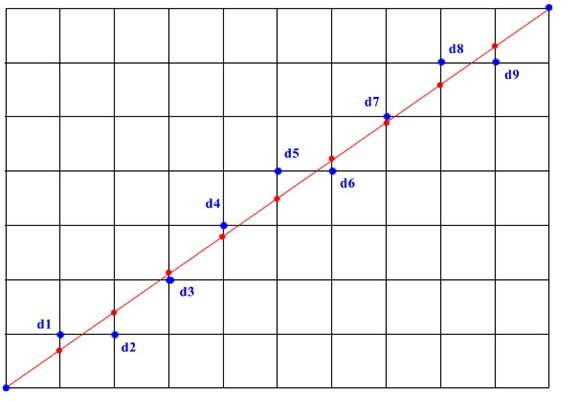
## 题目名：

中点Bresenham直线光栅化算法

## 算法原理



用蓝色像素点代替红色直线

每次在主位移（宽高中较长的为主位移）移动一个像素，另一个方向走不走取决于中点偏差判别式的值中点偏差判别式

f(x,y) = y - kx - b = 0 其中 k = (y2-y1)/(x2-x1)

根据f(x,y)的值与0 的关系（ < 0 , = 0 , > 0）判定中点与直线的位置关系

例如.主位移为x， 0 < k < 1 时，按a(x+1, y),b(x+1,y+1)中点与直线比较,然后进行相应处理

a(x+1, y),b(x+1,y+1) 的中点带入偏差判别式可得  d = f(x+1, y + 0.5) = y + 0.5 -  k\*(x+1) - b

若d < 0 中点在直线下方，下个像素点画在b，   下一步的d 值为 d = f(x + 2, y + 1.5) = y + 0.5 - k(x+1) - b + 1 - k = d + 1 - k

若d >= 0 中点在直线上方，下个像素点画在a， 下一步的d值为  d = f(x + 2, y + 0.5) = y + 0.5 - k(x+1) - b - k = d - k

过程中的每个d 都可以由上一步通过加减操作得出，操作快速！

d的初始值为 d = f(x + 1, y + 0.5) = y + 0.5 - k(x+1) - b = y - kx - b -k + 0.5

其中(x,y)为起始点，在直线y - kx - b = 0上代入得 d = 0.5 - k

效率较高.

## 实现

### 1.相关数据结构:

线段基于两点，即记录起始点即可

### 2.函数模块说明

MidpointBresenhamDrawLine 刻画线段

Putpixel 模拟设置像素

### 3.核心代码

// 中点 Bresenham

void ImageRenderer::MidpointBresenhamDrawLine(int x1, int y1, int x2, int y2){

m\_pRenderTarget->SetAntialiasMode(D2D1\_ANTIALIAS\_MODE\_ALIASED);

//

float k = float(y2 - y1) / float(x2 - x1);

int flag = 0;

if (k > 1 || k < -1) {

flag = 1;

x1 ^= y1 ^= x1 ^= y1;

x2 ^= y2 ^= x2 ^= y2;

k = float(y2 - y1) / float(x2 - x1);

}

float d = 0.5f - k;

if (x1 > x2) {

x1 ^= x2 ^= x1 ^= x2;

y1 ^= y2 ^= y1 ^= y2;

}

while (x1 != x2)

{

if (k > 0 && d < 0){

++y1, ++d;

}

else if (k < 0 && d > 0){

--y1, --d;

}

d -= k;

++x1;

if (flag) putpixel(y1, x1);

else putpixel(x1, y1);

}

//

m\_pRenderTarget->SetAntialiasMode(D2D1\_ANTIALIAS\_MODE\_PER\_PRIMITIVE);

}

// 设置像素点

void ImageRenderer::putpixel(int x, int y){

D2D1\_RECT\_F rect;

rect.left = static\_cast<float>(x);

rect.top = static\_cast<float>(y);

rect.right = rect.left + 1.f;

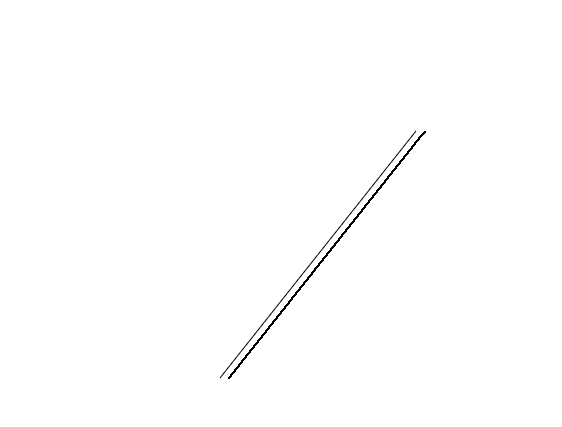
rect.bottom = rect.top + 1.f;

m\_pRenderTarget->DrawRectangle(&rect, m\_pBlackBrush);

}

程序基于Direct2D, 使用DrawRectangle模拟设置像素点

## 结果



左边的是Direct2D自带的方法，右边的是本算法

## 讨论

本算法本身效率不错，但是实际上效率较低，原因是GPU与CPU数据传送成本，假设纯粹的CPU渲染，效率还是不错。加上效果还不如默认的算法，b毕竟默认是反锯齿的