第3讲：基本图形生成算法

吴文明
计算机与信息学院

导论

直线

- (000, 000)
- (100, 100)

矩形

- (000, 000)
- (100, 000)
- (100, 100)
- (000, 100)

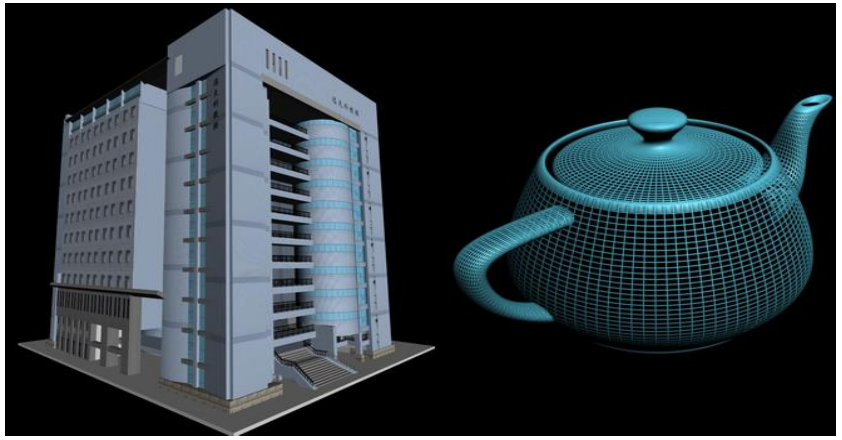
怪物

- (XXX, XXX)
- (XXX, XXX)
- ……

导论

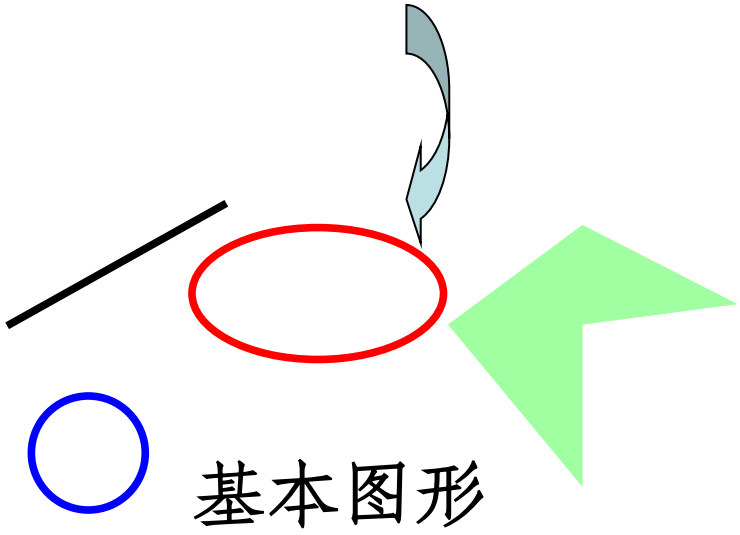
模型千变万化：

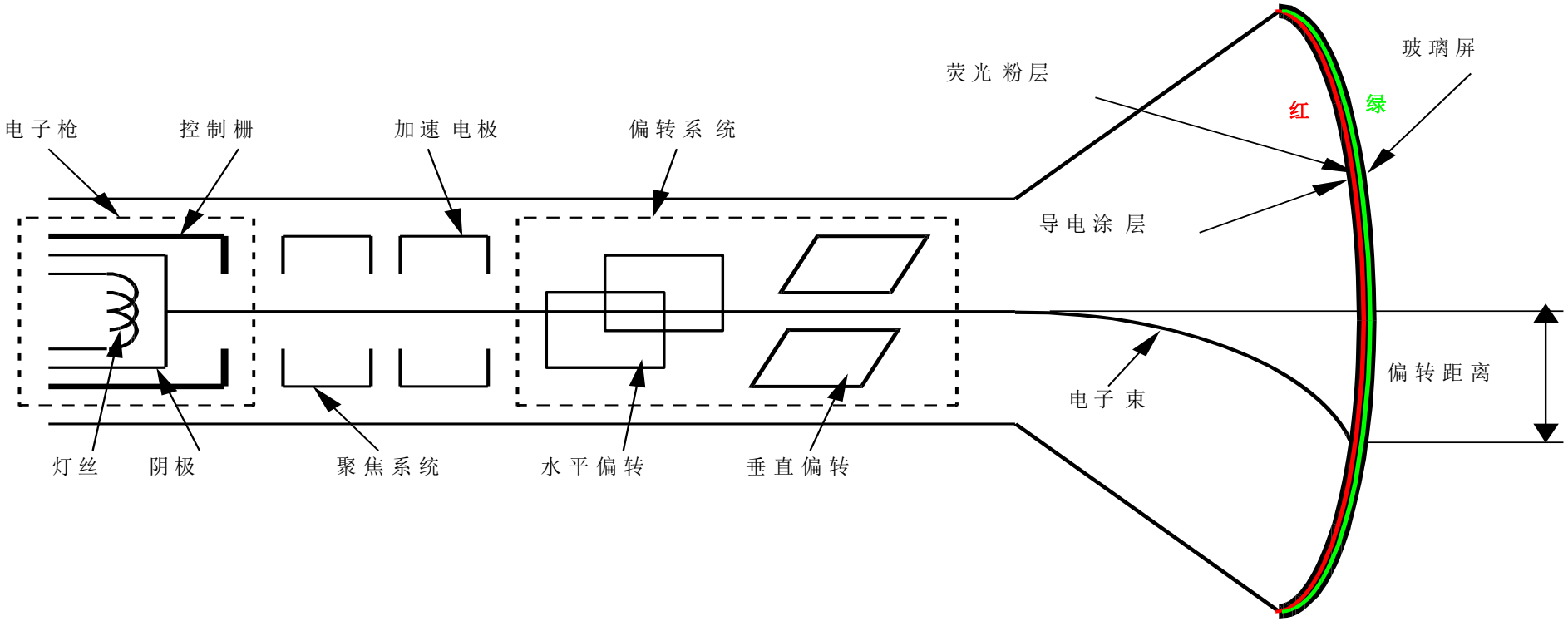
- 不可能为每个模型的设置一个绘制算法
- 也不需要为每个模型的设置一个绘制算法



任何模型都是由基本模型组成：

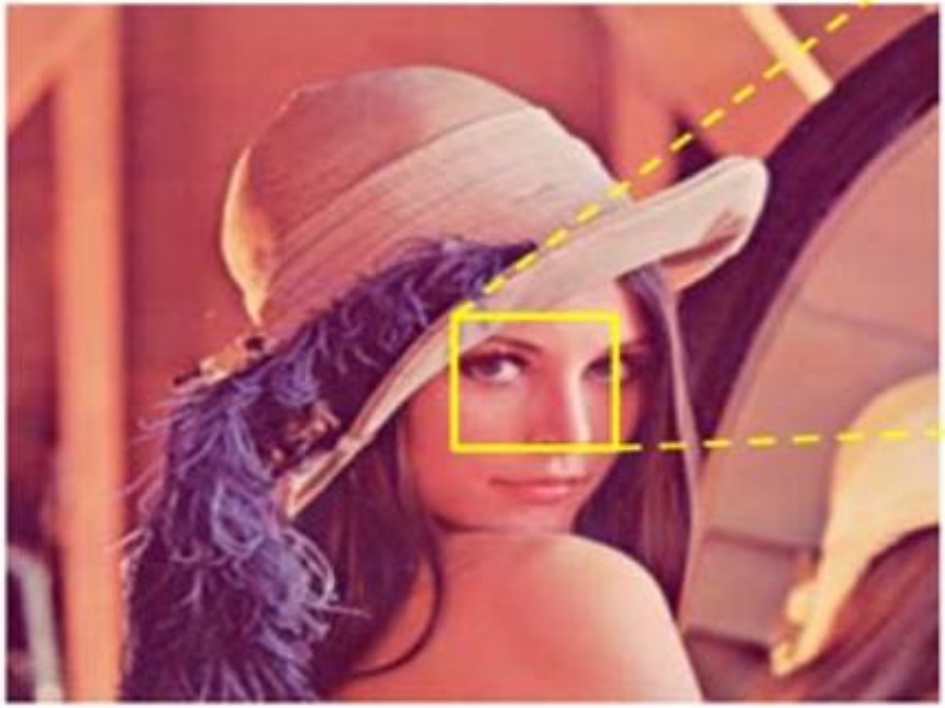
- 基本模型：直线、圆、椭圆、多边形
- 只需讨论基本模型

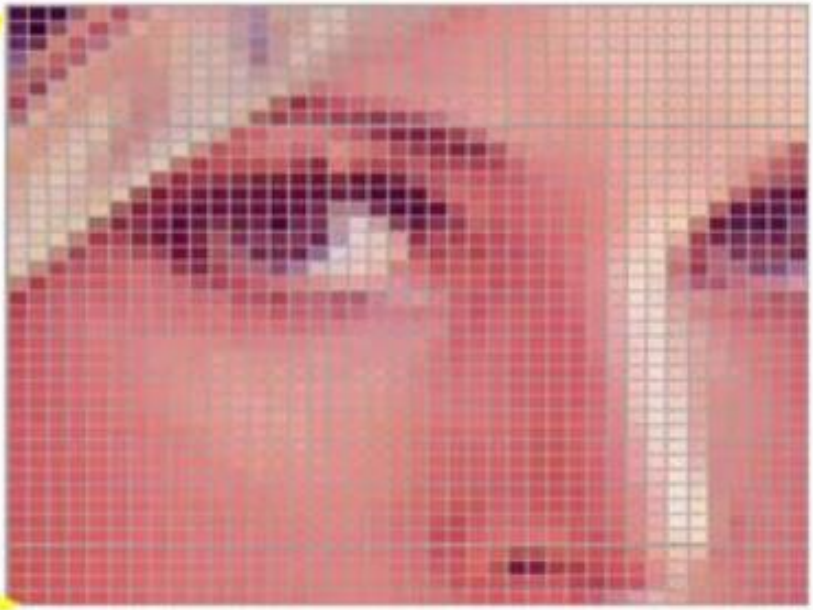




阴极射线管（CRT）剖面图（穿透式）

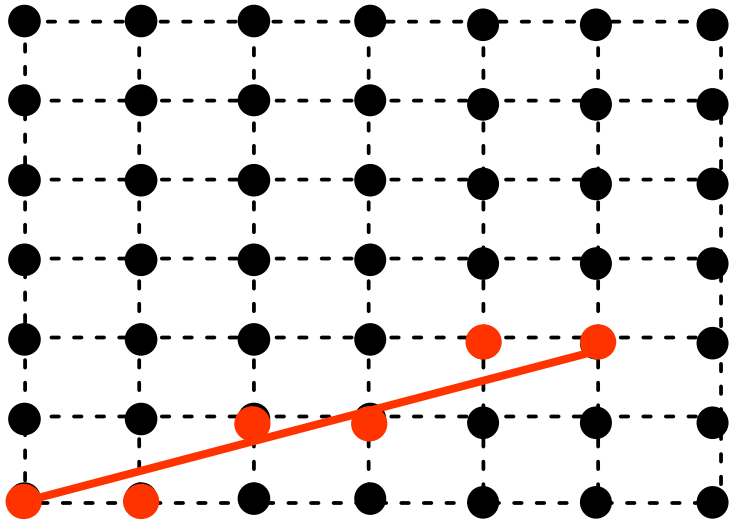
导论





光栅显示器：

- 二维点阵（像素）
- 最基本的绘图函数：
  DrawPixel(x, y)



分辨率：1024*768

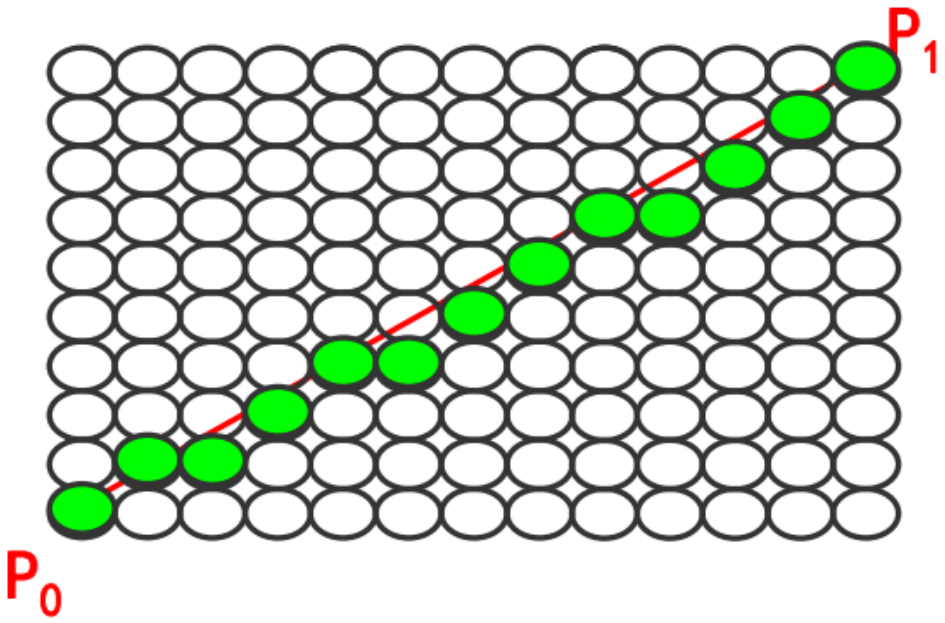
- 每行有1024个点；
- 每列有768个点；

讲义大纲

- 直线的扫描转换
- 多边形的扫描转换
- 多边形的区域填充
- 反走样
- 小结

直线的扫描转换

直线的扫描转换

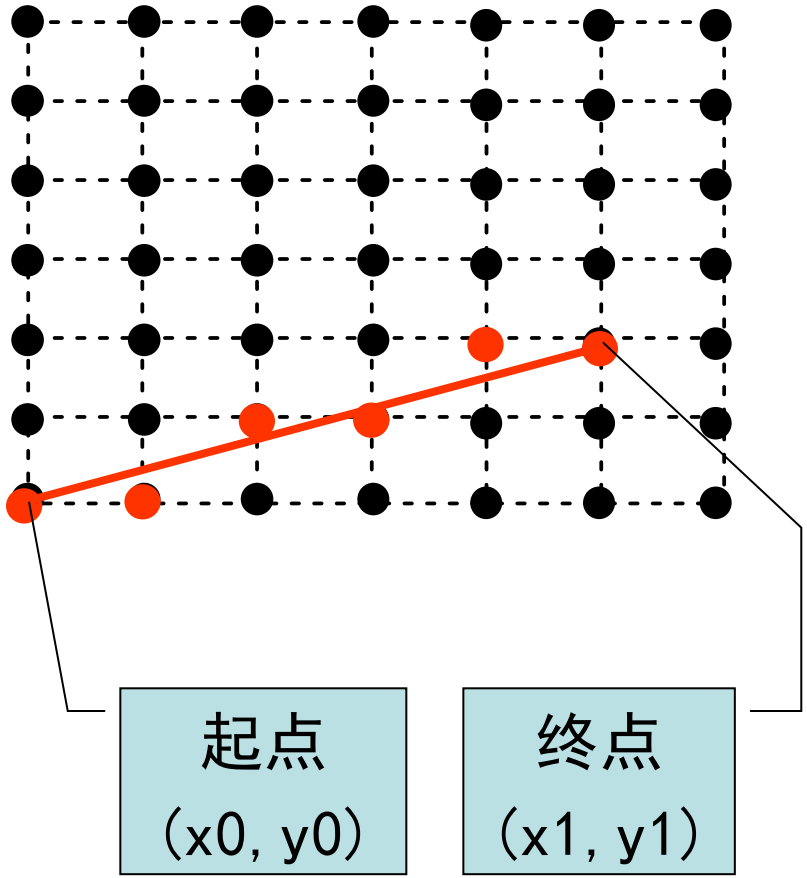


直线的扫描转换

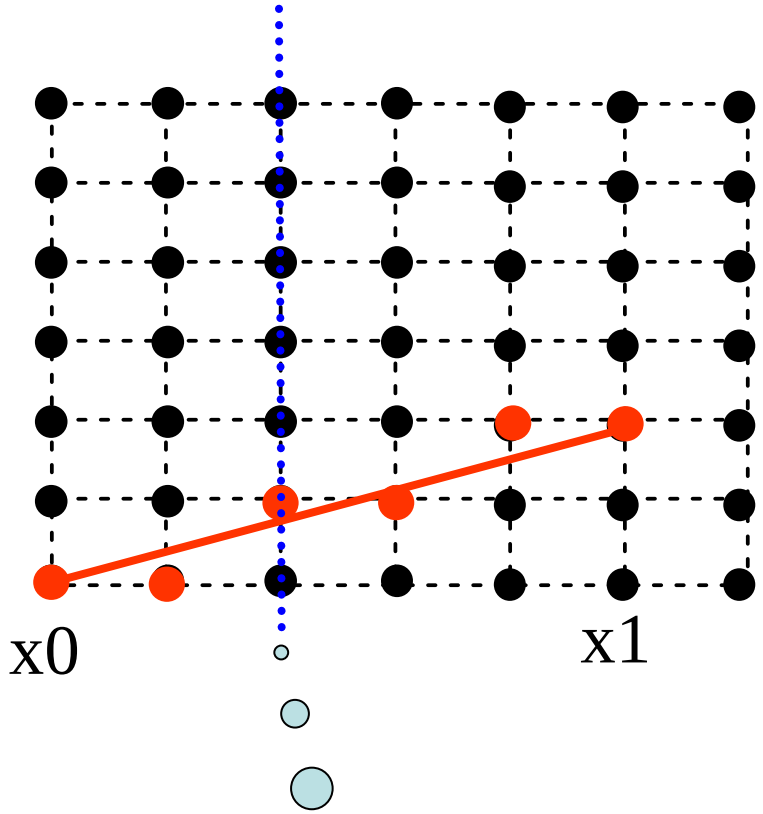
问题：对给定起点和终点的直线，在光栅显示器上显示直线

问题：给定直线的起点和终点，在光栅显示器的二维点阵中确定一组点最佳逼近所给直线！

- 理论上，线段由无数个点组成；
- 光栅显示器：用有限个点表示；



直线的扫描转换



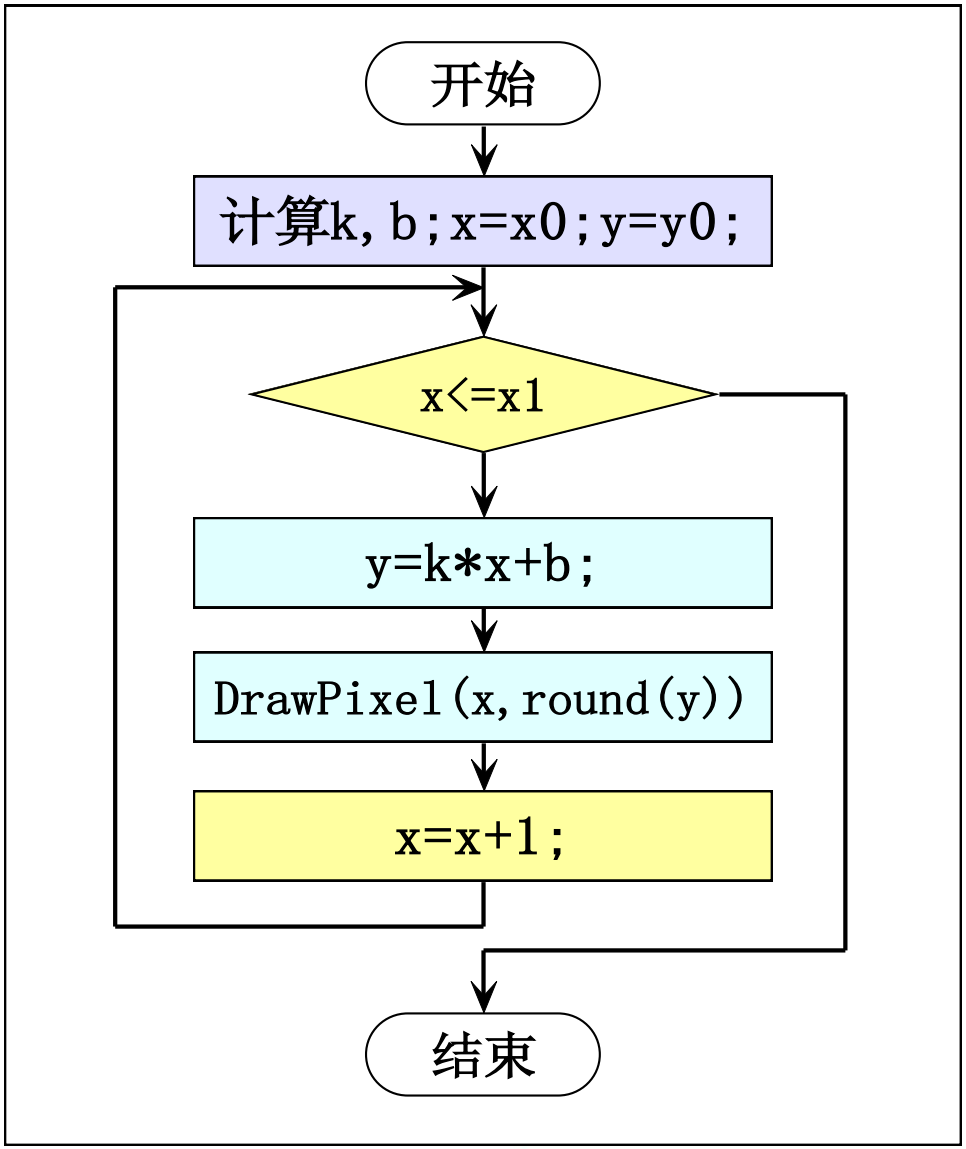
在所有x(x0≤x≤x1)列上，求点(x,y)，并绘制点(x,y)

- 求x：已知量
- 求y：y=kx+b
- 绘制：DrawPixel(x,y)

对任意x0≤x≤x1, 在第x列上存在一个点

循环结构

直线的扫描转换

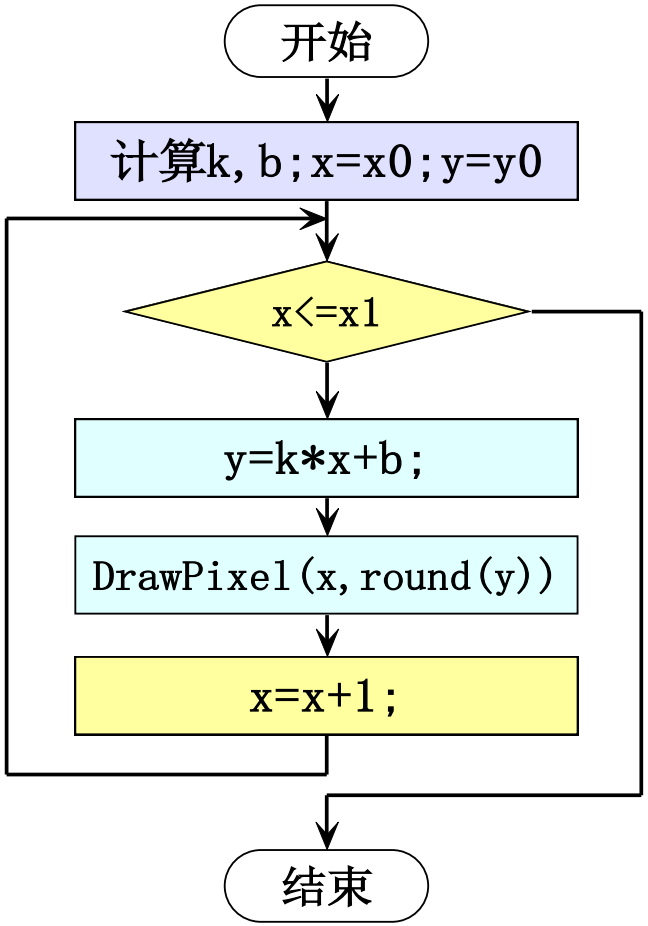


在所有x(x0≤x≤x1)列上，求点(x,y)，并绘制点(x,y)

- 求x：已知量
- 求y：y=kx+b
- 绘制：DrawPixel(x,y)

循环结构

直线的扫描转换



- 浮点数乘法
- 浮点数加法
- 整数加法
- 四舍五入

```
void DrawLine(int x0, int y0,
              int x1, int y1)
{
    int   x;
    float k, b, y;
    k = float(y1-y0)/
        float(x1-x0);
    b = float(x1*y0-x0y1)/
        float(x1-x0);
    x = x0;y=y0
    while ( x <= x1)
    {
        y = k*x + b;
        DrawPixel(x,round(y));
        x++;
    }
    return;
}
```

数值微分法

绘制直线的本质是确定点序列：

(x_0, y_0)

…

(x_i, y_i)

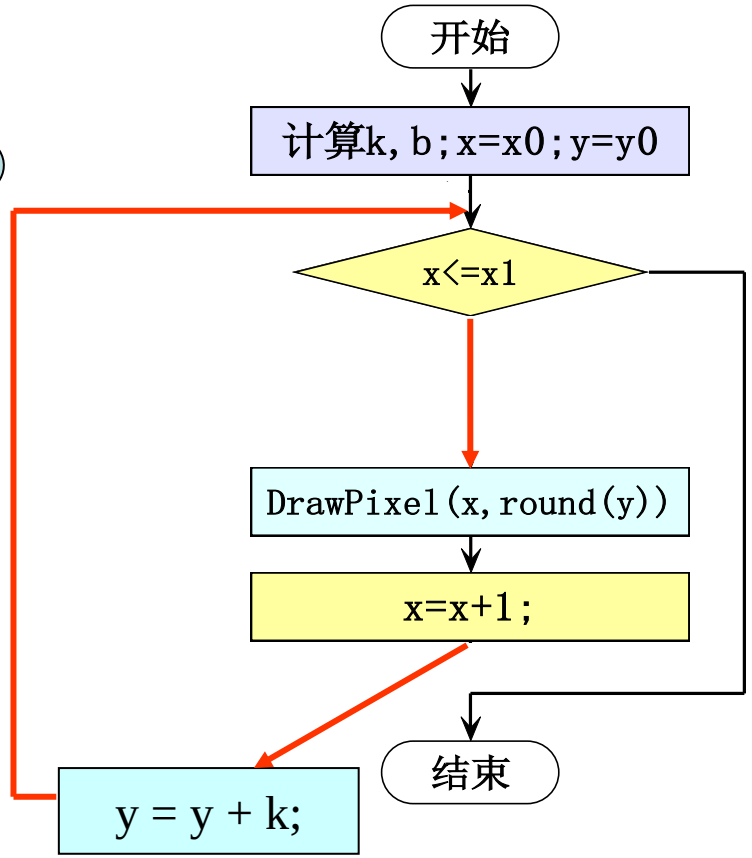
(x_{i+1}, y_{i+1})。

…

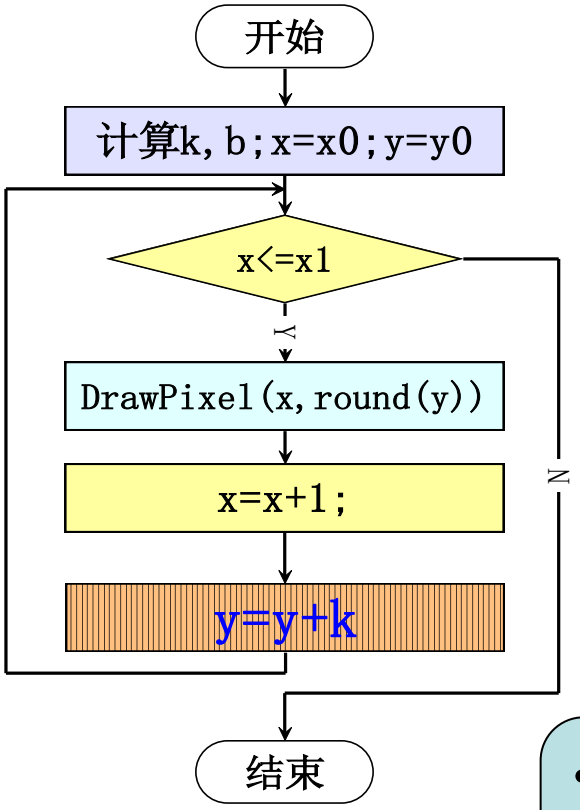
(x_n, y_n)

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

$$
\begin{aligned}
y_{i+1} &= kx_{i+1}+b \\
&= k(x_0+i+1)+b \\
&= k(x_0+i)+b+k \\
&= y_i+k;
\end{aligned}
$$



数值微分法




```
void DrawLine(int x0, int y0,
              int x1, int y1)
{
    int   x;
    float k, b, y;
    k = float(y1-y0)/
        float(x1-x0);
    b = float(x1*y0-x0y1)/
        float(x1-x0);
    x = x0;y=y0
    while ( x <= x1)
    {
        //y = k*x + b;
        DrawPixel(x,round(y));
        x++;
        y+=k;//增量运算
    }
    return;
}
```

- 浮点数加法
- 整数加法
- 四舍五入

数值微分法

改进多少？

- 浮点数乘法
- 浮点数加法
- 整数加法
- 四舍五入

```
d DrawLine(int x0, int y0,
           int x1, int y1)
{
    int   x;
    loat k, b, y;
     = float(y1-y0)/
       float(x1-x0);
    b = float(x1*y0-x0y1)/
       float(x1-x0);
    x = x0;y=y0
    while ( x <= x1)
    {
       y = k*x + b;
       DrawPixel(x,round(y));
       x++;
    }
    return;
}
```

增量运算！

- 浮点数加法
- 整数加法
- 四舍五入

```
void DrawLine(int x0, int y
              int x1, int y
{
    int   x;
    float k, b, y;
    k = float(y1-y0)/
       float(x1-x0);
    b = float(x1*y0-x0y1)/
       float(x1-x0);
    = x0;y=y0
    ile ( x <= x1)
    {
       //y = k*x + b;
       DrawPixel(x,round(y));
       x++;
       y+=k;//增量运算
    }
    return;
}
```

数值微分法

直线L1:

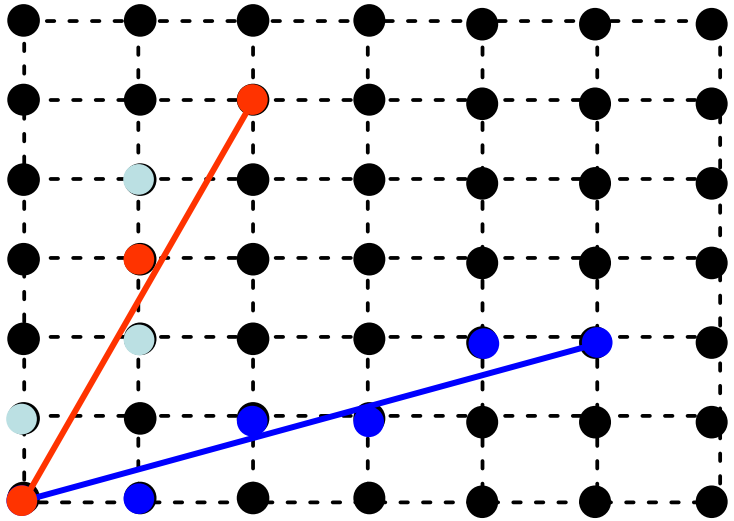
(0, 0)

(5, 2)

直线L2:

(0, 0)

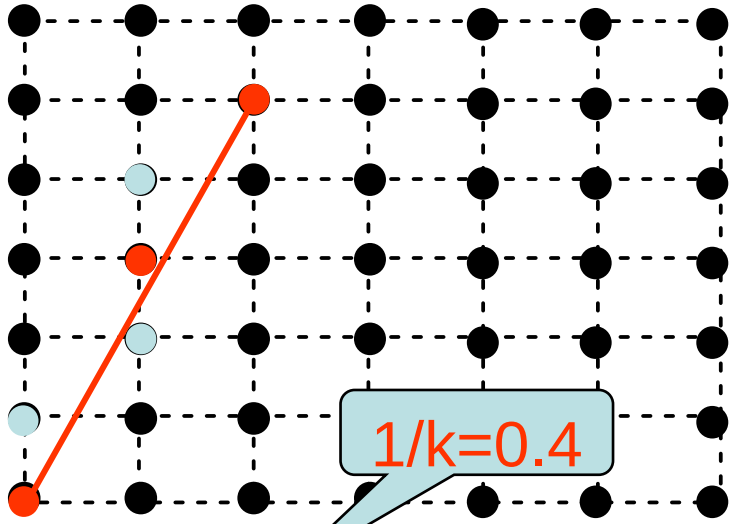
(2, 5)



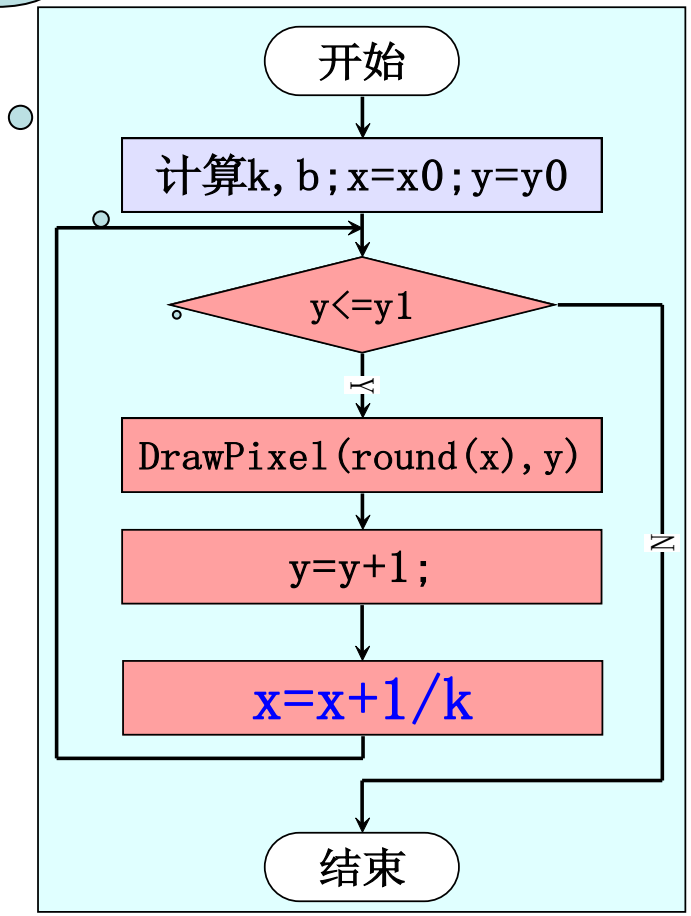
为什么要确定最大位移方向？

数值微分法

x与y对调



y	x	round (x)	y	x	round (x)
0	0	0	3	1.2	1
1	0.4	0	4	1.6	2
2	0.8	1	5	2.0	2



数值微分法

如何选择

x变化比y快
|k|<=1

```
void DrawLine(int x0, int y0,
              int x1, int y1)
{
    int   x;
    float k, b, y;
    k = float(y1-y0)/
        float(x1-x0);
    b = float(x1*y0-x0y1)/
        float(x1-x0);
    x = x0;y=y0
    while ( x <= x1)
    {
        //y = k*x + b;
        DrawPixel(x,round(y));
        x++;
        y+=k;//新增代码
    }
    return;
}
```

y变化比x快
|k|>1

```
void DrawLine(int x0, int y0,
              int x1, int y1)
{
    int   y;//int x;
    float k, b, x;//float y
    k = float(y1-y0)/
        float(x1-x0);
    b = float(x1*y0-x0y1)/
        float(x1-x0);
    x = x0;y=y0
    while ( y <= y1)//x<=x1
    {
        //DrawPixel(x,round(y));
        DrawPixel(round(x),y);
        y++;      //x++;
        x+=1/k;   //y+=k;
    }
    return;
}
```

最后代码

```
void DrawLine(int x0, int y0,
              int x1, int y1)
{
    float k, b;
    k = float(y1-y0)/
        float(x1-x0);
    b = float(x1*y0-x0y1)/
        float(x1-x0);
    if ( fabs(k) <= 1)
    {
        int x; float y;
        x = x0;y=y0
        while ( x <= x1)
        {
           DrawPixel(x,round(y));
           x++;
           y+=k;
        }
    }
    else
    {
        float x; int y;
        x = x0;y=y0
        while ( y <= y1)
        {
           DrawPixel(x,round(y));
           y++;
           x+=1/k;
        }
    }
    return;
}
```

教材中的代码

代码紧凑

```
void DDALine(int x0,int y0,int x1,int y1)
{
   int dx,dy, epsl,k;
   float x, y, xIncre,yIncre;
   dx = x1 - x0;  dy = y1 - y0;
   if ( abs(dx) > fabs(dy) ) epsl = abs(dx);
   else                      epsl = abs(dy);
   xIncre = (float)dx/(float)epsl;
   yIncre = (float)dy/(float)epsl;
   for ( k = 0; k <= epsl; k++)
   {
      DrawPixel(round(x), round(y));
      x += xIncre;       y += yIncre;
   }
   return;
}
```

DDA：总结

优点：

- 算法直观
- 易实现

不足：

- 涉及浮点数运算
- 不利于硬件实现

如何改进

优点：

- 算法直观
- 易实现

不足：

- 涉及浮点数运算。
- 不利于硬件实现

中点Bresenham算法

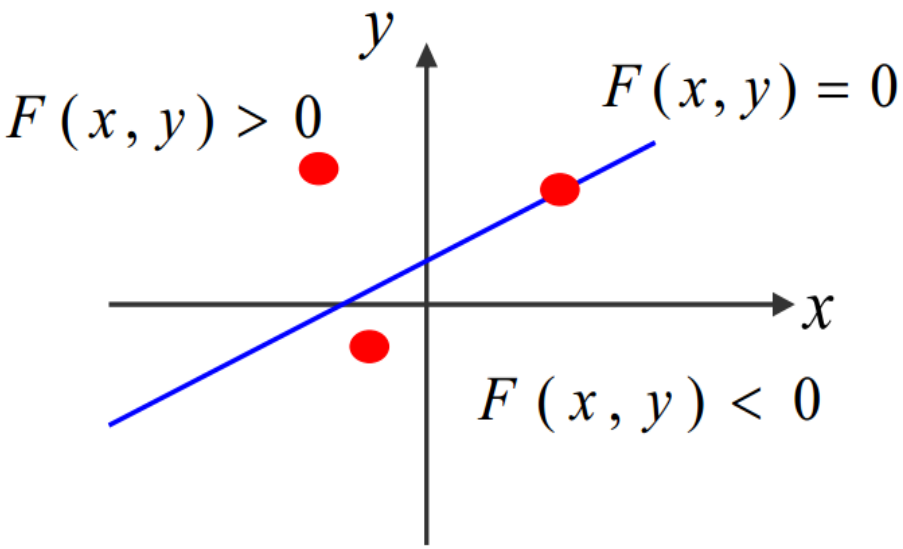
直线的一般式方程：

$$F(x, y) = 0$$

$$Ax + By + C = 0$$

其中：$A = -(\Delta y);\quad B = (\Delta x);\quad C = -B(\Delta x)$

中点Bresenham算法

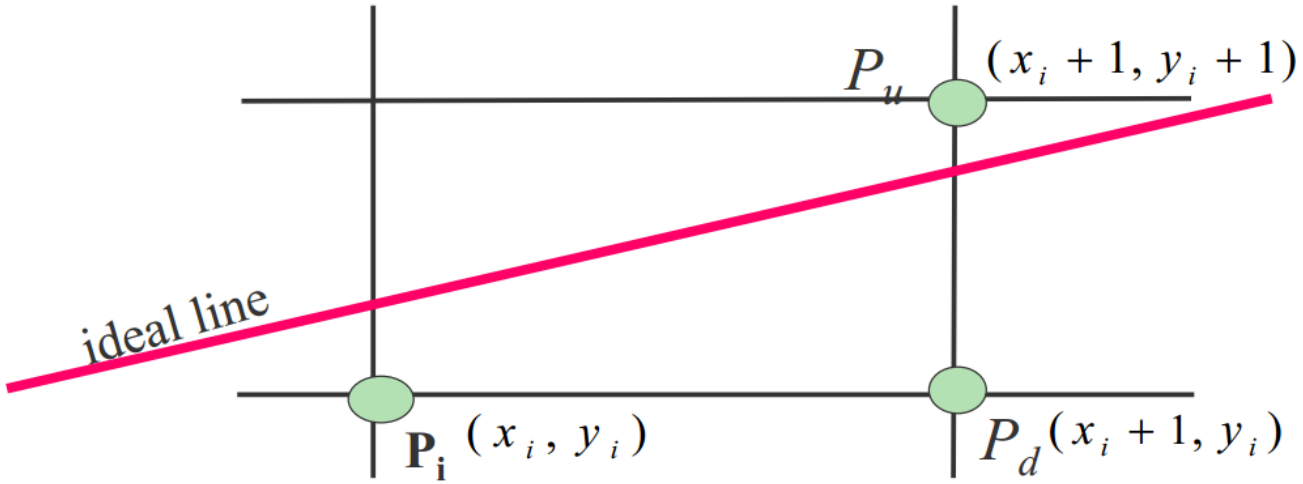


- 对于直线**上**的点：$F(x, y) = 0$
- 对于直线**上方**的点：$F(x, y) > 0$
- 对于直线**下方**的点：$F(x, y) < 0$

中点Bresenham算法

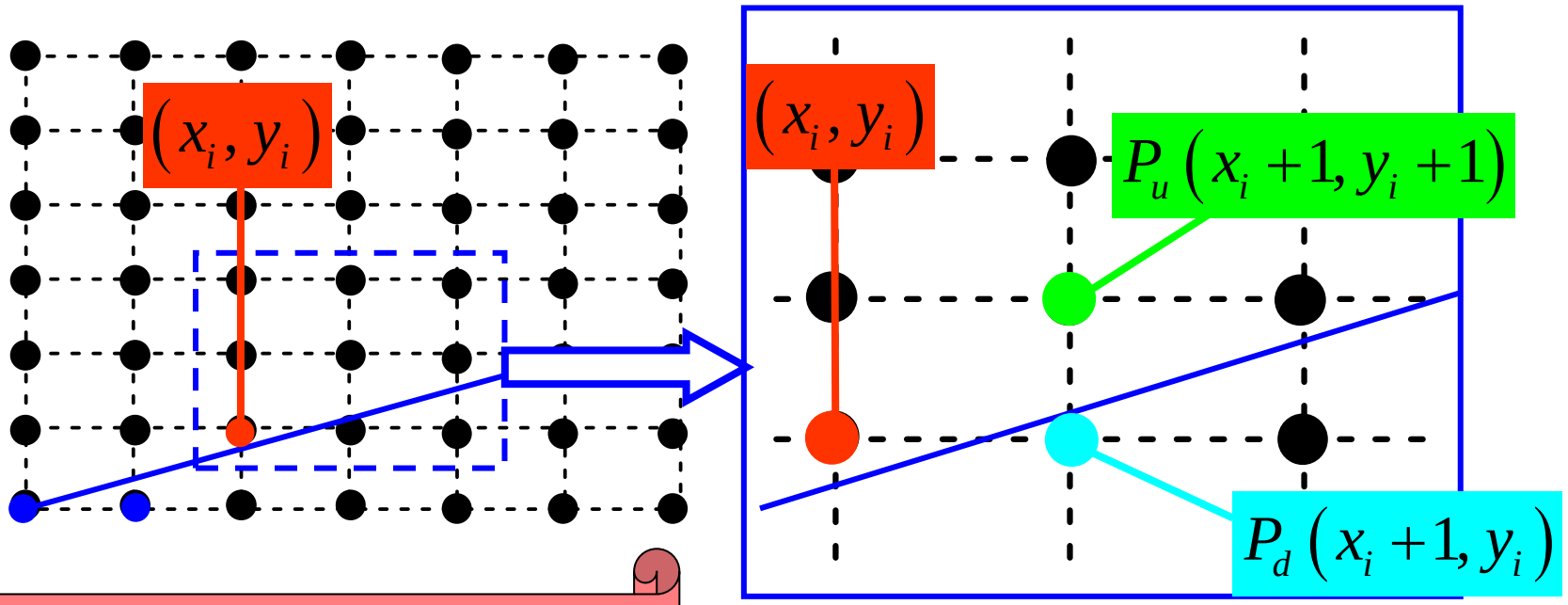
每次在最大位移方向上走一步，而另一个方向是走步还是不走步要取决于中点误差项的判断。

假定：0≤|k|≤1。因此，每次在x方向上加1，y方向上加1或不变需要判断。



中点Bresenham算法

$0 \le k \le 1$



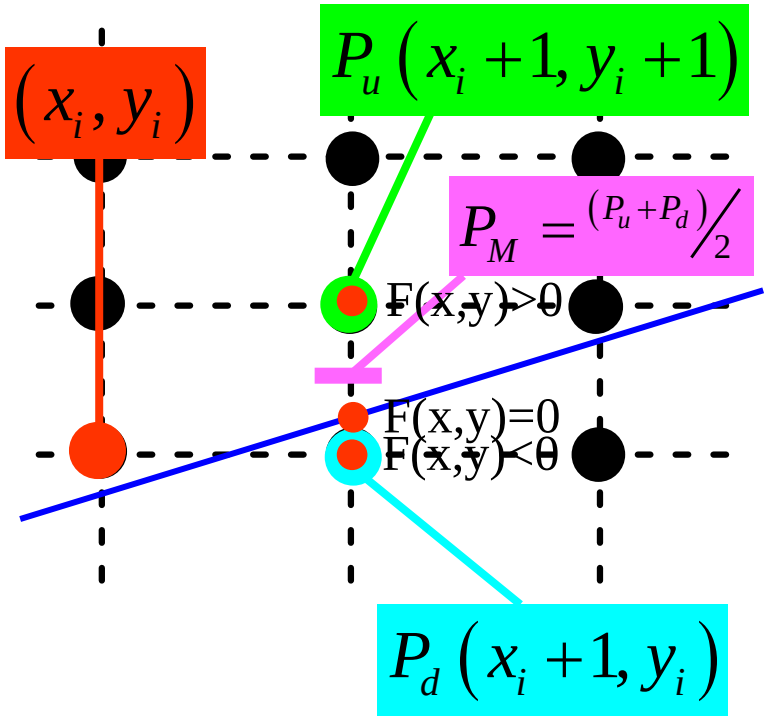
如何快速确定下一点（x_{i+1}, y_{i+1}）？

如何从Pu和Pd中进行选择？

为什么下一点必定是P_u或P_d?

中点Bresenham算法

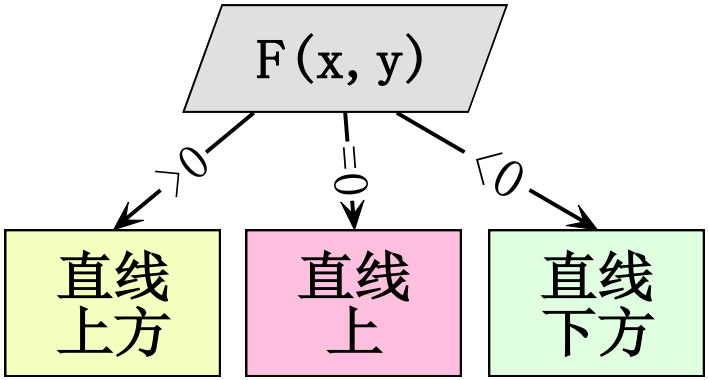
P_u和P_d的取舍



$$y = kx + b$$

$$\Rightarrow y - kx + b = 0$$

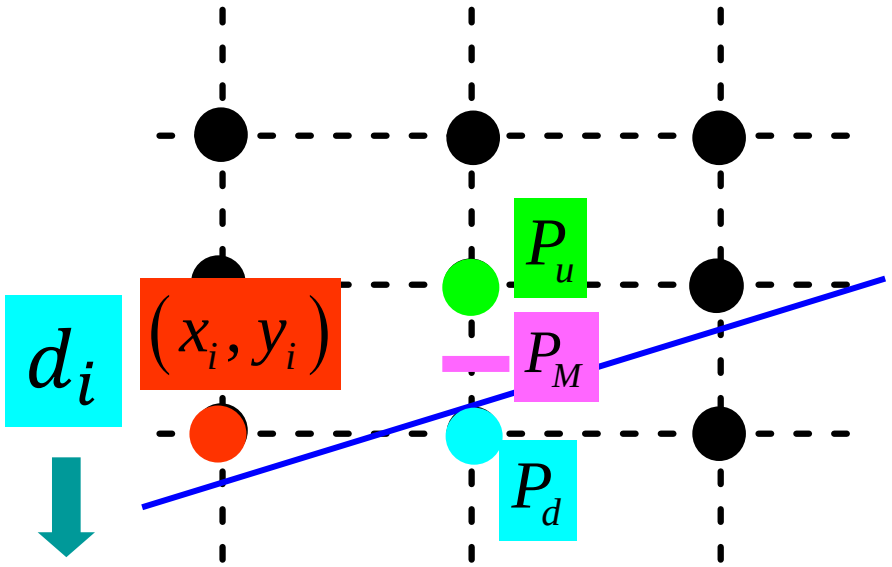
$$\Rightarrow F(x, y) = 0;$$



P_M在直线上方，取P_d

P_M在直线下方，取P_u

中点Bresenham算法



$$F(P_M) = F(x_i + 1, y_i + 0.5)$$
$$= (y_i + 0.5) - k \cdot (x_i + 1) - b$$

$$\begin{cases} d_i \geq 0 : \text{取} P_d \\ d_i < 0 : \text{取} P_u \end{cases}$$

```
void DrawLine(int x0,int y0,
              int x1,int y1 )
{
    float k, b;
    float d;
    int x, y;
    k =...;b=...;
    x = x0; y = y0;
    while ( x <= x1)
    {
        DrawPixel(x,y);
        d = (y+0.5)
            -k*(x+1)-b;
        if (d < 0) y = y +1;
        //else     y = y ;
        x++;
    }
    return;
}
```

浮点乘法
浮点加法

$d_i \rightarrow d_{i+1}$

中点Bresenham算法

判别式：

$$d = F(x_M, y_M) = F(x_i + 1, y_i + 0.5) = y_i + 0.5 - k(x_i + 1) - b$$

则有：

$$y = \begin{cases} y+1 & (d < 0) \\ y & (d \geq 0) \end{cases}$$

$$d_{i+1} = d_i + ?$$ 增量

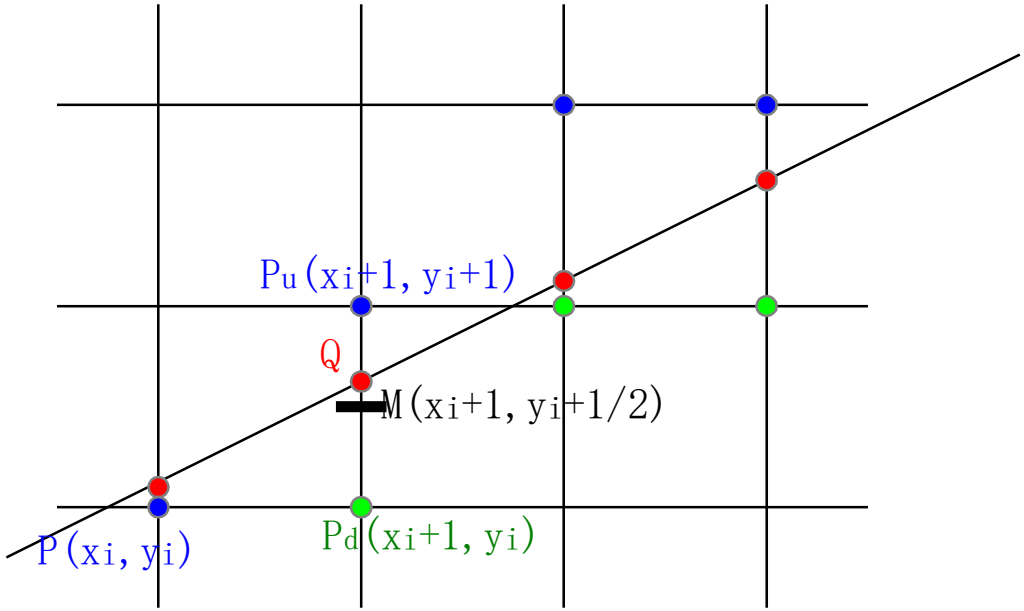


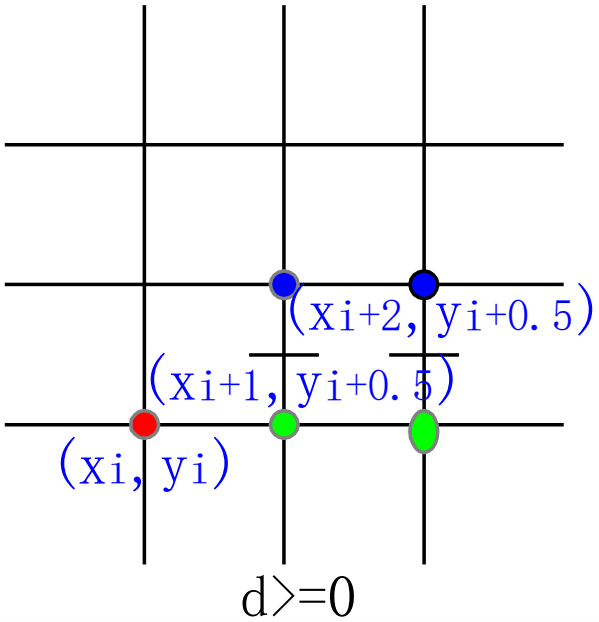
图5-5　Brensemham算法生成直线的原理

d的递推公式

$$d_i = F(x_i+1, y_i+0.5)$$
$$= (y_i+0.5) - k\cdot(x_i+1) - b$$
$$d_{i+1} = F(x_{i+1}+1, y_{i+1}+0.5)$$
$$= (y_{i+1}+0.5) - k\cdot(x_{i+1}+1) - b$$

$di \geq 0$: $x_{i+1} = x_i + 1; y_{i+1} = y_i;$

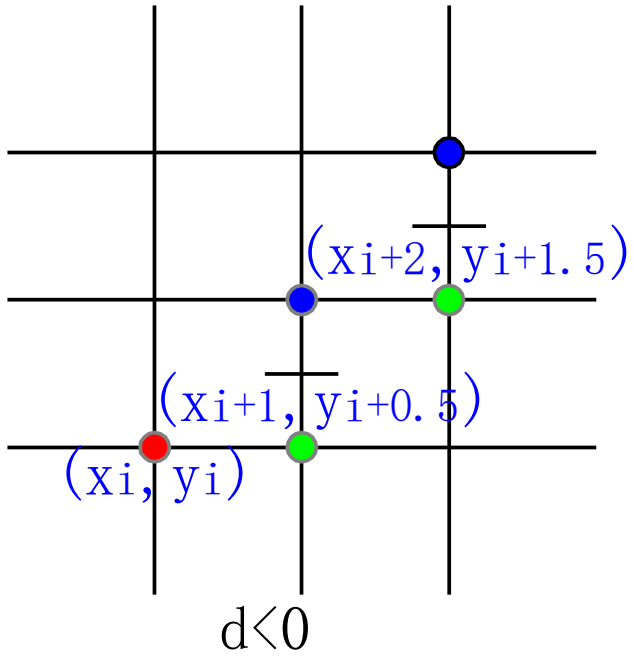
$$d_{i+1} = di - k$$
$$d_0 = (y0+0.5) - k\cdot(x0+1) - b$$
$$= [y0 - k\cdot x0 - b] + 0.5 - k = 0.5 - k$$



d的递推公式

$$d_i = F(x_i+1, y_i+0.5)$$
$$= (y_i+0.5) - k\cdot(x_i+1) - b$$

$$d_{i+1} = F(x_{i+1}+1, y_{i+1}+0.5)$$
$$= (y_{i+1}+0.5) - k\cdot(x_{i+1}+1) - b$$



$di < 0$: $x_{i+1} = x_i + 1; y_{i+1} = y_i + 1;$

$$d_{i+1} = (y_i + 1 + 0.5) - k\cdot(x_i + 1 + 1) - b$$
$$= \left[(y_i + 0.5) - k\cdot(x_i + 1) - b\right] + 1 - k$$
$$= d_i + 1 - k$$

d的递推公式

$$d_i = F(x_i+1, y_i+0.5)$$
$$= (y_i+0.5) - k\cdot(x_i+1) - b$$
$$d_{i+1} = F(x_{i+1}+1, y_{i+1}+0.5)$$
$$= (y_{i+1}+0.5) - k\cdot(x_{i+1}+1) - b$$

$di \geq 0$: $x_{i+1} = x_i + 1; y_{i+1} = y_i;$

$$d_{i+1} = di - k$$
$$d_0 = (y0+0.5) - k\cdot(x0+1) - b$$
$$= [y0 - k\cdot x0 - b] + 0.5 - k = 0.5 - k$$

$di < 0$: $x_{i+1} = x_i + 1; y_{i+1} = y_i + 1;$

$$d_{i+1} = (y_i + 1 + 0.5) - k\cdot(x_i + 1 + 1) - b$$
$$= [(y_i + 0.5) - k\cdot(x_i + 1) - b] + 1 - k$$
$$= d_i + 1 - k$$

$$d_0 = 0.5 - k$$

$$d_{i+1} = \begin{cases} d_i - k & , d_i \geq 0 \\ d_i + 1 - k & , d_i < 0 \end{cases}$$

改进算法

中点Bresenham算法（0≤k≤1）

$$d_0 = 0.5 - k$$

$$d_{i+1} = \begin{cases} d_i - k & , d_i \geq 0 \\ d_i + 1 - k & , d_i < 0 \end{cases}$$

```
void DrawLine(int x0,int y0,
              int x1,int y1 )
{
    float k, b, d;
    int x, y;
    k =...;b=...;
    x = x0; y = y0;
    d = 0.5 - k; //计算d0
    while ( x < x1)
    {
        DrawPixel(x,y);
        if ( d < 0)
        {   y = y +1;
            d = d + 1 - k;
        }
        else
        {
            d = d -k;
        }
        x++;
    }
    return;
}
```

- 只关心d的符号，
- 而不关心d的大小

浮点加法

改进算法

整数加法

$$d_0 = 0.5 - k$$

$$d_{i+1} = \begin{cases} d_i - k & , d_i \geq 0 \\ d_i + 1 - k & , d_i < 0 \end{cases}$$

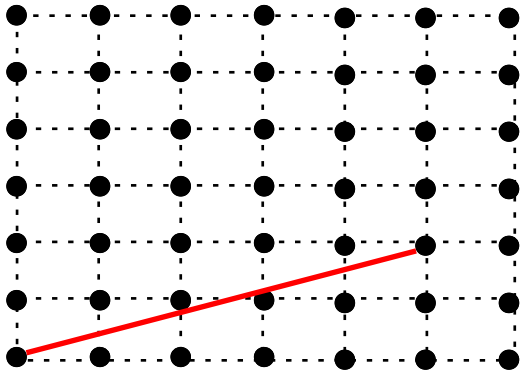
D_i=di*2(x1-x0);

$$D_0 = \Delta x - 2 \cdot \Delta y$$

$$D_{i+1} = \begin{cases} D_i - 2 \cdot \Delta y & , D_i \geq 0 \\ D_i + 2 \cdot \Delta x - 2 \cdot \Delta y & , D_i < 0 \end{cases}$$

```
void DrawLine(int x0,int y0,
              int x1,int y1 )
{
     int   D;
     int   dx,dy;
     int x, y;
     dx = x1-x0; dy = y1-y0;
     x = x0; y = y0;
     D = dx - 2*dy; //计算D0
     while ( x < x1)
     {
          DrawPixel(x,y);
          if ( d < 0)
          {   y = y +1;
              D = D + 2*dx - 2*dy;
          }
          else
          {
              D = D -2*dy;
          }
          x++;
     }
     return;
}
```

中点Bresenham算法



x0=0;
y0=0;
x1=5;
y1=2;

$$\Delta x = 5; \Delta y = 2;$$

$$D_0 = \Delta x - 2\Delta y = 1$$

-4

$$D_{i+1} = \begin{cases} D_i + (-2\Delta y), & D_i \geq 0 \\ D_i + (2\Delta x - 2\Delta y), & D_i < 0 \end{cases}$$

6

x	y	d
0	0	1
1	0	-3
2	1	3
3	1	-1
4	2	5
5	2	1

中点Bresenham算法

0≤k≤1时Bresenham算法的算法步骤为：

1. 输入直线的两端点P0(x0,y0)和P1(x1, y1)。

2. 计算初始值△x、△y、d=0.5-k、x=x0、y=y0；

3. 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+1-k；

否则(x, y)更新为(x+1, y)，d更新为d-k。

4. 当直线没有画完时，重复步骤3。否则结束。

中点Bresenham算法改进

改进：用2d△x代替d

1.输入直线的两端点P0(x0,y0)和P1(x1,y1)。

2.计算初始值△x、△y、d=△x-2△y、x=x0、y=y0。

3.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△x-2△y；

否则(x, y)更新为(x+1,y), d更新为d-2△y。

4.当直线没有画完时，重复步骤3。否则结束。

Bresenham算法

E.Jack Bresenham

communication of the ACM

Graphics and Image Processing — W. Newman* Editor

A Linear Algorithm for Incremental Digital Display of Circular Arcs

Jack Bresenham
IBM System Communications Division

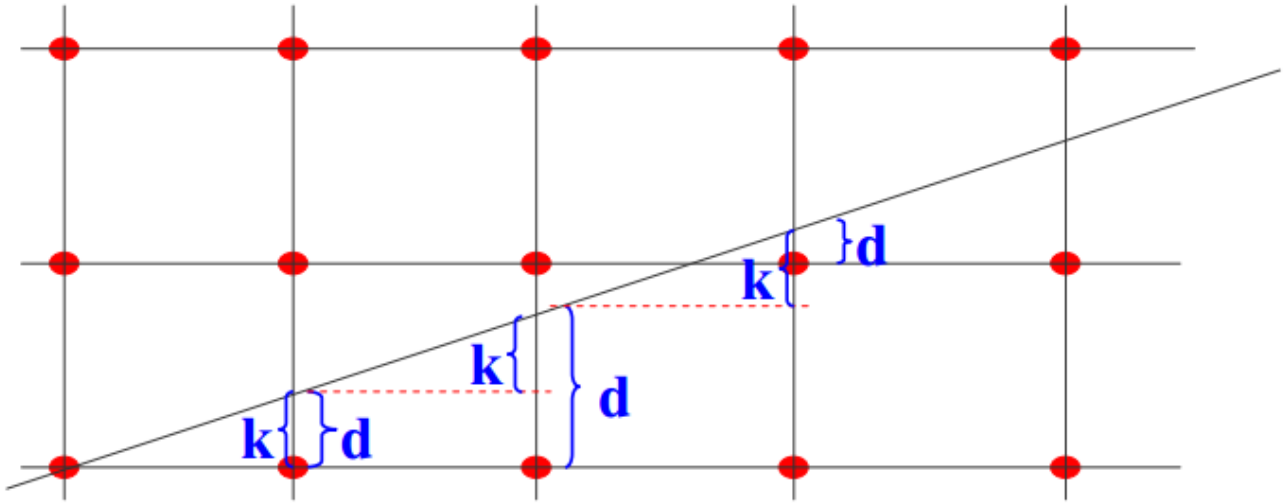
Circular arcs can be drawn on an incremental display device such as a cathode ray tube, digital plotter, or matrix printer using only sign testing and elementary addition and subtraction. This paper describes methodology for producing dot or step patterns closest to the true circle.

This paper describes an algorithm for circular arc mesh point selection using incremental display devices such as a cathode ray tube or digital plotter. Error criteria are explicitly specified and both squared and radial error minimization considered. The repetitive incremental stepping loop for point selection requires only simple addition/subtraction and sign testing; neither quadratic nor trigonometric evaluations are required. When a circle's center point and radius are integers, only integer calculations are required.

The circle algorithm complements an earlier line algorithm described in [1, 2]. The algorithm's minimum error point selection is appropriate for use in numerical control, drafting, or photo mask preparation applications where closeness of fit is a necessity. Its simplicity and use only of elementary addition/subtraction allow its use in small computers, programmable terminals, or direct hardware implementations where compactness and speed are desirable.

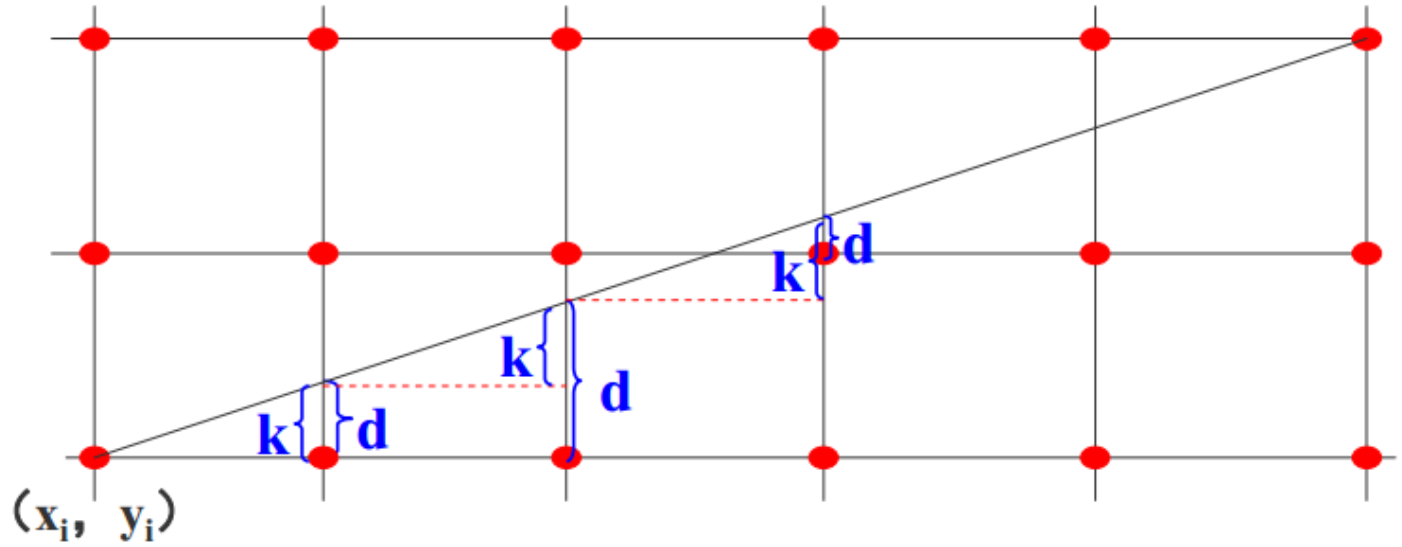
The display devices under consideration are capable of executing, in response to an appropriate pulse, any one of the eight linear movements shown in Figure 1. Thus incremental movement is from a point on a mesh to any of its eight adjacent points on the mesh.

Bresenham算法



该算法的思想是通过各行、各列像素中心构造一组虚拟网格线，按照直线起点到终点的顺序，计算直线与各垂直网格线的交点，然后根据误差项的符号确定该列象素中与此交点最近的象素。

Bresenham算法



假设每次x+1，y的递增（减）量为0或1，它取决于实际直线与最近光栅网格点的距离，这个距离的最大误差为0.5。

Bresenham算法

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

误差项的计算

- $d_{初}$=0，
- 每走一步：d=d+k
- 一旦y方向上走了一步，d=d−1

改进1

令e=d-0.5

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

- e初=-0.5,
- 每走一步有e=e+k。
- if (e>0) then e=e-1

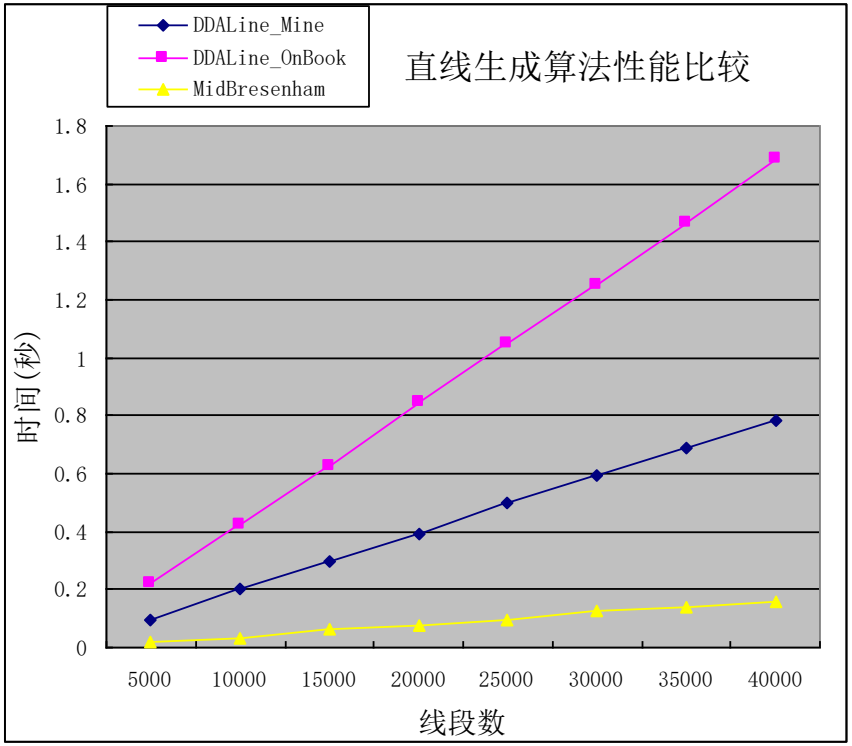
改进2

用2e△x来替换e

- e初=-△x,
- 每走一步有e=e+2△y。
- if (e>0) then e=e-2△x

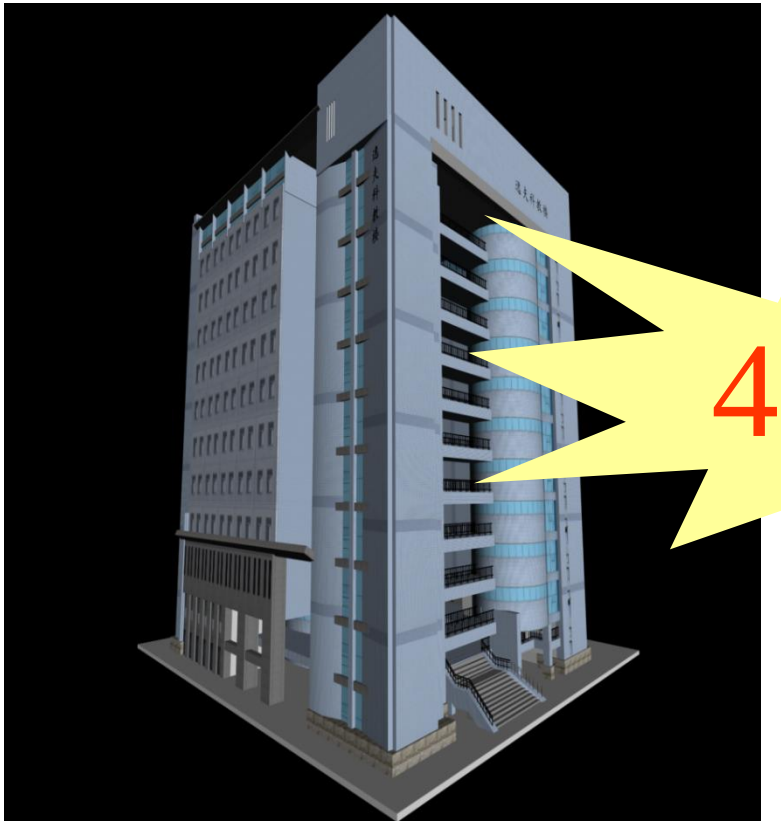
中点Bresenham算法

环境	CPU: 2G 内存： 512M		
线段	起点(0,0)	终点 （1000， 370)	
线段数	时间(单位： 秒)		
	DDALine_Mine	DDALine_OnBook	MidBresenham
05000	0.093	0.219	0.016
10000	0.203	0.422	0.031
15000	0.297	0.625	0.062
20000	0.391	0.844	0.078
25000	0.5	1.047	0.093
30000	0.594	1.25	0.125
35000	0.688	1.468	0.141
40000	0.781	1.688	0.156

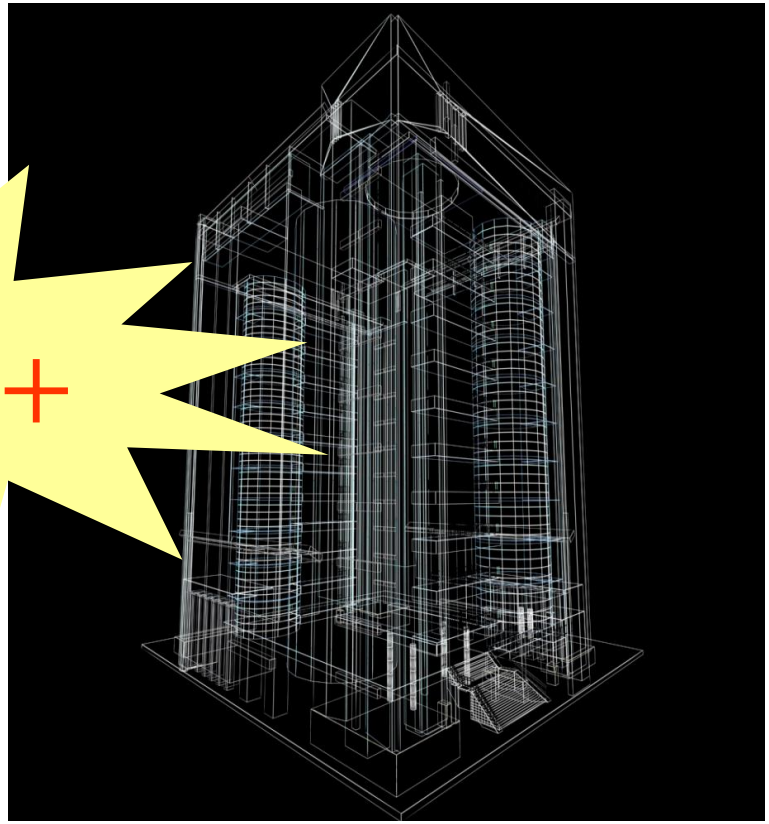


需要这么多线段吗?

请看….



46000+



请看….

DDA&中点Bresenham算法

介绍了扫描转换算法的作用

介绍了两种直线扫描转换算法：

- DDA方法
- 中点 Bresenham算法

“重要点”

- 利用固有的属性改进算法，提高性能！

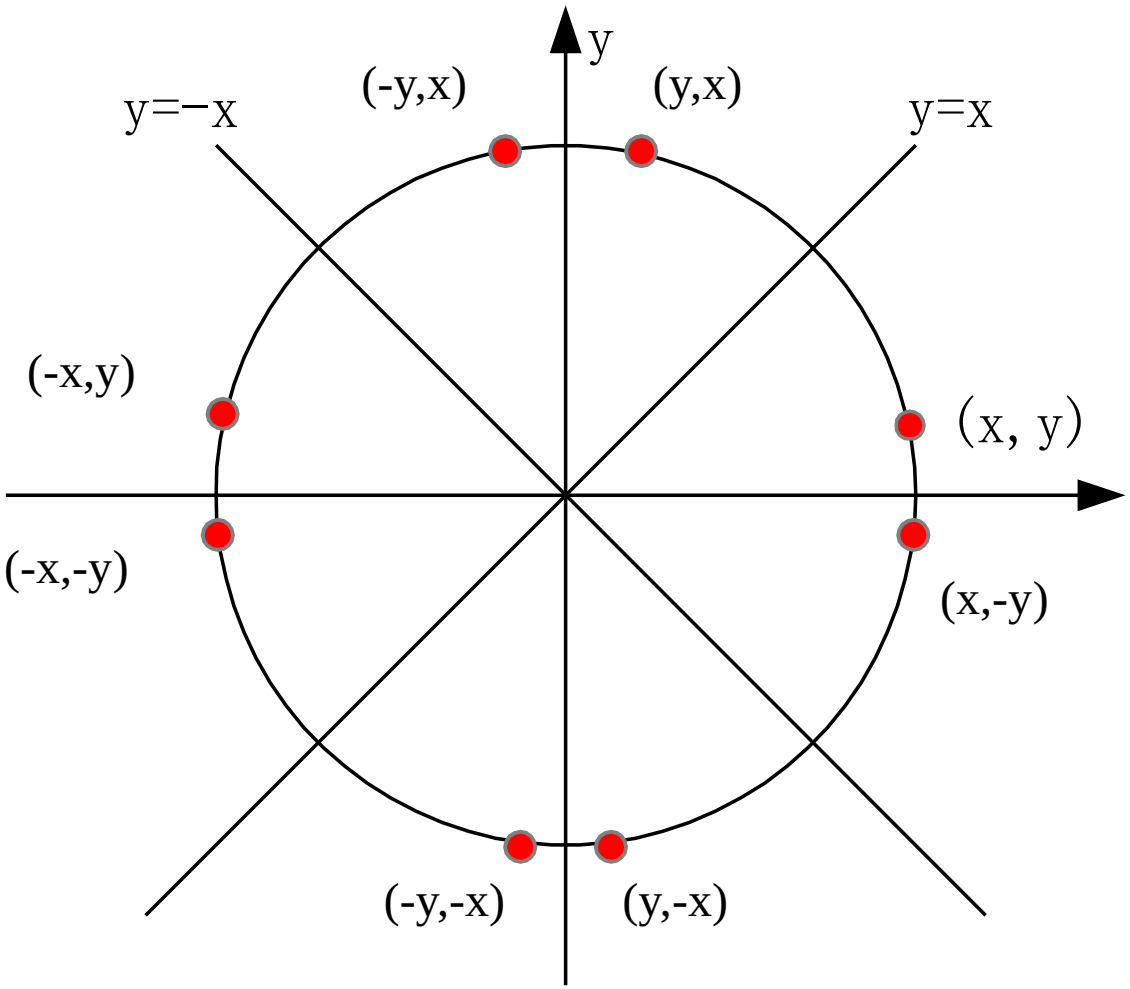
圆的扫描转换

解决的问题：

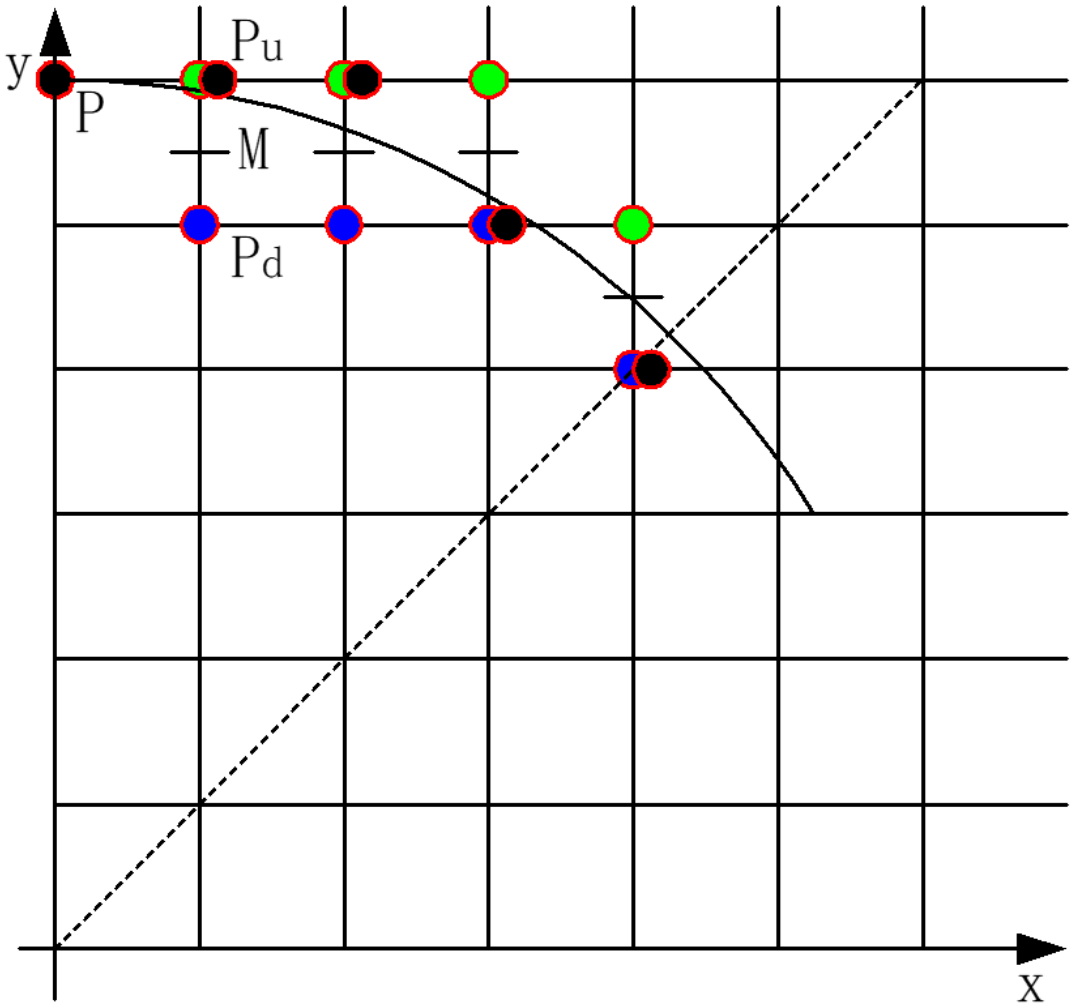
绘出圆心在原点，半径为整数R的圆$x^2+y^2==R^2$

八分法画圆

八分法画圆



八分法画圆



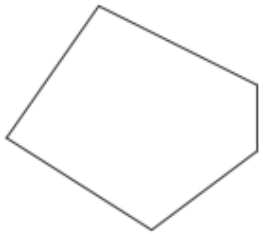
多边形的扫描转换

多边形的扫描转换

多边形分为凸多边形、凹多边形、含内环的多边形等：

（1）凸多边形

任意两顶点间的连线均在多边形内



（2）凹多边形

任意两顶点间的连线有不在在多边形内

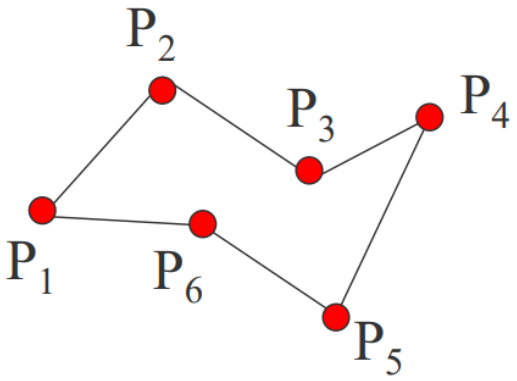
（3）含内环的多边形

多边形内包含多边形

多边形的扫描转换

顶点表示：

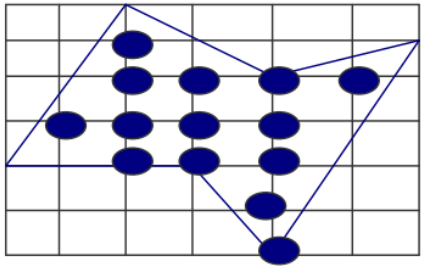
用多边形的顶点序列来刻画多边形



顶点表示

点阵表示：

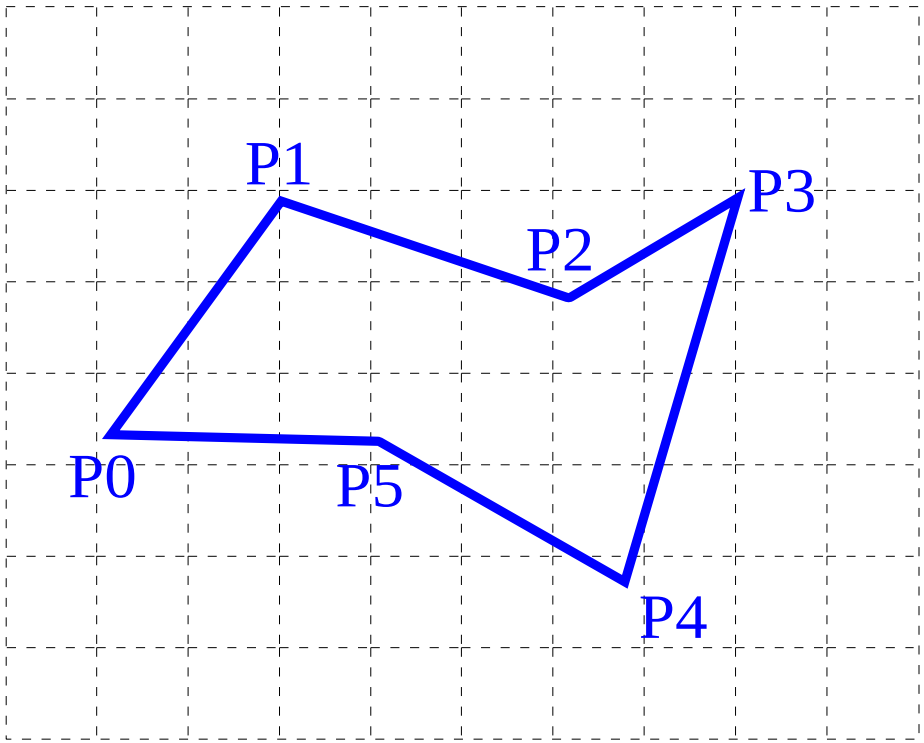
是用位于多边形内的象素的集合来刻画多边形



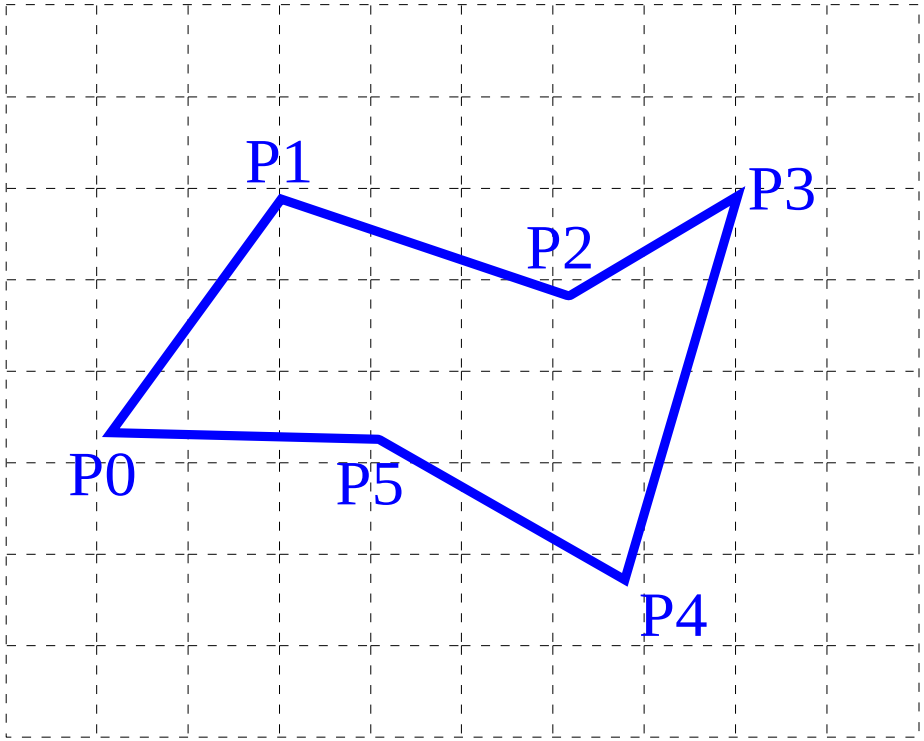
点阵表示

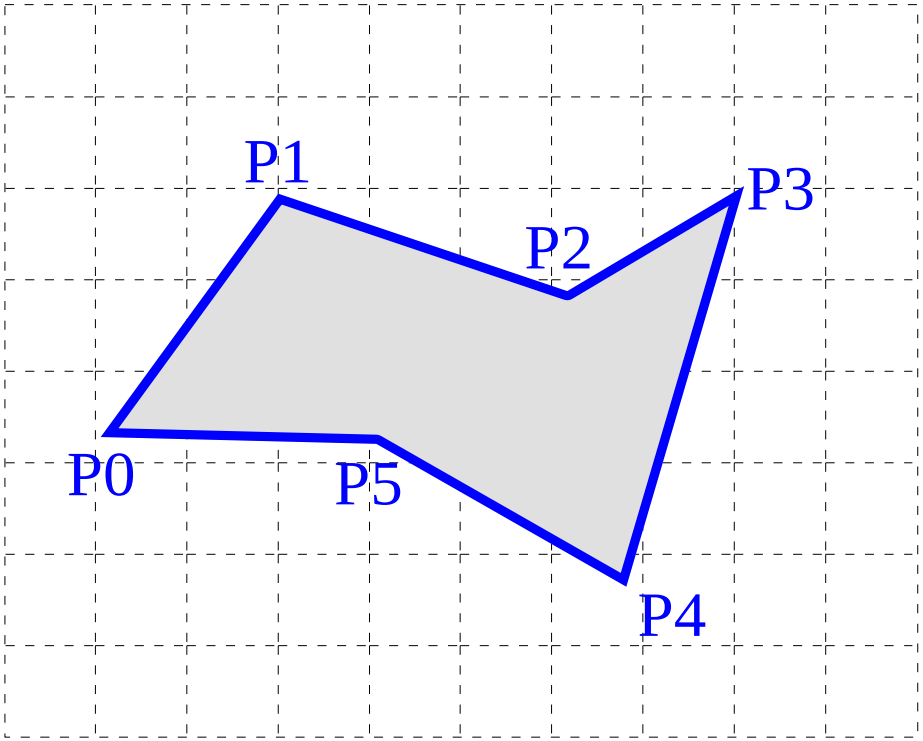
扫描转换多边形或多边形的填充：从多边形顶点表示到点阵表示的转换

多边形的扫描转换

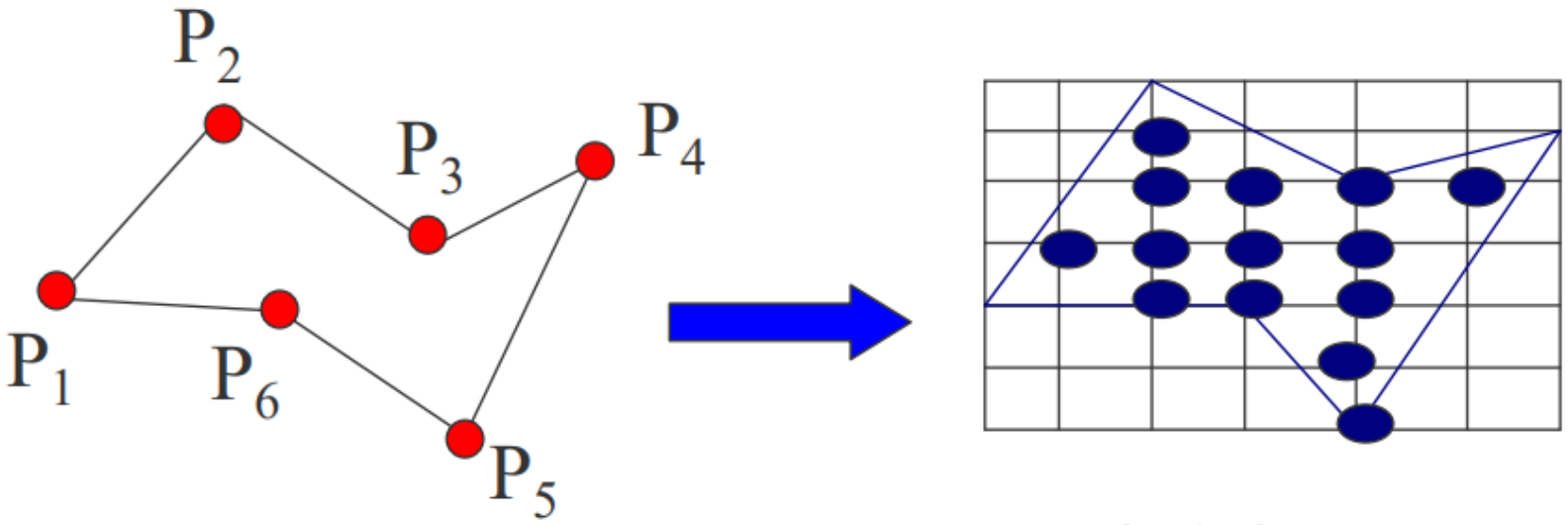


多边形的扫描转换



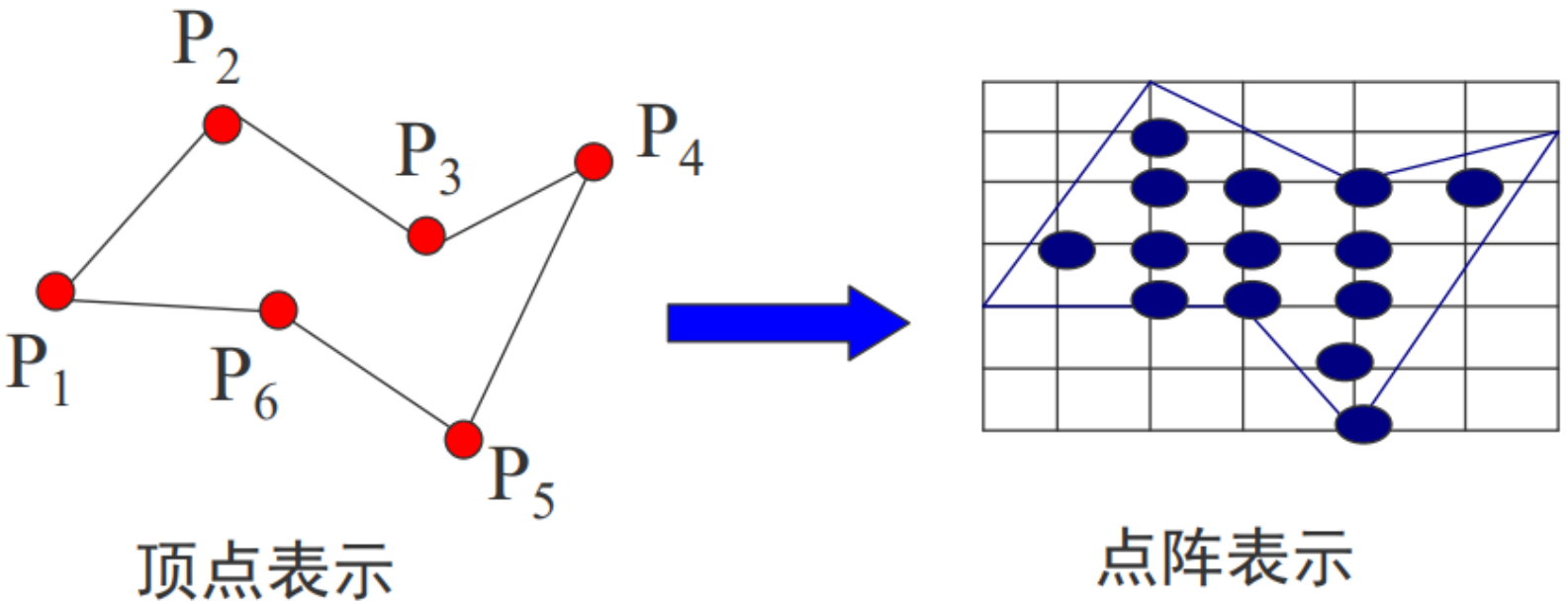


多边形的扫描转换

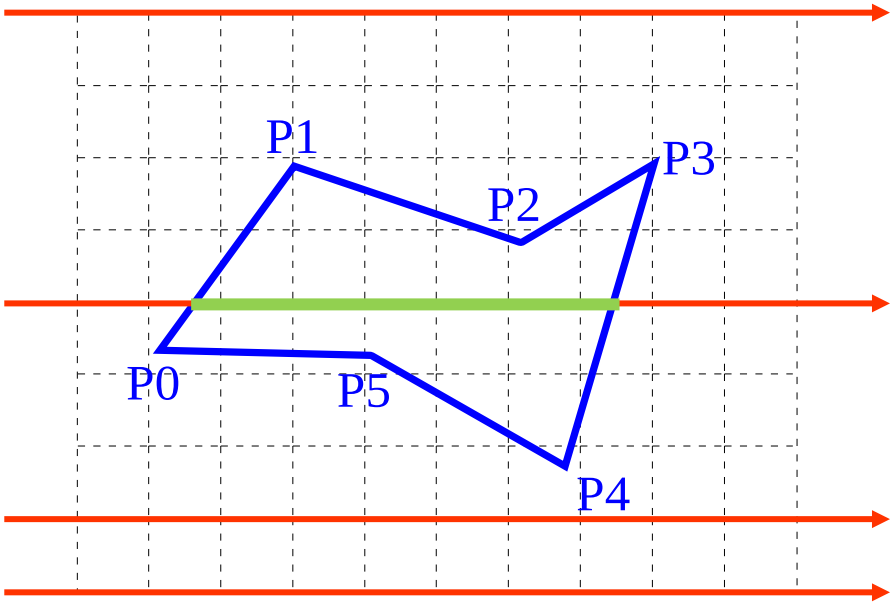


多边形的扫描转换

现在的问题是，知道多边形的边界，如何找到多边形内部的（像素）点，即把多边形内部填上颜色



X-扫描线算法



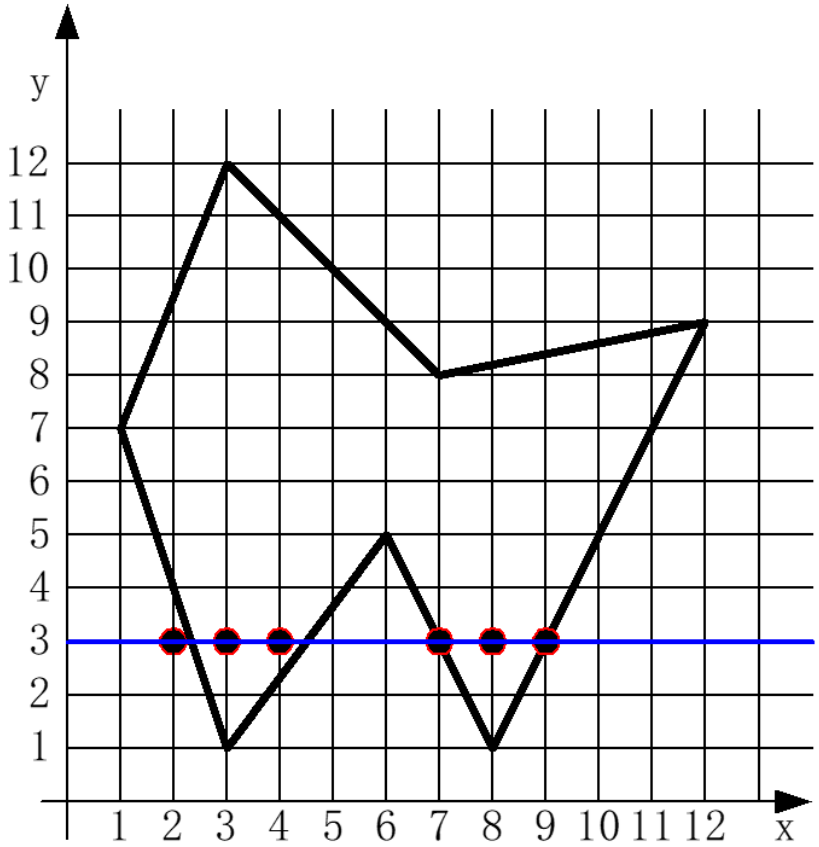
“从下到上”，逐条线的处理

- 扫描转换

扫描线平行于x轴：

- X-扫描线算法

X-扫描线算法



求交点：

A：（2，3）

B：（4，3）

C：（7，3）

D：（9，3）

交点取整

[A，B]，[C，D]在多边形内部

X-扫描线算法

（1）确定多边形所占有的最大扫描线数，得到多边形顶点的最小和最大y值（ymin和ymax）。

（2）从y=ymin到y=ymax，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：

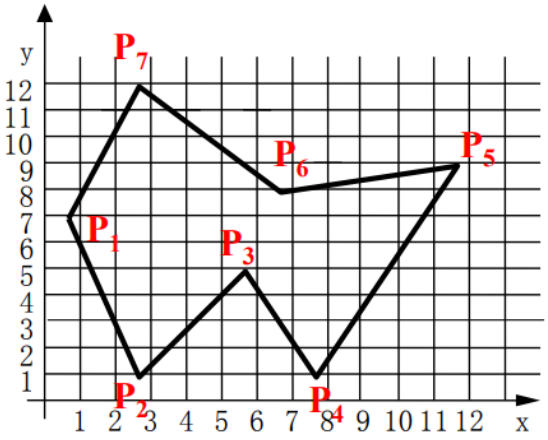
a.求交

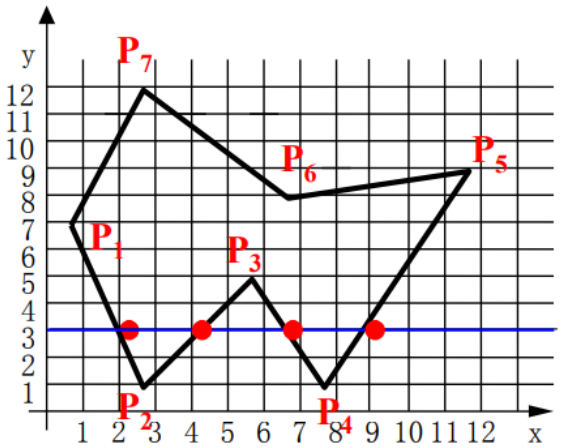
b.排序

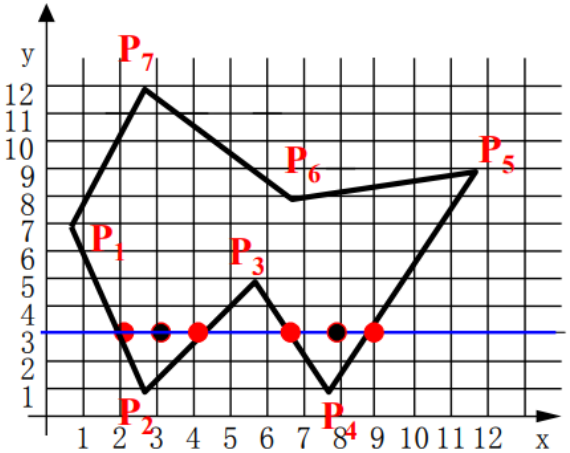
c.交点配对

d.区间填色

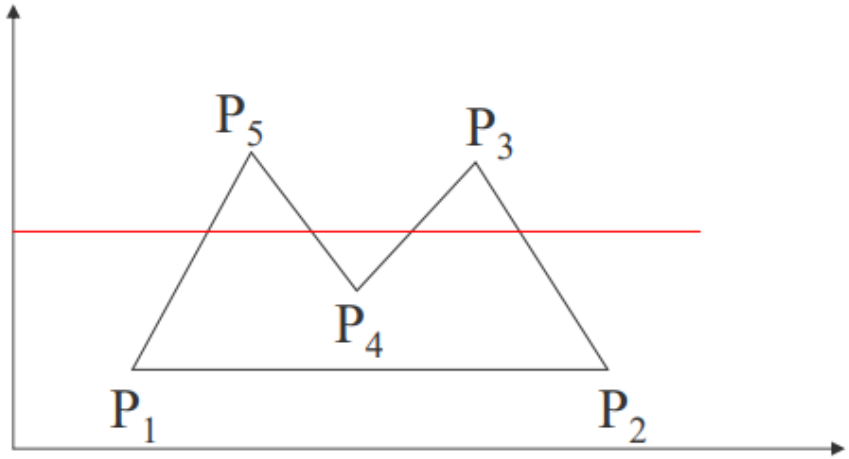
X-扫描线算法

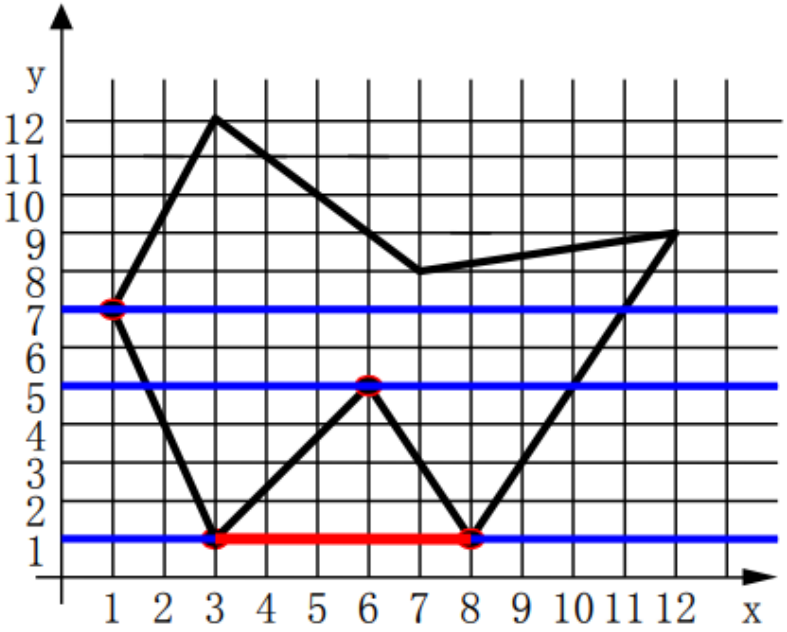




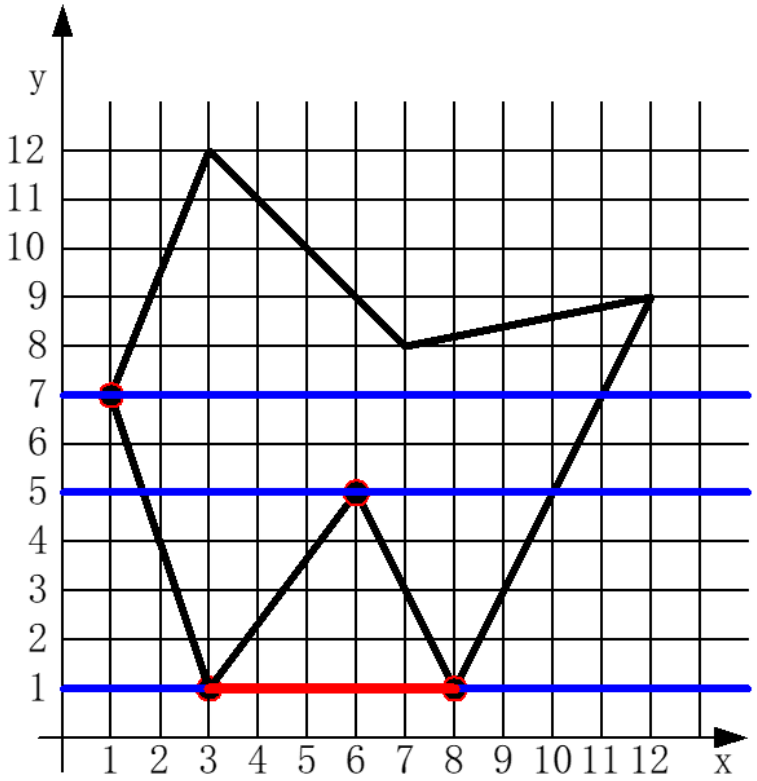


交点处理





交点处理



共享顶点的两条边在扫描线的同一边

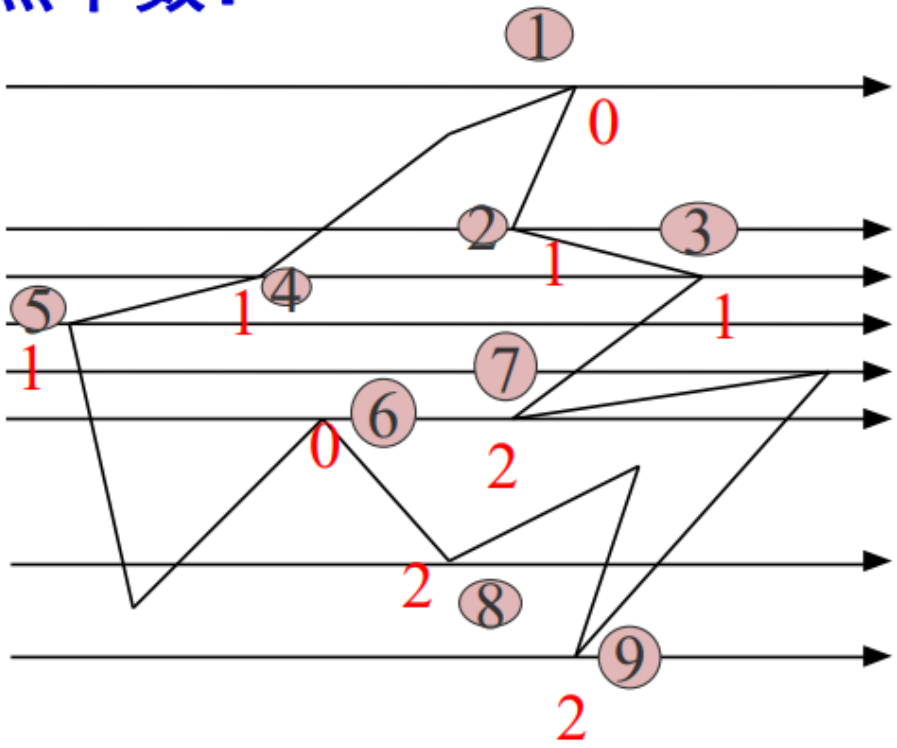
- 按0或2个处理

共享顶点的两条边分别落在扫描线的两边

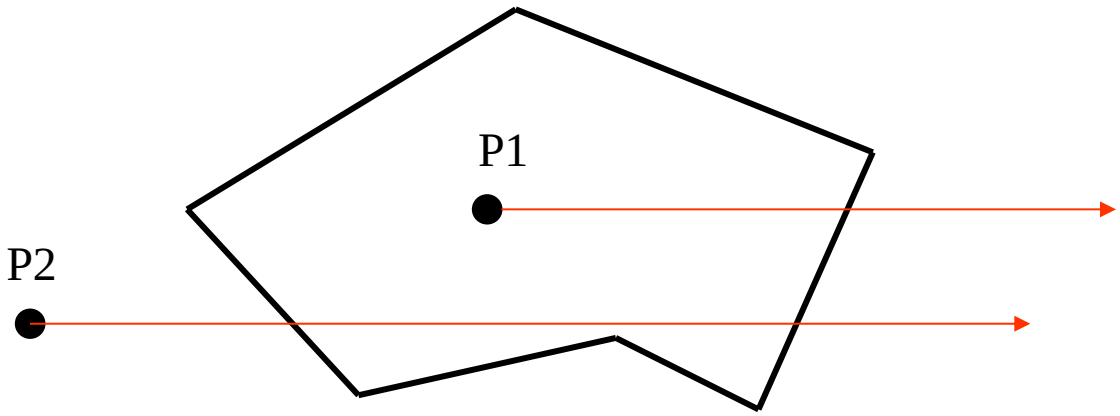
- 按1个处理

交点处理

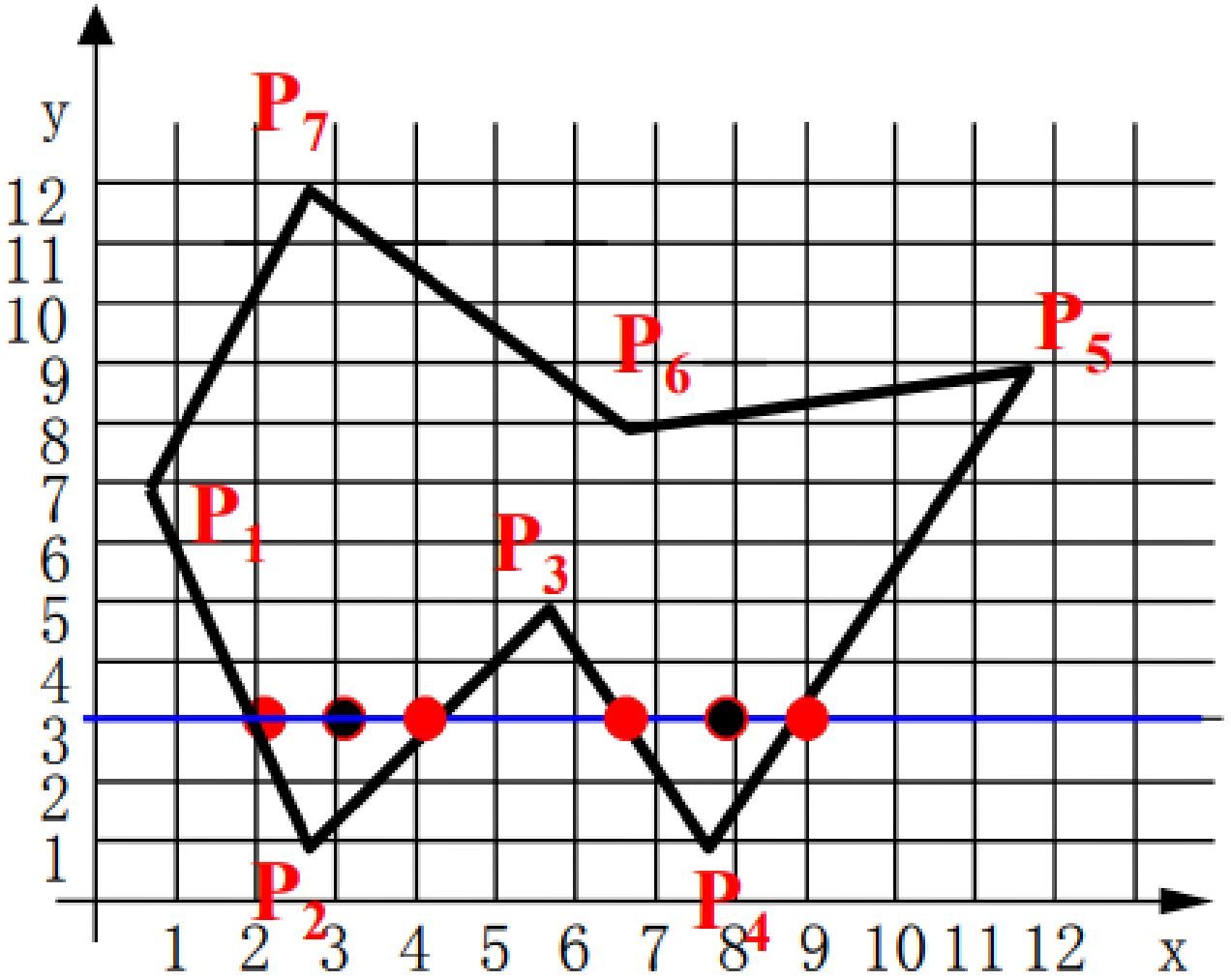
举例计算交点个数：



扫描线算法的应用：内外测试



x-扫描线算法



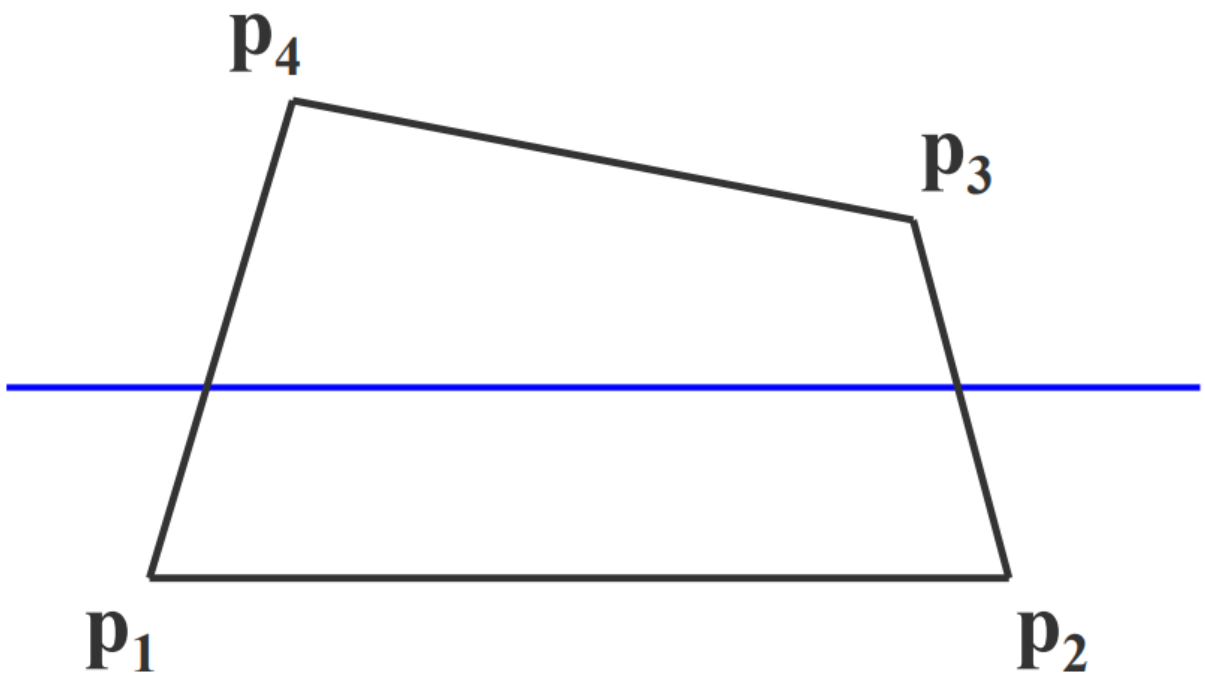
算法改进

扫描转换算法重要意义是提出了图形学里两个重要的思想：

（1）扫描线：当处理图形图像时按一条条扫描线处理

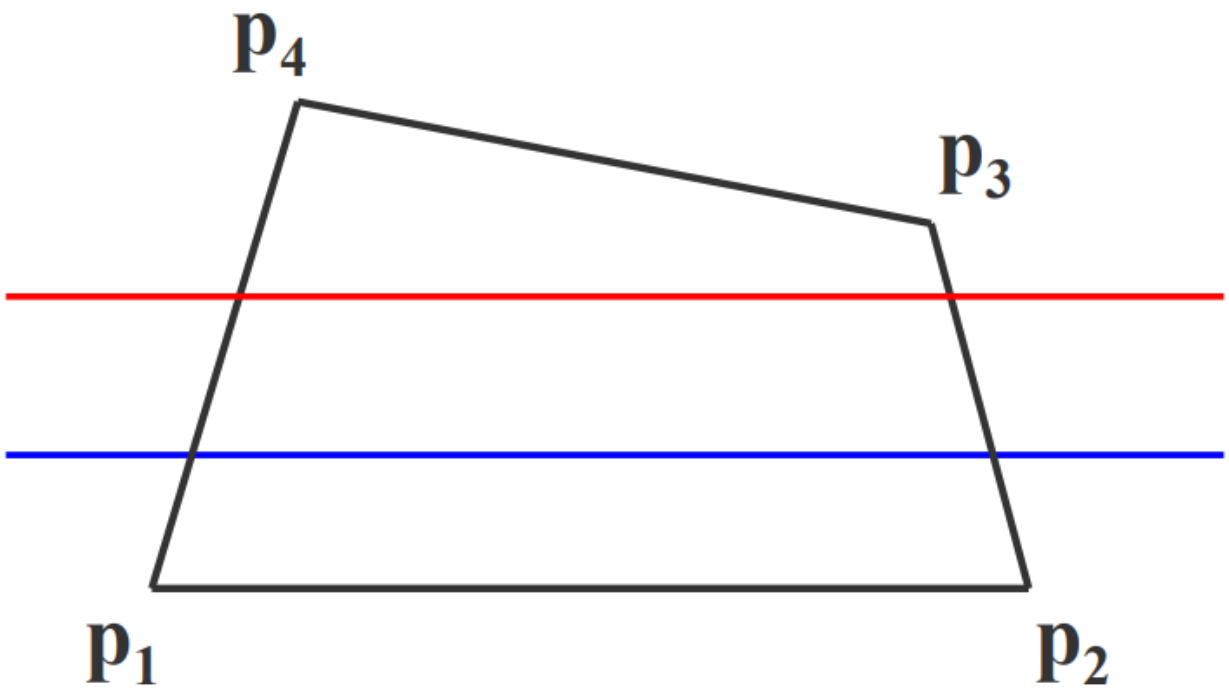
（2）增量的思想

算法改进



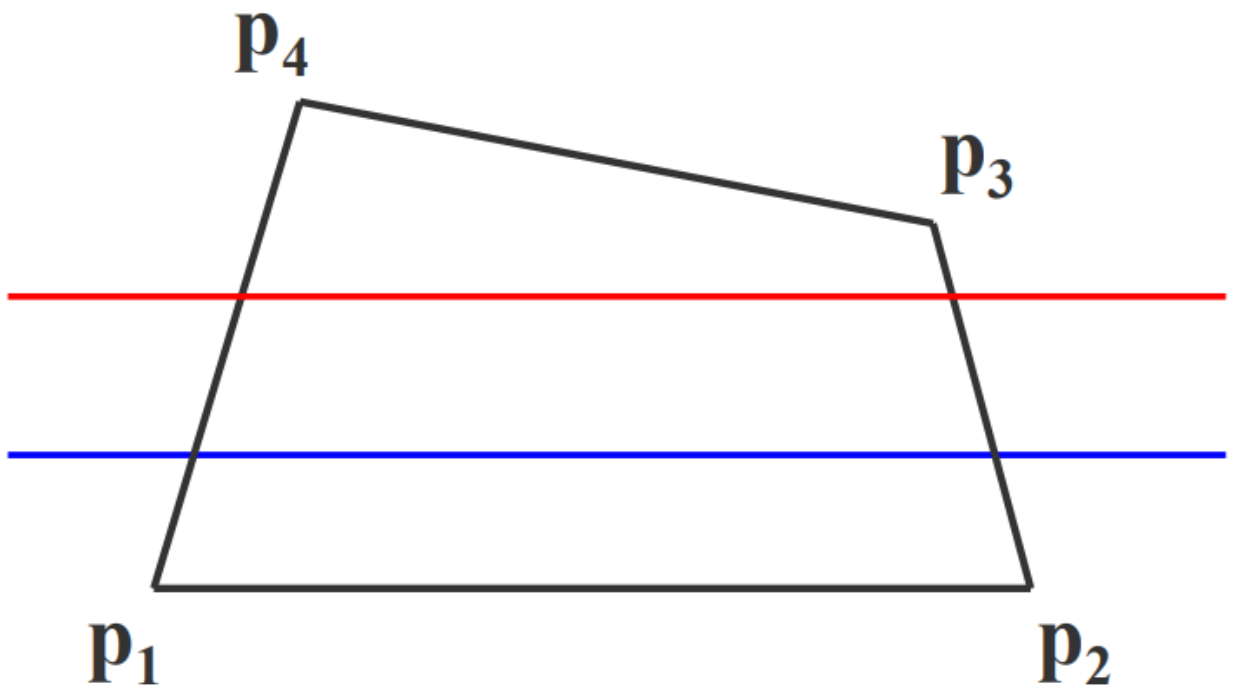
有效边

算法改进



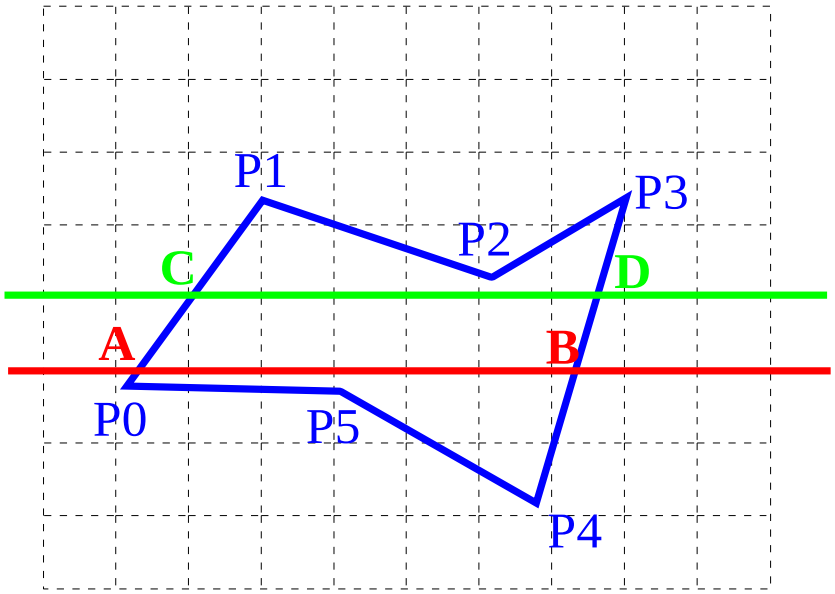
扫描线的连贯性

算法改进



多边形的连贯性

改进的有效边表算法



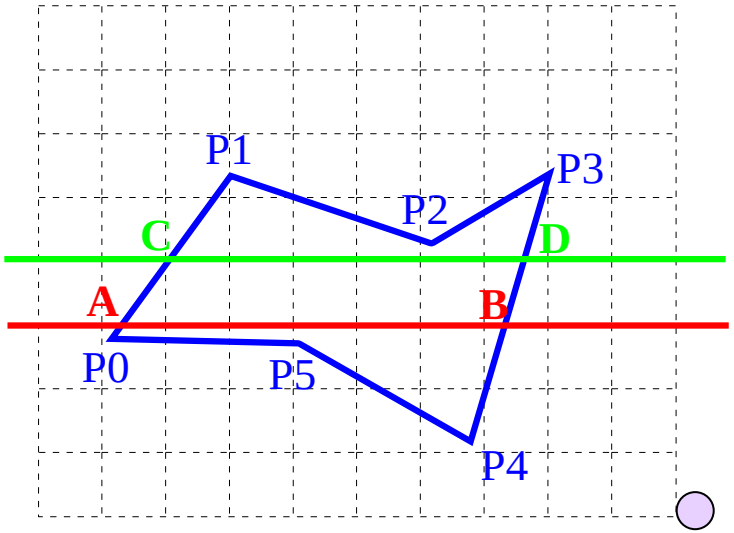
扫描线AB和多边形求交过程：

依次和P0P1、P1P2、…、P5P0求交

问题：

- P1P2、P2P3、 P4P5、 P5P0和AB不相交
- 求C、D可否利用A、B的信息？

改进的有效边表算法



只与它相交的边进行求交运算

利用交点的相关性

多边形边的连贯性

扫描线与边求交点

合理利用各种相关性

有效边表与边表

改进的有效边表算法

有效边（Active Edge）：指与当前扫描线相交的多边形的边，也称为活性边。

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。

改进的有效边表算法

```
void polyfill (polygon, color)
      int color;  多边形   polygon;
      {  for (各条扫描线i )
          { 初始化新边表头指针NET[i];
            把y_min = i 的边放进边表NET[i];
          }
         y = 最低扫描线号;
         初始化活性边表AET为空;
         for (各条扫描线i )
              {
                 把新边表NET[i] 中的边结点用插入排序法插入AET表,
                     使之按x坐标递增顺序排列;
                 遍历AET表，把配对交点区间(左闭右开)上的象素(x,y)
                    ，用putpixel(x,y,color) 改写象素颜色值;
                 遍历AET表，把y_max= i 的结点从AET表中删除，并把y_max>i
                   结点的x值递增Δx;
                 若允许多边形的边自相交，则用冒泡排序法对AET表重新排序;
              }
       }  /* polyfill */
```

扫描线算法

扫描线法可以实现已知任意多边形域边界的填充。

该填充算法是按扫描线的顺序，计算扫描线与待填充区域的相交区间，再用要求的颜色显示这些区间的像素，即完成填充工作。

无法实现对未知边界的区域填充

边缘填充算法

其基本思想是按任意顺序处理多边形的每条边。

在处理每条边时，首先求出该边与扫描线的交点，然后将每一条扫描线上交点右方的所有像素取补。

多边形的所有边处理完毕之后，填充即完成。

算法简单，但对于复杂图型，每一象素可能被访问多次。

输入和输出量比有效边算法大得多。

栅栏填充算法

为了减少边缘填充法访问像素的次数，可采用栅栏填充算法。

栅栏指的是一条过多边形顶点且与扫描线垂直的直线。

它把多边形分为两半。

在处理每条边与扫描线的交点时，将交点与栅栏之间的像素取补。

边界标志算法

访问每个像素的次数不超过两次。

由于边界标志算法不必建立维护边表以及对它进行排序，所以边界标志算法更适合硬件实现。

这时它的执行速度比有序边表算法快一至两个数量级。

自学内容

边缘填充算法

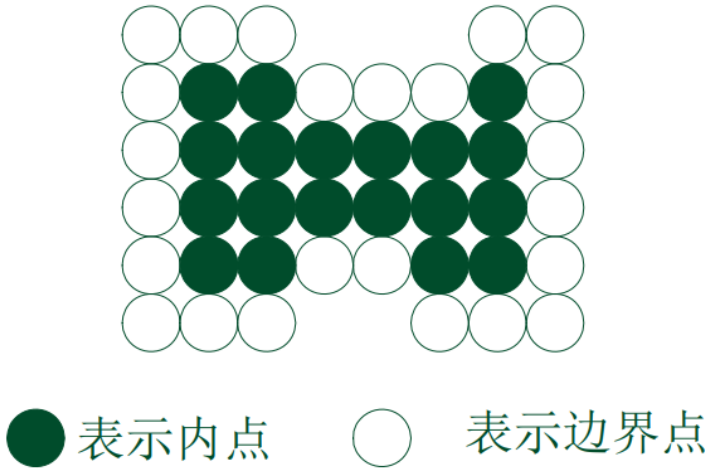
栅栏填充算法

边界标志算法

思考各算法背后的原理，为什么可行？

多边形的区域填充

区域填充



边界表示：枚举出边界上的所有像素，边界上的所有像素着同一个颜色，内部像素着与边界像素不同的颜色
（边界填充算法）

内点表示：枚举出区域内部的所有像素，内部的所有像素着同一个颜色，边界像素着与内部像素不同的颜色
（泛填充算法）

区域填充

区域填充算法要求区域是连通的，因为只有在连通区域中，才可能将种子点的颜色扩展到区域内的其它点。

区域可分为4向连通区域和8向连通区域

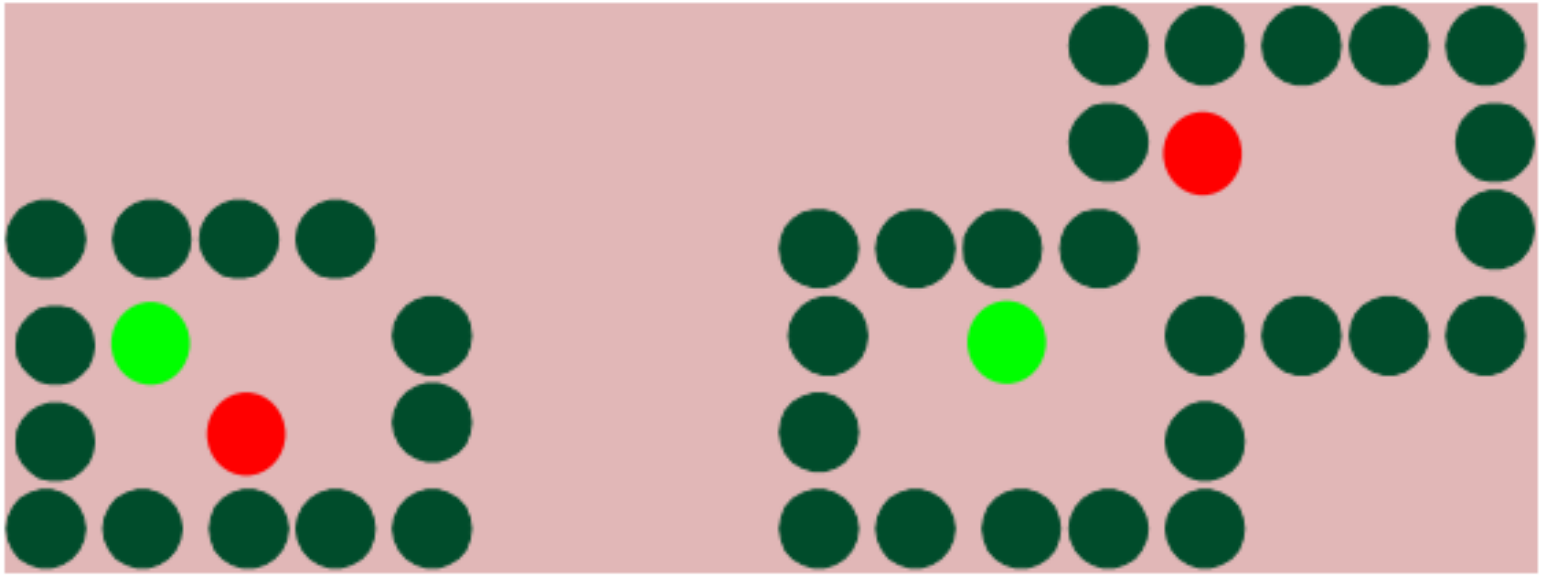
四个方向运动

八个方向运动

四连通区域

八连通区域

区域填充



四连通区域

八连通区域

边界填充算法

算法的输入：种子点坐标(x，y)，填充色和边界颜色。

栈结构实现边界填充算法的算法步骤为：

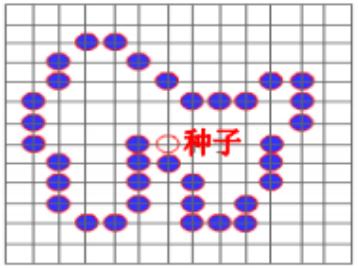
种子像素入栈；当栈非空时重复执行如下三步操作：

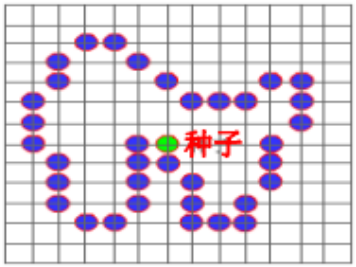
(1) 栈顶像素出栈；

(2) 将出栈像素置成填充色；

(3) 检查出栈像素的邻接点，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。

边界填充算法



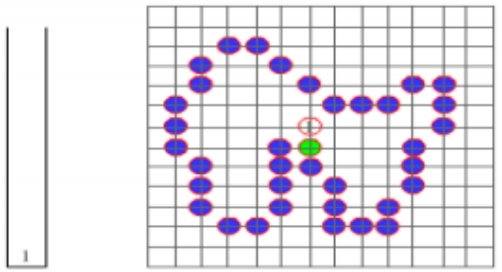


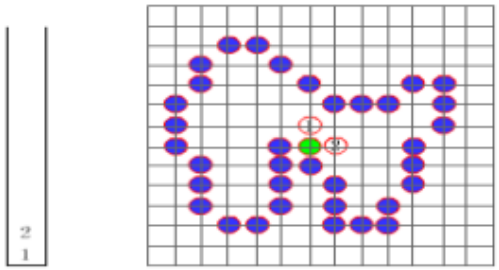
种子像素入栈，栈非空时重复执行如下三步：

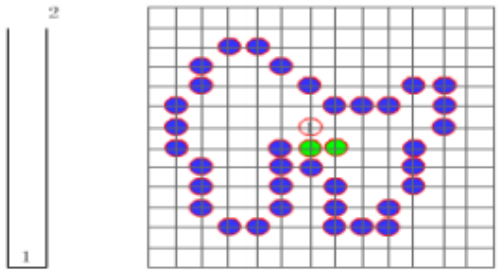
（1）栈顶像素出栈

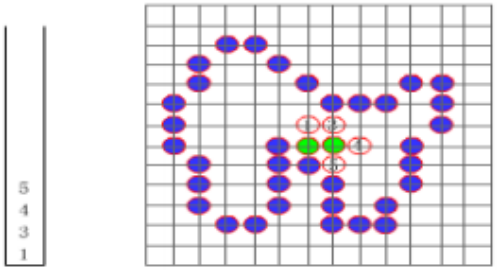
（2）将出栈像素置成要填充色

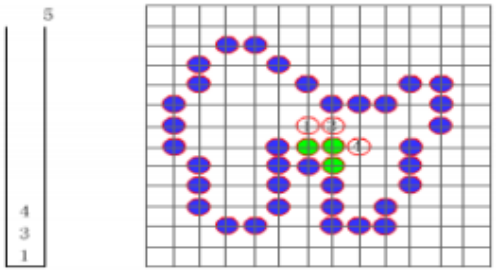
（3）按左、上、右、下顺序检查与栈像素相邻的四个像素，若其中某个像素不在边界且未置成填充色，则把该像素入栈

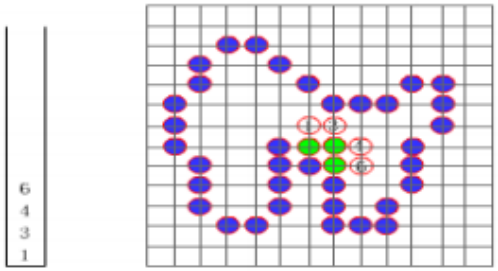




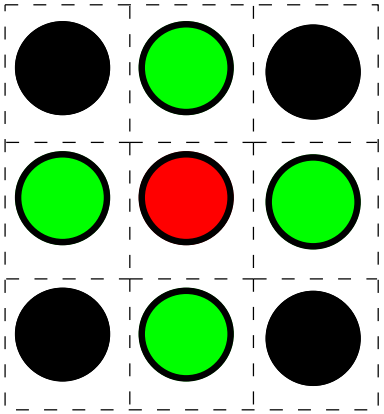




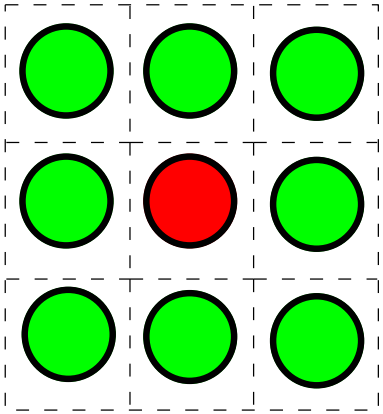




遍历路径



4-连通域



8-连通域

边界填充算法

种子填充算法的不足之处

（1）有些像素会入栈多次，降低算法效率；栈结构占空间

（2）递归执行，算法简单，但效率不高。区域内每一像素都引进一次递归，进/出栈，费时费内存

（3）改进算法，减少递归次数，提高效率

可以采用区域填充的扫描线算法

泛填充算法

算法的输入：种子点坐标(x, y)，填充色和内部点的颜色。

算法原理：

算法从指定的种子(x, y)开始，用所希望的填充色赋给所有当前为给定内部颜色的像素点。

泛填充算法

泛填充算法步骤如下：

种子像素入栈；当栈非空时重复执行如下三步操作：

(1) 栈顶像素出栈；

(2) 将出栈像素置成填充色；

(3) 检查出栈像素的邻接点，若其中某个像素点不是给定内部点的颜色且未置填充色，则把该像素入栈。

基于图遍历算法的区域填充

```
void dfs(Graph G, init v)
{
     int w;
     visit(v); visited[v] = TRUE;
     w = firstadj(G,v);
     while(w != 0)
     {
          if ( visited(w) == FALSE )
          {
               dfs(G,w);
          }
          w = nextadj(G,v,w);
     }
}
```

种子点

设置访问标志

是否访问过？

获取相邻节点

多边形的扫描转换与区域填充

- 基本思想不同
  - 多边形扫描转换是指将多边形的顶点表示转化为点阵表示
  - 区域填充只改变区域的填充颜色，不改变区域表示方法

- 基本条件不同
  - 在区域填充算法中，要求给定区域内一点作为种子点，然后从这一点根据连通性将新的颜色扩散到整个区域
  - 扫描转换多边形是从多边形的边界(顶点)信息出发，利用多种形式的连贯性进行填充的

自学内容

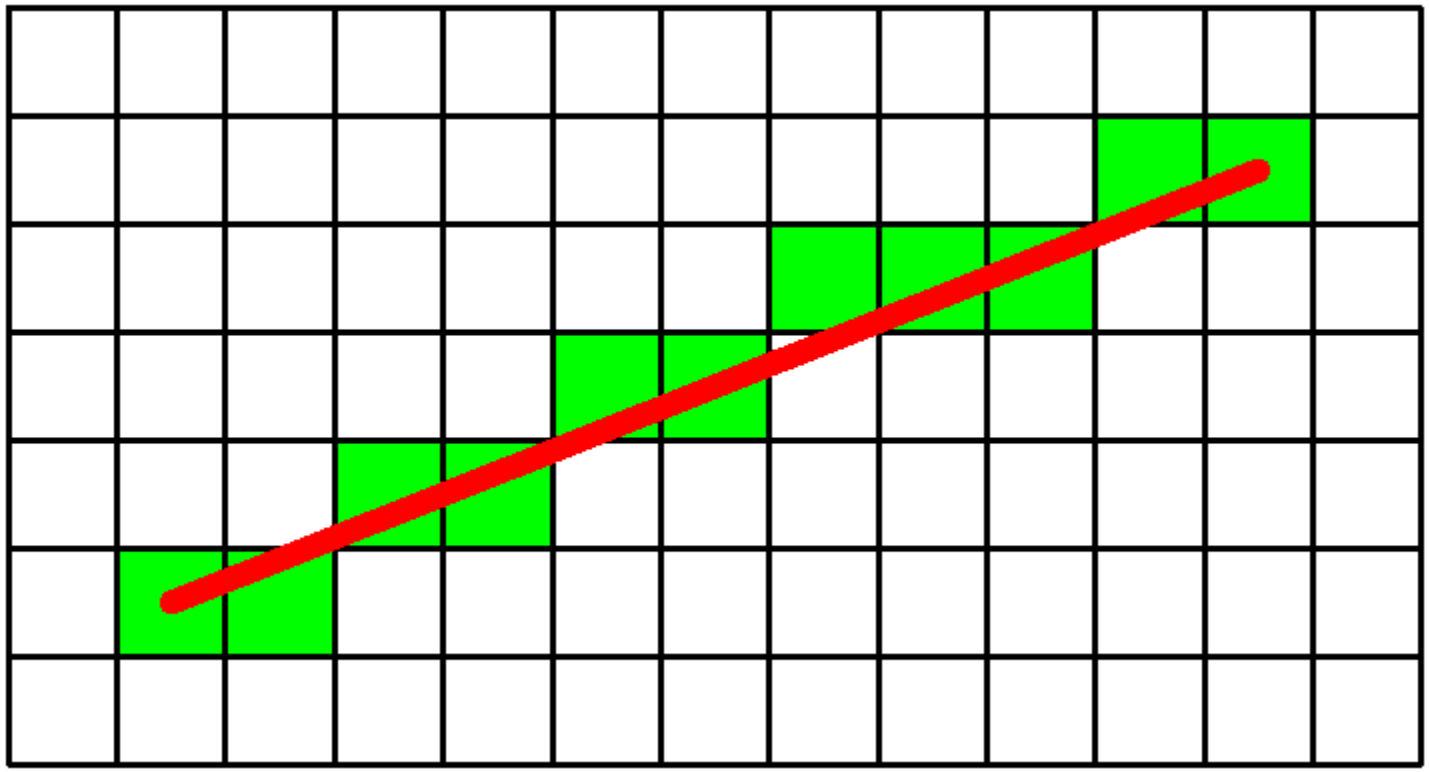
字符处理

属性处理

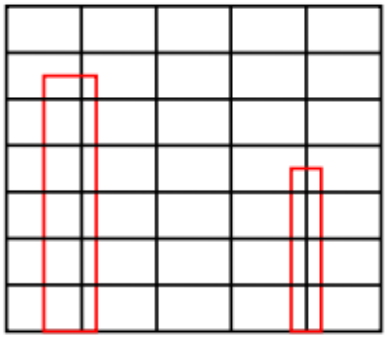
反走样

走样

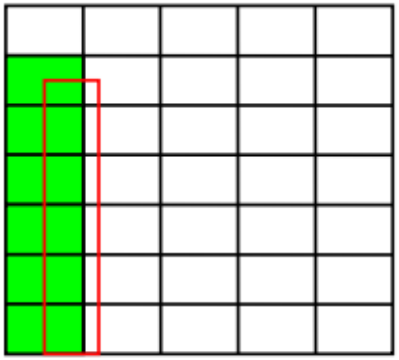
反走样



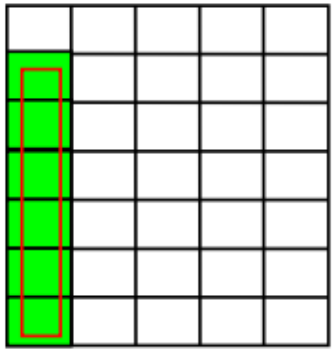
走样



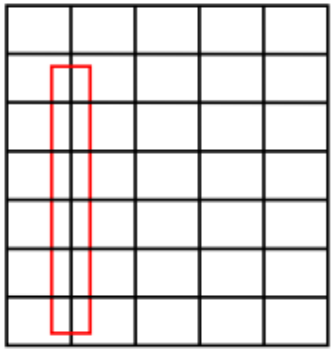
(a) 需显示的矩形



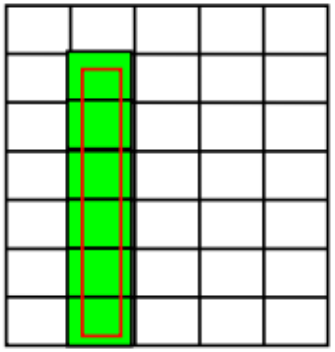
(b) 显示结果



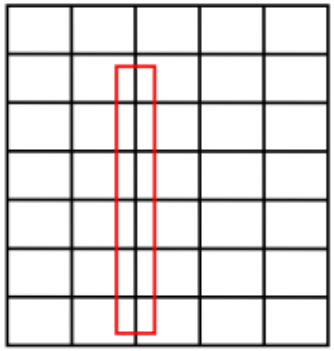
(a) 显示



(b) 不显示



(c) 显示



(d) 不显示

反走样

640*512

走样（Aliasing）：

用离散量表示连续量引起的失真，叫走样

走样是数字化的必然产物

320*256

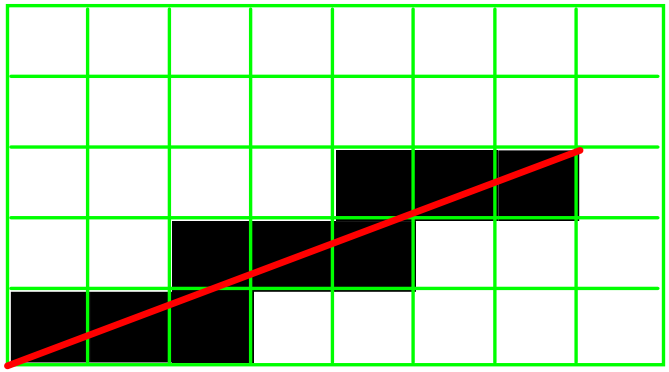
反走样（Antialiasing）：

用于消除、减少走样的技术

反走样

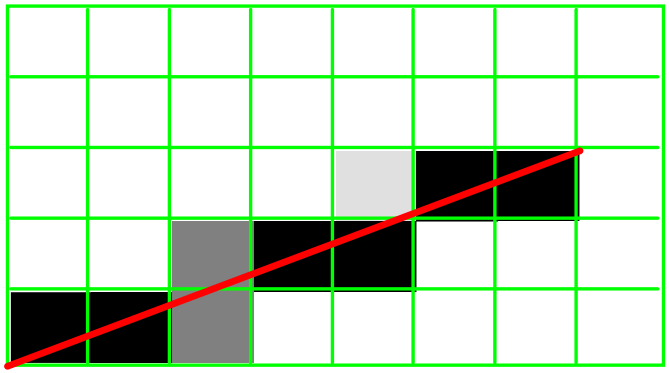
对像素点P，现在：

- 要么全在直线上
- 要么不在直线上



反走样：

P在直线L上的灰度值

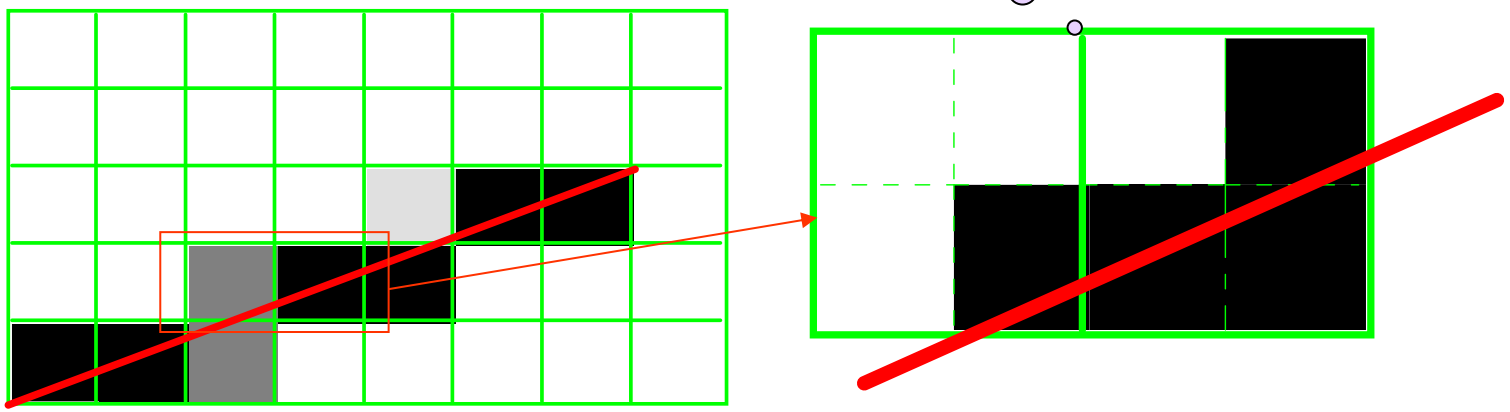


确定灰度值的方法：

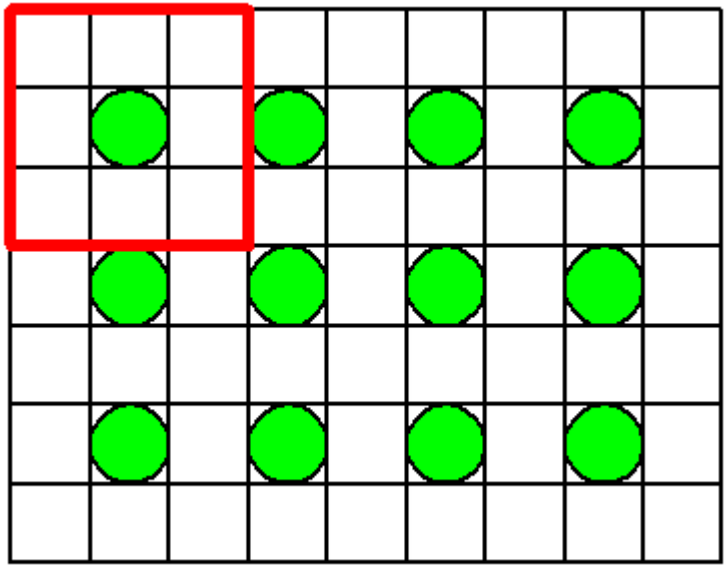
- 过取样
- 区域取样

简单的过取样

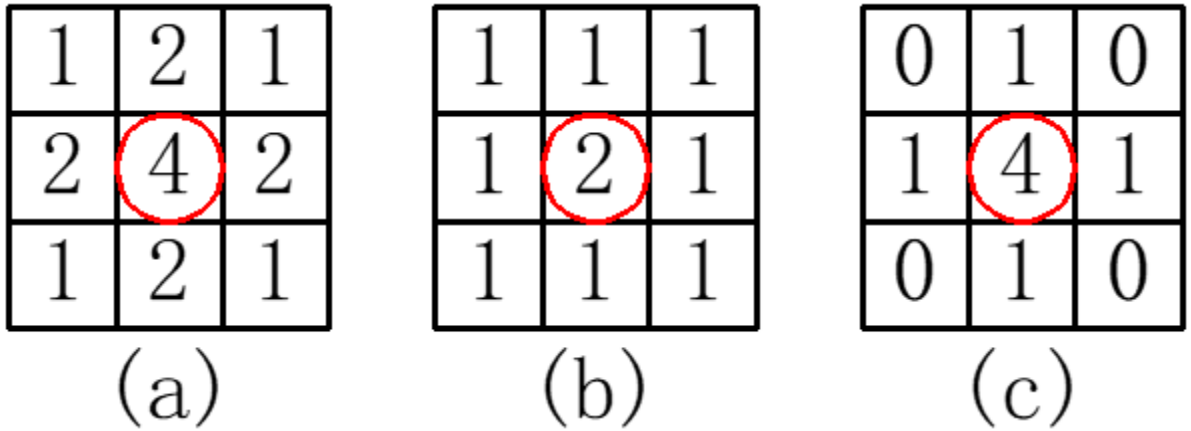
采样频率是实际频率的两倍



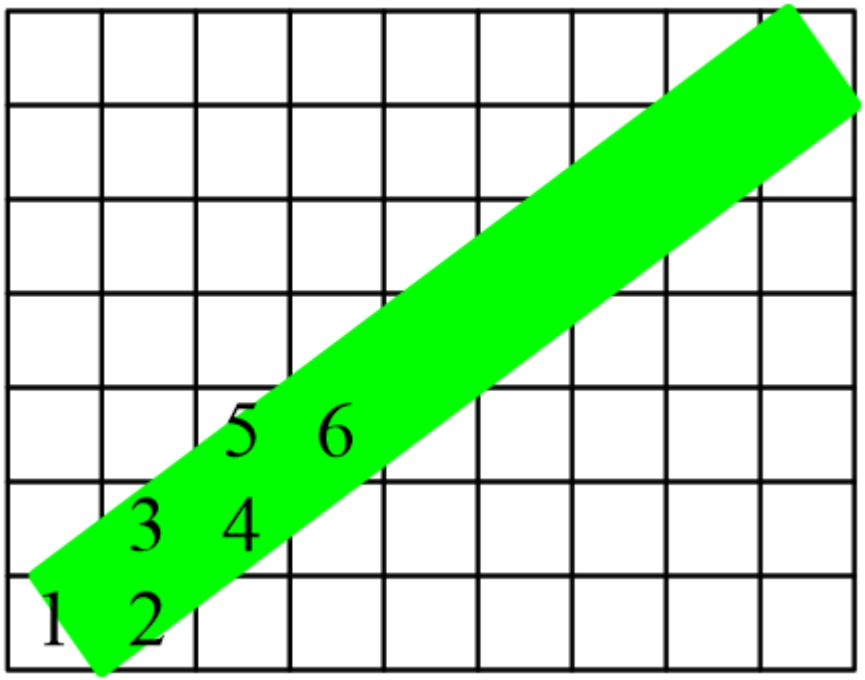
重叠过取样



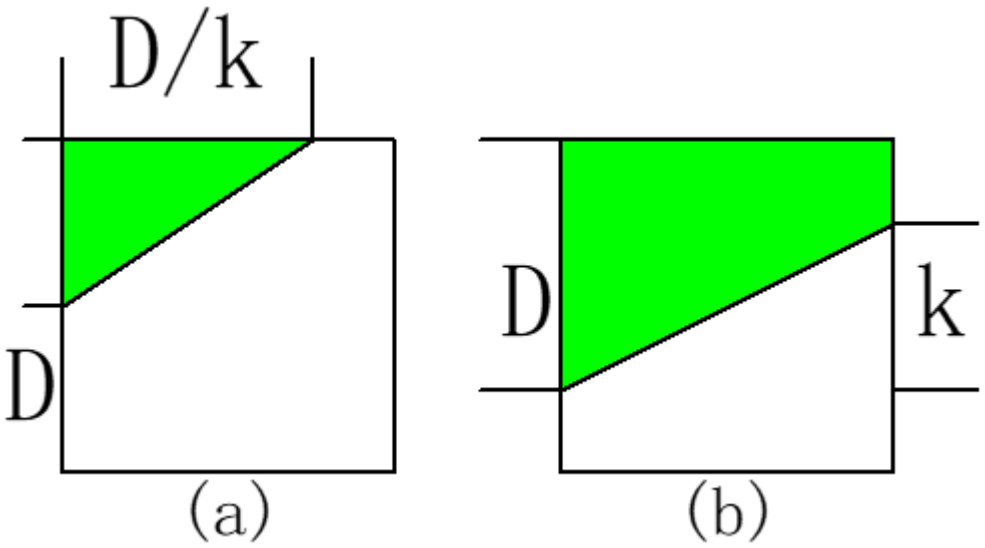
基于加权模板的过取样



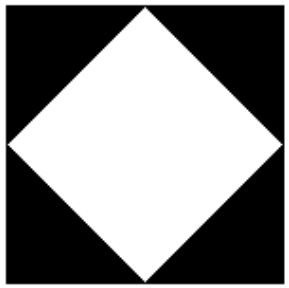
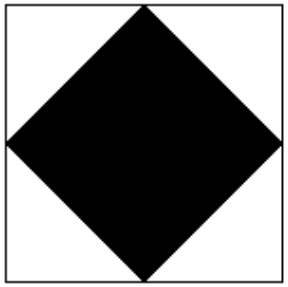
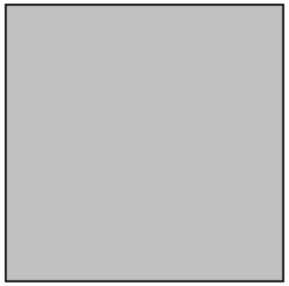
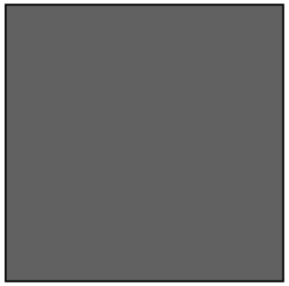
简单的区域取样



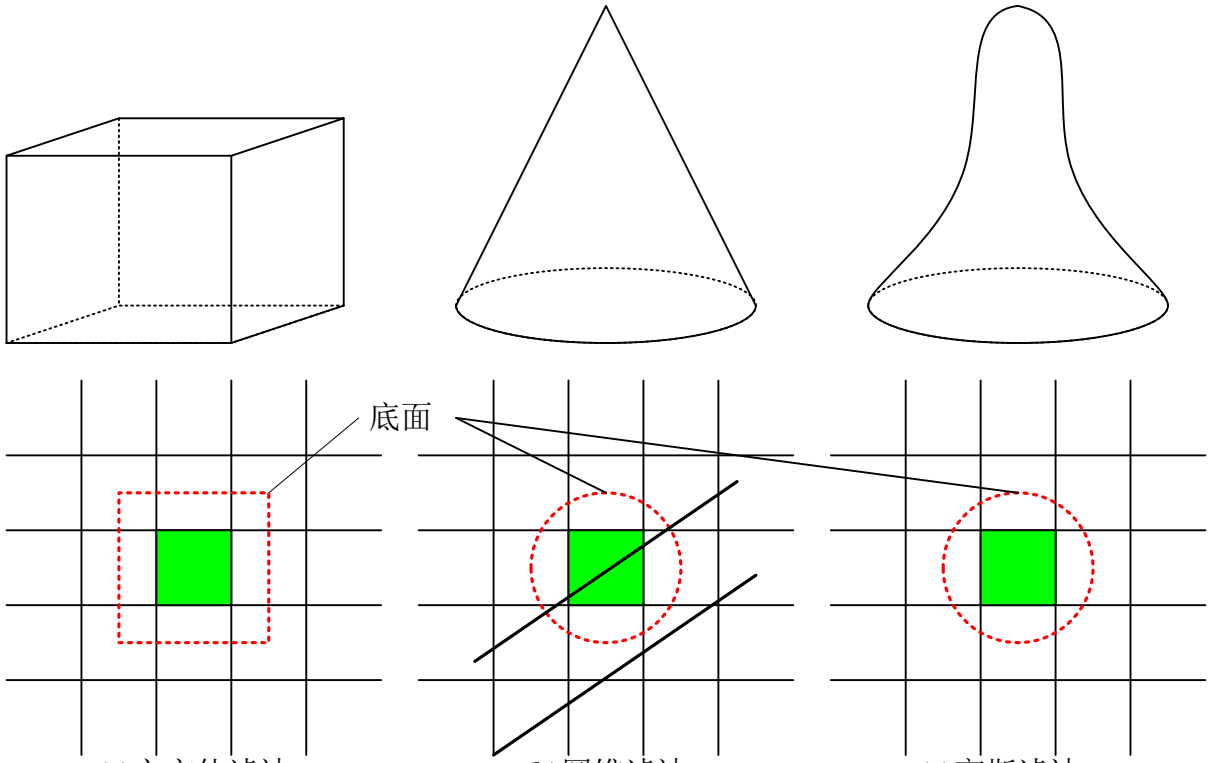
简单的区域取样



加权区域取样

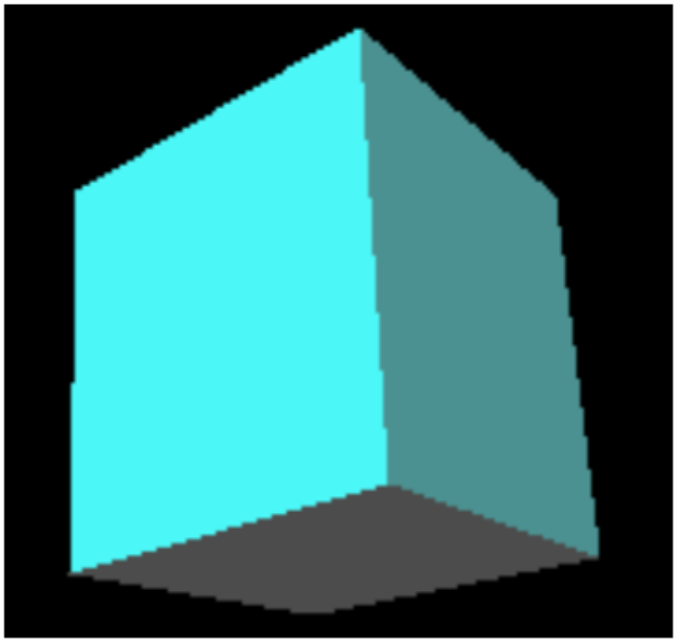
加权区域取样



(a)立方体滤波　(b)圆锥滤波　(c)高斯滤波

图5-57 常用的过滤函数

反走样



原 图

反走样图

小结

小结

- 直线的扫描转换
- 多边形的扫描转换
- 多边形的区域填充
- 反走样

直线段的扫描转换算法

（1）DDA算法主要利用了直线的斜截式方程（y=kx+b），在这个算法里引进了增量的思想，结果把一个乘法和加法变成一个加法

（2）中点法是采用的直线的一般式方程，也采用了增量的思想，比DDA算法的优点是采用了整数加法

（3）Bresenham算法也采用了增量和整数算法，优点是这个算法还能用于其它二次曲线

多边形的扫描转换算法

如何把边界表示的多边形转换成由像素逐点描述的多边形
这是二维图形显示的基础

有四个步骤：求交、排序、配对、填色。这里引进了一个新的思想—图形的连贯性。手段就是利用增量算法和特殊的数据结构（多边形y表、边y表、活化多边形表、活化边表），2个指针数组和2个指针链表

反走样

因为用离散量表示连续量，有限的表示无限的自然会导致一些失真，这种现象称为走样

反走样主要有三种方法：

提高分辨率

区域采样

加权区域采样。

反走样

提高分辨率无非是把分辨率增加，这样可以提高反走样的效果；但这个方法是有物理上的限制的—分辨率不能无限增加

区域采样算法是在关键的直线段、关键的区域上绘制的时候并非非黑即白，可以把关键部位变得模糊一点，有颜色的过渡区域，这样会产生一种好的视觉效果

加权区域是不但要考虑区域采样，而且要考虑不同区域的权重，用积分、滤波等技巧来做

谢　谢