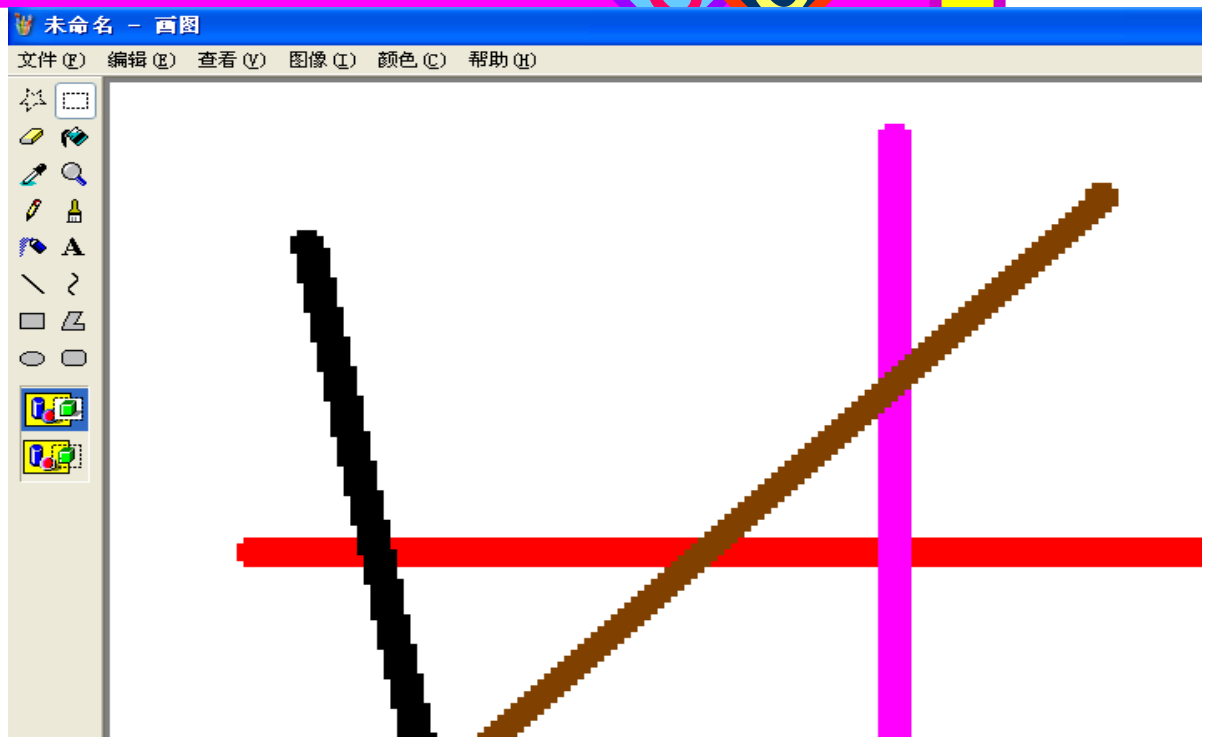
第3章 基本光栅图形生成技术

线的生成算法

区域填充

用OpenGL生成基本图形

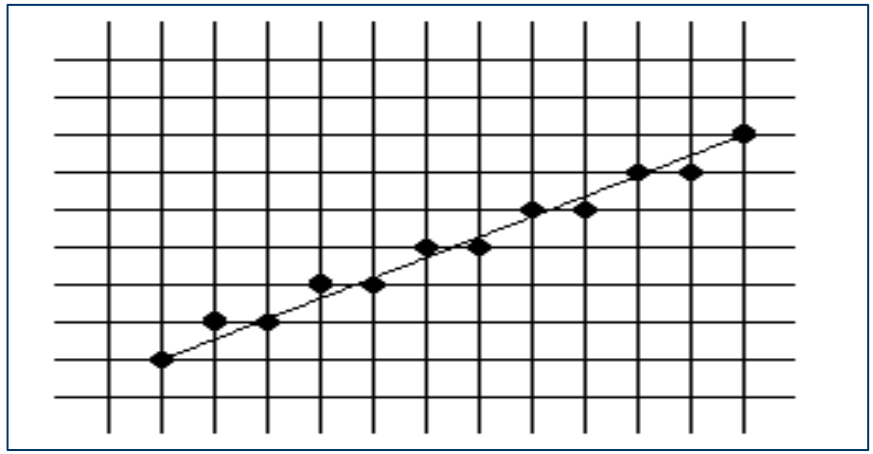
“画笔”程序



- 为什么有时绘制的直线会出现锯齿现象？
- 如何填充区域（如何区分内外部区域）？

直线的生成算法

1.扫描转换直线段： 求与直线段充分接近的像素集



2.三点假设

–直线段的宽度为1

–直线段的斜率：$m \in [-1,1]$

–有底层画点函数:SetPixel(*x*,*y*,*color*)）

条件

- 待扫描转换的直线段：$P_0(x0, y0)$、$P_1(x1, y1)$
- 斜率：$m = \Delta y / \Delta x$　　$y_0 = m * x_0 + b$

$$\Delta x = x1 - x0, \quad \Delta y = y1 - y0$$

- 直线方程：

$$y = m \bullet x + b$$

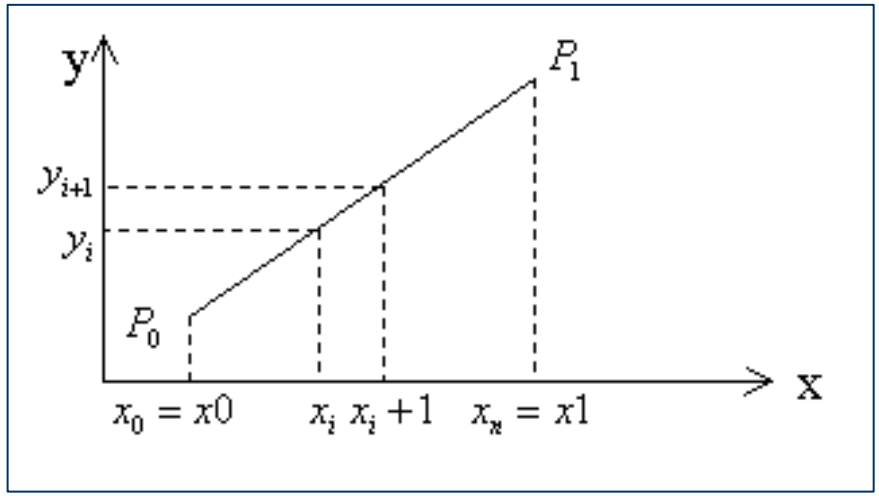
如何求表示直线段$\mathbf{P_0P_1}$的像素集？

利用直线方程直接计算

$m \in [-1,1]$

- 划分区间[x0,x1]:

$x_0, x_1, \cdots, x_n$，其中$x_{i+1} = x_i + 1$



- 计算纵坐标：

$$y_i = m \bullet x_i + b$$

$$\{(x_i, y_i)\}_{i=0}^{n} \longrightarrow \{(x_i, y_{i,r})\}_{i=0}^{n}$$

- 取整：why?

$$y_{i,r} = round(y_i) \quad = (\text{int})(y_i + 0.5)$$

利用直线方程直接计算

算法的复杂度

乘法

加法

取整

浮点数！准确度、存放

循环体内每次进行一次乘法和取整运算，还有浮点数，效率低下！

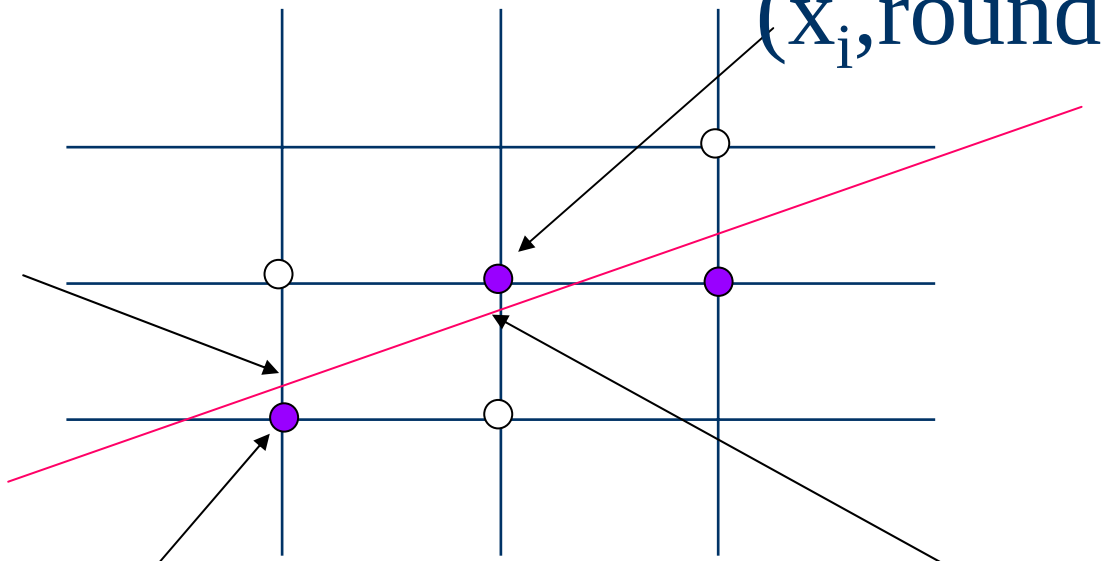
```
int dY=Y1-Y0,dX=X1-X0;
float m=dY/dX;b=Y0-m*X0;
for(int i=0;i<dX;i=i+1)
{
y=int(m*x+b+0.5); setpixel(x,y,color);
x=x+1;
}
```

数值微分(DDA)法

|m|=|y1-y0|/|x1-x0|<1

$$y_{i+1} = k \bullet x_{i+1} + b = k \bullet (x_i + \Delta x) + b = k \bullet x_i + b + k \bullet \Delta x = y_i + k \bullet \Delta x$$

△x＝1时…



增量算法

复杂度：加法+取整

适用于所有象限的DDA算法（p20-21）

```
void DDALine(int x0,int y0,int x1,int y1,int color)
{
int i;
float dx, dy, length,x,y;
if (fabs(x1-x0)>=fabs(y1-y0))
        length=fabs(x1-x0);
else
        length=fabs(y1-y0);
dx = (x1-x0)/length;
dy=(y1-y0)/length;
i=1;x= x0;y= y0;
while(i<=length)
{
SetPixel (int(x+0.5), int(y+0.5), color);
   x=x+dx;
   y=y+dy;
   i++;
   }
}
```

DDA算法与基本算法相比，减少了浮点乘法，提高了效率。但是x与dx、y与dy用浮点数表示，每一步要进行四舍五入后取整，不利于硬件实现，因而效率仍有待提高。

example1

$m=dy/dx=2/5=0.4$

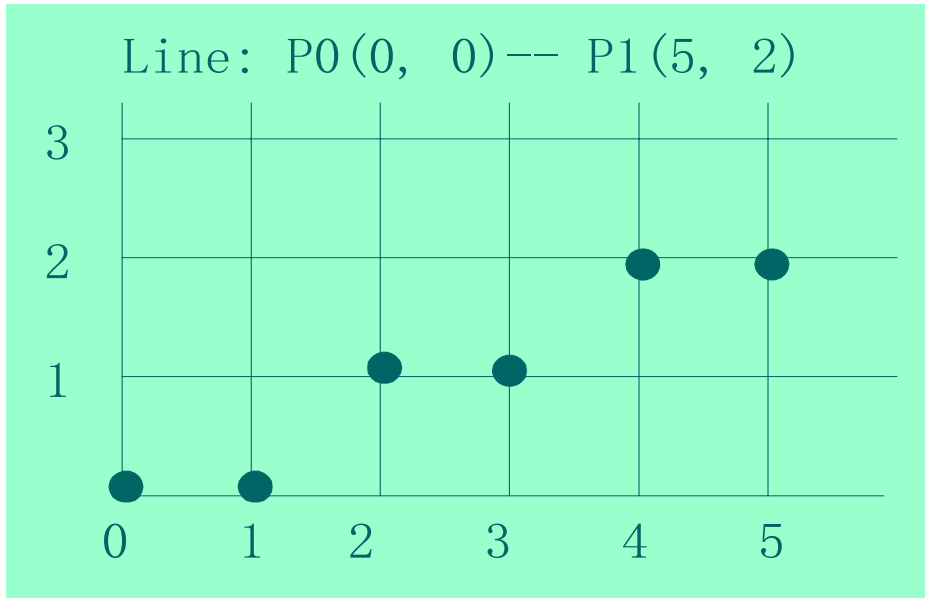
举例：画直线段P0(0,0)--P1(5,2)

$y_{i+1}=y_i+m\ \Delta x$

实际：$y_{i+1}=round(y_i+m)$

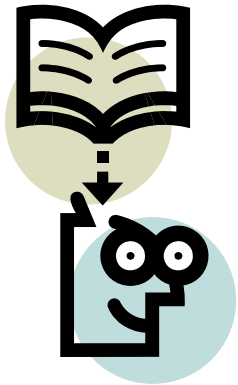
$=int(y_i+m+0.5)$

x	y	int(y+0.5)	y+0.5
0	0	0	0+0.5
1	0.4	0	0.4+0.5
2	0.8	1	0.8+0.5
3	1.2	1	1.2+0.5
4	1.6	2	1.6+0.5
5	2.0	2	2.0+0.5

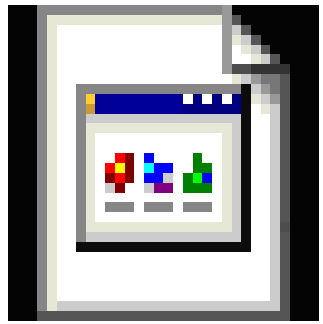


注：网格点表示像素

P0(0,0)--P1(5,4)？？



如何用上述方法生成斜率 $|m|>1$ 的直线段？



DDA扫描转换直线演示

Bresenham算法

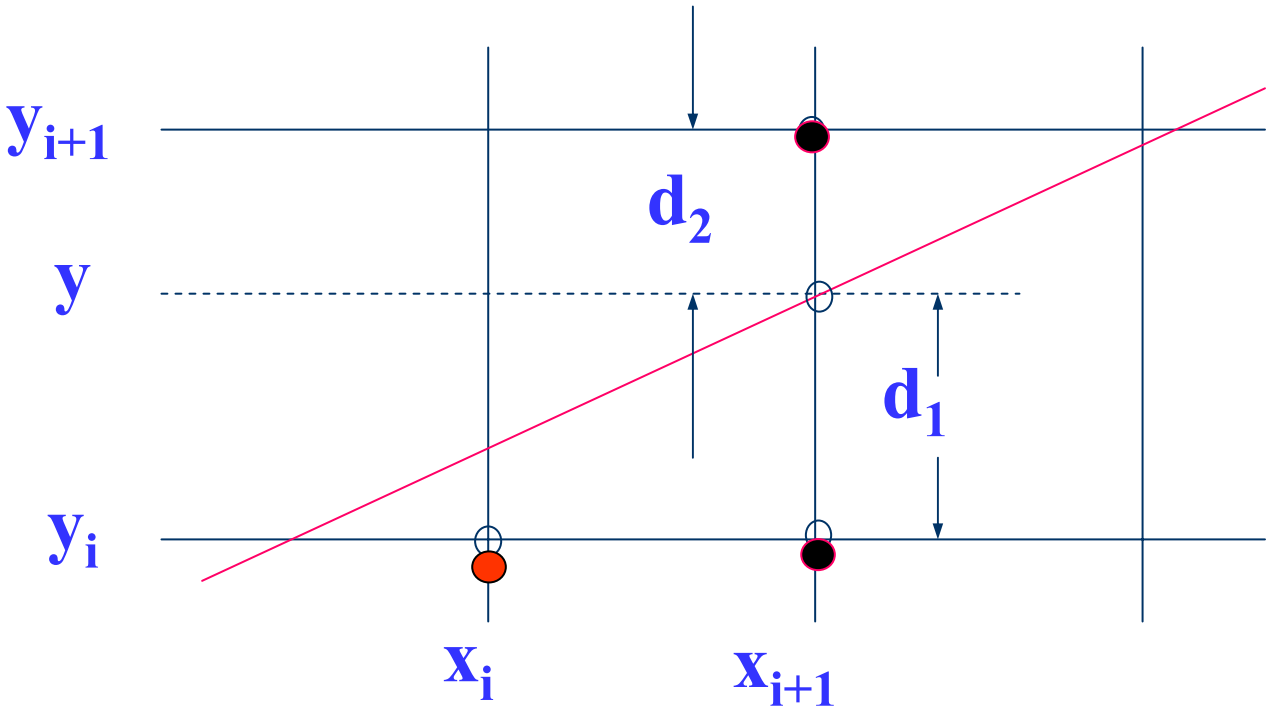
基本思想

Bresenham算法**1965**年提出，基本原理是：借助于一个误差量(直线与当前实际绘制像素点的距离)，来确定下一个像素点的位置。算法的巧妙之处在于采用增量计算，使得对于每一列，只要检查误差量的符号，就可以确定该下一列的像素位置。

应用最广泛的直线扫描转换算法！！

Bresenham算法

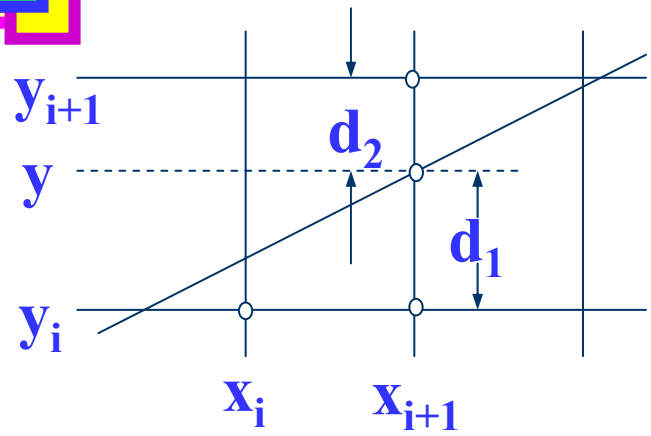
如图所示，对于直线斜率k在0~1之间的情况，从给定线段的左端点$P_0(x_0, y_0)$开始，逐步处理每个后续列(x位置)，并在扫描线y值最接近线段的像素上绘出一点。



$$d_1 = y - y_i = (k（x_i + 1) + b) - y_i$$

$$d_2 = (y_i + 1) - y = y_i + 1 - (k（x_i + 1) + b)$$

$$d_1 - d_2 = 2k(x_i + 1) - 2y_i + 2b - 1$$



设 $\Delta y = y_1 - y_0$，$\Delta x = x_1 - x_0$，则 $k = \Delta y / \Delta x$，代入上式，得：

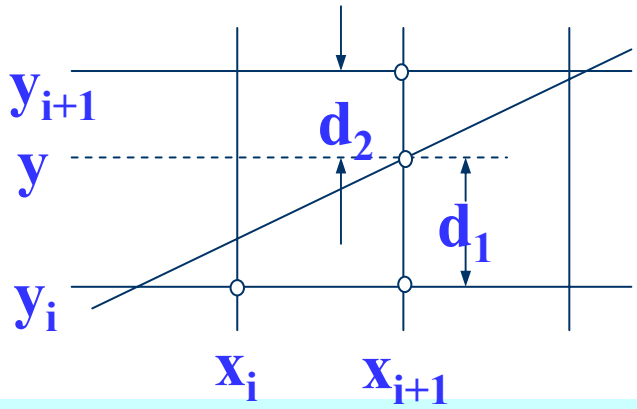
$$\Delta x(d_1 - d_2) = 2 \cdot \Delta y \cdot x_i - 2 \cdot \Delta x \cdot y_i + c$$

$c = 2\Delta y + \Delta x(2b - 1)$ 是常量，与像素位置无关

令 $d_i = \Delta x(d_1 - d_2)$

令$d_i = \Delta x(d_1 - d_2)$

$\Delta x = x1 - x0 > 0$



则d_i的符号与(d_1-d_2)的符号相同。

✓当$d_i<0$时，直线上理想位置与像素(x_i+1, y_i)更接近，应取右方像素；

✓当$d_i>0$时，右上方像素(x_i+1, y_i+1)与直线上理想位置更接近；

✓当$d_i=0$时，两个像素与直线上理想位置一样接近，可约定取(x_i+1, y_i+1)：右上方像素。

$$d_i = \Delta x(d_1 - d_2) = 2\cdot\Delta y\cdot x_i - 2\cdot\Delta x\cdot y_i + c$$

对于**k+1**步，误差 $d_{i+1} = 2\cdot\Delta y\cdot x_{i+1} - 2\cdot\Delta x\cdot y_{i+1} + c$

$$d_{i+1} - d_i = 2\cdot\Delta y\cdot(x_{i+1} - x_i) - 2\cdot\Delta x\cdot(y_{i+1} - y_i)$$

因为$x_{i+1}=x_i+1$ $d_{i+1} = d_i + 2\cdot\Delta y - 2\cdot\Delta x\cdot(y_{i+1} - y_i)$

如果选择右上方像素，$y_{i+1} - y_i = 1$，则：$d_{i+1} = d_i + 2\Delta y - 2\Delta x$

如果选择右方像素，$y_{i+1} = y_i$，则：$d_{i+1} = d_i + 2\Delta y$

对于每个整数x，从线段的坐标端点开始，循环的进行误差量的计算。在起始像素(x_0，y_0)的第一个参数d_0为：

$$d_0 = 2\Delta y - \Delta x$$

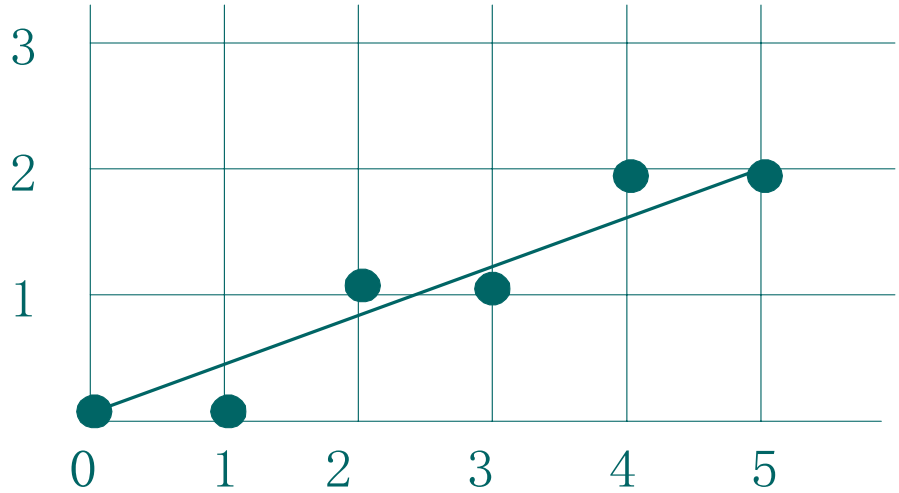
d_0在后面的程序中称为e

Bresenham画线算法实例

例：Line: $P_0(0, 0)$, $P_1(5,2)$　k=dy/dx=0.4　dy=2　dx=5

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

$$e_0 = 2dy - dx = -1$$



$e_i<0, e_{i+1}=e_i+2dy$

$e_i>0, e_{i+1}=e_i+2dy-2dx$

$e_i<0$,取右方；

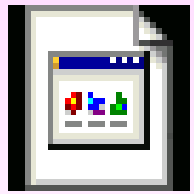
$e_i \geqslant 0$,右上方

Bresenham算法

以下是当0<k<1时的Bresenham画线算法程序：

```
void Bresenham_Line (int x0,int y0,int x1, int y1,int color)
{
   int dx,dy,e,i,x,y;
   dx = x1-x0;   dy = y1- y0;   e=2*dy-dx;
   x=x0;   y=y0;
   for (i=0; i<=dx; i++)
   {
            SetPixel (x, y, color);
            if (e>=0)
            {
              y++;
              e=e-2*dx;
            }
             x++;
             e=e+2*dy;
    }
}
```

只有加法和判断了！



Bresenham扫描转换直线演示

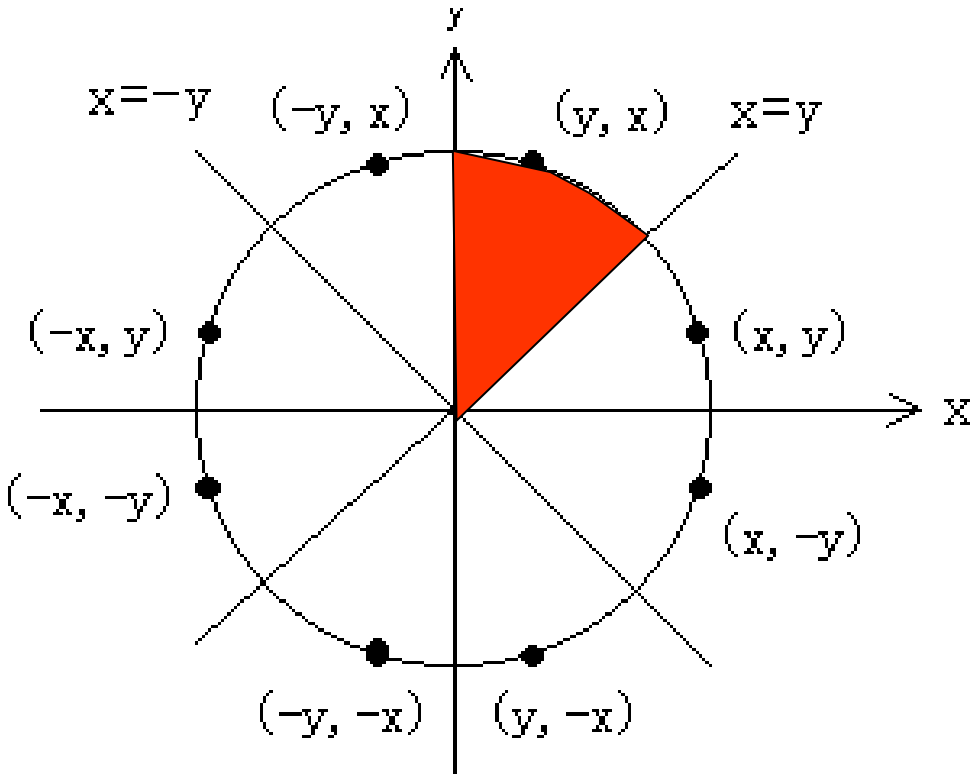
纠正教材p33的错误！

- 第二次上机作业：
  - **p75习题3.1，增加DDA方法: DDA，Bresenham方法在一个窗口各画2条线－－上机作业2**
  - 选作：**p75习题3.5**
  - 思考：如何画出一个坐标系，并用适当大小的圆表示画出的点？然后演示画线程序？

- ftp://192.168.83.240,
- 学生用户名:cq 密码:1234

扫描转换圆弧算法

- 处理对象：圆心在原点的圆弧
- 利用圆的八对称性（教材**p34**）



- 前提条件：只考虑中心在原点、半径为整数R的圆。
- **中心不在原点的圆，可通过以下三个步骤来绘制：**

①首先通过平移变换，将圆化为中心在原点的圆；

②然后再进行扫描转换，得到圆心在坐标原点的像素集合；

③最后把像素集合中每一个像素坐标加上一个位移量即得所需绘制圆的像素坐标。

圆弧基本绘制方法

- 两种直接离散方法：

  1/离散点：利用隐函数方程 $x^2 + y^2 = R^2$

  $$(x_i, y_i = \sqrt{R^2 - x_i^2}) \xrightarrow{\text{取整}} (x_i,\ y_{i,r})$$

  2/离散角度：利用参数方程 $\begin{cases} x = R\cos\theta \\ y = R\sin\theta \end{cases}$

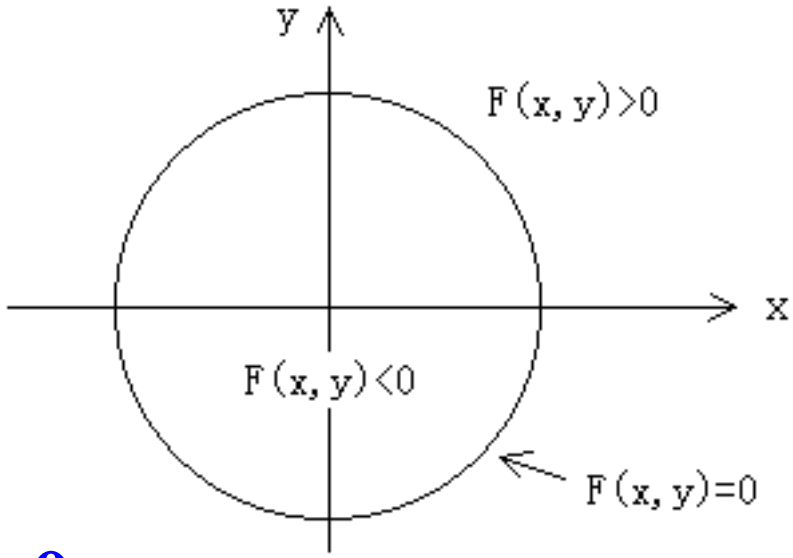
  $$(round(R\cos\theta_i), round(R\sin\theta_i))$$

- 开根，三角函数运算，计算量大，不实用。

圆弧中点画法的引出

圆弧的正负划分性：构造函数

$$F(x,y)=x^2+y^2-R^2=0$$

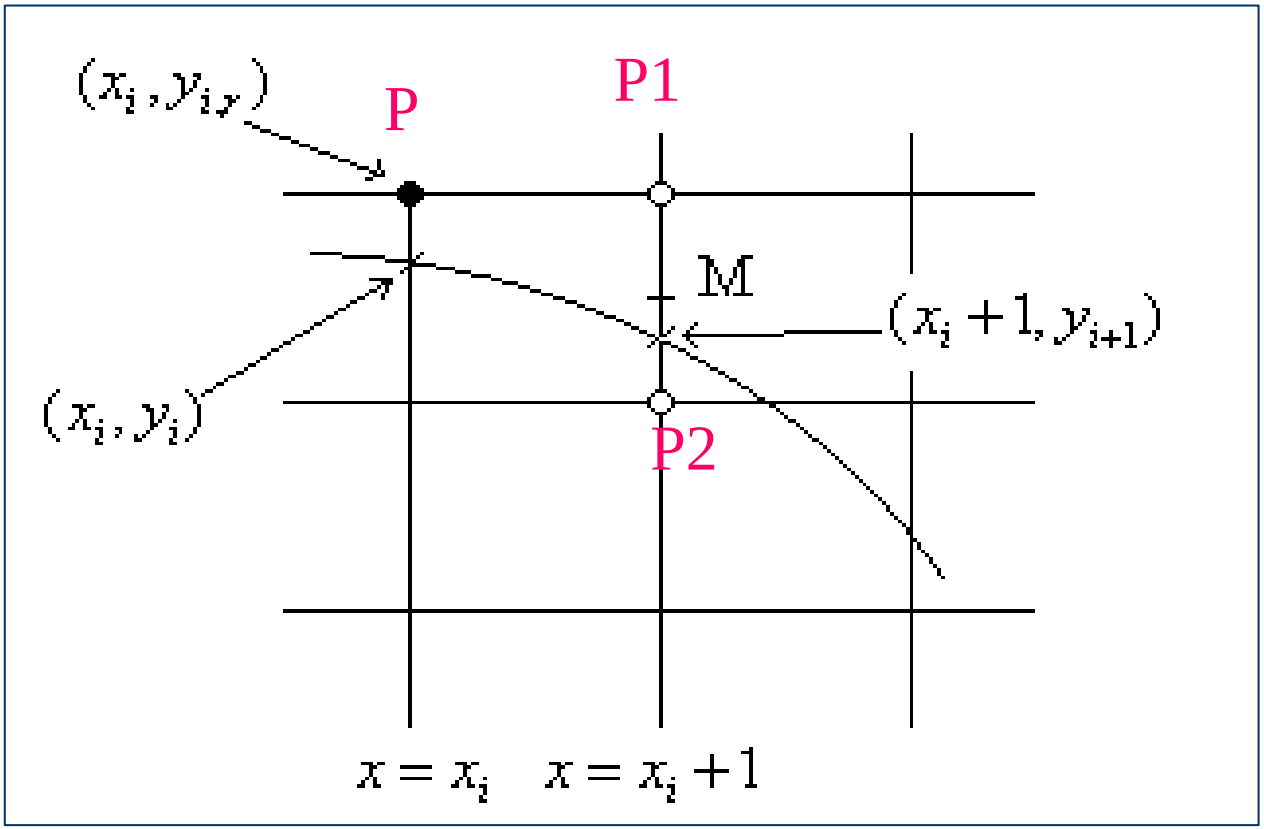


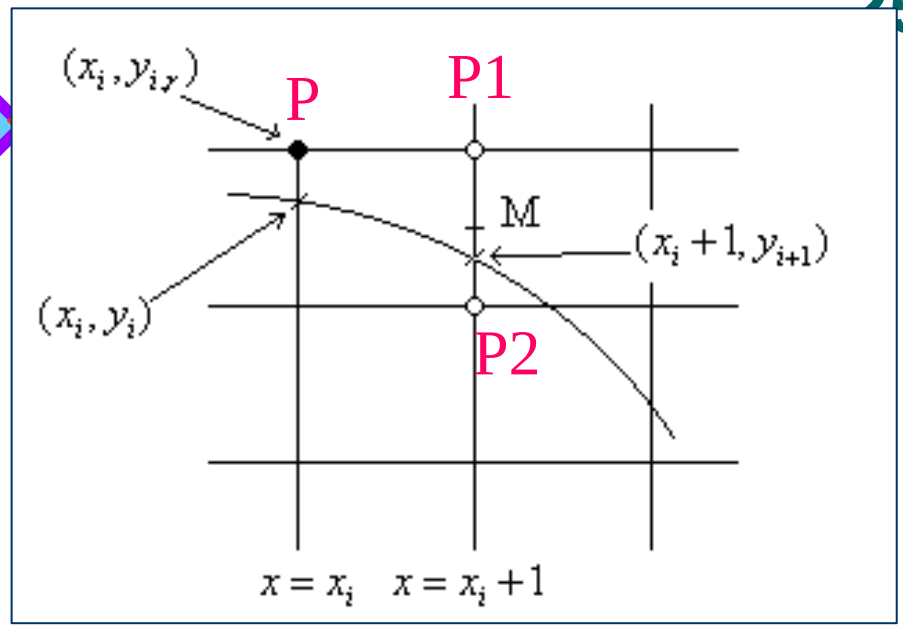
圆弧外的点：**F(X，Y)＞0**

圆弧内的点：**F(X，Y)＜0**

圆弧中点算法

考虑对象： 第一象限的八分之一圆弧





- 问题：
- 圆弧的隐函数： $F(X,Y)=X^2+Y^2-R^2=0$

  切线斜率min [-1,0]
- 中点： M=(Xp+1,Yp-0.5),如何取P1 or P2?

  当F(M)＜0时，M在圆内，说明P1距离圆弧更近，取P1;

  当F(M)＞0时，M在圆外，说明P2距离圆弧更近，取P2;

  当F(M) = 0时，可在P1和P2中任选1个，这里约定P2.

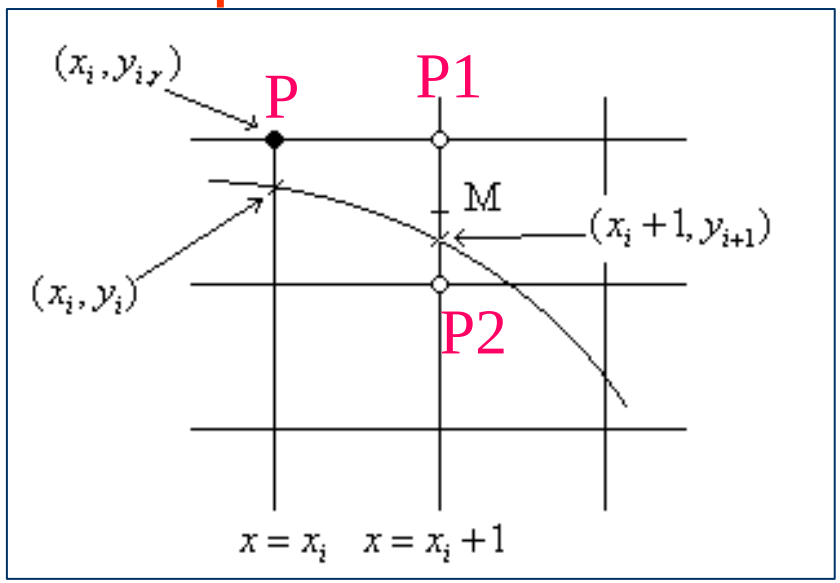
- 构造判别式

$$d_p=F(M)=F(X_p+1,Y_p-0.5)=(X_p+1)^2+(Y_p-0.5)^2-R^2$$

1）若$d_p<0$，取P1，再下一个像素的判别式为：

$$d_{p+1}=F(X_p+2,Y_p-0.5)=d_p+2X_p+3,$$

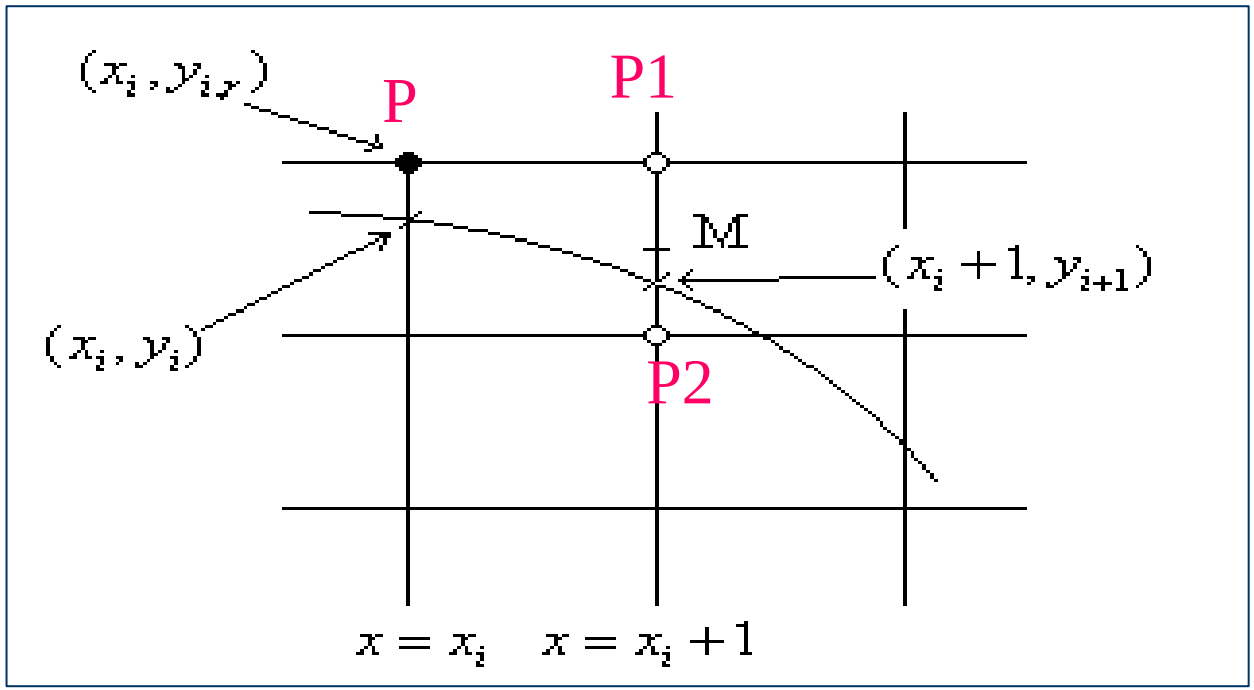
沿正右方向，d的增量为$2X_p+3$；



2）若d≥0，取P2，再下一个像素的判别式为：

$$d_{p+1}=F(X_p+2,Y_p-1.5)=d_p+(2X_p+3)+(-2Y_p+2)$$

沿右下方向，d的增量为2(Xp-Yp)+5



- d的初始值(在第一个像素(0,R)处),

  $d_0=F(1, R-0.5)=1.25-R$

- 算法中有浮点数，故用e=d-0.25代替，即：

  d = 1.25-R 对应 e = d-0.25 = 1-R;

  判别式 d < 0 对应于 e < -0.25;

- e的初值为整数,且运算中增量也为整数，故e始终为整数，所以e < -0.25可以用e < 0来代替。

- 结论：
  - if(e<0),e+=2*x+3;x++;
  - if(e>=0), e += 2*(x-y)+5 ;x+ +;y- -;

- 代码如下：

Circlepoints函数的定义在P34

```
MidpointCircle(r,color)
int r, color;
{
  int x,y,e;
  x = 0; y = r; e = 1-r;
  circlepoints(x,y,color);
  while( x < y)
   {  if (e <0)
         { e += 2*x+3; x++;  }
      else
         { e += 2*(x-y)+5;
           x++ ; y--;  }
    circlepoints(x,y,color);
    }
}
```

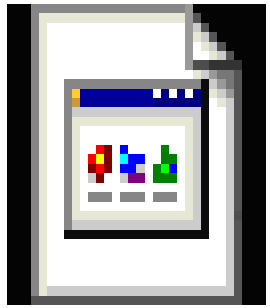
思考：本程序与p75例3.4的区别？

显示圆弧上的八个对称点的算法：

```
void CirclePoints ( int x, int y, int cx, int cy, COLORREF color)
//cx,cy为圆心的位置
{
    CDC *pDC = this->GetDC( );
    pDC->SetPixel(x+cx,y+cy,color);
    pDC->SetPixel(x+cx,-y+cy,color);
    pDC->SetPixel(-x+cx,y+cy,color);
    pDC->SetPixel(-x+cx,-y+cy,color);
    pDC->SetPixel(y+cx,x+cy,color);
    pDC->SetPixel(y+cx,-x+cy,color);
    pDC->SetPixel(-y+cx,x+cy,color);
    pDC->SetPixel(-y+cx,-x+cy,color);
    this->ReleaseDC(pDC);
}
```

程序只有简单加法和整数运算

思考：本程序与p34程序的区别？



中点法扫描转换圆弧演示

其它圆弧画法：

Bresenham…

中点画圆法可以推广到一般二次曲线的生成

3．椭圆的绘制

对于一般位置的椭圆，例如，

$$(x-x_c)^2/a^2+(y-y_c)^2/b^2=1$$

可将中心平移到坐标原点，确定好中心在原点的标准位置的椭圆像素点集后，再平移到位置，将问题转变为标准位置的椭圆的绘制问题。

3．椭圆的绘制

如果椭圆的长轴和短轴方向不与坐标轴**x**和**y**平行，那么可以采用旋转坐标变换的方式，同样将问题转变为标准位置的椭圆的绘制，

$$x^2/a^2+y^2/b^2=1$$

定义下面椭圆中点算法的判别式：

$$F(x,y)=b^2x^2+a^2y^2-a^2b^2=0 \qquad \textbf{(3-3)}$$

???

则：

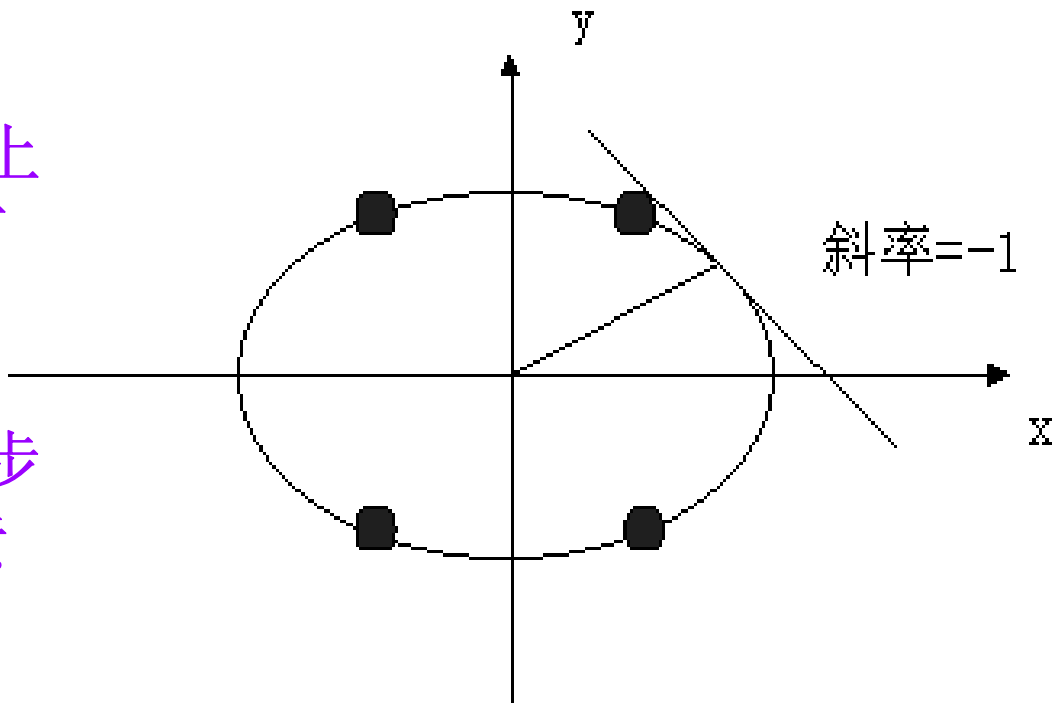
若 $F(x,y)<0$ ，说明**(x，y)**在椭圆边界内

若 $F(x,y)=0$ ，说明**(x，y)**在椭圆边界上

若 $F(x,y)>0$ ，说明**(x，y)**在椭圆边界外

由于椭圆的对称性，这里只讨论第一象限椭圆弧的生成。在处理这段椭圆弧时，进一步把它分为两部分：上部分和下部分，以弧上斜率为-1的点作为分界：

在上部分，在x方向上取单位步长，确定下一像素的位置；
在斜率小于-1的下部分，在y方向取单位步长，来确定下一像素的位置。



椭圆的斜率可从方程**(3-3)**中计算出来：

$$dy/dx = -2b^2x/2a^2y$$

在上部分和下部分的交界处，$dy/dx = -1$

则上式为；

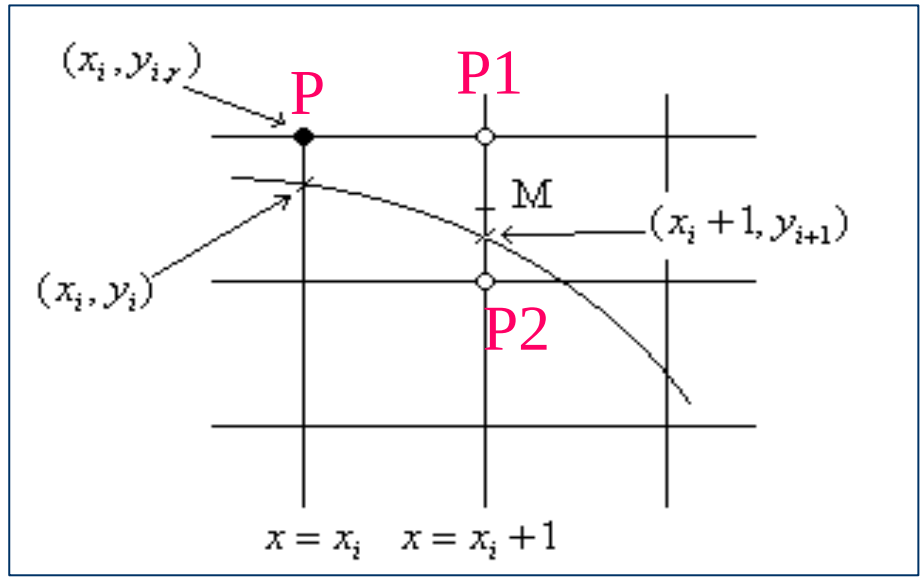
$$2b^2x = 2a^2y$$

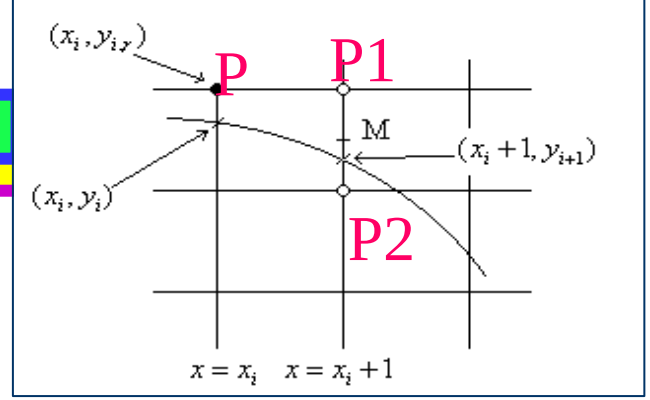
椭圆的绘制思路与中点画圆算法类似，当我们确定一个像素后，接着在两个候选像素的中点计算一个判别式的值。并根据判别式符号确定两个候选像素哪个离椭圆更近。

首先讨论椭圆弧的上部分，假设当前已确定的椭圆弧上的像素点为 (x_p, y_p)，那么下一对候选像素的中点是 $(x_p+1, y_p-0.5)$ 。因此判别式为

$$d_p = F(x_p+1, y_p-0.5) = b^2(x_p+1)^2 + a^2(y_p-0.5)^2 - a^2b^2$$

它的符号决定下一个像素是取正右方的那个像素，还是右下方的那个像素。





若 $d_p < 0$ ，中点在椭圆内，则应取正右方像素，且判别式更新为

$$d_{p+1} = F(x_p + 2, y_p - 0.5) = b^2(x_p + 2)^2 + a^2(y_p - 0.5)^2 - a^2b^2$$

$$= (b^2(x_p + 1)^2 + a^2(y_p - 0.5)^2 - a^2b^2) + b^2(2x_{p+1} + 1) = d_p + b^2(2x_{p+1} + 1)$$

因此，往正右方向，判别式的增量为

$$b^2(2x_{p+1} + 1)$$

若 $d_p > 0$ ，中点在椭圆外，则应取右下方像素，且判别式更新为

$$d_{p+1} = F(x_p + 2, y_p - 1.5) = b^2(x_p + 2)^2 + a^2(y_p - 1.5)^2 - a^2b^2$$

$$= (b^2(x_p + 1)^2 + a^2(y_p - 0.5)^2 - a^2b^2) + b^2(2x_{p+1} + 1) - 2a^2y_{p+1}$$

$$= d_p + b^2(2x_{p+1} + 1) - 2a^2y_{p+1}$$

因此，沿右下方向，判别式 d_1 的增量为

$$b^2(2x_{p+1} + 1) - 2a^2y_{p+1}$$

d_p 的初始条件是：根据弧起点(0，b)，因此，第一个中点是(1，b-0.5)，对应的判别式是

$$d_{p0} = F(1, b-05) = b^2 + a^2(b-0.5)^2 - a^2b^2 = b^2 + a^2(-b+0.25)$$

中点椭圆绘制算法的程序如下。其中，每步迭代过程中，需要随时计算和比较从上部分转入下部分的条件是否成立，从而将步进方向由x改为y。

```
void MidpointEllipse(int xc,int yc,
int a, int b, int color)
{
        float aa=a*a, bb=b*b;
        float twoaa=2*aa,
twobb=2*bb;
        float x=0,y=b;
        float d;
        float dx=0;
        float dy=twoaa*y;
        d=int(bb+aa*(-
b+0.25)+0.5);
        SetPixel(xc+x,yc+y,color);
        SetPixel(xc+x,yc-y,color);
        SetPixel(xc-x,yc+y,color);
        SetPixel(xc-x,yc-y,color);
```

```
While(dx<dy)
{       x++;
        dx++twobb;
        if(d<0) d+=bb+dx;
        else
        {  dy-=twoaa;
           d+=bb+dx-dy;
           y--;
        }
       SetPixel(xc+x,yc+y,color);
        SetPixel(xc+x,yc-y,color);
        SetPixel(xc-x,yc+y,color);
        SetPixel(xc-x,yc-y,color);
}
d=int(bb*(x+0.5)*(x+0.5)+aa*(y
-1)*(y-1)-aa*bb+0.5);
```

```
        while(y>0)
        {       y--;
                dy-=twoaa;
                if(d>0)
                        d+=aa-dy;
                else
                {       x++;
                        dx+=twobb;
                        d+=aa-dy+dx;
                }
                SetPixel(xc+x,yc+y,color);
                SetPixel(xc+x,yc-y,color);
                SetPixel(xc-x,yc+y,color);
                SetPixel(xc-x,yc-y,color);
        }
}
```

上机作业第三次

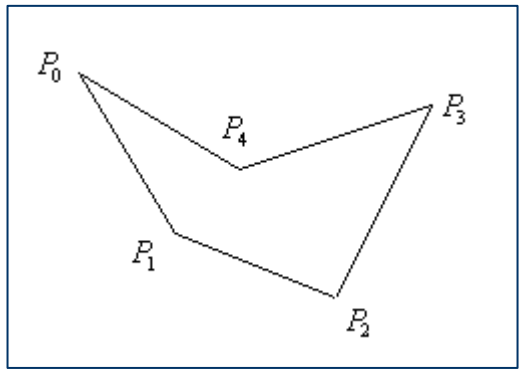
- 继续完成**DDA**和**Bresenham**画直线的程序；
  - **＃include “math.h”——fabs**函数
  - **this->GetDC( )->SetPixel(x,y,color)**
  - **DDA(int x_Start, int y_Start,int x_End,int y_End, COLORREF color)**
  - **.h** 与 **.cpp**？；函数如何定义？
  - 看不清楚要提问；不要抄袭！
  - 工程（**project**）以自己名字学号全拼来命名；
  - 程序打包:注意删除**debug**目录下的文件；
  - 注意发挥：调试、增加变量、提高适应性、画棋盘格？

上机作业第四次

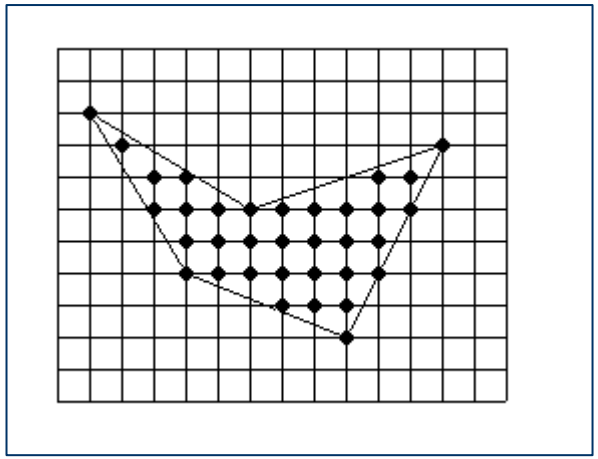
- 完成中点画圆法的程序：**p34，36**
  - 添加相应函数（比如**MidPointCircle（int r, COLORREF color）,CirclePoint(int x,int y,CORLORREF color)**
  - 圆心不在原点时如何处理？添加进第一个函数？
  - 由于**DC**坐标轴原点在左上角，只有**1/4**圆能画出，其余部分呢？
- 同样当场验收。
- 注意看编译时的英文提示。

区域填充

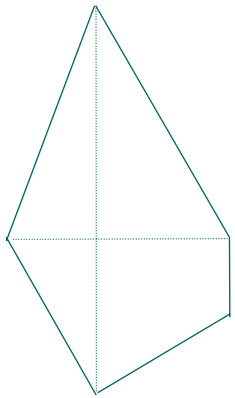
- 多边形区域的表示方法
  - 顶点表示

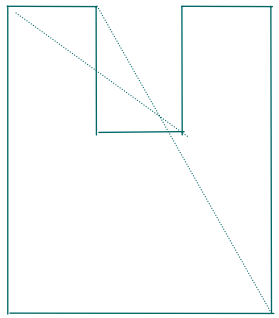


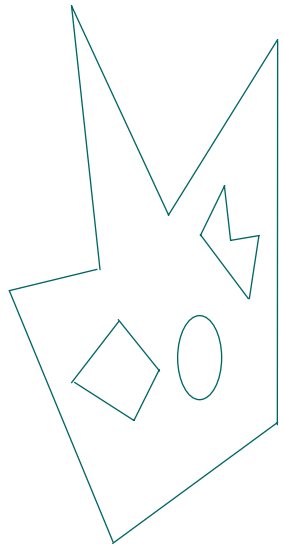
  - 点阵表示



- 多边形分为凸多边形、凹多边形、含内环的多边形。







最简单的区域填充算法是检查屏幕上的每一个像素是否位于区域多边形内(?)。由于大多数像素不在多边形内，因此该算法的效率很低

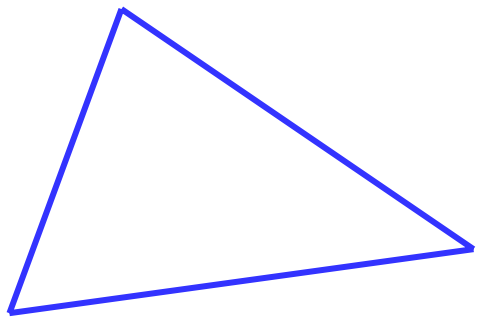
根据区域的定义，可以采用不同的填充算法

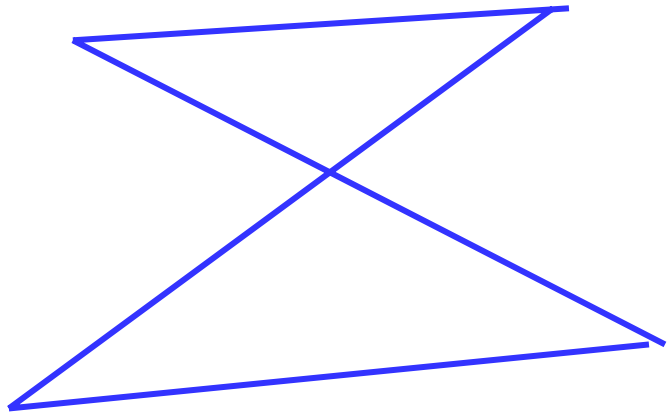
用于顶点表示的扫描线类算法

用于点阵表示的种子填充算法

扫描线算法

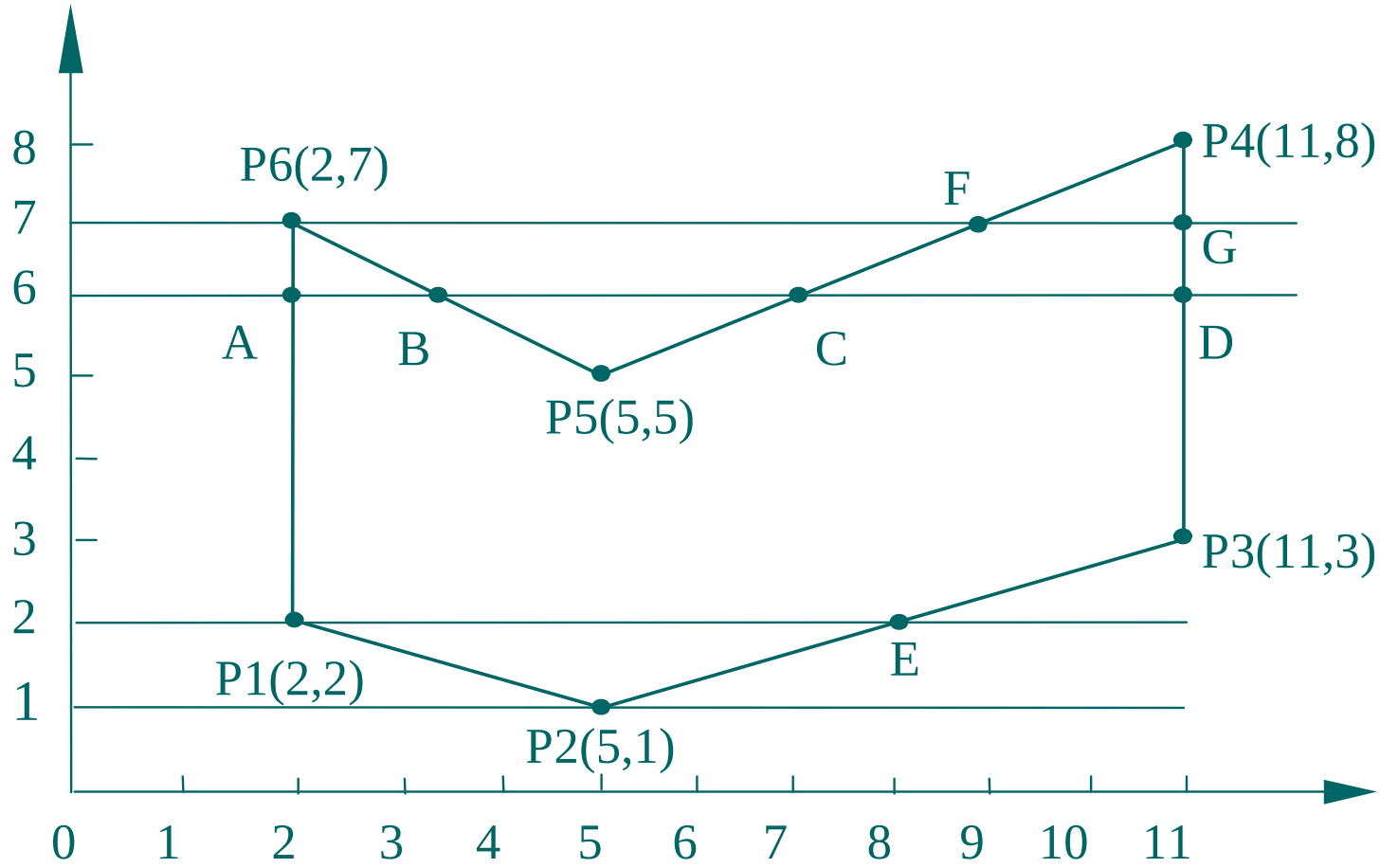
- 扫描线算法
  - 目标：利用相邻像素之间的连贯性，提高算法效率
  - 处理对象：非自交多边形 （边与边之间除了顶点外无其它交点）



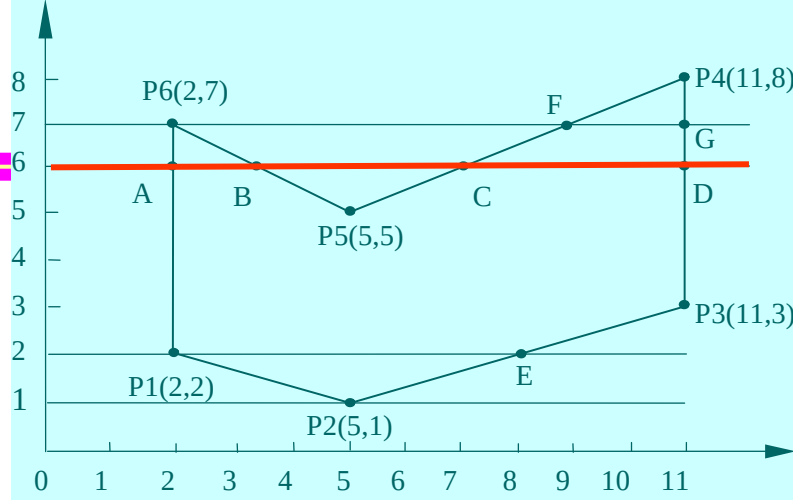


- 扫描线算法
  - 基本思想：
    - 按扫描线顺序，计算扫描线与多边形的相交区间，再用要求的颜色显示这些区间的像素，即完成填充工作。
  - 对于一条扫描线填充过程可以分为四个步骤：
    - （1）求交：计算扫描线与多边形各边的交点
    - （2）排序：把所有交点按x值递增顺序排序
    - （3）配对：第一个与第二个，第三个与第四个……
    - （4）填色：把相交区间内的像素置成多边形颜色，把相交区间外的像素置成背景色

X-扫描线算法填充.SWF


8
7
6
5
4
3
2
1
0
1
2
3
4
5
6
7
8
9
10
11
P6(2,7)
P4(11,8)
F
G
A
B
C
D
P5(5,5)
P3(11,3)
P1(2,2)
E
P2(5,1)

扫描线的拓扑信息



- 数据结构
  - 活性边表(AET)：把与当前扫描线相交的边称为活性边，并把它们按与扫描线交点x坐标递增的顺序存放在一个链表中

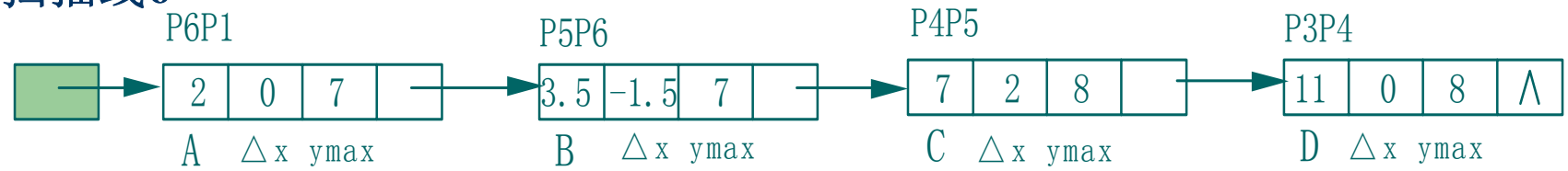
    结点内容

    x：当前扫描线与边的交点坐标

    △x：从当前扫描线到下一条扫描线间x的增量

    ymax：该边所交的最高扫描线号y_{max}

对于扫描线6



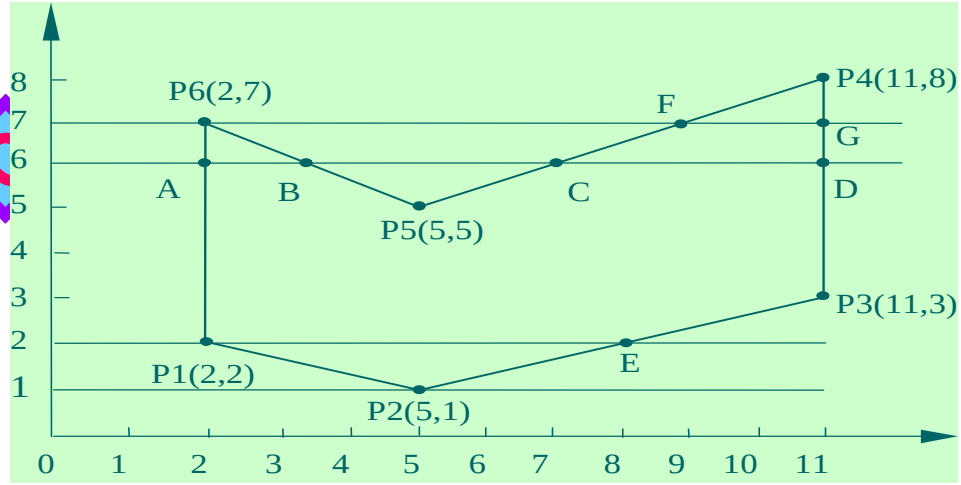
假定当前扫描线与多边形某一条边的交点的x坐标为x，则下一条扫描线与该边的交点不要重计算，只要加一个增量△x。

设该边的直线方程为：ax+by+c=0；

- 若y＝y_i，x=x_i；则当y = y_{i+1}时，

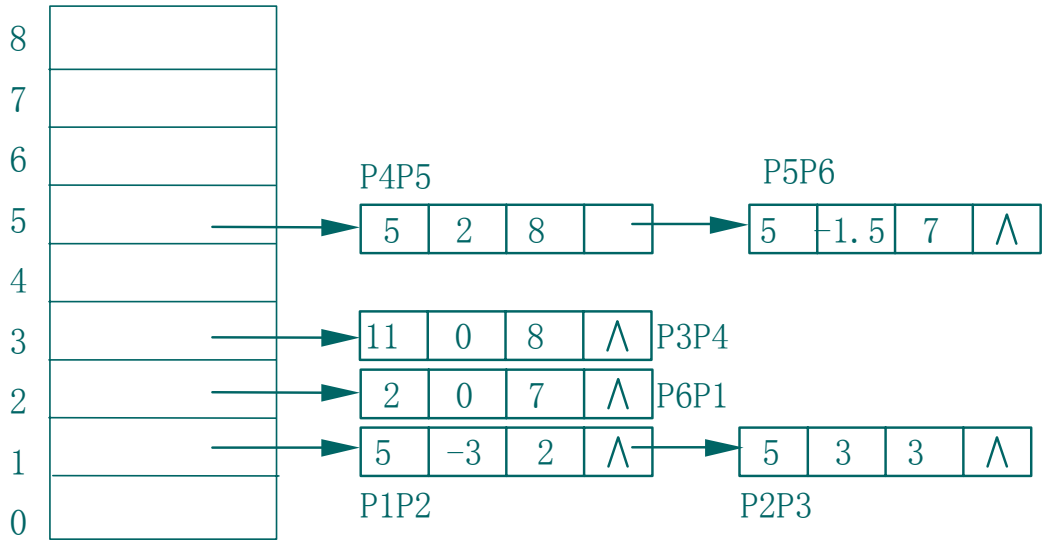
$$x_{i+1} = \frac{1}{a}(-b \cdot y_{i+1} - c_i) = x_i - \frac{b}{a};$$

其中 $\Delta x = -b/a$ 为常数



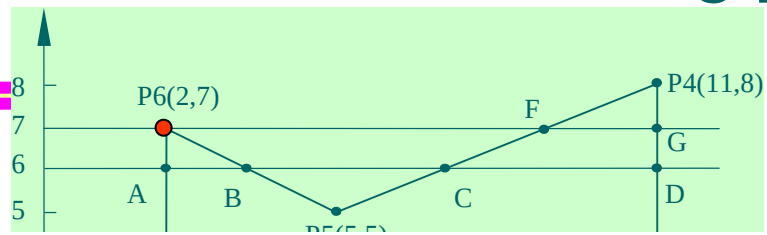
新边表（NET）：

- 存放在该扫描线第一次出现的边。若某边的较低端点为y_{min}，则该边就放在扫描线y_{min}的新边表中



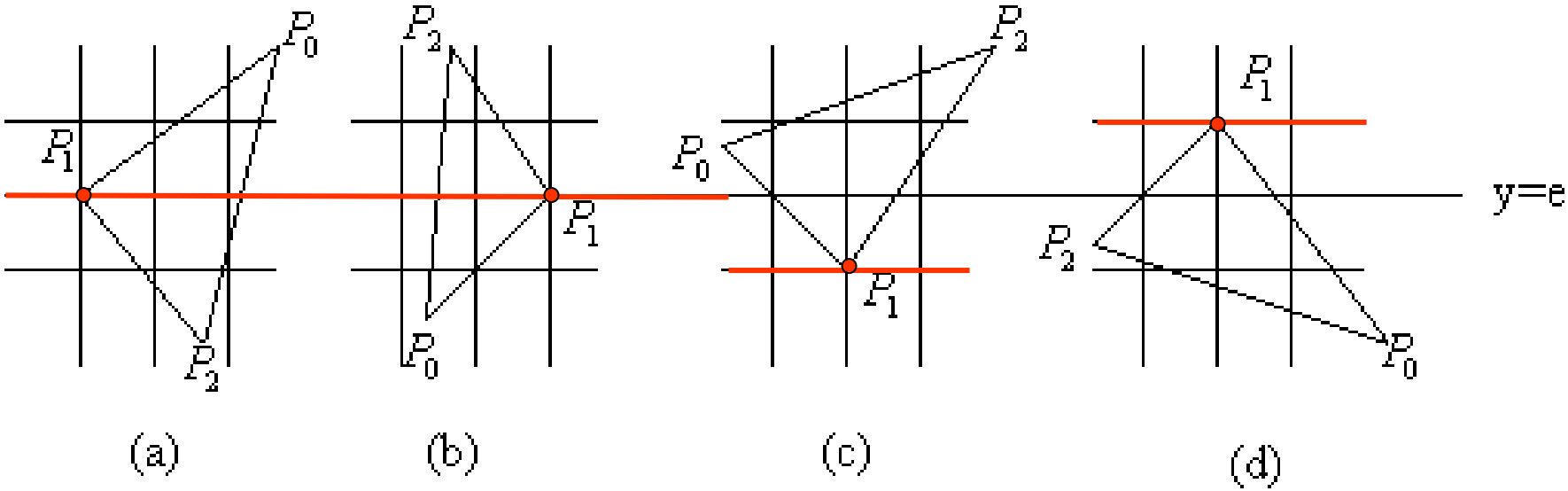
上图所示各条扫描线的新边表NET

填充过程必须解决的两个特殊问题

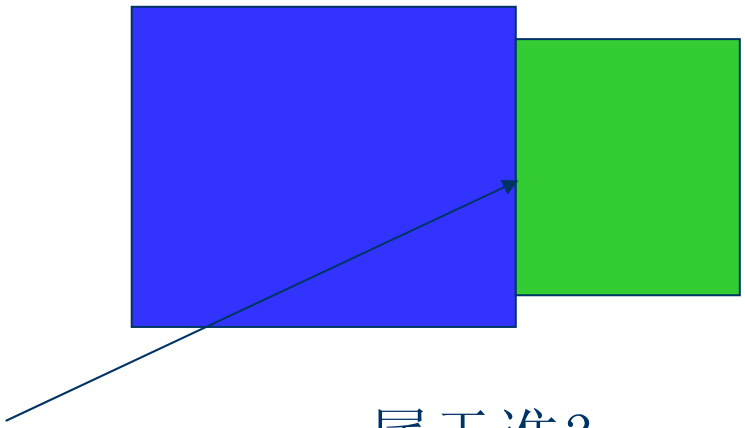


扫描线与多边形的顶点或边界相交时，必须正确进行交点的取舍。只需检查顶点的两条边的另外两个端点的y值。按这两个y值中大于交点y值的个数是0, 1, 2来决定。

- **(a)** 算作**1**个交点　　**(b)** 算作**1**个交点
- **(c)** 算作**2**个交点　　**(d)** 算作**0**个交点



问题二～边界上像素的取舍问题



属于谁？

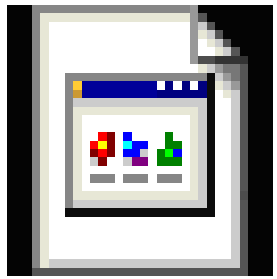
共享边界如何处理？

规定：落在右/上边界的像素不予填充，而落在左/下边界的像素予以填充，即左闭右开，下闭上开

算法过程

```
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)
    { 处理新边表和活性边表}
  }
```

- 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（左闭右开）上的像素（x,y），用drawpixel(x,y,color)改写像素颜色值；
- 遍历AET表，把ymax＝i的结点从AET表中删除，并把ymax>i结点的x值递增△x；
- 若允许多边形的边自交，则用冒泡法对AET表重新排序；



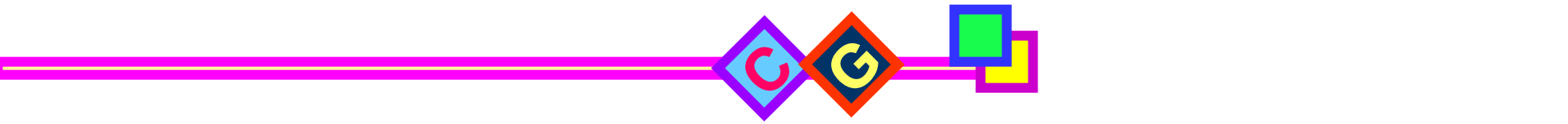
多边形扫描线算法演示

边界标志算法

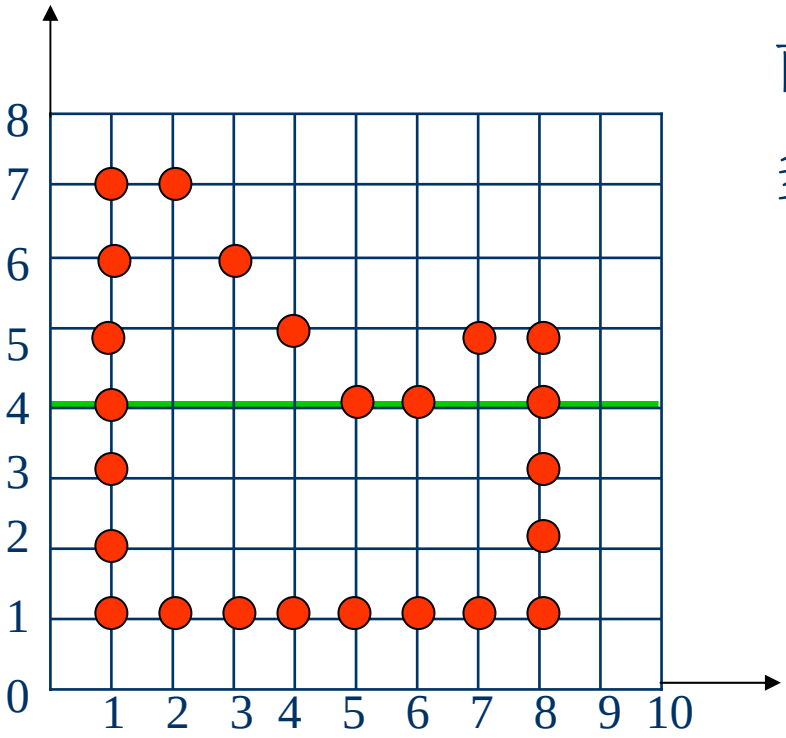
- 基本思想：
  - 帧缓冲器中对多边形的每条边进行直线扫描转换，亦即对多边形边界所经过的像素打上标志。
  - 然后再采用和扫描线算法类似的方法将位于多边形内的各个区段着上所需颜色。
  - 使用一个布尔量inside来指示当前点是否在多边形内的状态。

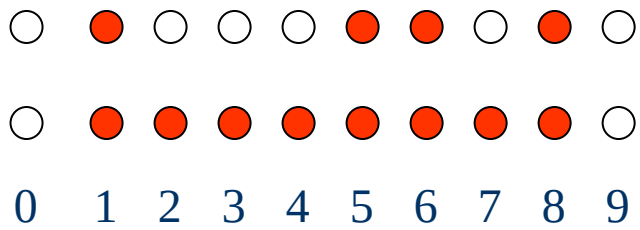
算法过程

```
void edgemark_fill(polydef, color)
多边形定义  polydef;     int color;
{    对多边形polydef 每条边进行直线扫描转换;
     inside = FALSE;
     for (每条与多边形polydef相交的扫描线y )
     for (扫描线上每个像素x )
     { if(像素 x 被打上边标志)
           inside = ! (inside);
       if(inside! = FALSE)
            drawpixel (x, y, color);
       else drawpixel (x, y, background);
      }
 }
```



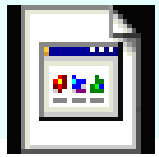
对第四条扫描线的像素在填充前后的状态，填红者表示置为多边形色。





- 用软件实现时，扫描线算法与边界标志算法的执行速度几乎相同，
- 但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。

分步边界标志算法演示

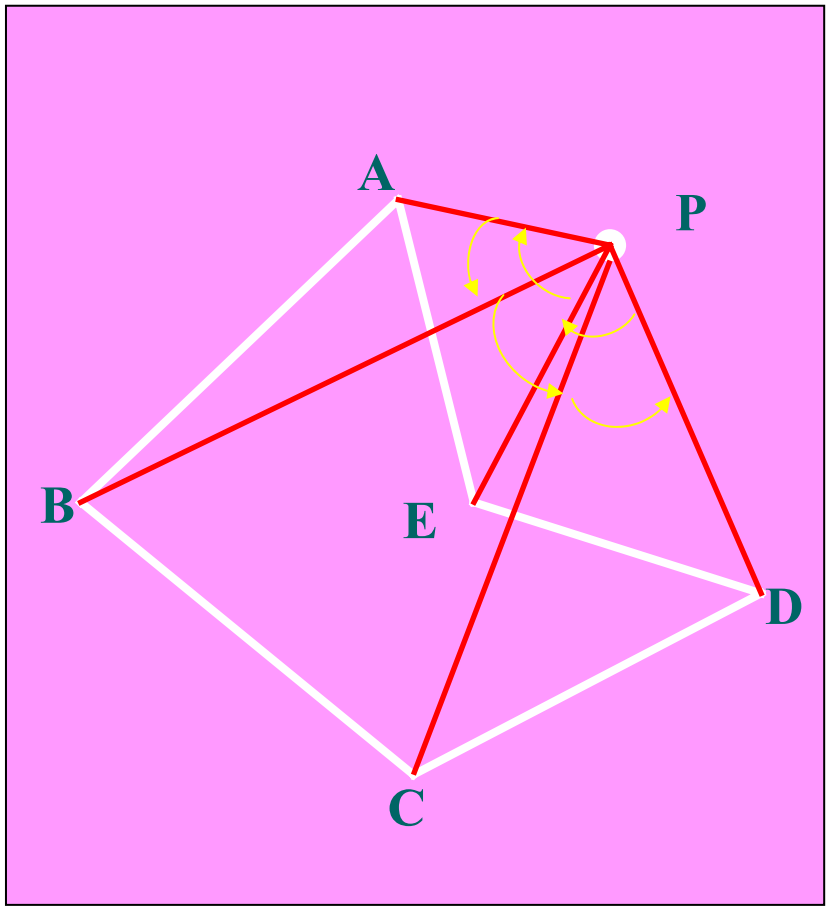


边界标志算法演示.swf

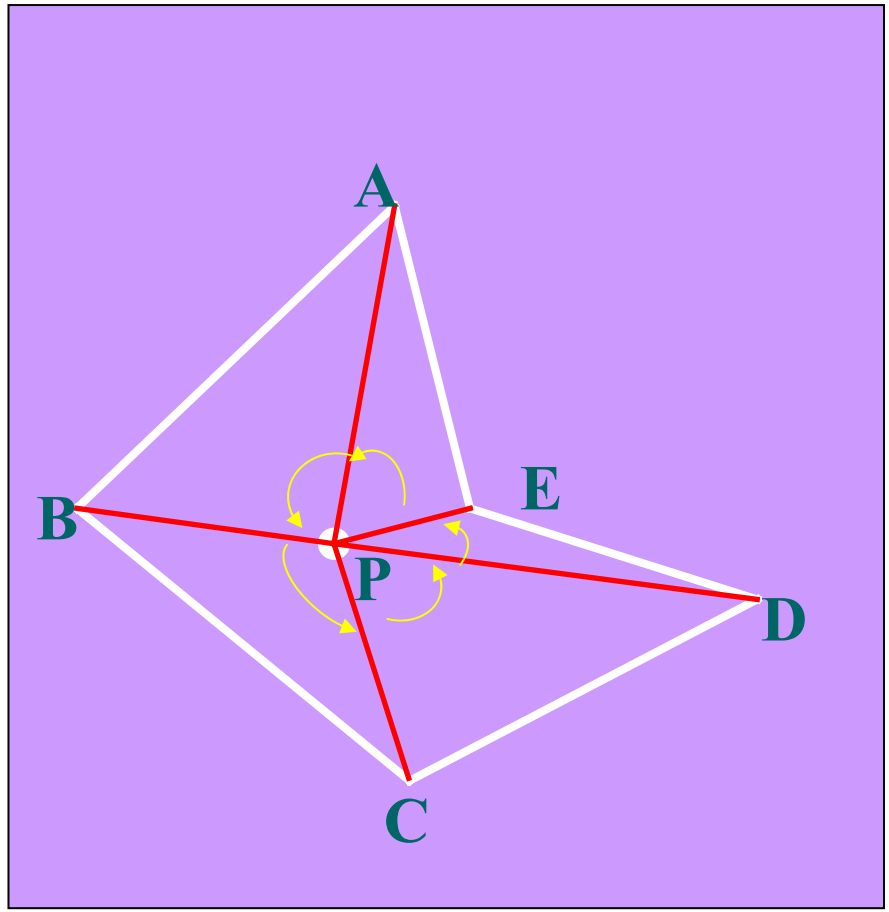
点在多边形内的包含性检验

- 检验夹角之和
- 射线法检验交点数

检验夹角之和



若夹角和为0，则点p在多边形外

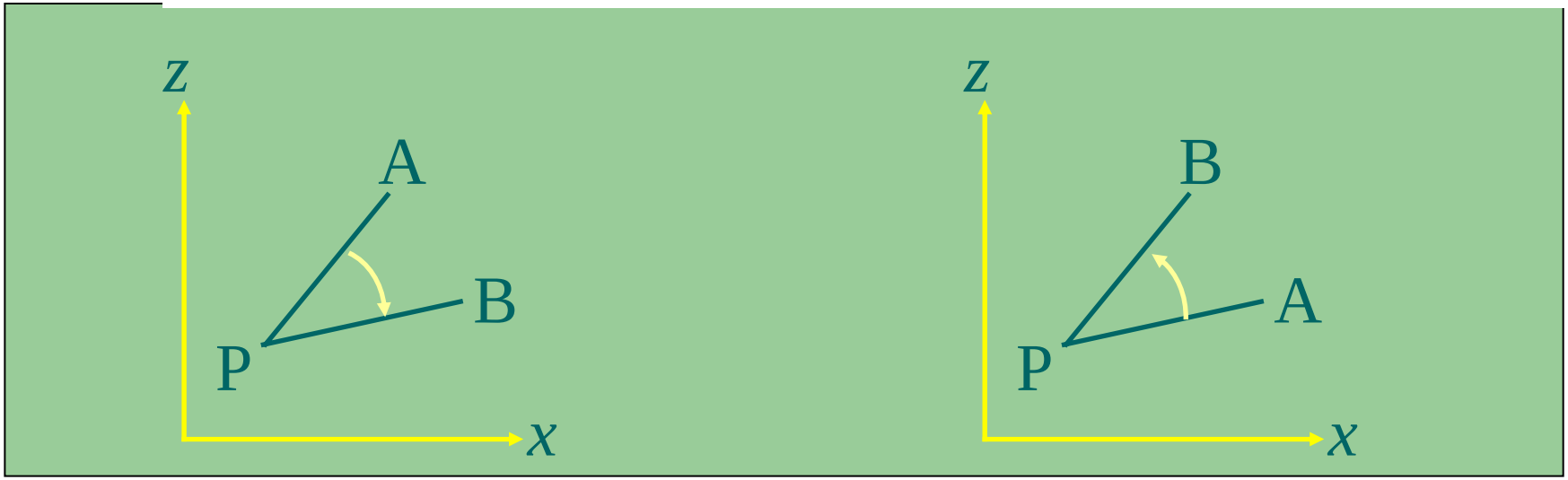


若夹角和为360°，则点p在多边形内

夹角如何计算？

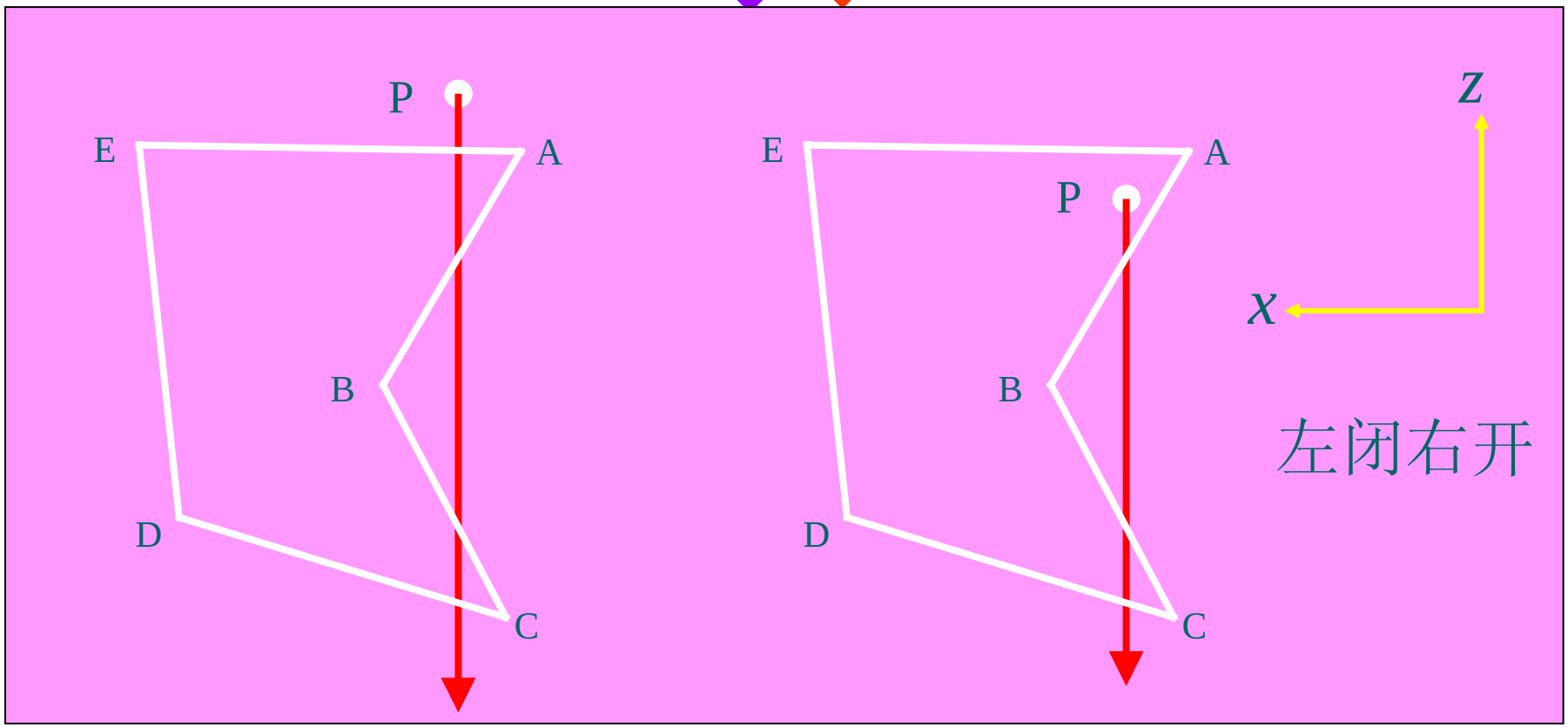
- 大小：利用余弦定理
- 方向：令

$$T=\begin{vmatrix} x_A-x_P & z_A-z_P \\ x_B-x_P & z_B-z_P \end{vmatrix}=(x_A-x_P)(z_B-z_P)-(x_B-x_P)(z_A-z_P)$$



当T<0时，AP斜率>BP斜率，为顺时针角　　当T>0时，AP斜率<BP斜率，为逆时针角

射线法检验交点数



交点数=偶数（包括**0**）

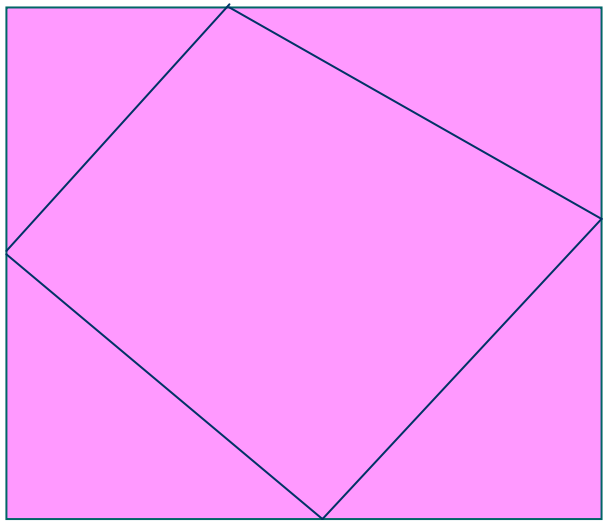
点在多边形之外

交点数=奇数

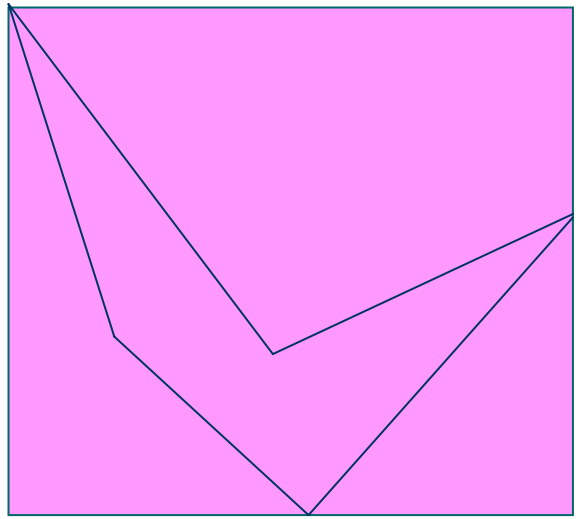
点在多边形之内

逐点测试效率低不实用怎么办？

- 包围盒法



凸多边形



凹多边形

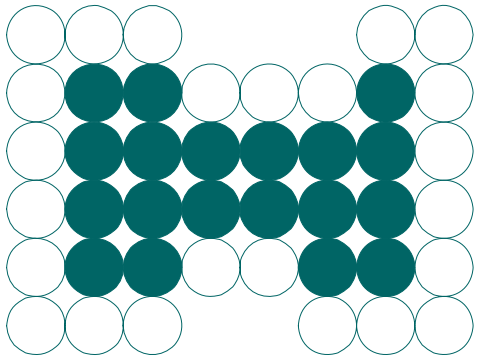
区域填充算法

- 区域指已经表示成点阵形式的填充图形，它是像素的集合。
- 区域可采用内点表示和边界表示两种表示形式。
- 区域可分为**4**向连通区域和**8**向连通区域。
- 区域填充指先将区域的一点赋予指定的颜色，然后将该颜色扩展到整个区域的过程。
- 区域填充算法要求区域是连通的

- 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
- 边界表示
  - 枚举出边界上所有的像素
  - 边界上的所有像素着同一颜色
  - 内部像素着与边界像素不同的颜色

- 4向连通区域和8向连通区域

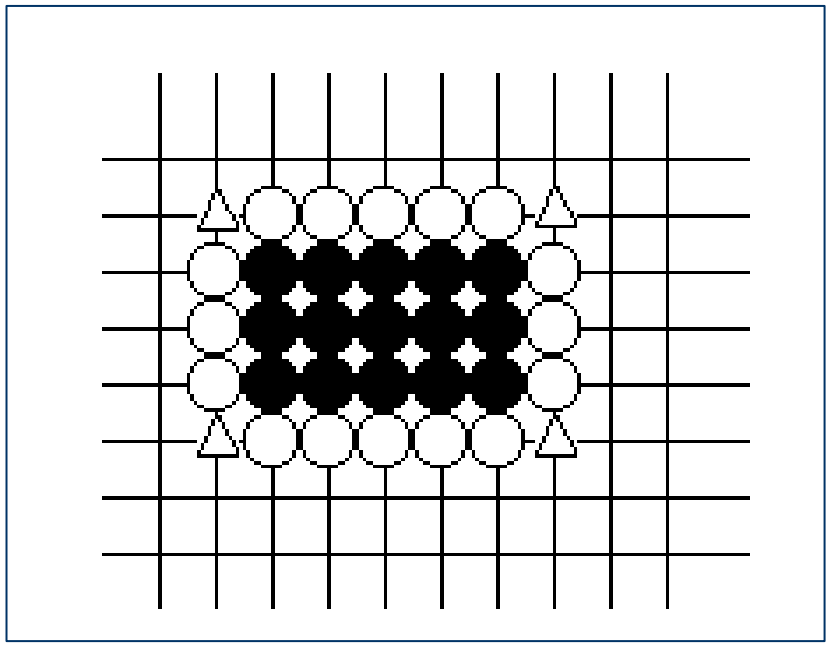
四个方向运动　八个方向运动　四连通区域　八连通区域



表示内点　表示边界点

种子填充法

- 4连通与8连通区域的区别
  - 连通性： 4连通可看作8连通区域，但对边界有要求
  - 对边界的要求



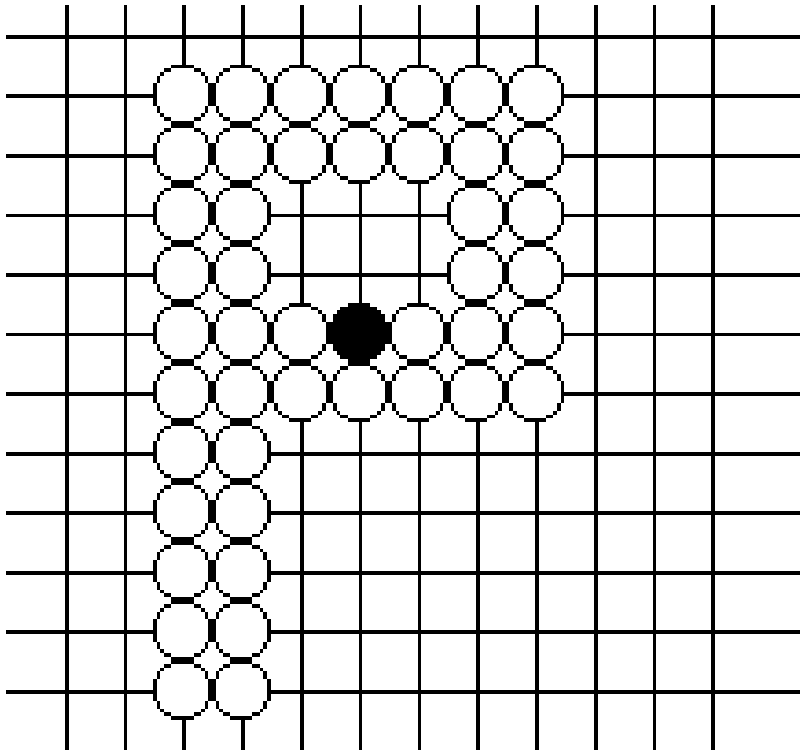
区域填充的递归算法

内点表示的**4**连通区域的递归填充算法**:**

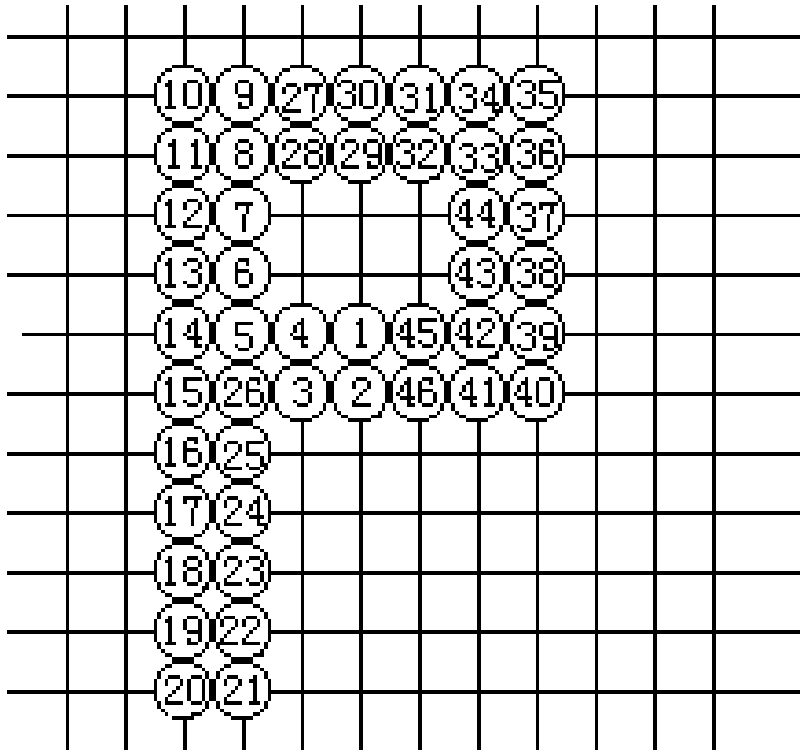
```
void FloodFill4(int x,int y,int oldcolor,int newcolor)
{   if(getpixel(x,y)==oldcolor)   //属于区域内点oldcolor
    {     drawpixel(x,y,newcolor);
          FloodFill4(x,y+1,oldcolor,newcolor);  //上
          FloodFill4(x,y-1,oldcolor,newcolor);  //下
          FloodFill4(x-1,y,oldcolor,newcolor);  //左
          FloodFill4(x+1,y,oldcolor,newcolor);  //右
      }
}
```

- 注意坐标系的方向！

区域填充(种子填充法)



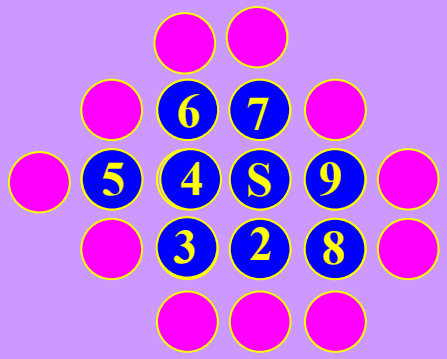
(a)



(b)

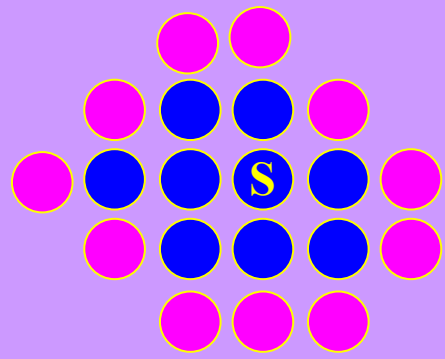
内点表示的4连通区域的递归填充过程（上、下、左、右）

填充算法演示



缺点？

按左、上、右、下顺序



				5								
		3	4	6	6	7						
	2	8	8	8	8	8	8	9				
	4	4	4	4	4	4	4	4	4			
	7	7	7	7	7	7	7	7	7	7		
S	9	9	9	9	9	9	9	9	9	9	9	9

区域填充的递归算法

边界表示的**4**连通区域的递归填充算法**:**

```
void BoundaryFill4(int x,int y,int boundarycolor,int newcolor)
{   if(color!=newcolor &&color!=boundarycolor)
    {     drawpixel(x,y,newcolor);
          BoundaryFill4(x,y+1,boundarycolor,newcolor);
          BoundaryFill4(x,y-1, boundarycolor,newcolor);
          BoundaryFill4(x-1,y, boundarycolor,newcolor);
          BoundaryFill4(x+1,y, boundarycolor,newcolor);
      }
}
```

区域填充(种子填充法)

4-连通边界填充.SWF

8-连通边界填充.SWF

区域填充(种子填充法)

- 问题：

  内点表示与边界表示的8连通区域？

- 缺点：

(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起一次递归，进/出栈，费时费内存。

- 改进算法，减少递归次数，提高效率。

  方法之一使用扫描线种子填充算法；

区域填充的扫描线算法

- 算法步骤：
  - 首先填充种子点所在扫描线上的位于给定区域的一个区段
  - 然后确定与这一区段相连通的上、下两条扫描线上位于给定区域内的区段，并依次保存下来。
  - 反复这个过程，直到填充结束。

区域填充的扫描线算法实现步骤

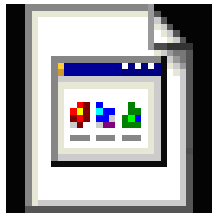
(1)初始化：堆栈置空。将种子点（x，y）入栈。

(2)出栈：若栈空则结束。否则取栈顶元素（x，y），以y作为当前扫描线。

(3)填充并确定种子点所在区段：从种子点（x，y）出发，沿当前扫描线向左、右两个方向填充，直到边界。分别标记区段的左、右端点坐标为xl和xr。

(4)并确定新的种子点：在区间[xl，xr]中检查与当前扫描线y上、下相邻的两条扫描线上的像素。若存在非边界、未填充的像素，则把每一区间的最右像素作为种子点压入堆栈，返回第（2）步。

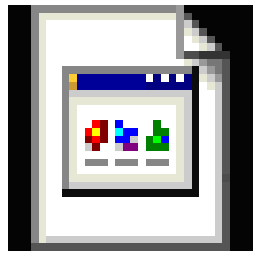
上述算法对于每一个待填充区段，只需压栈一次；因此，扫描线填充算法提高了区域填充的效率。



扫描线种子填充演示

区域图案填充算法

```
if(pattern(x%M,y%N))
    SetPixel(x,y,color);
//color为填充前景色
```



图案填充演示

字符（自学）

- 字符指数字、字母、汉字等符号。
- 计算机中字符由一个数字编码唯一标识。
- “美国信息交换用标准代码集”简称ASCII码。它是用7位二进制数进行编码表示128个字符
- 汉字编码的国家标准字符集。每个符号由一个区码和一个位码（2字节）共同标识。
- 区分ASCII码与汉字编码，采用字节的最高位来标识（最高位为0表示ASCII码）

上机作业第五次

- **VC下实现种子填充算法的编程：**
  - 在**C**View类里添加一个实现函数FloodFill4；－p44或p45**
  - 在**OnDraw函数里确定种子点，调用该实现函数；
  - 注意：使用**SetPixel与GetPixel函数前先取得DC：this->GetDC( )…最后还要释放DC；
  - 由于种子填充算法需要不断地压栈操作，因此，当需要填充的区域很大时，可能会使计算机内存空间中的栈空间满，而出现错误。为了保证程序的正常运行，请将需要填充的多边形区域设置的较小。
- 思考：如何画出**flash中演示的效果？
- 选作：实现扫描线种子填充算法（上机实验成绩为优！）－－压栈、出栈、上下扫描线判定

OpenGL介绍

一、OpenGL的背景知识

- 是近几年发展起来的一个性能卓越的三维图形标准；
- 是在SGI等多家计算机公司倡导下，以SGI的GL三维图形库为基础制定的一个通用共享的开放式三维图形标准；
- **Microsoft、SGI、IBM、DEC、SUN、HP**等都采用了**OpenGL**作为三维图形标准；
- 许多硬件厂商提供对**OpenGL**的支持，是一个工业标准
- **OpenGL**独立于硬件设备、窗口系统和操作系统，以其为基础开发的应用程序可以在各种平台间移植。
- **OpenGL**可以用各种编程语言进行调用

OpenGL介绍

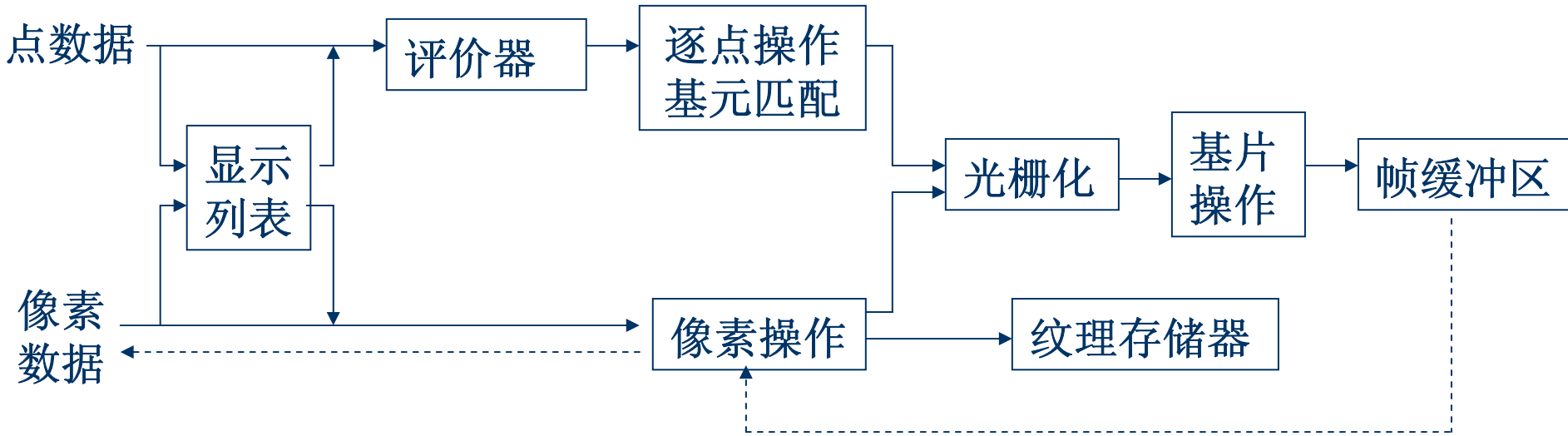
OpenGL：

图形硬件的软件接口

OpenGL是一种应用程序编程接口，而不是一种编程语言

Microsoft提供的OpenGL软件实现位于Opengl32.dll动态链接库中，位于Windows的System32目录

OpenGL的绘制流程和原理



OpenGL介绍

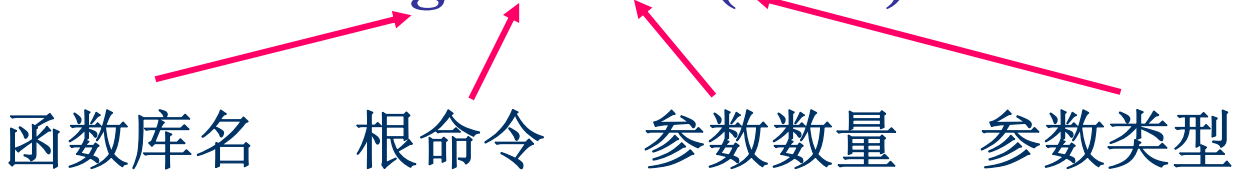
OpenGl函数及结构

- OpenGL核心库，函数以gl开头
- OpenGL实用库，函数以glu开头
- 辅助库，函数以aux开头(帮助初学者练习之用)
- Windows专用函数，用于连接OpenGL和Windows窗口系统，以wgl开头
- OpenGL实用函数工具包(GLUT)提供了与任意屏幕窗口系统进行交互的函数库，函数以glut开头。

OpenGL介绍

基本的OpenGL语法

函数命名的约定：



- 函数库名指明函数来自于哪个库
- 根命令表示这个函数相对应的OpenGL命令
- 参数数量和参数类型表示这个函数将接受的参数的个数和类型

OpenGL定义的常量都以GL开头，所有的字母大写，单词间以下划线来分隔，如GL_COLOR_BUFFER_BIT

OpenGL介绍

头文件：

```
#include  <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
```

注意，若使用

```
#include <GL/glut.h>
```

则不需要引入gl.h和glu.h，因为GLUT保证了它们的正确引入

可以引入C++程序需要的头文件

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

OpenGL介绍—命令后缀参数类型

后缀定义	数据类型	相应的C语言类型	OpenGL类型定义
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

二、OpenGL的主要功能

- 绘制模型：提供了绘制点、线、多边形、球、锥、多面体、茶壶等复杂的三维物体以及贝塞尔、NURBS等复杂曲线或曲面的绘制函数。
- 各种变换：提供了平移、旋转、变比和镜像四种基本变换以及平行投影和透视投影两种投影变换。通过变换实现三维的物体在二维的显示设备上显示。
- 着色模式：提供了RGBA模式和颜色索引两种颜色的显示方式。
- 光照处理：在自然界我们所见到的物体都是由其材质和光照相互作用的结果，OpenGL提供了辐射光(Emitted Light)、环境光(Ambient Light)、漫反射光(Diffuse Light)和镜面光(Specular Light)。材质是指物体表面对光的反射特性，在OpenGL中用光的反射率来表示材质。

二、OpenGL的主要功能

➢纹理映射(Texture Mapping)： 将真实感的纹理粘贴在物体表面，使物体逼真生动。纹理是数据的简单矩阵排列，数据有颜色数据、亮度数据和alpha数据。

➢位图和图像：提供了一系列函数来实现位图和图像的操作。位图和图像数据均采用像素的矩阵形式表示。

➢制作动画：提供了双缓存(Double Buffering)技术来实现动画绘制。双缓存即前台缓存和后台缓存，后台缓存用来计算场景、生成画面，前台缓存用来显示后台缓存已经画好的画面。当画完一帧时，交互两个缓存，这样循环交替以产生平滑动画。

二、OpenGL的主要功能

- 选择和反馈：OpenGL为支持交互式应用程序设计了选择操作模式和反馈模式。在选择模式下，则可以确定用户鼠标指定或拾取的是哪一个物体，可以决定将把哪些图元绘入窗口的某个区域。而反馈模式，OpenGL把即将光栅化的图元信息反馈给应用程序，而不是用于绘图。
- 反走样技术
- 深度暗示(Depth Cue)
- 运动模糊(Motion Blur)
- 雾化(Fog)

OpenGL生成基本图形

OpenGL提供了描述点、线、多边形的绘制机制。它们通过glBegin()函数和glEnd()函数配对来完成。glBegin()函数有一个类型为Glenum的参数，它的取值见下表。gLEnd()函数标志着形状的结束，该函数没有参数。

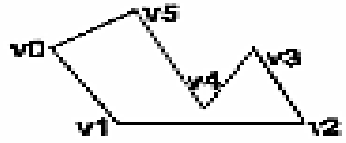
Mode 的值	解释
GL_POINTS	一系列独立的点
GL_LINES	每两点相连成为线段
GL_POLYGON	简单凸多边形的边界
GL_TRIANGLES	三点相连成为一个三角形
GL_QUADS	四点相连成为一个四边形
GL_LINE_STRIP	顶点相连成为一系列线段
GL_LINE_LOOP	顶点相连成为一系列线段，连接最后一点与第一点
GL_TRIANGLE_STRIP	相连的三角形带
GL_TRIANGLE_FAN	相连的三角形扇形
GL_QUAD_STRIP	相连的四边形带

OpenGL生成基本图形

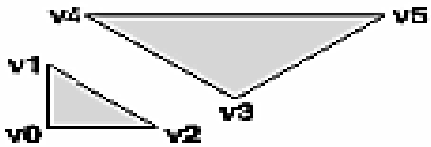
GL_POINTS

GL_LINES

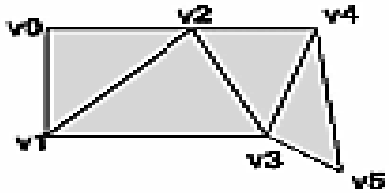
GL_LINE_STRIP



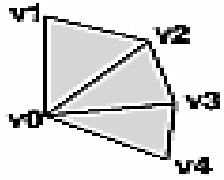
GL_LINE_LOOP



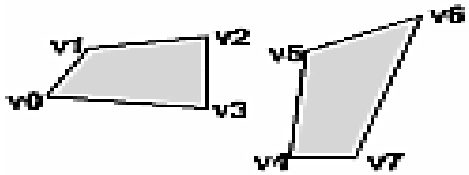
GL_TRIANGLES



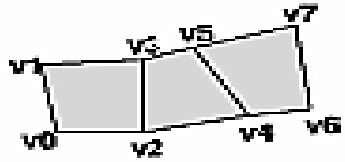
GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN



GL_QUADS



GL_QUAD_STRIP

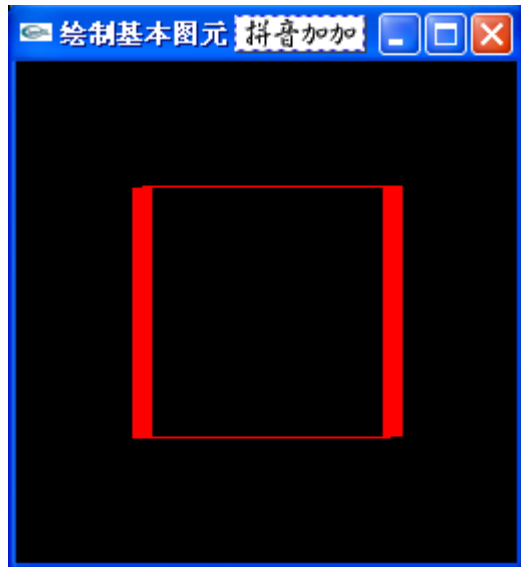
GL_POLYGON

OpenGL生成基本图形—点

```
//绘制点
   glColor3f (1.0, 0.0, 0.0);
   glBegin(GL_POINTS);
      glVertex3f (0.25, 0.25, 0.0);
      glVertex3f (0.75, 0.25, 0.0);
   glEnd();
   glColor3f(0.0, 1.0, 0.0);
   glPointSize(10.0f);//绘制有宽度的点
   glBegin(GL_POINTS);
      glVertex3f (0.75, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
   glEnd();
```

OpenGL生成基本图形—线

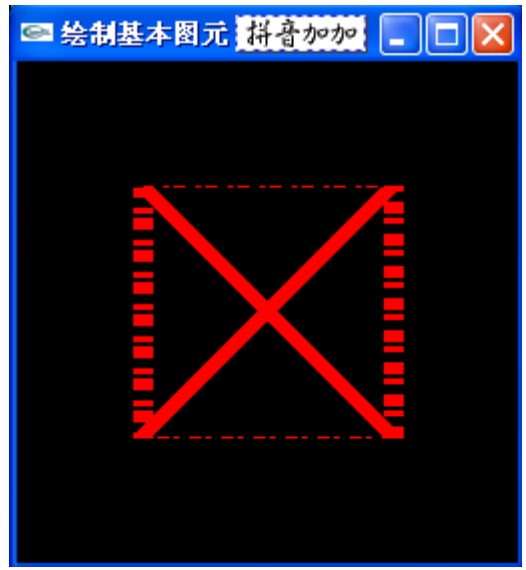

```
//绘制线
  glColor3f (1.0, 0.0, 0.0);
  glBegin(GL_LINES);
    glVertex3f (0.25, 0.25, 0.0);
    glVertex3f (0.75, 0.25, 0.0);
    glVertex3f (0.75, 0.75, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
  glEnd();
  glLineWidth(10.0f);//设置线的宽度
  glBegin(GL_LINES);
    glVertex3f (0.25, 0.25, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
   glVertex3f (0.75, 0.75, 0.0);
   glVertex3f (0.75, 0.25, 0.0);
  glEnd();
```



OpenGL生成基本图形－点划线

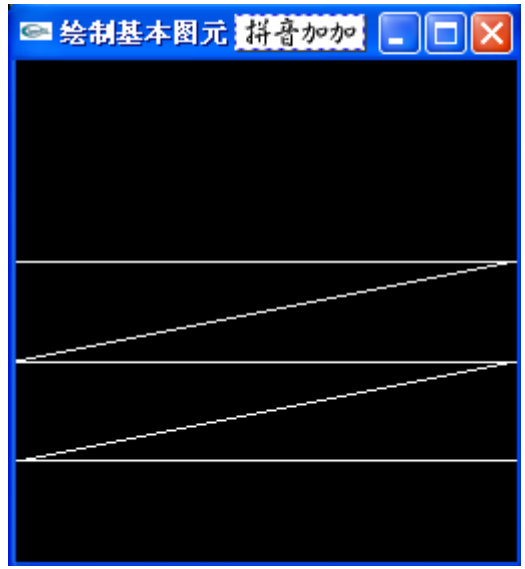

```
//绘制点划线
  glLineStipple(1,0x3f07);//设置点画模式
  glEnable(GL_LINE_STIPPLE);//激活点画线模
  glColor3f (1.0, 0.0, 0.0);
  glBegin(GL_LINES);
    glVertex3f (0.25, 0.25, 0.0);
    glVertex3f (0.75, 0.25, 0.0);
    glVertex3f (0.75, 0.75, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
  glEnd();
  glLineWidth(10.0f);//设置线的宽度
  glBegin(GL_LINES);
    glVertex3f (0.25, 0.25, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
    glVertex3f (0.75, 0.75, 0.0);
    glVertex3f (0.75, 0.25, 0.0);
  glEnd();
```

```
glDisable(GL_LINE_STIPPLE);//取消点画模式
  glBegin(GL_LINES);
    glVertex3f (0.25, 0.25, 0.0);
    glVertex3f (0.75, 0.75, 0.0);
    glVertex3f (0.75, 0.25, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
  glEnd();
```



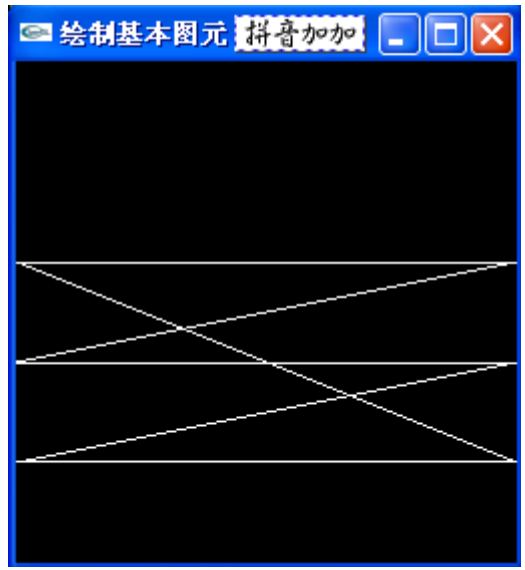
OpenGL生成基本图形—折线

```
//绘制折线
  glBegin(GL_LINE_STRIP);
      glVertex2f(0.0f,0.6f);
      glVertex2f(1.0f,0.6f);
      glVertex2f(0.0f,0.4f);
      glVertex2f(1.0f,0.4f);
     glVertex2f(0.0f,0.2f);
      glVertex2f(1.0f,0.2f);
glEnd();
```



OpenGL生成基本图形－闭合折线


```
glBegin(GL_LINE_LOOP);
    glVertex2f(0.0f,0.6f);
    glVertex2f(1.0f,0.6f);
    glVertex2f(0.0f,0.4f);
    glVertex2f(1.0f,0.4f);
    glVertex2f(0.0f,0.2f);
    glVertex2f(1.0f,0.2f);
glEnd();
```



OpenGL生成基本图形－绘制多边形

- OpenGL的多边形必须至少有三个顶点
- 直线不能相交，多边形构成单连通的凸区域
- 一个多边形有前面和后面之分
- 前面和后面可以有不同的属性

Void glPolygonMode(Glenum face,Glenum mode);

该函数控制多边形绘制模式是正面还是反面，是以点、轮廓还是填充的形式画出。默认为正面和反面都以填充的形式画出

face可取：GL_FRONT_AND_BACK、GL_FRONT、GL_BACK

mode可取：GL_POINT、GL_LINE、GL_FILL

void glFrontFace(Glenum mode);

该函数控制如何确定正面多边形，默认情况下，参数mode为GL_CCW(逆时针)，也可以指定为GL_CW(顺时针)

void glCullFace(Glenum mode);

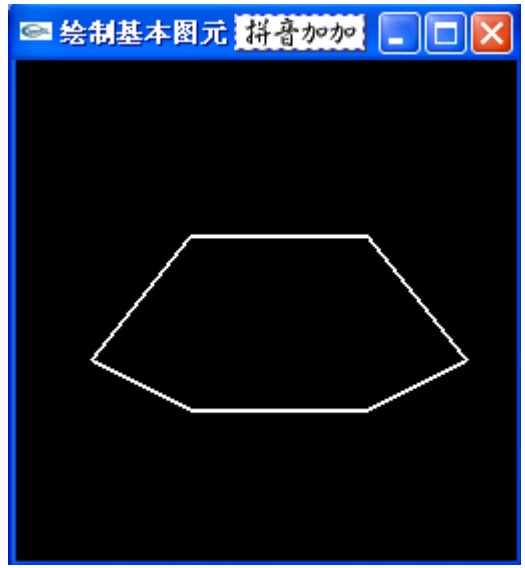
该函数指出在变换到屏幕坐标之前，舍弃哪个面的多边形。mode可以为GL_FRONT、GL_BACK 、GL_FRONT_AND_BACK，必须使用

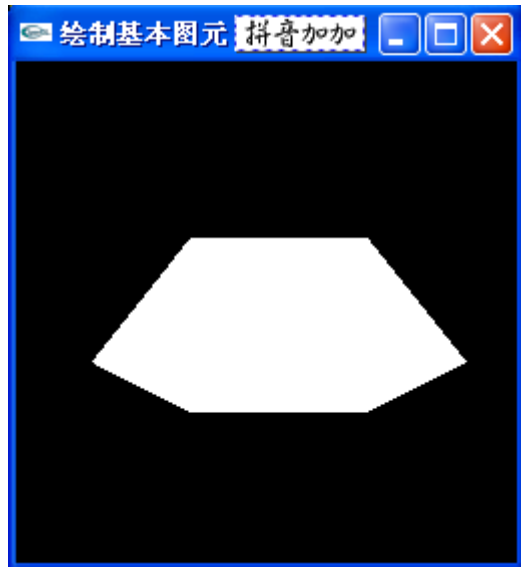
```
glEnable(GL_CULL_FACE);//激活拣选模式

glDisable(GL_CULL_FACE);//使拣选失效
```

OpenGL生成基本图形－绘制多边形


```
//绘制多边形
 glLineWidth(2.0f);
 glPolygonMode(GL_FRONT,GL_LINE);
 glPolygonMode(GL_BACK,GL_FILL);
 glFrontFace(GL_CCW);
// glFrontFace(GL_CW);
 glBegin(GL_POLYGON);
   glVertex2f(-0.3f,0.3f);
   glVertex2f(-0.7f,-0.2f);
   glVertex2f(-0.3f,-0.4f);
   glVertex2f(0.4f,-0.4f);
   glVertex2f(0.8f,-0.2f);
   glVertex2f(0.4f,0.3f);
 glEnd();
```

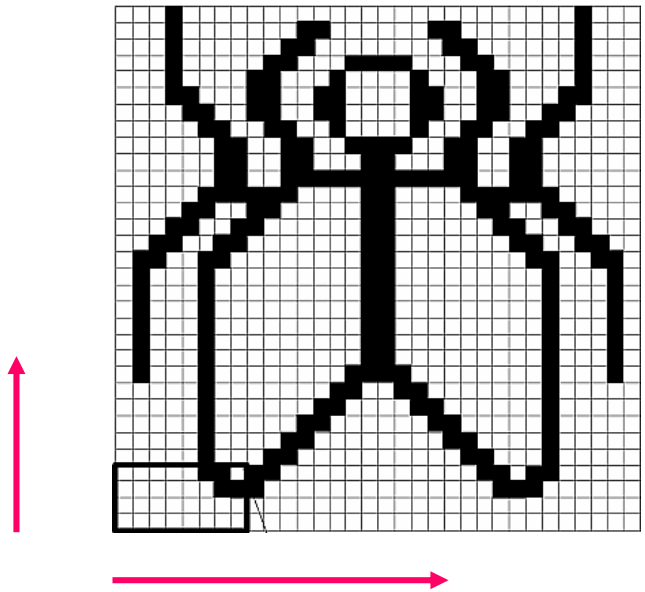
若替换为此函数，结果为





OpenGL生成基本图形－多边形填充模式

- OpenGL可以让用户自定义多边形填充模式，该填充模式必须是一个32×32的位图
- 其中的每一个位代表屏幕上的一个点。被填充的位的值为1，未被填充的值为0
- 将填充模式表示成16进制，表示该位图以一个字节为单位
- 从左下角开始自左而右自下而上地表示。



OpenGL生成基本图形－多边形填充模式

void glPolygonStipple(const Glubyte *mask);

该函数为填充多边形定义当前的点画模式

void glEnable(GL_POLYGON_STIPPLE);

该函数使多边形点画模式激活

void glDisable(GL_POLYGON_STIPPLE);

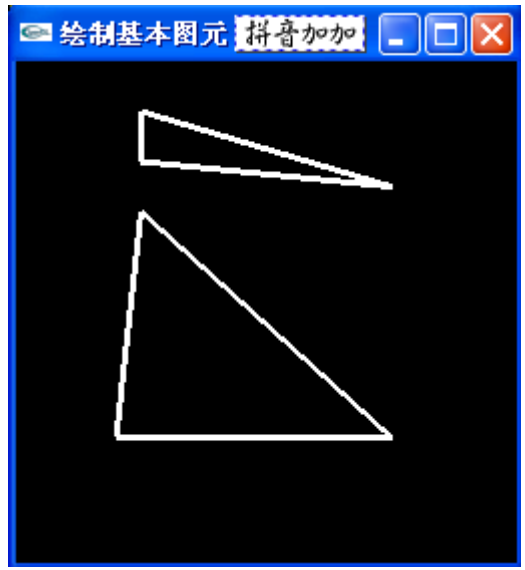
该函数使多边形点画模式失效

OpenGL生成基本图形－多边形填充模式

```
//定义多边形点画模式 为fly
    glLineWidth(2.0f);
    glPolygonMode(GL_FRONT,GL_FILL);
    glFrontFace(GL_CCW);
   glEnable(GL_POLYGON_STIPPLE);
   glPolygonStipple(fly);
   glBegin(GL_POLYGON);
        glVertex2f(-0.3f,0.3f);
                glVertex2f(-0.7f,-0.2f);
        glVertex2f(-0.3f,-0.4f);
        glVertex2f(0.4f,-0.4f);
        glVertex2f(0.8f,-0.2f);
        glVertex2f(0.4f,0.3f);
   glEnd();
   glDisable(GL_POLYGON_STIPPLE);
```

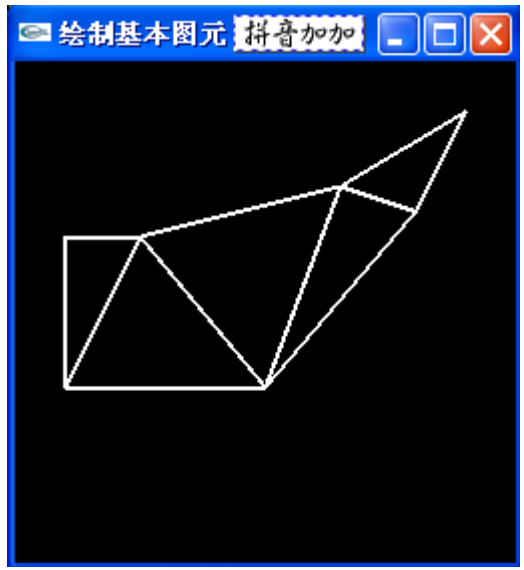
OpenGL生成基本图形—绘制三角形


```
//绘制三角形
  glLineWidth(3.0f);
  glPolygonMode(GL_FRONT,GL_LINE);
  glBegin(GL_TRIANGLES);
     glVertex2f(-0.5f,0.4f);
     glVertex2f(-0.6f,-0.5f);
     glVertex2f(0.5f,-0.5f);

     glVertex2f(-0.5f,0.8f);
     glVertex2f(-0.5f,0.6f);
     glVertex2f(0.5f,0.5f);
  glEnd();
```



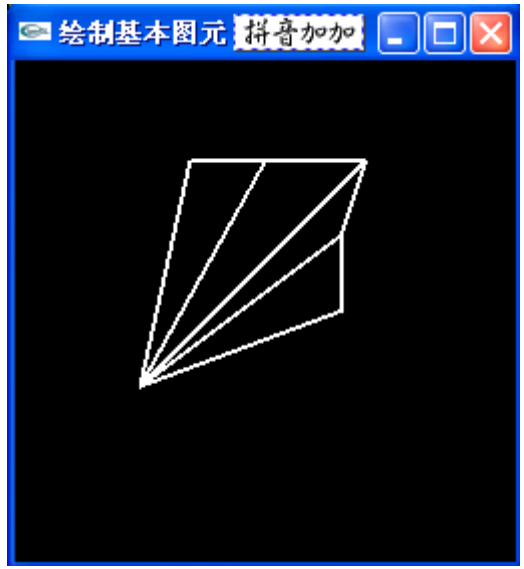
OpenGL生成基本图形—绘制三角形片

```
//绘制三角形片
 glLineWidth(2.0f);
 glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
 glBegin(GL_TRIANGLE_STRIP);
    glVertex2f(-0.8f,0.3f);
    glVertex2f(-0.8f,-0.3f);
    glVertex2f(-0.5f,0.3f);
    glVertex2f(0.0f,-0.3f);
    glVertex2f(0.3f,0.5f);
    glVertex2f(0.6f,0.4f);
    glVertex2f(0.8f,0.8f);
 glEnd();
```



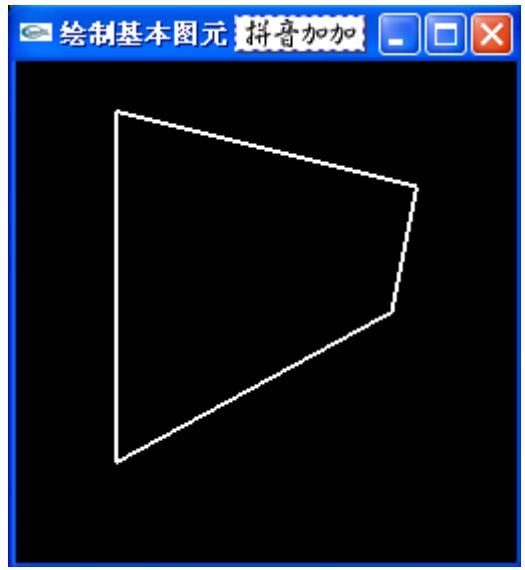
OpenGL生成基本图形—绘制三角形扇

```
//绘制三角形扇
  glLineWidth(2.0f);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
  glBegin(GL_TRIANGLE_FAN);
     glVertex2f(-0.5f,-0.3f);
     glVertex2f(0.3f,-0.0f);
     glVertex2f(0.3f,0.3f);
     glVertex2f(0.4f,0.6f);
     glVertex2f(0.0f,0.6f);
     glVertex2f(-0.3f,0.6f);
  glEnd();
```



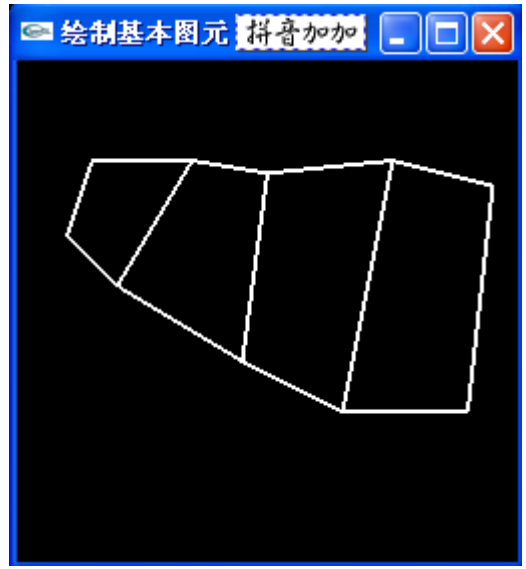
OpenGL生成基本图形—绘制四边形

```
//绘制四边形
  glLineWidth(2.0f);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
  glBegin(GL_QUIDS);
    glVertex2f(-0.6f,0.8f);
    glVertex2f(-0.6f,-0.6f);
    glVertex2f(0.5f,0.0f);
    glVertex2f(0.6f,0.5f);
  glEnd();
```

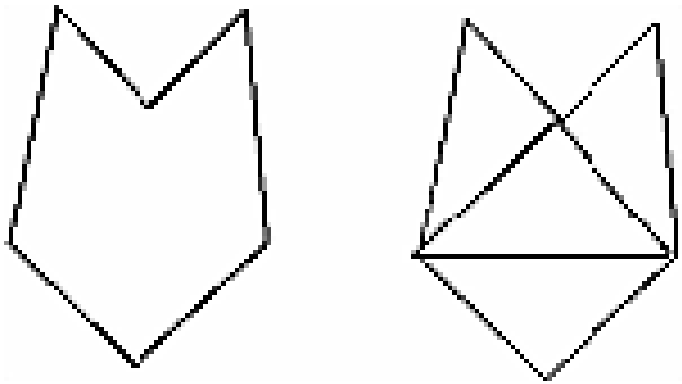


OpenGL生成基本图形—绘制四边形片


```
//绘制四边形片
glLineWidth(2.0f);
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
glBegin(GL_QUAD_STRIP);
   glVertex2f(-0.8f,0.3f);
   glVertex2f(-0.7f,0.6f);
   glVertex2f(-0.6f,0.1f);
   glVertex2f(-0.3f,0.6f);
   glVertex2f(-0.1f,-0.2f);
   glVertex2f(0.0f,0.55f);
   glVertex2f(0.3f,-0.4f);
   glVertex2f(0.5f,0.6f);
   glVertex2f(0.8f,-0.4f);
   glVertex2f(0.9f,0.5f);
glEnd();
```



OpenGL生成基本图形—边的可见性

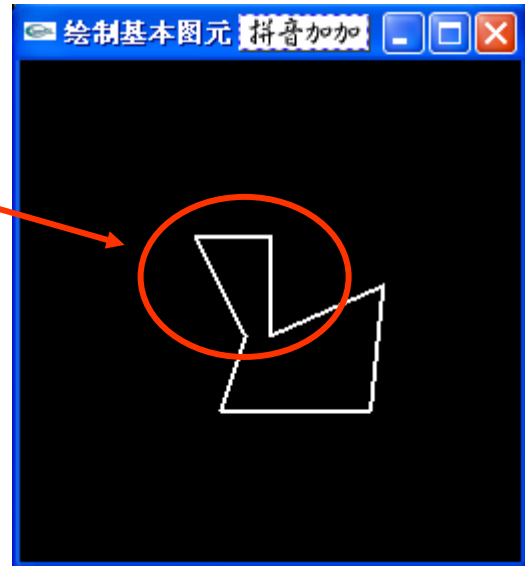


void glEdgeFlag(GLboolean flag);

该函数说明边的可见性，其参数为**GL_TRUE**，则该边可见，为**GL_FALSE**，则该边不可见的，该函数位于两个**glVertex()**函数之前，说明这两个顶点构成的边的可见性。该函数不适合为三角形切片或四边形切边指定顶点。

OpenGL生成基本图形—边的可见性

```
//边的可见性
  glLineWidth(2.0f);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
  glBegin(GL_POLYGON);
    glEdgeFlag(GL_TRUE);
    glVertex2f(-0.3f,0.3f);
    glEdgeFlag(GL_FALSE);
    glVertex2f(-0.1f,-0.1f);
    glEdgeFlag(GL_TRUE);
    glVertex2f(0.0f,-0.1f);
    glVertex2f(0.0f,0.3f);
  glEnd();
 ……
```

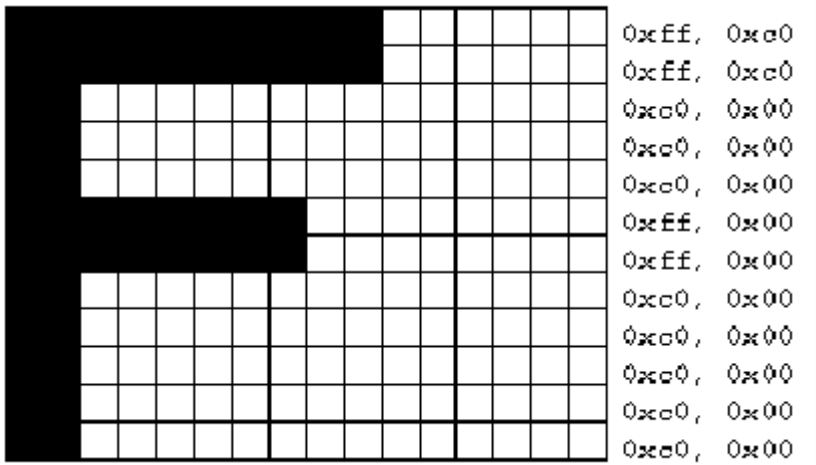


OpenGL生成基本图形—生成英文字符

- 位图字体
- 轮廓字体
- 纹理映射字体

} 三种渲染字体的方法

位图：位图是0和1组成的矩阵，只有每一像素的一位信息。



位图化的F及其数据

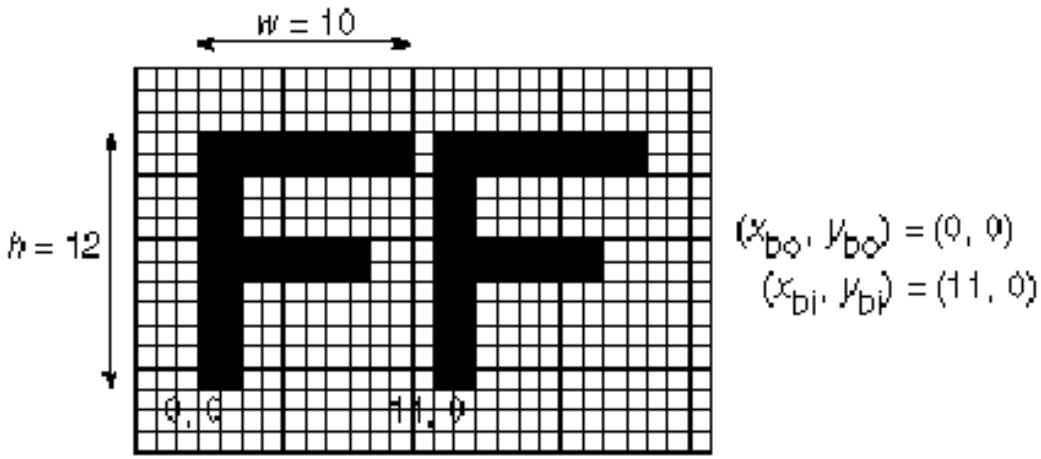
OpenGL生成基本图形—生成英文字符

glRasterPos{234}{sifd}(TYPE x,TYPE y,TYPE z,TYPE w);

该函数设置当前光栅位置，即将要绘制的位图的原点。

void glBitmap(GLsizei width,GLsizei height,GLfloat x_{bo},GLfloat y_{bo},GLfloat x_{bi},GLfloat y_{bi},const GLubyte *bitmap);

该函数用来绘制位图



OpenGL生成基本图形—生成英文字符

```
glPixelStorei(GL_UNPACK_ALIGNMENT,1
);
//设置像素存储模式
 GLubyte rasters[24] = {
      0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0,
0x00,       0xc0, 0x00,   0xff, 0x00, 0xff, 0x00,
0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,   0xff, 0xc0,
0xff, 0xc0};
   glRasterPos2i(0, 0);
   glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
   glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
   glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
```