第二章 光栅图形学

2.1 直线段的扫描转换算法

2.2 圆弧的扫描转换算法

2.3 多边形的扫描转换与区域填充

2.4 字符

2.5 裁剪

2.6 反走样

2.1 直线段的扫描转换算法

直线的扫描转换: 确定最佳逼近于该直线的一组象素，并且按扫描线顺序，对这些象素进行写操作。 三个常用算法:

2.1.1 **数值微分法（DDA）**

2.1.2 **中点画线法**

2.1.3 **Bresenham算法**

2.1.1 数值微分(DDA)法
--- Digital Differential Analyzer

基本思想：

已知过端点$P_0(x_0, y_0)$、$P_1(x_1, y_1)$的直线段L：$y = kx + B$，其中直线斜率：

$$k = \frac{y_1 - y_0}{x_1 - x_0}$$

从x的左端点x_0开始，向x右端点步进。步长 = 1(个象素)，计算相应的y坐标$y = kx + B$；取象素点$(x, \text{round}(y))$作为当前点的坐标。

$$y_{i+1} = kx_{i+1} + B = k(x_i + \Delta x) + B$$

$$= k\,x_i + B + k\Delta x$$

$$= y_i + k\Delta x$$

当Δx =1时，$y_{i+1} = y_i + k$，即：当x每递增1，y递增k(直线斜率)。

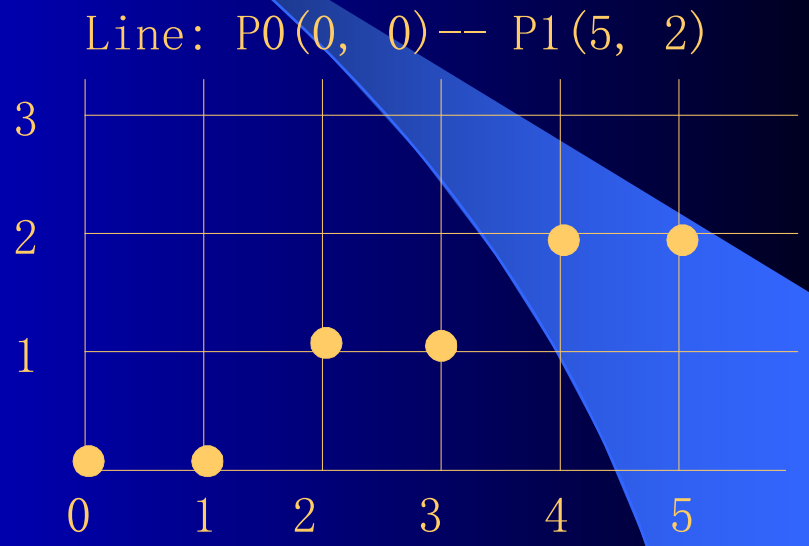
上述分析的算法仅适用于$|k| \leq 1$的情形。在这种情况下，x每增加1，y最多增加1；当$|k| > 1$时，必须把x、y地位互换

例：画直线段 $P_0(0, 0)$ -- $P_1(5, 2)$

斜率 *k* = (2-0)/(5-0) = 0.4

四舍五入！

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



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

下面是一个适合于所有象限的简单DDA算法

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

DDA算法的缺点：

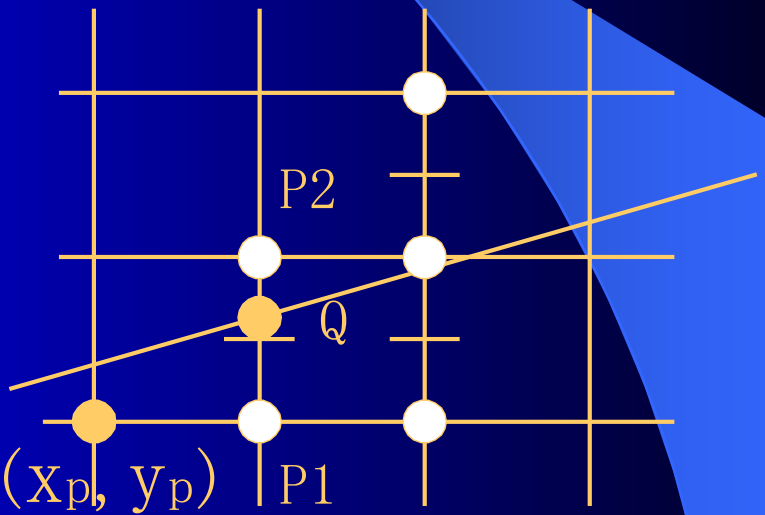
在这个算法中，y 与 k 必须用浮点数表示，而且每一步都要对 y 进行四舍五入后取整。

DDA算法不利于硬件实现！

2.1.2 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之中点，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**，则P_2应为下一个象素点；
- **M在Q的上方**，应取P_1为下一点。

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

令直线段P_0P_1的方程式为：

$F(x,y) = ax+by+c = 0$

当 $\boldsymbol{d<0}$，**M在Q点下方**，取右上方P_2为下一个象素；

当 $\boldsymbol{d>0}$，**M在Q点上方**，取右方P_1为下一个象素；

当 $\boldsymbol{d=0}$，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。

初值 $\boldsymbol{d_0} = F(x_0+1, y_0+0.5) = a + 0.5b$

根据$\boldsymbol{d_0}$的符号判断在P_0左边取哪一个象素点

若当前象素处于$d \geq 0$情况，则取正右方象素$P_1(x_p+1, y_p)$, 要判下一个象素位置，应计算:

$$d_1 = F(x_p+2, y_p+0.5) = a(x_p+2) + b(y_p+0.5) + c = d + a$$

增量为a

若当前象素处于$d < 0$情况，则取右上方象素$P_2(x_p+1, y_p+1)$。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。

- 例：用中点画线法画直线段$P_0(0, 0)$ -- $P_1(5, 2)$

$a = y0 - y1 = -2 \quad b = x1 - x0 = 5$

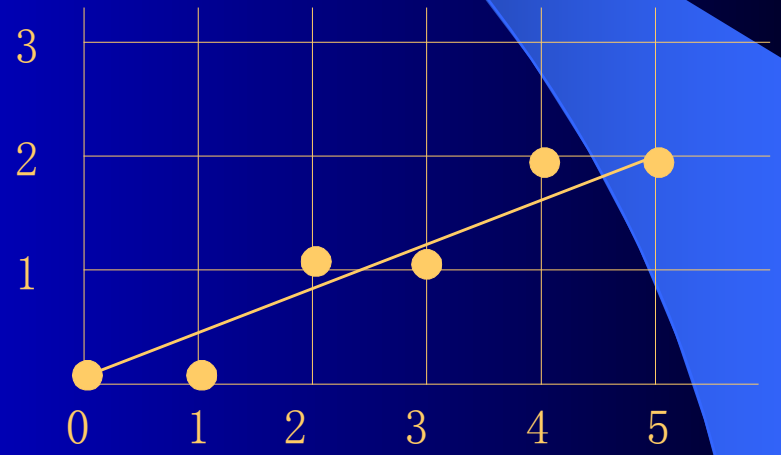
$d0=2a+b=1 \quad d1=2a=-4 \quad d2=2(a+b)=6$

i	x_i	y_i	d
1	0	0	1
2	1	0	-3
3	2	1	3
4	3	1	-1
5	4	2	5
6	5	2	1

用增量法，c不用计算!

$d0+2a$

$d0+2(a+b)$



```
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;
    drawpixel(x, y, color);
    while (x < x1) {
        if (d < 0) {x++;  y++;  d += d2; }
        else {x++;  d += d1;}
        drawpixel (x, y, color);
    }  /* while */
} /* mid PointLine */
```

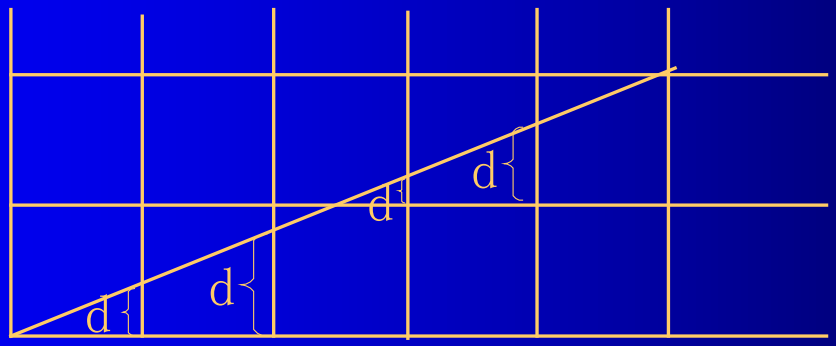
d_0

*M*在*Q*的下方，取右上方象素

2.1.3 Bresenham算法

基本思想:

过各行各列象素中心构造一组**虚拟网格线**(与DDA算法与中点画线算法中的网格线是一样的)，从起点到终点的**直线*L***与各垂直网格线相交。假定*Q*为*L*与垂线$x = x_i$的交点，则根据**误差项 *d* 的大小**可以确定该列象素中与*Q*点最近的象素。



设**直线L的方程为**：$y=kx+b$ ，其中k是斜率，则

$$y=k(x+\Delta x)+b=kx+b+k\Delta x \Rightarrow \Delta y=k\Delta x$$

DDA算法中已经推导过！

即x增加1，y增加斜率k。因为直线的起始点在象素中心，所以**误差项d**的初值$d_0=0$。x下标每增加1，d的值相应递增直线的斜率值k，即$d=d+k$。

- **当 $d\geq 0.5$ 时**，最接近于当前象素的右上方象素(x_i+1, y_i+1)
- **当 $d<0.5$ 时**，更接近于右方象素(x_i+1, y_i)

为方便计算，**令$e=d-0.5$**，则e的初值为-0.5，增量为k。

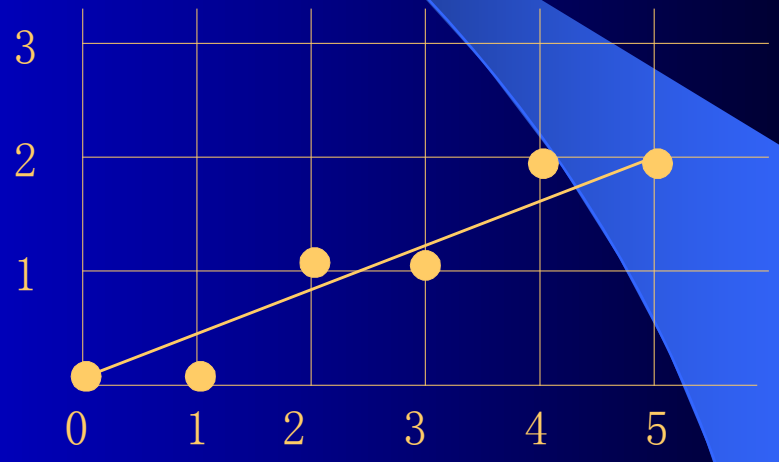
- **当 $e\geq 0$ 时**，取当前象素(x_i, y_i)的右上方象素(x_i+1, y_i+1)，**在考虑下一个象素以前，必须将误差项重新初始化，即将它减1**；
- **当 $e<0$ 时**，更接近于右方象素(x_i+1, y_i)。

由于算法中只用到误差项的符号，因此，可以作如下替换，以避免除法运算：　$e' = 2 * e * dx$

例：用Bresenham画线法画直线段

$P_0(0, 0)$ -- $P_1(5, 2)$，$k = dy/dx = 0.4$

x	y	d	e
0	0	0	-0.5
1	0	0.4	-0.1
2	1	0.8→-0.2	0.3→-0.7
3	1	0.2	-0.3
4	2	0.6→-0.4	0.1→-0.9
5	2	0	-0.5



- **d的增量为** $k = dy/dx$，**初值** $d_0 = 0$
- $e = d - 0.5$ **的增量也为** $k = dy/dx$，**初值** $e_0 = d_0 - 0.5 = -0.5$
- $e' = 2 * e * dx$ **的增量为** $2*(k = dy/dx)*dx = 2*dy$，**初值**

  $e_0' = 2 * e_0 * dx = 2 * 0.5 * dx = -dx$

```
void Bresenhamline (int x0, int y0, int x1, int y1, int color)
{
        int 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++) {
                drawpixel (x, y, color);
                x = x + 1; e = e + k;
                 if (e >= 0) { y++; e = e - 1;}
        }
 }
```

通用Bresenham算法的实现需要对其它象限的直线生成作一些修改。根据直线斜率和它所在象限很容易实现这一点。实际上，**当直线斜率的绝对值大于1时，*y*总是增1**，再用Bresenham误差判别式以确定*x*变量是否需要增1。***x*或*y*是增加1还是减去1取决于直线所在的象限**。通用Bresenham算法如下。

适用于所有象限的通用Bresenham算法

```
void Bresenhamline (int x0,int y0,int x1, int y1,int color){

    float  x, y, dx, dy, e, temp;  int s1,s2, interchange;
     x = x0; y = y0 //初始化变量
     dx = fabs(x1-x0); dy = fabs(y1- y0);
     s1 = sign(x1-x0); s2 = sign(y1-y0); //s1,s2取-1,0或1
     if (dy > dx){ //根据直线的斜率， 交换dx和dy
        temp = dx; dx = dy; dy = temp; interchange = 1;}
     else interchange = 0;
     e = 2*dy-dx; //初始化误差项
     for (i = 1; i <= int(dx); i++){ //主循环
         drawpixel (int(x), int(y), color);
         if (e > 0){
             if (interchange == 1) x = x + s1; else y = y + s2;
             e = e-2*dx;}
         if (interchange == 1) y = y + s2; else x = x + s1;
         e = e + 2*dy;
}
```

$e_0 = -0.5$

$e_1 = k + e_0 = \frac{dy}{dx} - 0.5$

$e = e - 1$

y步进1

x再步进1

$e = e + k$

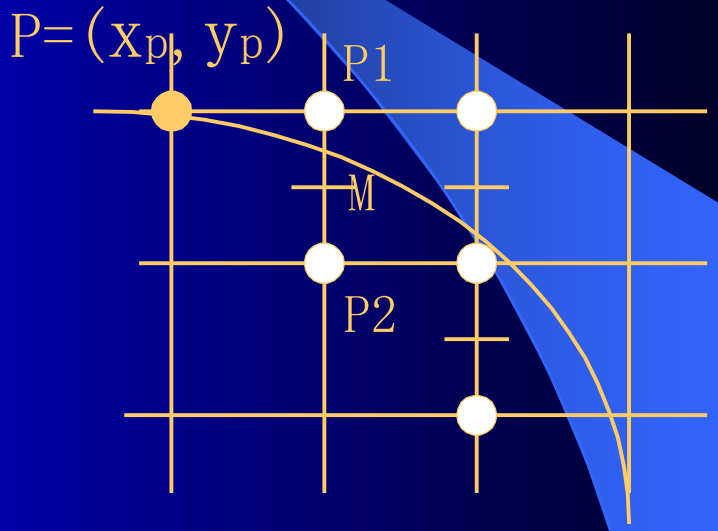
2.2 圆弧的扫描转换算法

圆的特征:八对称性。只要扫描转换八分之一圆弧，就可以求出整个圆弧的象素集

中点画圆法

考虑中心在原点，半径为R的第二个(从x正向、逆时针记数)8分圆，构造判别式(圆方程):

$$d = F(M) = F(x_p + 1, y_p - 0.5)$$
$$= (x_p + 1)^2 + (y_p - 0.5)^2 - R^2$$



若 $d<0$(即M在圆内部), 则取P_1为下一象素，而且再下一象素的判别式为:

$$d'=F(x_p+2,y_p-0.5)=(x_p+2)^2+(y_p-0.5)^2-R^2=d+2x_p+3$$

若$d>=0$ (即M在圆外部), 则应取P_2为下一象素，而且再下一象素的判别式为:

$$d'=F(x_p+2,y_p-1.5)=(x_p+2)^2+(y_p-1.5)^2-R^2=d+2(x_p-y_p)+5$$

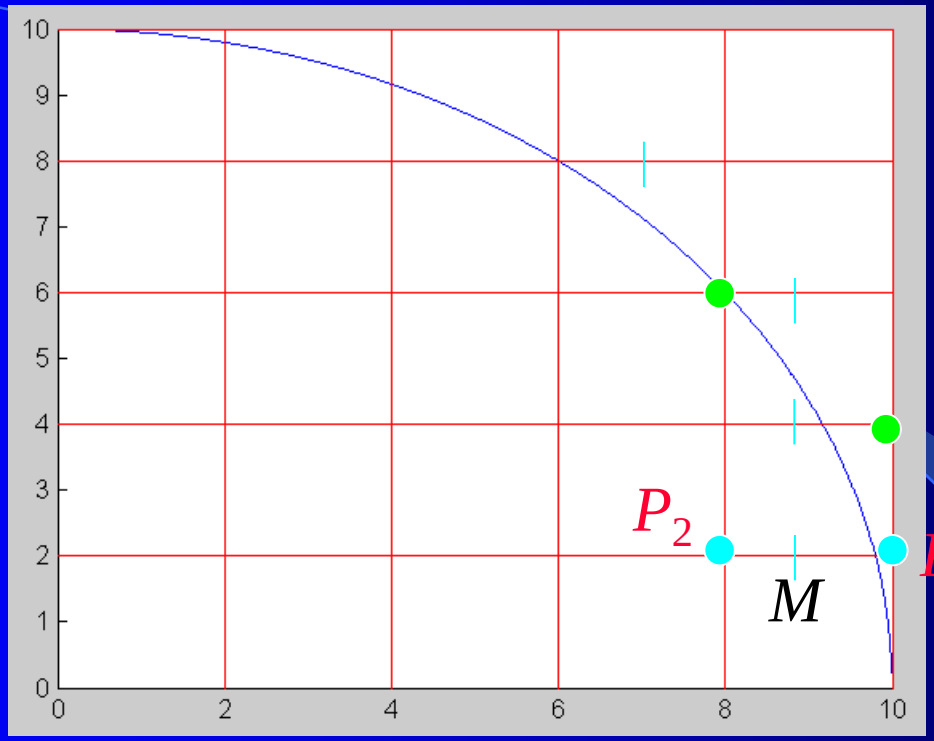
第一个象素是 (0, R)，判别式d的初始值为:

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

为了进一步提高算法的效率，可以将上面的算法中的浮点数改写成整数，将乘法运算改成加法运算，即仅用整数实现中点画圆法。

若使用$e = d - 0.25$代替d，则 $e_0 = 1 - R$。

```
void MidPointCircle(int r, int color) {
        int x, y;
        float d;
        x = 0;  y = r;  d = 1.25 - r;
        circlepoints (x, y, color);  //显示圆弧上的八个对称点
        while (x <= y) {
            if (d < 0)   d += 2*x + 3;
            else { d += 2*(x - y) + 5;  y--;}
            x++;
            circlepoints (x, y, color);
        }
}
```



$P = (x_P, y_P)$

扫描圆的第一个八分之一圆弧!

$$d = F(M) = F(x_p - 0.5, y_p + 1)$$

$$= (x_p - 0.5)^2 + (y_p + 1)^2 - R^2$$

若 $d < 0$ (即M在圆内部), 则取P_1为下一象素，而且再下一象素的判别式为:

$$d' = F(x_p - 0.5, y_p + 2) = (x_p - 0.5)^2 + (y_p + 2)^2 - R^2 = d + 2y_p + 3$$

若$d >= 0$ (即M在圆外部), 则应取P_2为下一象素，而且再下一象素的判别式为:

$$d' = F(x_p - 1.5, y_p - 2) = (x_p - 1.5)^2 + (y_p + 2)^2 - R^2 = d - 2(x_p - y_p) + 5$$

第一个象素是 **(R , 0)**，判别式d的初始值为:

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