像素置成填充色；

(c)检查出栈像素的8-邻接点，若其中某个像素点不是边界色且未置成填充色，则把该像素入栈。

# 区域填充算法——边界填充算法

实例：



# 第3章 基本图形生成算法

## 主要内容

- 直线段的生成算法
- 圆弧的生成算法
- 多边形扫描转换
- 区域填充算法
- 反走样技术

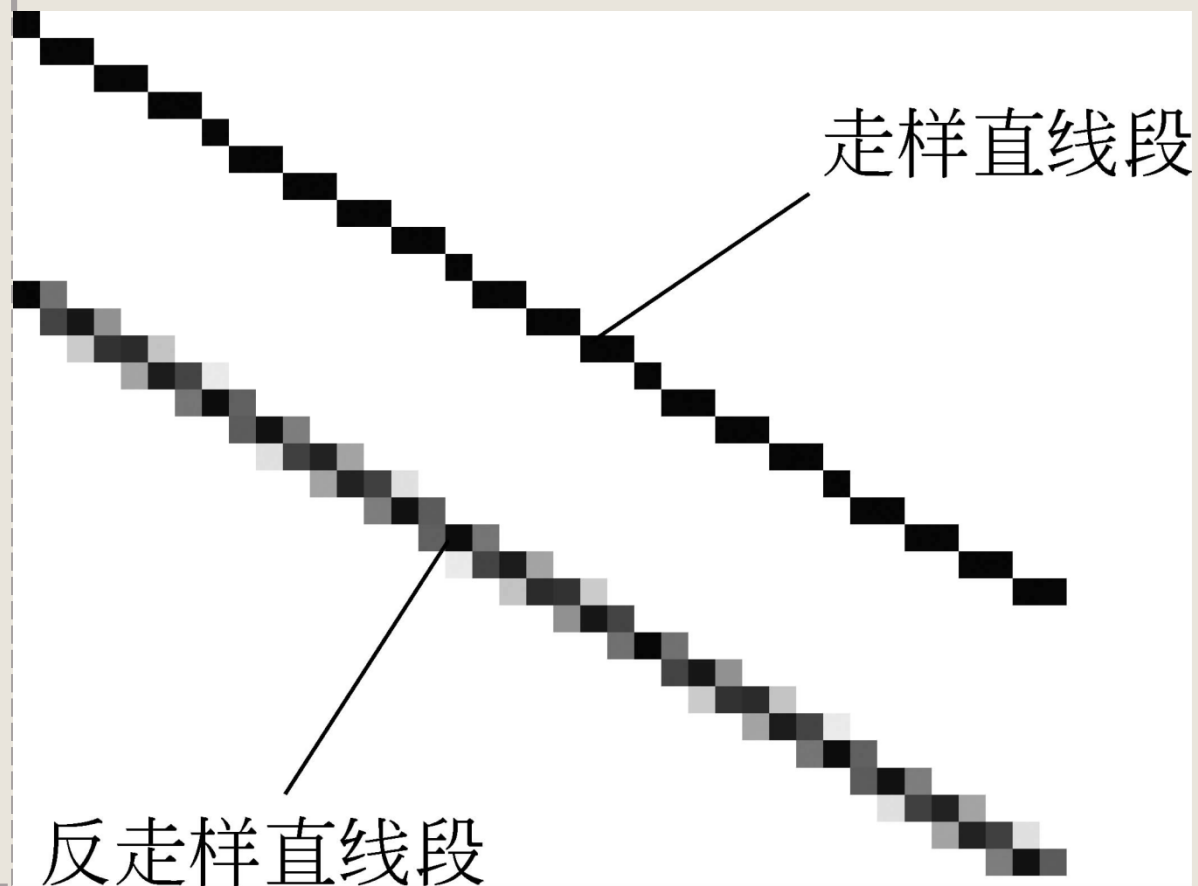
**走样：**在光栅图形显示器上绘制的非水平且非垂直的直线或多边形边界，会呈现**锯齿状**。

**反走样：**用于减轻或消除走样现象的技术称为反走样。

# 第3章 基本图形生成算法

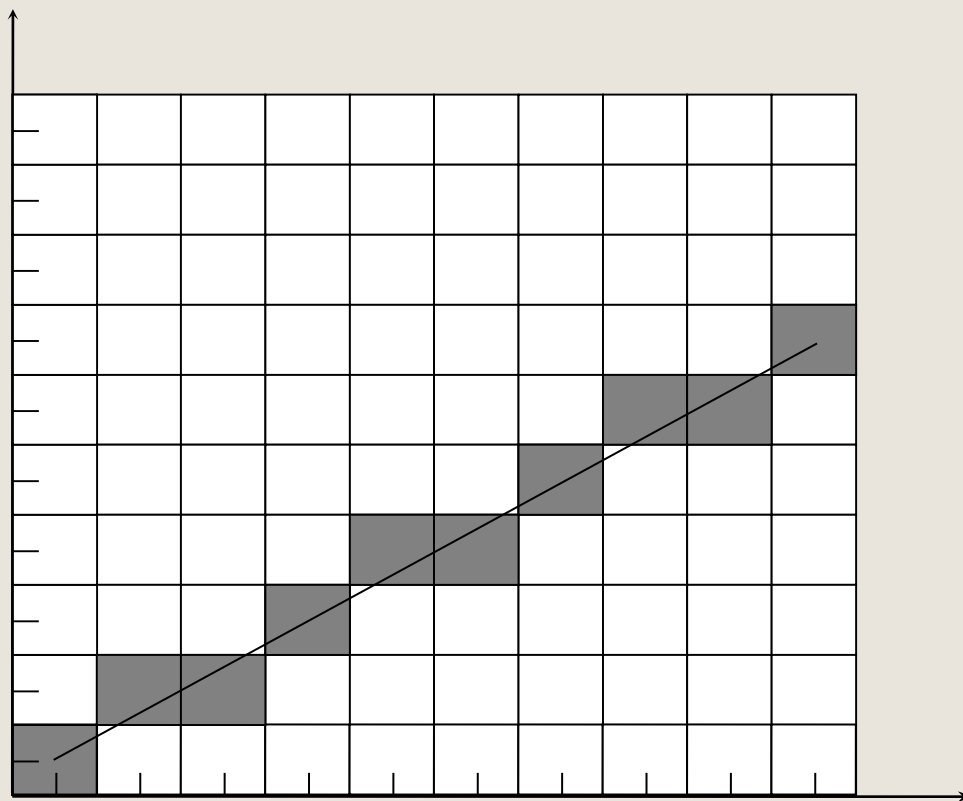
## 主要内容

- 直线段的生成算法
- 圆弧的生成算法
- 多边形扫描转换
- 区域填充算法
- 反走样技术



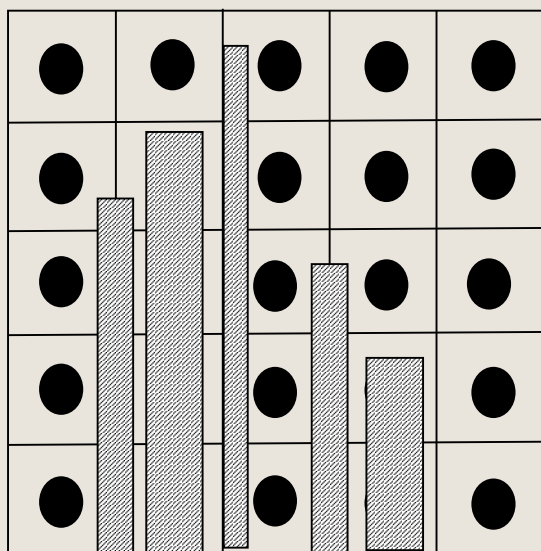
# 走样现象

## 1. 产生阶梯状的边界

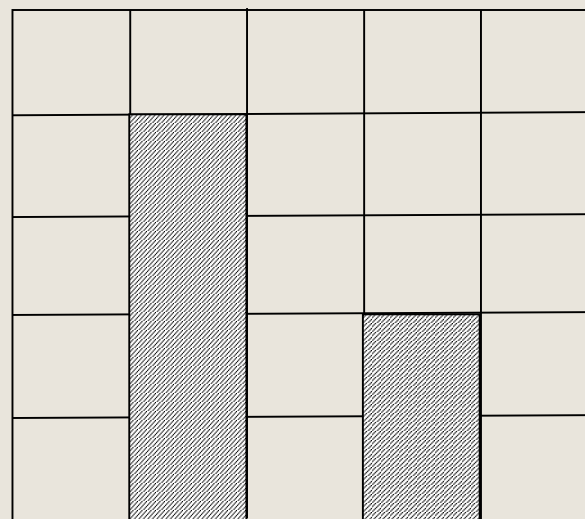


# 走样现象

## 2. 图形的细节失真



(a) 待显示的细小图形

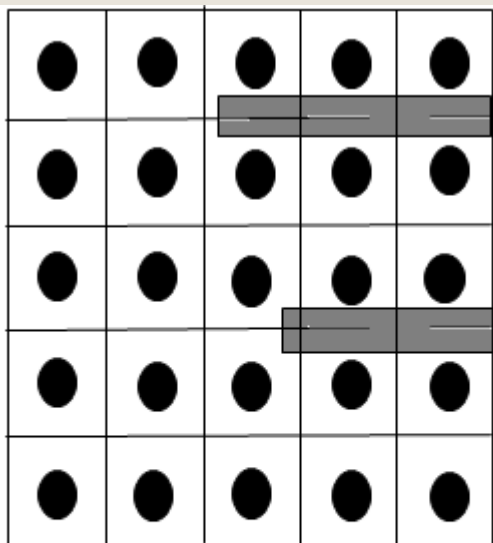


(b) 显示结果

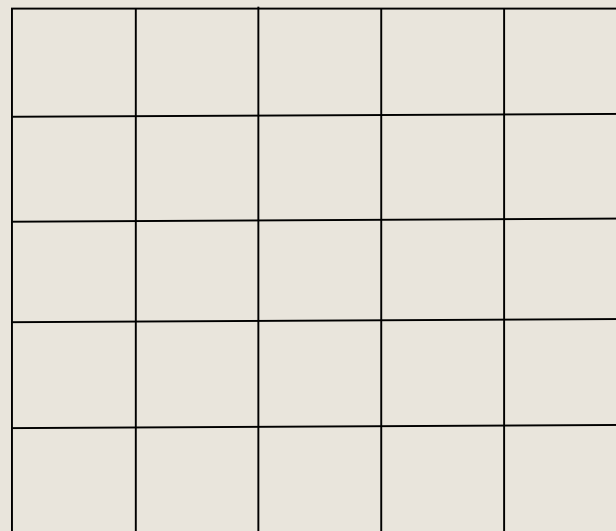


# 走样现象

## 3. 狭小图形遗失等



(a) 待显示的狭小矩形

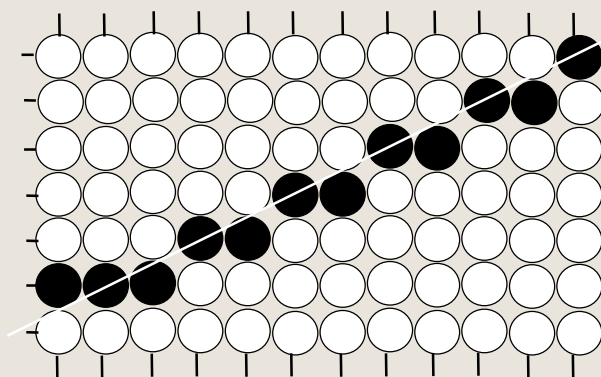
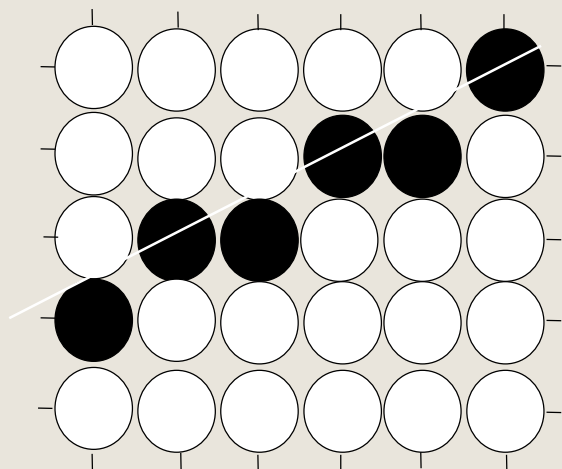


(b) 显示结果

# 反走样方法

1. 提高硬件的分辨率
2. 过取样
3. 区域采样

➤ 锯齿的宽度减小，所以显示出的直线段看起来就平直光滑了一些。



# 反走样方法

1. 提高硬件的分辨率

2. 过取样

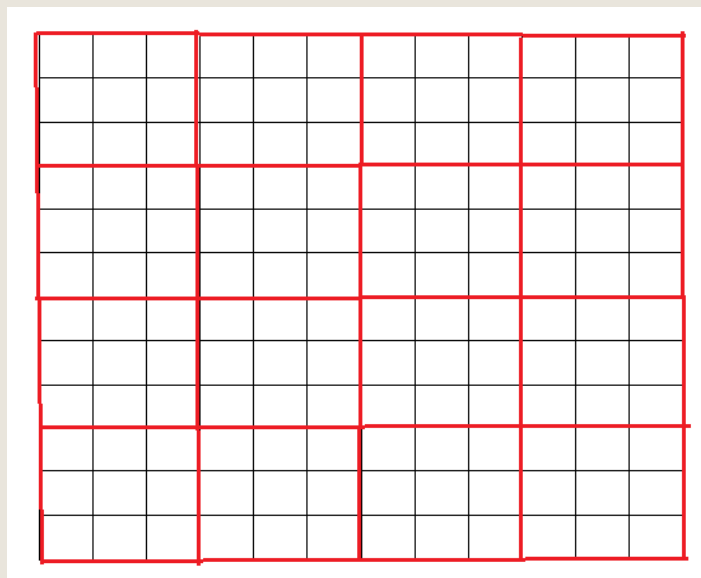
3. 区域采样

- 方法简单，但代价非常大。
- 显示器的水平、竖直分辨率各提高一倍，帧缓存容量则增加到原来的4倍，而扫描转换同样大小的图要花4倍时间。
- 只能减轻而不能消除锯齿问题。

# 反走样方法

1. 提高硬件的分辨率
2. 过取样
3. 区域采样

✓ 在高于显示分辨率的较高分辨率下用点取样方法计算，然后对几个像素的属性进行平均得到较低分辨率下的像素属性——高分辨率计算，低分辨率显示。

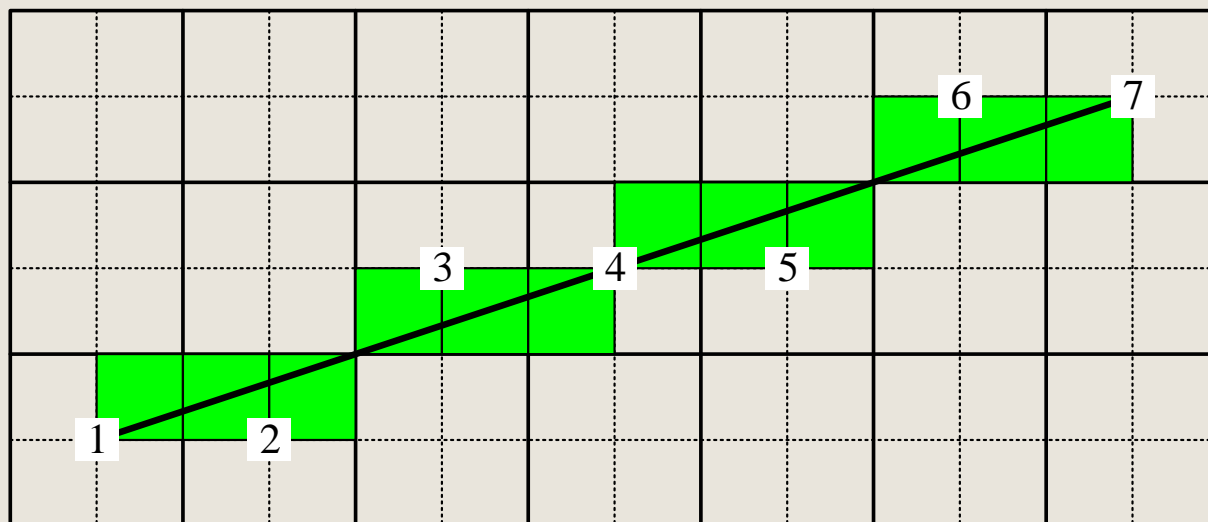


**红色像素为真实像素**

# 反走样方法

1. 提高硬件的分辨率
2. 过取样
3. 区域采样

➤ 把每个像素分为四个子像素，扫描转换算法求得各子像素的灰度值，然后对四像素的灰度值**简单平均**，作为该像素的灰度值。



# 反走样方法

1. 提高硬件的分辨率
2. 过取样
3. 区域采样

**基于加权模板的过取样：**接近像素中心的子像素赋予较大的权值，对所有子像素的亮度进行**加权平均**。

1	2	1
2	4	2
1	2	1

(a)

1	1	1
1	2	1
1	1	1

(b)

0	1	0
1	4	1
0	1	0

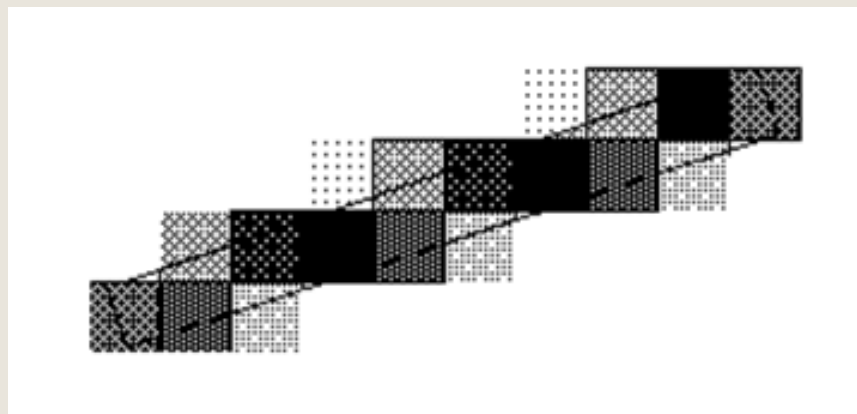
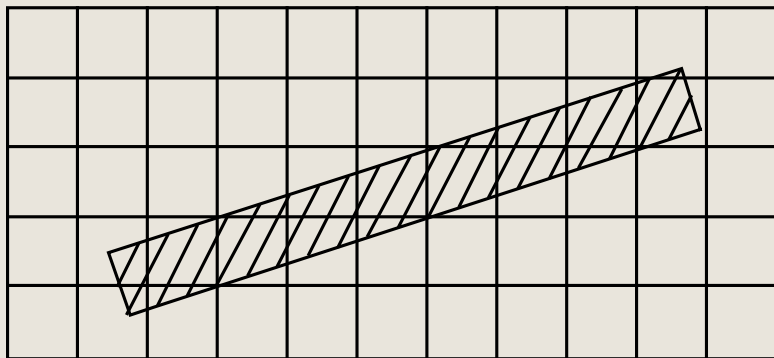
(c)

常用的加权模板

# 反走样方法

1. 提高硬件的分辨率
2. 过取样
3. 区域采样

- 将直线段看作具有一定宽度的狭长矩形；
- 当直线段与某像素有交时,求出两者相交区域的面积；
- 根据相交区域的面积，确定该像素的亮度值。

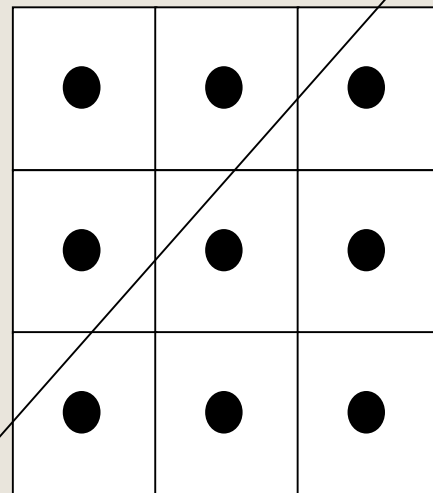


# 反走样方法

1. 提高硬件的分辨率
2. 过取样
3. 区域采样

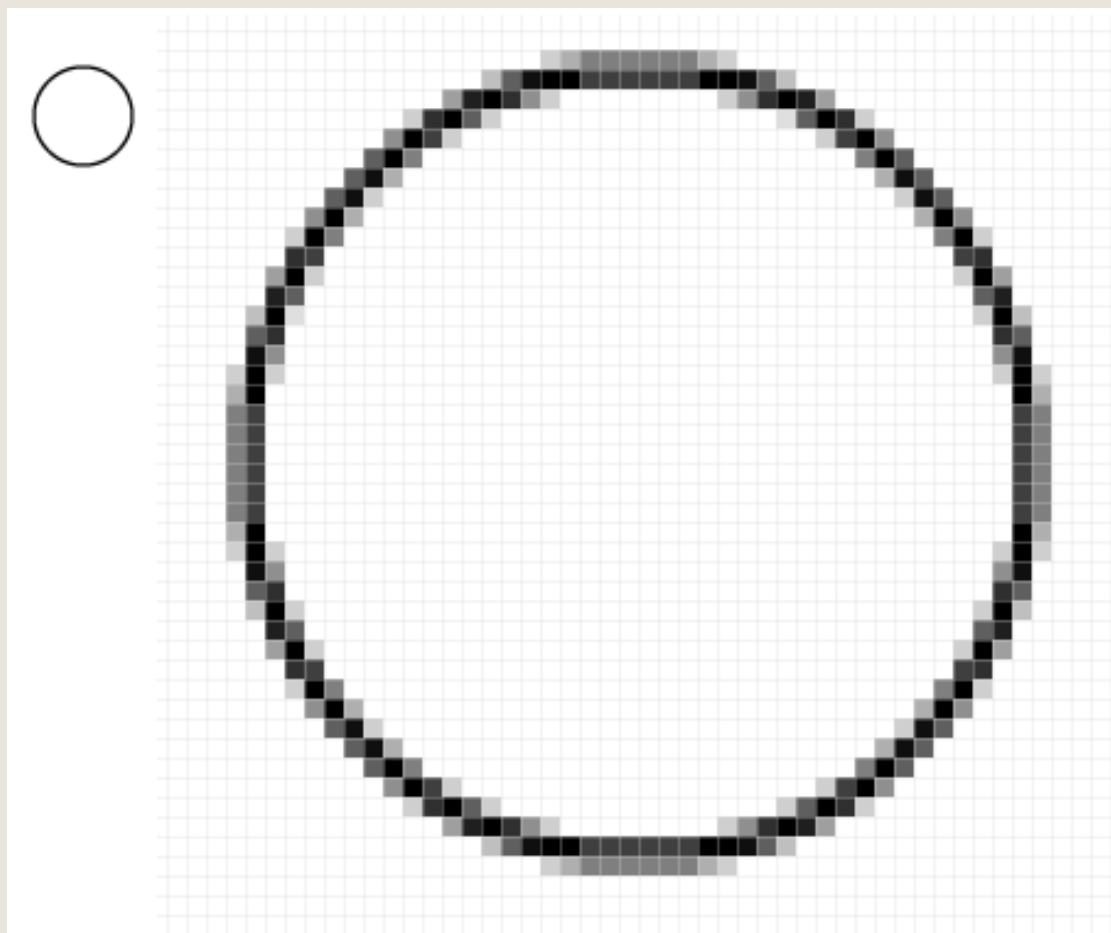
## 计算像素与直线相交的面积

1. 将屏幕一个像素均分成  $n$  个子像素；
2. 计算中心点落在直线段内的子像素的个数  $k$ ；
3. 近似值  $k/n$ 。



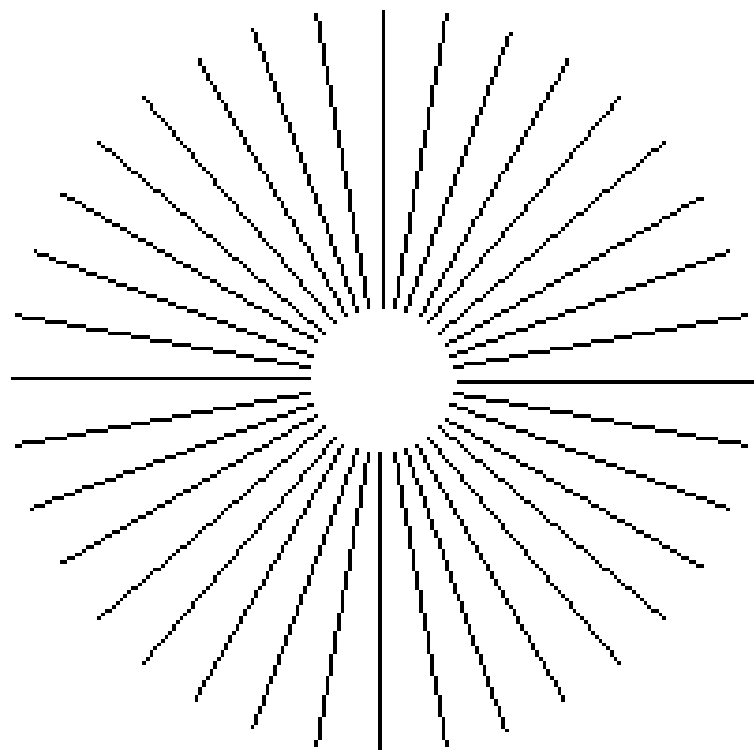


# 反走样结果

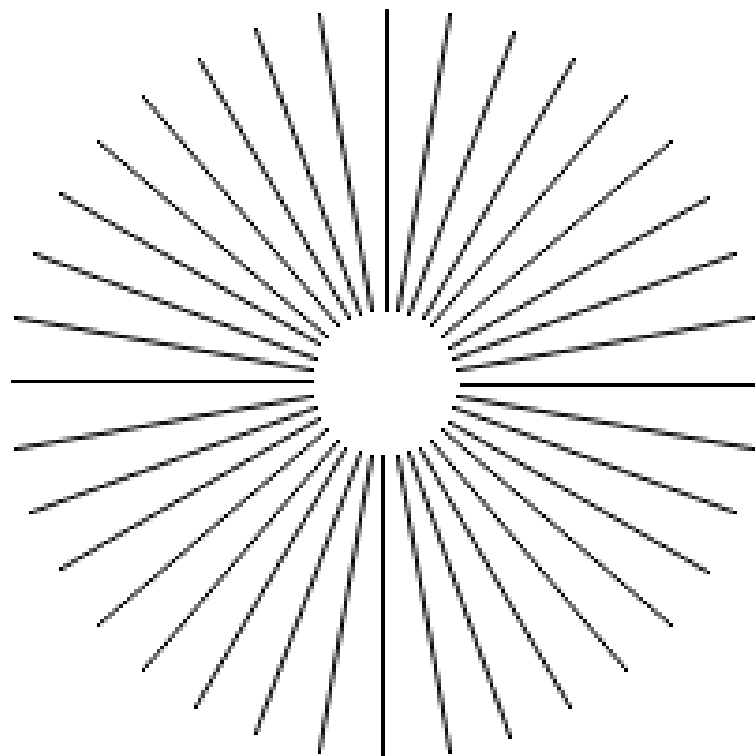


# 反走样结果

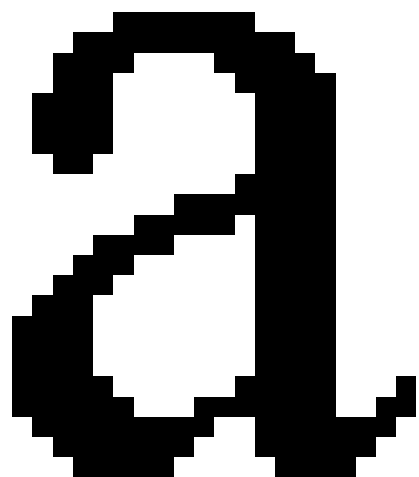
Normal



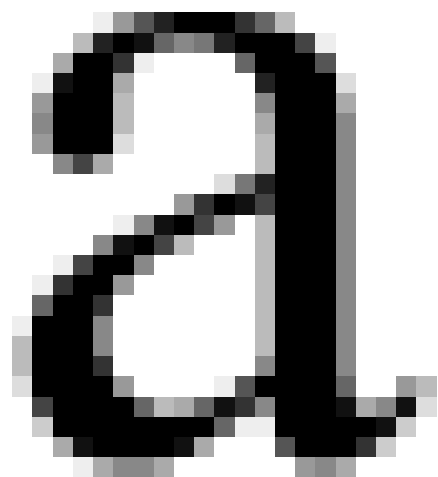
Anti-aliased



## 反走样结果

A large, black, pixelated lowercase letter 'a' on a white background. The edges are jagged and blocky, characteristic of low-resolution digital art without anti-aliasing.

Alias

A large, black, pixelated lowercase letter 'a' on a white background. The edges are smooth and blurred, showing the result of an anti-aliasing filter applied to the original pixelated image.

Anti-aliased