**5 画线算法**

要在光栅监视器上显示一线段，图形系统必须先将两端点投影到整数屏幕坐标，并确定离两端点间的直线路径最近的像素位置。接下来将颜色值装入帧缓冲区相应的像素坐标处，视频控制器从帧缓冲读出写入的颜色值并绘制屏幕像素

**1）DDA（digital differential analyzer数字微分分析仪）算法**

在一个坐标轴上以单位间隔对线段取样，从而确定另一个坐标轴上最靠近线路经的对应整数

yk=mxk+b

y(k+1)=m(xk+1)+b

y(k+1)-yk=m

按照上面的方法不断递归进行下去，然后向下取整就可以得到一条直线

DDA方法计算像素位置要比直接计算得到结果速度更快。它利用光栅特性消除了惩罚，而在x或y方向使用合适的增量，从而沿线路经逐步得到各像素的位置，但在浮点数增量的连续叠加过程中，取整误差的累积使得对于较长线段所极端的像素位置偏离实际线段，而且该过程中的取整操作和浮点运算仍然十分耗时

void drawLineDDA(GLint x0, GLint y0, GLint x1, GLint y1)

{

int dx = x1 - x0, dy = y1 - y0, steps = 0;

float x = x0, y = y0, xIncrement = 0.0f, yIncrement = 0.0f;

if(fabs((float)dx) > fabs((float)dy))

{

steps = fabs((float)dx);

}

else

{

steps = fabs((float)dy);

}

xIncrement = (float)dx / (float)steps;

yIncrement = (float)dy / (float)steps;

drawPixel(round(x0), round(y0));

for(int i = 0; i < steps; i++)

{

x += xIncrement;

y += yIncrement;

drawPixel(round(x), round(y));

}

}

**2）Bresenham算法**

Bresenham算法是由bresenham提出的一种精确而有效的光栅线生成算法，该算法仅仅使用增量整数计算。另外，bresenham算法还可用于显示圆和其他曲线

Bresenham算法将对整型参数的符号进行检测，整型参数的值正比于两像素与实际线段之间的偏移，其核心问题是在（xk，yk）处，下一个像素点应该取（xk+1，yk）还是（xk+1，yk+1），（斜率在0到1之间）下面是推导：

假设直线方程为y=mx+b通过了（xk，yk）点

那么下一个像素点的水平位置为xk+1，此时y=m（xk+1）+b

Upper=yk+1-y=yk+1- m（xk+1）+b

Down=y-yk= m（xk+1）+b-yk

此时的差值为：down-upper=2m（xk+1）-2\*yk+2\*b-1

假设dx与dy分别表示两个端点的水平和垂直方向的偏移量

那么m=dy/dx

Down-upper=2\*dy(xk+1)/dx-2\*yk+2\*b-1

dx(Down-upper)=2\*dy(xk+1)-2\*dx\*yk+(2\*b-1)\*dx

此时dx(Down-upper)与(Down-upper)的符号是一样的，因为（m>0&&m<1）

dx(Down-upper)=2\*dy(xk+1)-2\*dx\*yk+(2\*b-1)\*dx

=2\*dy\*xk-2\*dx\*yk+2\*dy+(2\*b-1)\*dx=2\*dy\*xk-2\*dx\*yk+c

假设决策参数pk=2\*dy\*xk-2\*dx\*yk+c

**那么p(k+1)-pk=2\*dy\*x(k+1)-2\*dx\*y(k+1)-2\*dy\*xk-2\*dx\*yk**

**=2\*dy\*(x(k+1)-xk)-2\*dy\*(y(k+1)-yk)=2\*dy-2\*dx(y(k+1)-yk)，其中y(k+1)-yk=0或1**

**P0=2\*dy\*x0-2\*dx\*y0+2\*dy+(2b-1)\*dx=2\*dy\*x0-2\*dx\*(m\*x0+b)+2\*dy+(2b-1)\*dx=2\*dy\*x0-2\*dy\*x0-2\*b\*dx+2\*dy+2b\*dx –dx=2\*dy-dx**

因此，斜率<=1的情况算法描述为：

1）计算dx、dy、2\*dy以及2\*dy-2\*dx，同时计算初始参数为p0=2\*dy-dx

2）如果pk>0，说明y(k+1)=yk+1则p(k+1)=pk+2\*dy-2\*dx，否则y(k+1)=yk，则p(k+1)=pk+2\*dy

对于斜率大于1的情况，只需要将dx与dy对调即可（已经验证）

对于|斜率|<=1的情况代码如下：

void drawLineBresenham(GLint x0, GLint y0, GLint x1, GLint y1)

{

float dx = x1 - x0, dy = y1 - y0;

float x, y, twoDx = 2 \* dx, twoDy = 2 \* dy;

float towDyMinusDx = twoDy - twoDx, p = 2 \* dy - dx;

// 寻找起始点，因此主要是基于dx>0推导出来的

if(x0 > x1)

{

x = x1;

y = y1;

x1 = x0;

}

else

{

x = x0;

y = y0;

}

drawPixel(x, y);

while(x < x1)

{

x++;

if(p >= 0)

{

y++;

p += towDyMinusDx;

}

else

{

p += twoDy;

}

drawPixel(x, y);

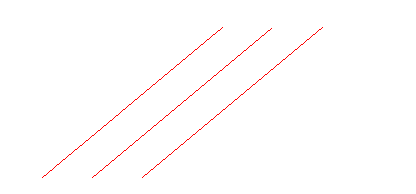
}

}

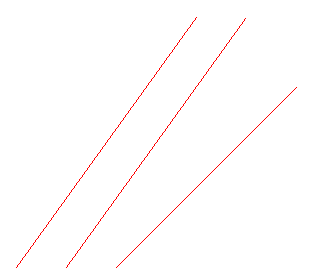
比较两者的代码，发现bresenham算法要先计算起始点，那是因为在推导过程中有一个默认dx>0，也就是每次是从左向右绘制

按照上面的代码：（左边为DDA，中间为opengl，右边为bresenham）

（20,1,200,160）



（20,10,200,260）



**为什么上面的bresenham算法在斜率大于1的时候始终为1呢？**

因为计算的p始终大于0，且此时的x++，y++即为45度角，显得很平滑

对于所有的情况的代码（排除水平，垂直以及45度角）

void drawLineBresenham(GLint x0, GLint y0, GLint x1, GLint y1)

{

float m = fabs((float)(y1 - y0)) / fabs((float)(x1 - x0));

if(m <= 1)

{

drawLineBresenhamLow(x0, y0, x1, y1);

}

else

{

drawLineBresenhamHigh(x0, y0, x1, y1);

}

}

void drawLineBresenhamLow(GLint x0, GLint y0, GLint x1, GLint y1)

{

float dx = x1 - x0, dy = y1 - y0;

float x, y, twoDx = 2 \* dx, twoDy = 2 \* dy;

float towDyMinusDx = twoDy - twoDx, p = 2 \* dy - dx;

// 寻找起始点，因此主要是基于dx>0推导出来的

if(x0 > x1)

{

x = x1;

y = y1;

x1 = x0;

}

else

{

x = x0;

y = y0;

}

drawPixel(x, y);

while(x < x1)

{

x++;

if(p >= 0)

{

y++;

p += towDyMinusDx;

}

else

{

p += twoDy;

}

drawPixel(x, y);

}

}

void drawLineBresenhamHigh(GLint x0, GLint y0, GLint x1, GLint y1)

{

float dx = x1 - x0, dy = y1 - y0;

float x, y, twoDx = 2 \* dx, twoDy = 2 \* dy;

float towDxMinusDy = twoDx - twoDy, p = 2 \* dx - dy;

// 寻找起始点，因此主要是基于dy>0推导出来的

if(y0 > y1)

{

x = x1;

y = y1;

y1 = y0;

}

else

{

x = x0;

y = y0;

}

drawPixel(x, y);

while(y < y1)

{

y++;

if(p >= 0)

{

x++;

p += towDxMinusDy;

}

else

{

p += twoDx;

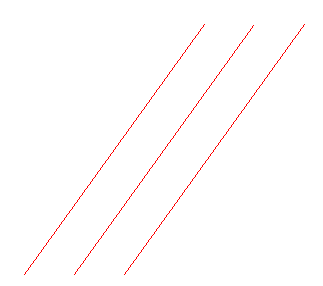
}

drawPixel(x, y);

}

}

（20,10,200,260）



**为什么都要取步长为1呢？**

个人理解：由于最终的画线都是以像素为单位的，如果斜率大于1，由于不断的累积会导致像素点断断续续的。例如y=kx，当k=1.5时，先取（0,0），再取（1,1.5）变为（1，1），再取（2,3），就会发现出现了断点，而如果斜率小于1，那么下一个像素点必定是上一个顶点相邻的像素点

下面是直接画的效果，斜率大于1的情况：

void drawLineDirection(GLint x0, GLint y0, GLint x1, GLint y1)

{

float m = (float)(y1 - y0) / (x1 - x0);

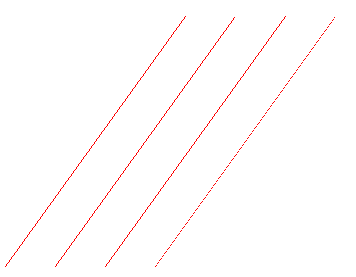
for(int x = x0; x < x1; x++)

{

drawPixel(round(x), round(m \* (x - x0) + y0));

}

}



最后一条直接由直接绘制而来，发现明显断断续续的