# 第3章 光栅图形学

# 课程目标

- 二维图形补充知识
- 普通直线段扫描转换算法
- 数值微分画线算法
- 中点画线算法
- Bresenham画线算法

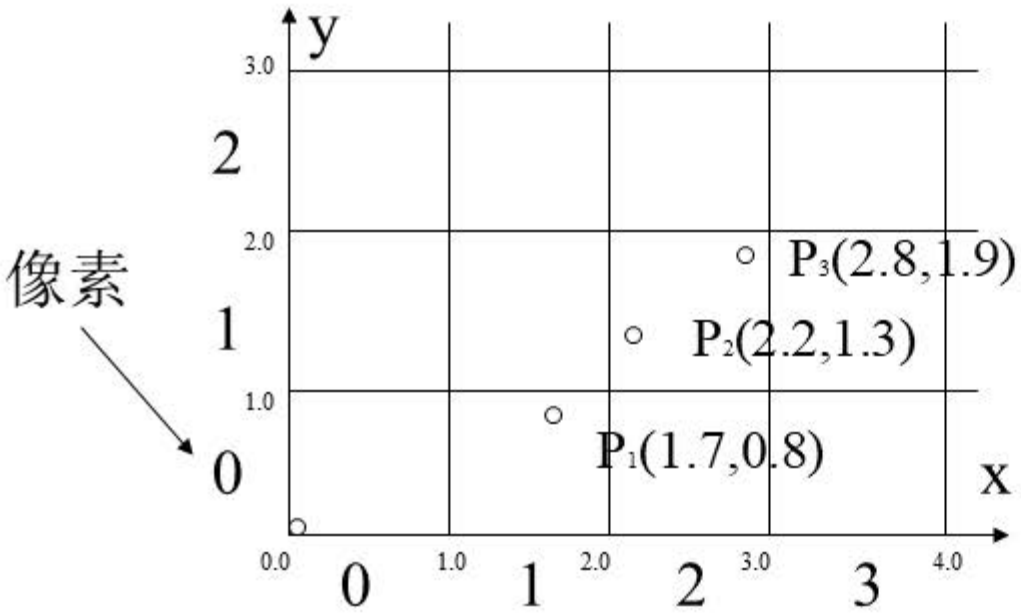
# 像素的扫描转换

- 实质：将一个图形区域上的数学点(x,y)转化为像素点(x',y')。
- 实现方法1：取x的整数部分作为x'，取y的整数部分作为y'。

函数：$x' = \mathrm{Int}(x) \quad x' \le x < x'+1$

$y' = \mathrm{Int}(y) \quad y' \le y < y'+1$

# 像素的扫描转换




$P_1(1.7,0.8)$ ⇨ 像素点（1,0）

$P_2(2.2,1.3)$ ⇨ 像素点（2,1）

$P_3(2.8,1.9)$ ⇨ 像素点（2,1）

# 像素的扫描转换

- 实现方法2：用把像素坐标排列(x,y)所在连续坐标系统的取整数值

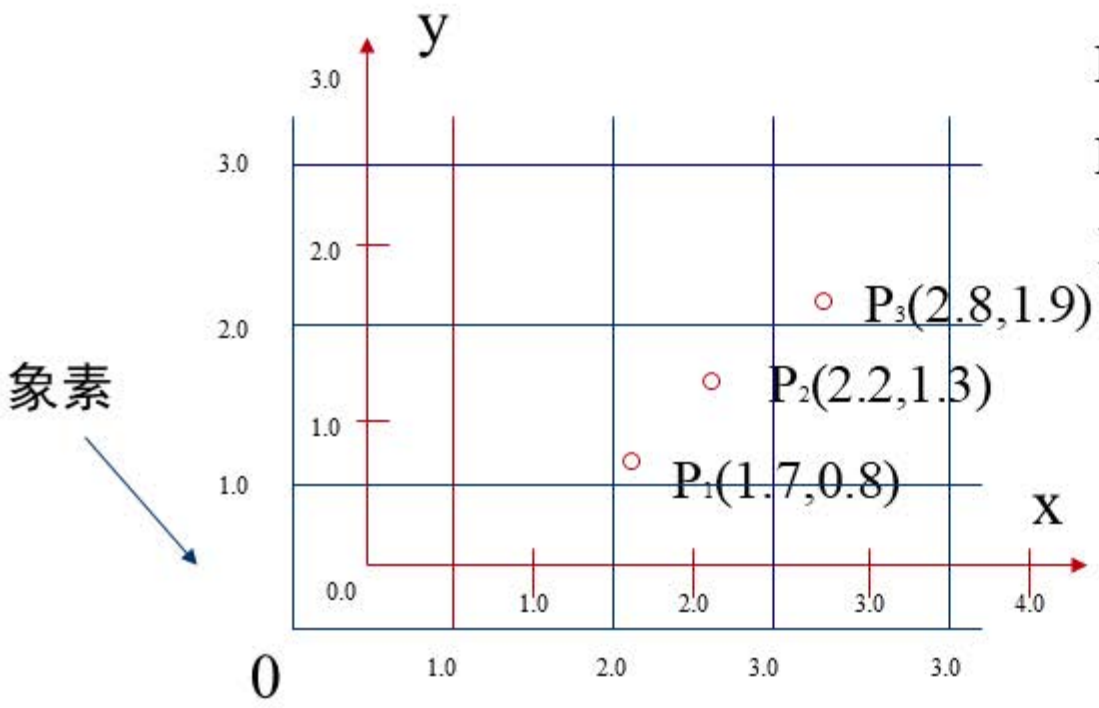
函数：$x' = \text{Round}(x+0.5)$

$y' = \text{Round}(y+0.5)$

$x' - 0.5 \le x < x' + 0.5$

$y' - 0.5 \le y < y' + 0.5$

# 像素的扫描转换




$P_1(1.7,0.8)$ ➡ 像素点（**2,1**）

$P_2(2.2,1.3)$ ➡ 像素点（**2,1**）

$P_3(2.8,1.9)$ ➡ 像素点（**3,2**）

# 3.2直线段扫描转换

- 数值微分法(DDA)
- 中点画线法
- Bresenham算法

# 画直线的基本要求

- 直线必须有精确的起点和终点，外观要直，线宽应当均匀一致、且与直线的长度和方向无关，最后，算法速度要快。

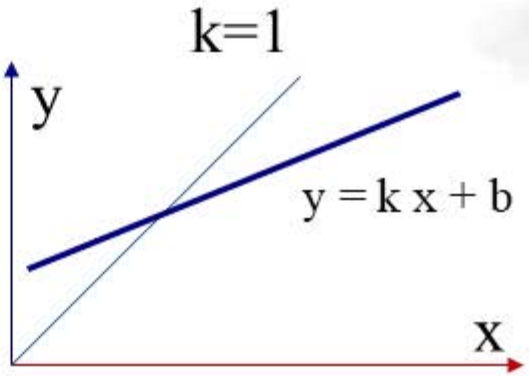
# 方法一：直接使用直线方程

$P_1$和$P_2$转换到象素坐标$(x_1', y_1')$，$(x_2', y_2')$

$$k = (y_2' - y_1') / (x_2' - x_1')$$

$$b = y_1 - k x_1'$$

$$y = k x + b$$




若$|k| \le 1$，对所有$(x_1', x_2')$间的整数x，计算y，得(x,y)

若$|k| > 1$，对所有$(y_1', y_2')$间的整数y，计算x，得(x,y)

缺点：每一步都要使用浮点乘法和除法运算

# 方法二：DDA算法

数字微分分析器算法
(Digital Differential Analyzer)

特点：每一步计算都要用到上一步的结果，增量扫描转换方法，

假设已算出$(x_i, y_i)$，如何求下一点$(x_{i+1}, y_{i+1})$呢？

$$k=\triangle y / \triangle x \qquad (\triangle y = y_{i+1} - y_i \qquad \triangle x = x_{i+1} - x_i)$$

假定：x从起点到终点变化，每步递增1，

计算对应的y坐标：y=kx+b 并取象素(x,Round(y))

优点：直观，可行

缺点：效率低，每步的运算都需一个浮点乘法与舍入运算.

# 方法二：DDA算法

假设： $y_i = kx_i + b$

$y_{i+1} = kx_{i+1} + b$

$= k(x_i + \triangle x) + b$

$= kx_i + b + k\triangle x$

$= y_i + k\triangle x$

故： $y_{i+1} = y_i + k\triangle x$ 或者 $x_{i+1} = x_i + \triangle y / k$

(1) 当 $|k| \le 1$，从 $x = x_1'$，$y = y_1'$ 开始，设 $\triangle x = 1$

$y_{i+1} = y_i + k\triangle x$

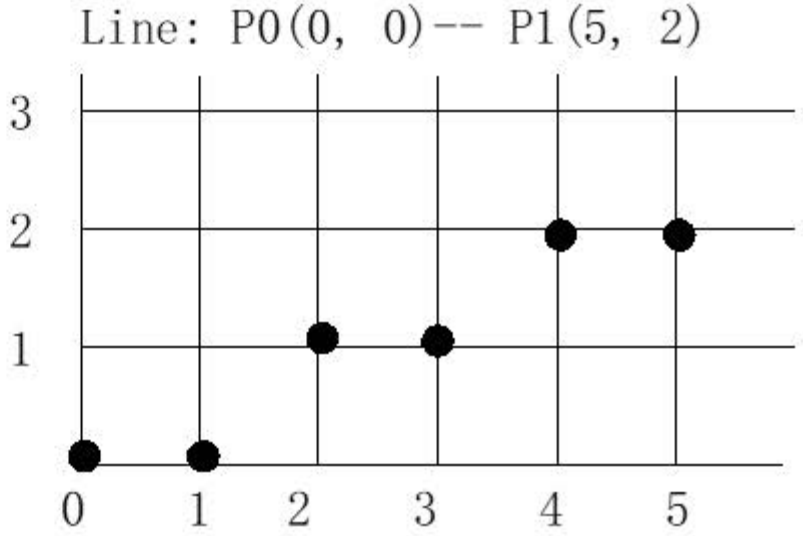
(2) 当 $|k| > 1$，从 $x = x_1'$，$y = y_1'$ 开始，设 $\triangle y = 1$

$x_{i+1} = x_i + \triangle y / k$

# 方法二：DDA算法

例：画直线段P0(0,0)—P1(5,2)　k=0.4

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



# 方法二：DDA算法

```
DDAline(int x0,int y0,int x1,int y1,int color)
{
    int x;
    float dx,dy,k,y;
    dx=x1-x0;
    dy=y1-y0;
    k=dy/dx;
    y=y0;
    for (x=x0;x<=x1;x++)
    {
      putpixel(x,int(y+0.5),color);
      y=y+k;
    }
}
```

注:算法只适应于|m|≤1，x每增加1，y最多增加1，在迭代的每一步,只要确定一个象素.

$$y_{i+1} = y_i + k \triangle x$$

当|m|>1时候,必须把x,y的地位交换

$$x_{i+1} = x_i + \triangle y / k$$

没有考虑垂直线和水平线的情况

# 方法二：DDA算法

```
#include<graphics.h>
#include<conio.h>
#include<math.h>

void DDALine(int x1,int y1,int x2,int y2,int color)
{
    int x;
    float k,y=y1;
    k=1.0*(y2-y1)/(x2-x1);
    for(x=x1;x<=x2;x++)
    {
        putpixel(x,(int)(y+0.5),color);
        y=y+k;
    }
}
void DDALine_all(int x1,int y1,int x2,int y2,int color)
{
    int i,length;
    float dx,dy,x=x1,y=y1;
    if (abs(x2-x1)>=abs (y2-y1))
       length=abs(x2-x1);
    else
       length=abs(y2-y1);
    dx=(float)(x2-x1)/length;
    dy=(float)(y2-y1)/length;

    putpixel((int)(x+0.5),(int)(y+0.5),color);
    for(i=1;i<=length;i++)
    {
        x=x+dx;
        y=y+dy;
    putpixel((int)(x+0.5),(int)(y+0.5),color);
    }
}
```

# 方法二：DDA算法

- 本质：用数值方法解微分方程，通过x和y各增加一个小增量，计算下一步的x,y值.
- 增量算法：每一步的x,y的值用前一步的值加上一个增量来获得.
- 缺陷：y与k必须用浮点数表示，且需要舍入取整，不利于硬件实现.

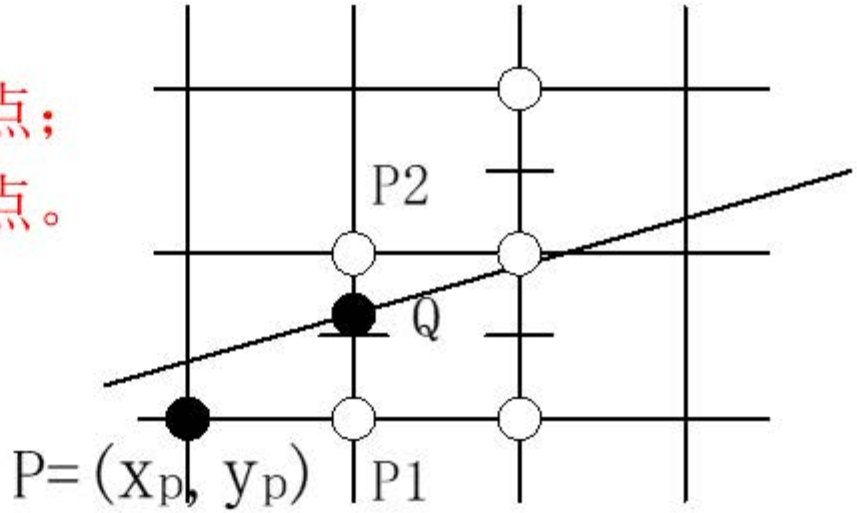
# 方法三:中点画线法

- 基本思想

当前象素点为$(x_p, y_p)$，下一个象素点为$P_1$或$P_2$。

设$M=(x_p+1, y_p+0.5)$，为$P_1$与$P_2$的中点，Q为理想直线与$x=x_p+1$垂线的交点。将Q与M的y坐标进行比较。

当M在Q的下方，应取$P_2$为下一点；
当M在Q的上方，应取$P_1$为下一点。



# 方法三:中点画线法

构造判别式:$d=F(M)=F(x_p+1,y_p+0.5)$

$=a(x_p+1)+b(y_p+0.5)+c$

其中$a=y_0-y_1$, $b=x_1-x_0$, $c=x_0y_1-x_1y_0$

当$d<0$，M在L(Q点)下方，取右上方$P_2$为下一个象素；

当$d>0$，M在L(Q点)上方，取右方$P_1$为下一个象素；

当$d=0$，选$P_1$或$P_2$均可，约定取$P_1$为下一个象素；

d是$x_p$, $y_p$的线性函数，因此可采用增量计算，提高运算效率。

# 方法三:中点画线法

若d≥0时，则取正右方象素P1($x_p$+1, $y_p$)
要判下一个象素位置，则要计算

$d_1$=F($x_p$+2, $y_p$+0.5)=a($x_p$+2)+b($y_p$+0.5)=d+a； 增量为a

若d<0时，则取右上方象素P2（$x_p$+1, $y_p$+1）
要判断再下一象素，则要计算

$d_2$= F($x_p$+2, $y_p$+1.5)=a($x_p$+2)+b($y_p$+1.5)+c=d+a+b ； 增量为a＋b

画线从($x_0$, $y_0$)开始，d的初值

$d_0$=F($x_0$+1, $y_0$+0.5)=F($x_0$, $y_0$)+a+0.5b =a+0.5b。

可以用2d代替d来摆脱小数，提高效率。

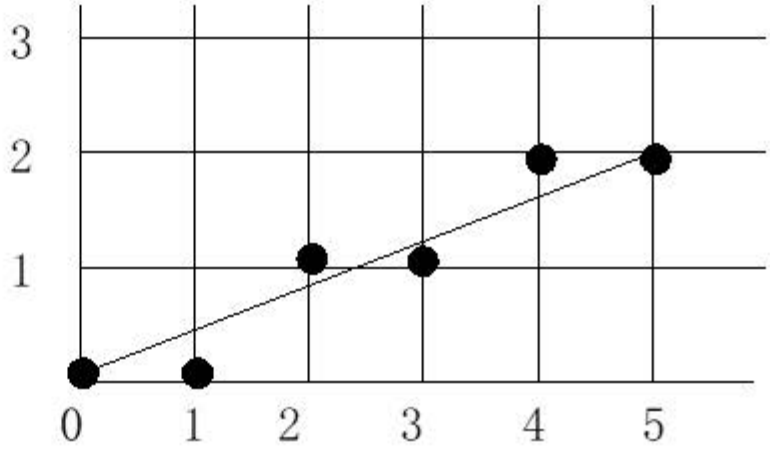
# 方法三:中点画线法

- 例：用中点画线法P0（0,0）P1（5,2）

$a=y_0-y_1=-2$　　$b=x_1-x_0=5$

$d_0=2a+b=1$　　$d_1=2a=-4$　　$d_2=2(a+b)=6$

| i | xi | yi | d |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | -3 |
| 3 | 2 | 1 | 3 |
| 4 | 3 | 1 | -1 |
| 5 | 4 | 2 | 5 |



# 方法三:中点画线法

```
void MidpointLine (int x0,int y0,int x1, int y1,int color)
{   int a, b, d1, d2, d, x, y;
    a=y0-y1, b=x1-x0, d=2*a+b;
    d1=2*a, d2=2* (a+b);
    x=x0, y=y0;
    putpixel(x, y, color);
    while (x<x1)
    {   if (d<0)      {x++, y++, d+=d2; }
        else      {x++, d+=d1;}
        putpixel (x, y, color);
    } /* while */
} /* midPointLine */
```

# 方法三:中点画线法

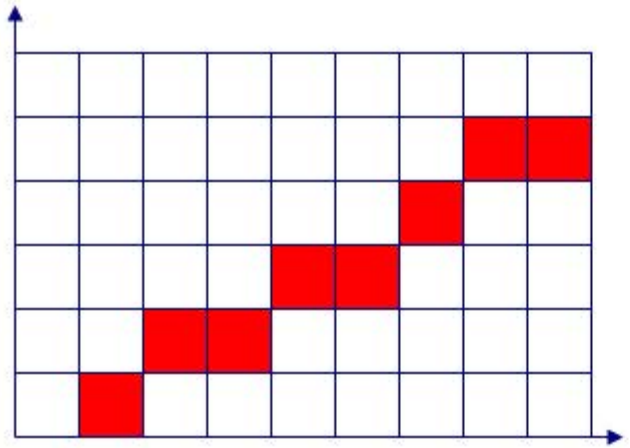
```
int Sign(int x)
{
    if(x<0) return -1;
    else if(x==0) return 0;
    else return 1;
}
void MPLine(int x1,int y1,int x2,int y2,int color)
{
    int x,y,a,b,d,d1,d2;
    a=y1-y2; b=x2-x1;
    y=y1;
    d=2*a+b; d1=2*a; d2=2*(a+b);
    putpixel(x,y,color);
    for(x=x1;x<=x2;x++)
    {
    if(d<0) { y++; d+=d2;}
        else {d+=d1;}
        putpixel(x,y,color);
    }
}
void MPLine_all(int x1,int y1,int x2,int y2,int color)
{
    int x,y,a,b,d1,d2,d,i,s1,s2,temp,swap;
    a=-abs(y2-y1); b=abs(x2-x1);
    x=x1; y=y1;
    s1=Sign(x2-x1);s2=Sign(y2-y1);
    if(-a>b) { temp=b;b=-a;a=-temp;swap=1;}
    else  swap=0;
    d=2*a+b; d1=2*a; d2=2*(a+b);
    putpixel(x,y,color);
    for(i=1;i<=b;i++)
    {
        if(swap==1)  y=y+s2;
        else  x=x+s1;
    if(d<0)
    {
        if(swap==1)  x=x+s1;
            else  y=y+s2;
        d+=d2;
    }
        else {d+=d1;}
        putpixel(x,y,color);
    }
}
```

# 方法四:Bresenham算法

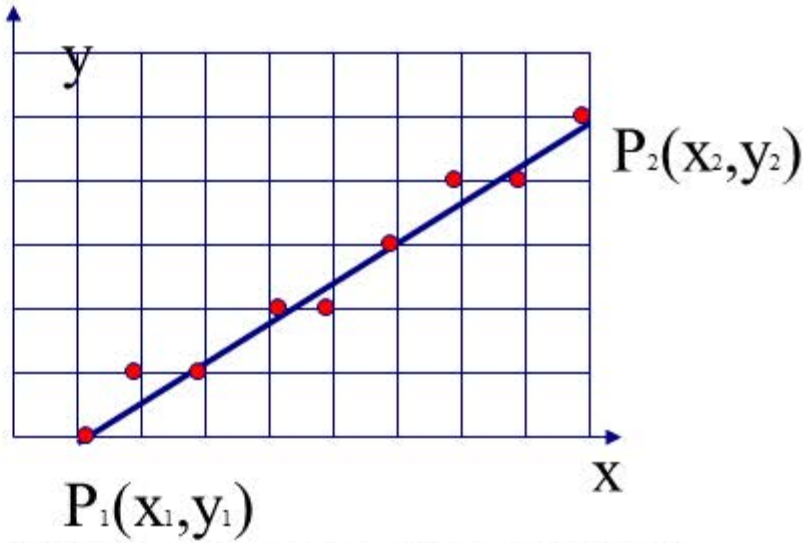
- 背景：Bresenham在1965年提出，最初是为数字绘图仪设计的，后来被广泛地应用于光栅图形显示和数控加工。
- 优点:只需要检查判别式的符号，和进行整型数的计算
- 工作原理:根据直线的斜率在x或y的方向每次都只递增一个象素单位，另一个方向的增量为0或1

# 方法四:Bresenham算法

- 工作原理:根据直线的斜率在x或y的方向每次都只递增一个象素单位，另一个方向的增量为0或1



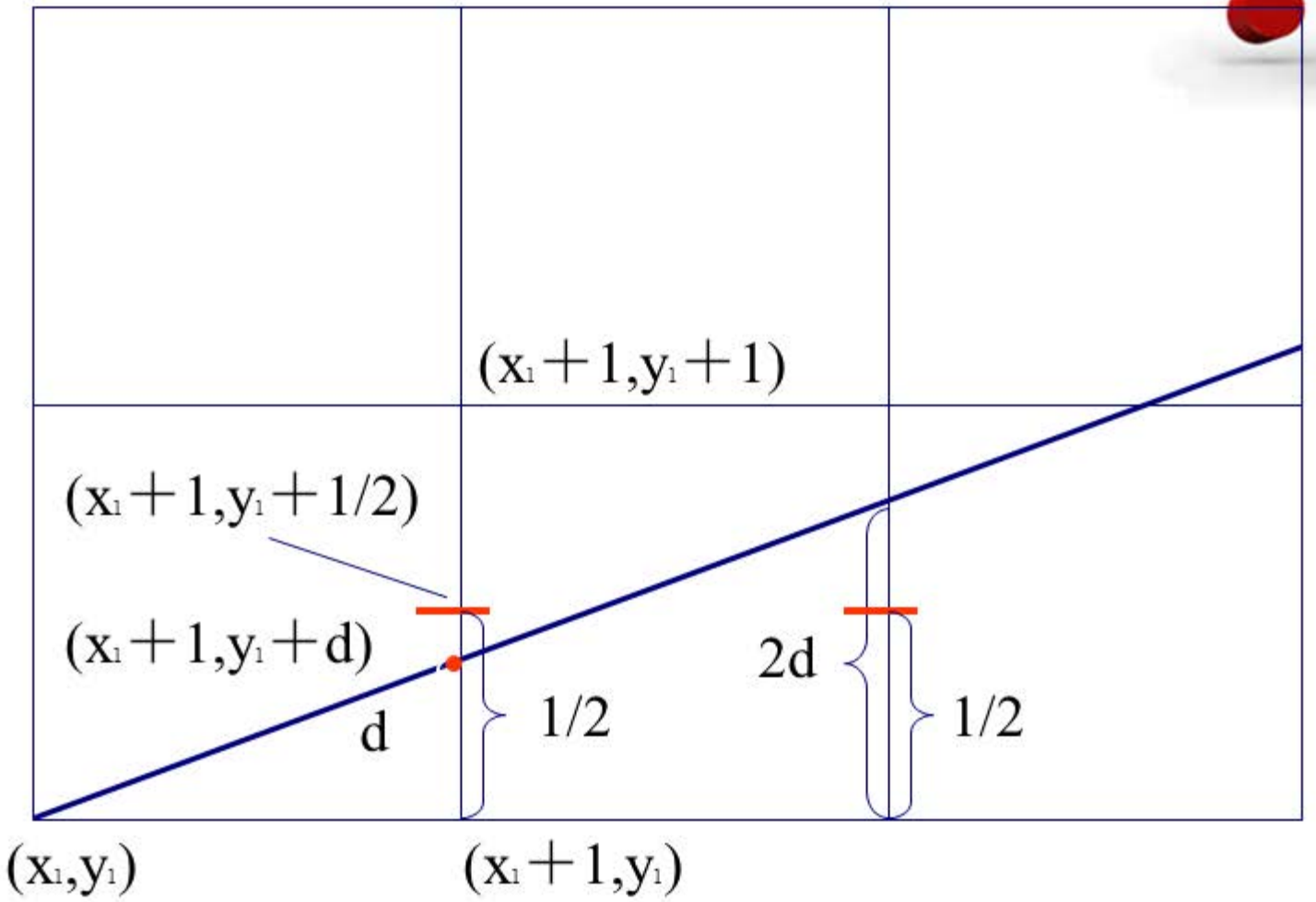
屏幕上的直线近似图



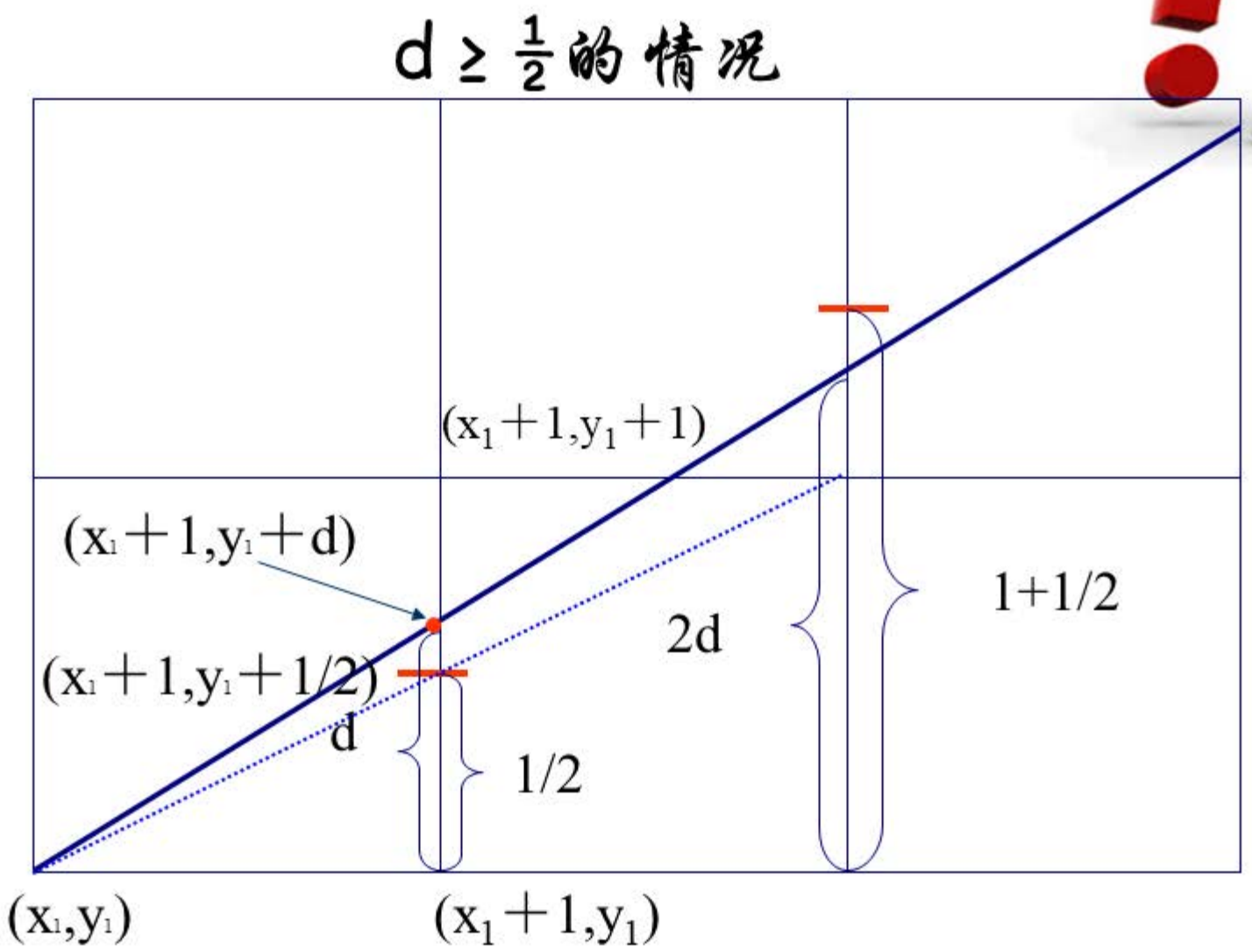

二维笛卡儿坐标系上的直线近似图

# 方法四:Bresenham算法

d < 1/2的情况



# 方法四:Bresenham算法



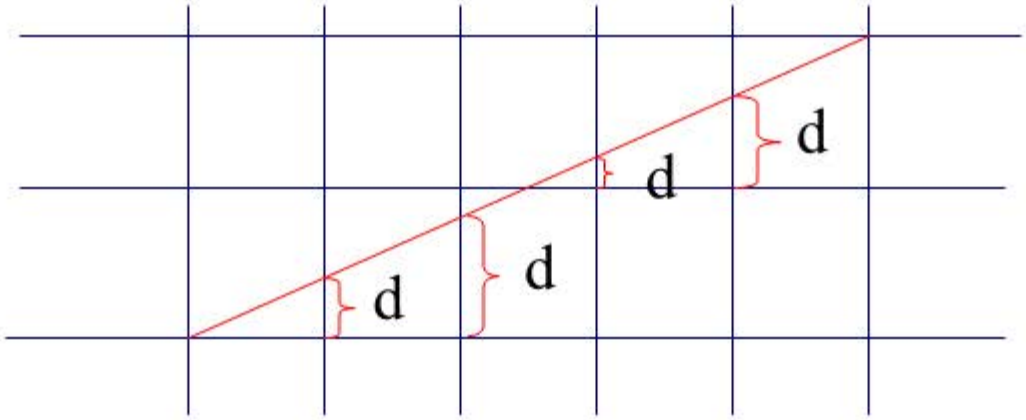
# 方法四:Bresenham算法

令e=d-0.5 则:

（1）e>=0时，下一象素的x下标增加1

（2）e ＜0时，下一象素的x下标不增，e的初值为-0.5

$e_{i+1}$与 0.5 进行比较，如此进行循环



# 方法四:Bresenham算法

设直线方程为：$y_{i+1}=y_i+k$ ，其中k=dy/dx。

因为直线的起始点在象素中心，所以误差项d的初值d0＝0。

x下标每增加1，d的值相应递增直线的斜率值k，即d＝d＋k。

一旦d≥1，就把它减去1，这样保证d在0、1之间。

- 当d≥0.5时，取右上方象素（xi＋1，yi＋1）
- 当d＜0.5时，取右方象素（xi＋1，yi）

为方便计算，令e＝d-0.5，

e的初值为-0.5，增量为k。

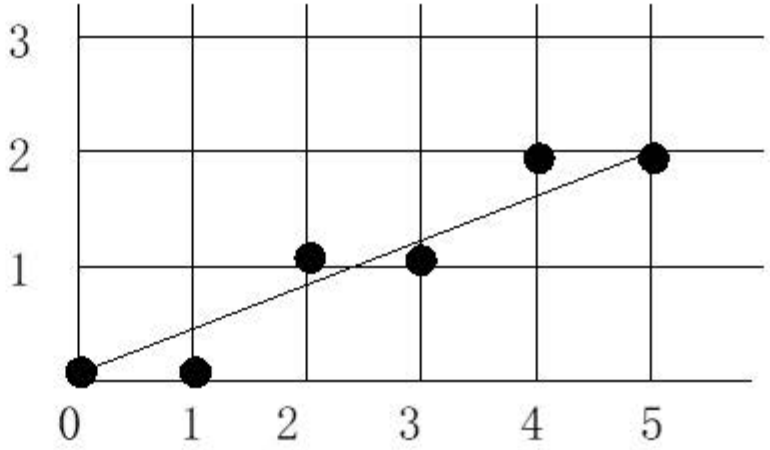
- 当e≥0时，取右上方象素（xi＋1，yi＋1）
- 当e＜0时，取右方象素（xi＋1，yi）

# 方法四:Bresenham算法

例：Line: P0(0, 0), P1(5,2)

k=dy/dx=0.4

| x | y | e |
|---|---|---|
| 0 | 0 | -0.5 |
| 1 | 0 | -0.1 |
| 2 | 1 | -0.7 |
| 3 | 1 | -0.3 |
| 4 | 2 | -0.9 |
| 5 | 2 | -0.5 |



# 方法四:Bresenham算法

```
void Bresenhamline(int x0,int y0,int x1, int y1,int color)
{
   int i,x, y, dx, dy;
   float k, e;
   dx = x1-x0;dy = y1- y0;k=dy/dx;
   e=-0.5; x=x0,y=y0;
   for (i=0;i<dx;i++)
   {  putpixel (x, y, color);
      x=x+1;e=e+k;
      if (e>=0)
       { y++; e=e-1;}
    }
}
```

用到小数与除法。
可以改用整数以避免除法。
由于算法中只用到误差项的符号，因此可作如下替换：$2*e*dx$。

# 方法四:Bresenham算法

d与 0.5 进行比较 ⇒ e与 0 进行比较

d = dy/dx ⇒ e = dy/dx − ½

放大2dx倍 ⇓

e ' =2e*dx= 2dy − dx

d ＜ 0.5时　　e＜ 0时

di＋1 ＝di＋ dy/dx ⇒ e ' = e ' + 2dx dy/dx

即e ' = e ' + 2dy

d ≥ 0.5时　　e ≥0时

di＋1 ＝di＋ dy/dx−1 ⇒ e ' = e ' + 2dx dy/dx − 2dx

即e ' = e ' + 2(dy − dx)

# 方法四:Bresenham算法

改进算法的形式描述

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

# 方法四:Bresenham算法

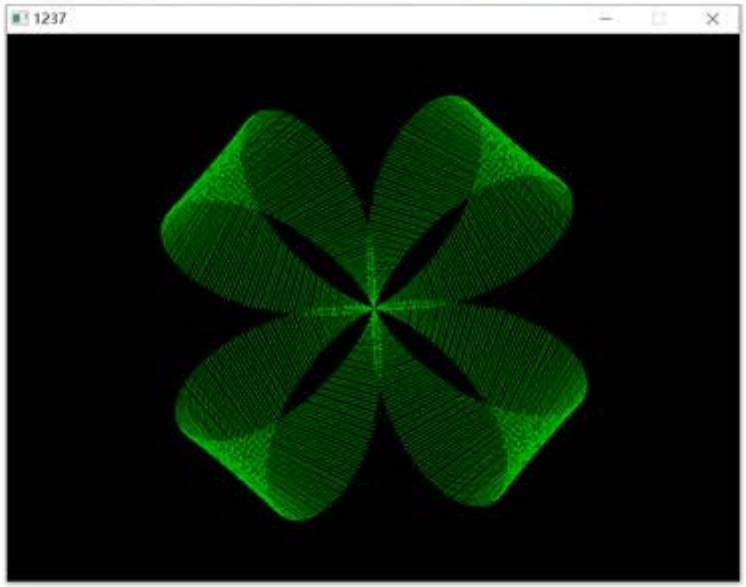
```
void BHLine(int x1,int y1,int x2,int y2,int color)
{
    int x,y,dx,dy,dk;
    dx=abs(x2-x1);dy=abs(y2-y1);
    dk=2*dy-dx;y=y1;
    for(x=x1;x<=x2;x++)
    {
    putpixel(x,y,color);
        dk=dk+2*dy;
        if(dk>=0)
    {  y++; dk=dk-2*dx;}
    }
}
void BHLine_all(int x1,int y1,int x2,int y2,int color)
{
    int x,y,dx,dy,dk,i,s1,s2,temp,swap;
    dx=abs(x2-x1);dy=abs(y2-y1);
    x=x1;y=y1;
    s1=Sign(x2-x1);s2=Sign(y2-y1);
    if(dy>dx) { temp=dx;dx=dy;dy=temp;swap=1;}
    else  swap=0;
    dk=2*dy-dx;
    for(i=1;i<=dx;i++)
    {
    putpixel(x,y,color);
        if(swap==1)  y=y+s2;
        else  x=x+s1;
        dk=dk+2*dy;
        if(dk>=0)
    {
        if(swap==1)  x=x+s1;
            else  y=y+s2;
            dk=dk-2*dx;
    }
    }
}
```

# 方法四:Bresenham算法

```
int main()
{
    initgraph(640, 640);
    setbkcolor(RGB(237, 202, 160));
    cleardevice();
    setfillcolor(RGB(249, 211, 172));
    DDALine(10,100,200,20,1);
    DDALine_all(10,10,200,120,1);
    DDALine_all(210,210,20,100,1);
    MPLine(40,160,200,60,2);
    MPLine_all(40,100,200,160,2);
    MPLine_all(250,120,60,160,2);
    BHLine(50,250,200,200,3);
    BHLine_all(50,250,200,200,3);
    BHLine_all(220,300,40,60,3);
    _getch();
    return 0;
}
```


line.cpp

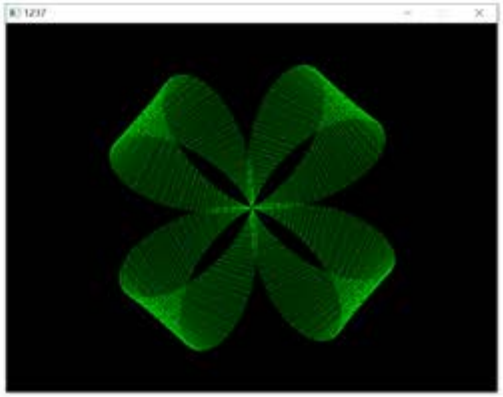
# 四叶草




动态绘制四叶草

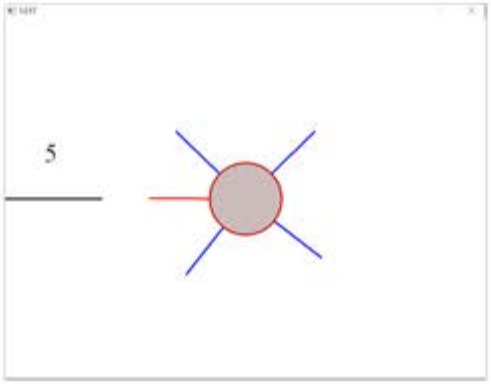
# 四叶草

```
#include <graphics.h>
#include <math.h>
#include <conio.h>

#define PI 3.1415926535

void main(void)
{
    // 初始化绘图窗口
    initgraph(640, 480);          // 创建绘图窗口
    setcolor(GREEN);              // 设置绘图颜色
    setorigin(320, 240);          // 设置原点坐标

    // 画花朵
    double e;
    int x1, y1, x2, y2;
    for(double a = 0; a < 2 * PI; a += 2 * PI / 720)
    {
        e = 100 * (1 + sin(4 * a));
        x1 = (int)(e * cos(a));
        y1 = (int)(e * sin(a));
        x2 = (int)(e * cos(a + PI / 5));
        y2 = (int)(e * sin(a + PI / 5));

        line(x1, y1, x2, y2);

        Sleep(20);       // 延迟函数，实现慢速绘制的动画效果
    }

    // 按任意键退出
    _getch();
    closegraph();
}
```



clover.cpp

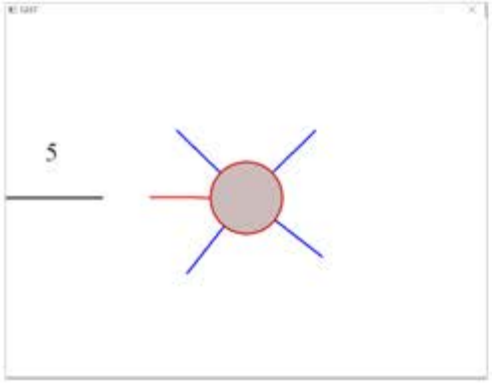
# 见缝插针




- 见缝插针的游戏。按下空格键后发射一根针到圆盘上，所有针逆时针方向转动；如果新发射的针碰到已有的针，游戏结束。
- 首先进行了圆盘与针的绘制，利用三角函数实现了针的旋转；然后利用数组实现了多根针的效果；利用批量绘制函数改进了绘制效果；最后实现了针的发射与增加、游戏失败判断、得分与显示效果的改进。

# 见缝插针

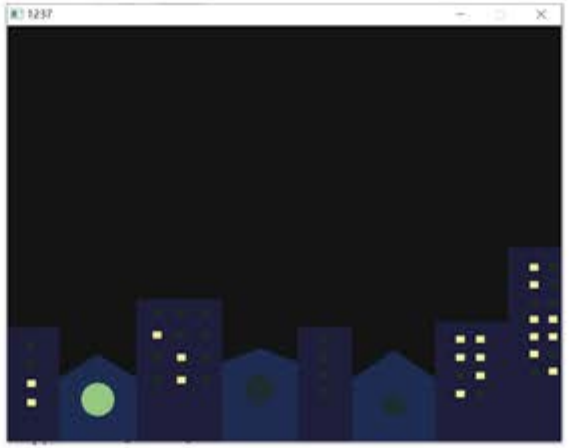
```
#include <graphics.h>
#include <conio.h>
#include <stdio.h>
#include <math.h>
int main()
{
    const float Pi = 3.1415926; // Pi常量
    int width = 800; // 画面宽度
    int height = 600; // 画面高度
    initgraph(width,height); // 新开一个画面
    setbkcolor(RGB(255,255,255)); // 背景为白色
    setlinestyle(PS_SOLID,3); // 线宽为3，这样针看起来更明显

    float lineLength = 160; // 针的长度
    float xEnd,yEnd; // 针的终点位置坐标（针起始位置为圆心）
    float rotateSpeed = Pi/360; // 针的旋转速度
    int lineNum = 0;  // 在旋转的针的个数
    float Angles[1000]; // 浮点数数组，存储所有针的旋转角度，最多1000根针
    int score = 0; // 得分
    int i;
```




jiànfèngchāzhēn

```
    BeginBatchDraw(); // 开始批量绘制
    while (1) // 重复循环
    {
        cleardevice(); // 以背景色清空背景
        setlinecolor(RGB(0,0,0)); // 设置针颜色为黑色
        line(0,height/2,lineLength,height/2); // 左边发射区域的一根针

        for (i=0;i<lineNum;i++) // 对所有旋转针进行遍历
        {
            Angles[i] = Angles[i] + rotateSpeed; // 角度增加
            if (Angles[i]>2*Pi) // 如果超过2*Pi，就减去2*Pi，防止角度数据无限增加
                Angles[i] = Angles[i] - 2*Pi; //
            xEnd = lineLength*cos(-Angles[i]) +width/2; // 计算针的末端坐标
            yEnd = lineLength*sin(-Angles[i]) +height/2;
            setlinecolor(RGB(0,0,255)); // 设定旋转针的颜色为蓝色
            if (i==lineNum-1) // 最新发射的一根针，设定颜色为红色
                setlinecolor(RGB(255,0,0));
            line(width/2,height/2,xEnd,yEnd); // 绘制一根针
        }

        if (kbhit() && rotateSpeed!=0) // 如果按键，并且旋转速度不等于0
        {
            char input = _getch(); // 获得用户按键输入
            if (input==' ') // 如果为空格键
            {
                lineNum++; // 针的个数加1
                Angles[lineNum-1] = Pi; // 这根新增加针的初始角度
                xEnd = lineLength*cos(-Angles[lineNum-1]) +width/2; //新增针的末端坐标
                yEnd = lineLength*sin(-Angles[lineNum-1]) +height/2;
                line(width/2,height/2,xEnd,yEnd); // 绘制出这根新增加的针
                for (i=0;i<lineNum-1;i++) // 拿新增加的针和之前所有针比较
                {
                    // 如果两根针之间角度接近，认为碰撞，游戏失败
                    if (abs(Angles[lineNum-1]-Angles[i]) < Pi/60)
                    {
                        rotateSpeed = 0; // 旋转速度设为0
                        break; // 不用再比较了，循环跳出
                    }
                }
                score = score + 1; // 得分+1
            }
        }
        setfillcolor(HSVtoRGB(0,lineNum/60.0,0.8)); // 绘制中间的圆盘，针越多，其颜色越鲜艳
        setlinecolor(HSVtoRGB(0,0.9,0.8)); // 设置圆盘线条颜色为红色
        fillcircle(width/2,height/2,60); // 绘制中间的圆盘
        TCHAR s[20]; // 定义字符串数组
        _stprintf(s, _T("%d"), score); // 将score转换为字符串
        settextstyle(50, 0, _T("Times")); // 设置文字大小、字体
        settextcolor(RGB(50,50,50));  // 设置字体颜色
        outtextxy(65, 200, s); // 输出得分文字

        FlushBatchDraw(); // 批量绘制
        Sleep(10); // 暂停10毫秒
    }
    closegraph();  // 关闭画面
    return 0;
}
```

# 随机闪电



- 绘画出 简单的随机街道，
- 街道上空出现 各种形态的闪电，
- 并实现街道在闪电时的 闪光变化

lighting.cpp

# 作业

- 编写完整的直线生成算法(实线，虚线)。
- 结合自己编写的直线函数，绘制作品。