## 题目名：

中点Bresenham圆光栅化算法

## 算法原理

中点画圆算法就是每部单位间隔取样并且计算离圆最近的位置。

在继续之前，我这里补充一个关于圆对称性的知识点，通过在圆中计算考虑使用对称性计算开销可以减小到原来的1/8。

对称性质原理：

（1）圆是满足x轴对称的，这样只需要计算原来的1/2点的位置；

（2）圆是满足y轴对称的，这样只需要计算原来的1/2点的位置；

（3）圆是满足y = x or y = -x轴对称的，这样只需要计算原来的1/2点的位置；

通过上面三个性质分析得知，对于元的计算只需要分析其中1/8的点即可。

例如：分析出来目标点(x,y)必然存在(x,-y),(-x,y),(-x,-y),(y,x),(y,-x),(-y,x),(-y,-x)的另外7个点。

关于中心画圆算法，通过计算x = 0到 x = y的1/8圆的范围，然后通过对称原理得到其他7/8个点的信息。

这里和Bresenham算法有很多相似之处，同样有一个决定下一个位置的关键值P来做权衡处理。

在中点画圆算法中，通过平移的方法将假设圆心在坐标原点，然后计算，最后再平移到真实原心位置。

中点画圆算法内容：

1，输入圆心位置和圆的半径，得到圆周上的第一个点Point1;

（假设起始点为坐标原点，后面将通过坐标平移来处理非圆心在圆点）

2，计算决策关键参数的初始值，P = 5/4 - r;

3，在每个Xn的位置，从n = 0开始,更具决策值P来判断：

    如果P<0,下一个点的位置为(Xn+1,Yn);

       并且执行P = P + 2\*(Xn+1)+1;

    如果P>=0,下一个点的位置为(Xn+1,Yn-1);

       并且执行P = P + 2\*(Xn+1)+1 - 2\*(Yn-1);

4，通过对称原理计算其他7个对称相关点；

5，移动坐标到圆心点(x1,y1)

      X = X + x1;

      Y = Y + y1;

6，如果X<Y重复执行3到5的步骤，否则结束该算法

效率较高.

## 实现

### 1.相关数据结构:

圆基于点与半径，记录这些即可

### 2.函数模块说明

MidpointBresenhamDrawCircle刻画圆

Putpixel 模拟设置像素

### 3.核心代码

// 中点 Bresenham

void ImageRenderer::MidpointBresenhamDrawCircle(int ox, int oy, float r){

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

//

float d = 1.25f - r;

int x = 0, y = static\_cast<int>(r), fx = static\_cast<int>(r / 1.4f);

while (x != fx)

{

if (d < 0)

d += 2 \* x + 3;

else

{

d += 2 \* (x - y) + 5;

--y;

}

this->putpixel(ox + x, oy + y);

this->putpixel(ox + x, oy - y);

this->putpixel(ox - x, oy + y);

this->putpixel(ox - x, oy - y);

this->putpixel(ox + y, oy - x);

this->putpixel(ox + y, oy + x);

this->putpixel(ox - y, oy + x);

this->putpixel(ox - y, oy - x);

++x;

}

//

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;

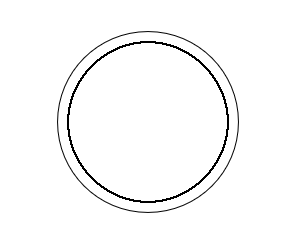
rect.bottom = rect.top + 1.f;

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

}

程序基于Direct2D, 使用DrawRectangle模拟设置像素点

## 结果



外面的是Direct2D自带的方法，里面的是本算法

## 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，毕竟默认是反锯齿的