Ex1:图片的旋转、缩放以及绘制

实验环境：

Ubuntu14.04

CImg 库

编译指令：

g++ -o test.exe test.cpp myFunction.h -O2 -L/usr/X11R6/lib -lm -lpthread -lX11

运行指令：

./test.exe

Task1：完成缩放

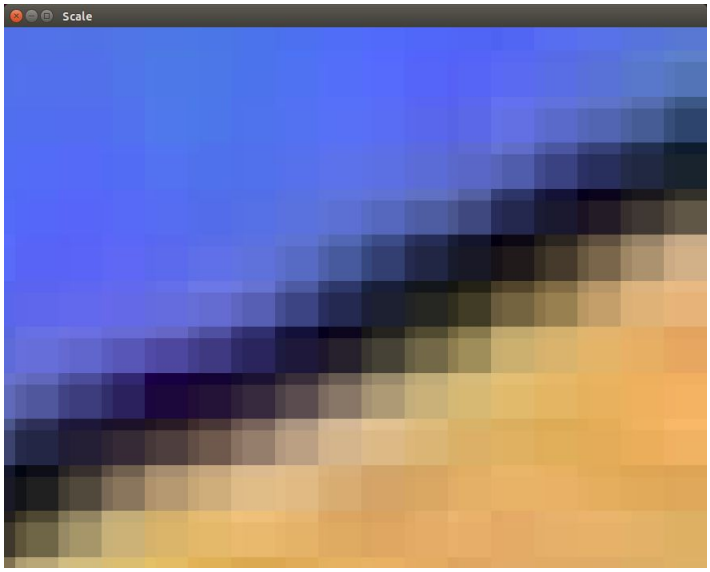
函数原型：

```
CImg<unsigned char> myScale(CImg<unsigned char>& inputImage, const float scale, bool interpolation)
```

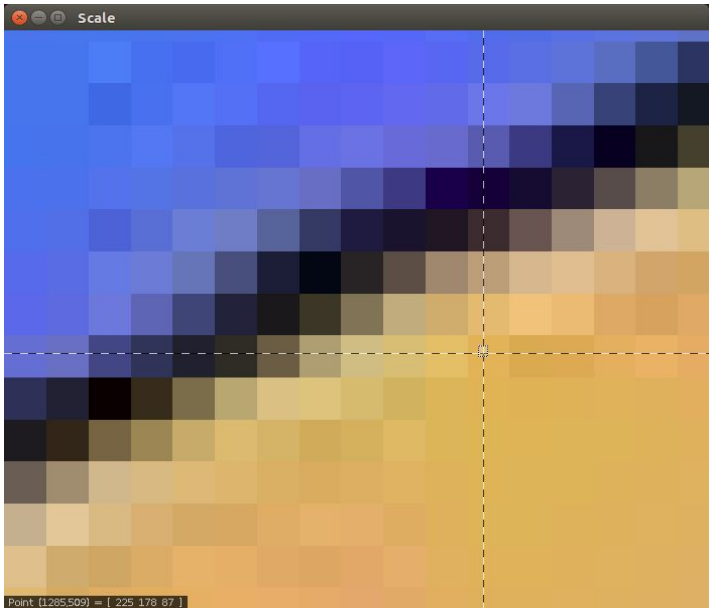
inputImage 表示原图，scale 表示缩放倍数，interpolation 表示缩放插值方式（1 为双线性插值，0 为最邻近插值）

最后结果 1（image3.bmp 放大 4 倍的边界效果图）

双线性插值：



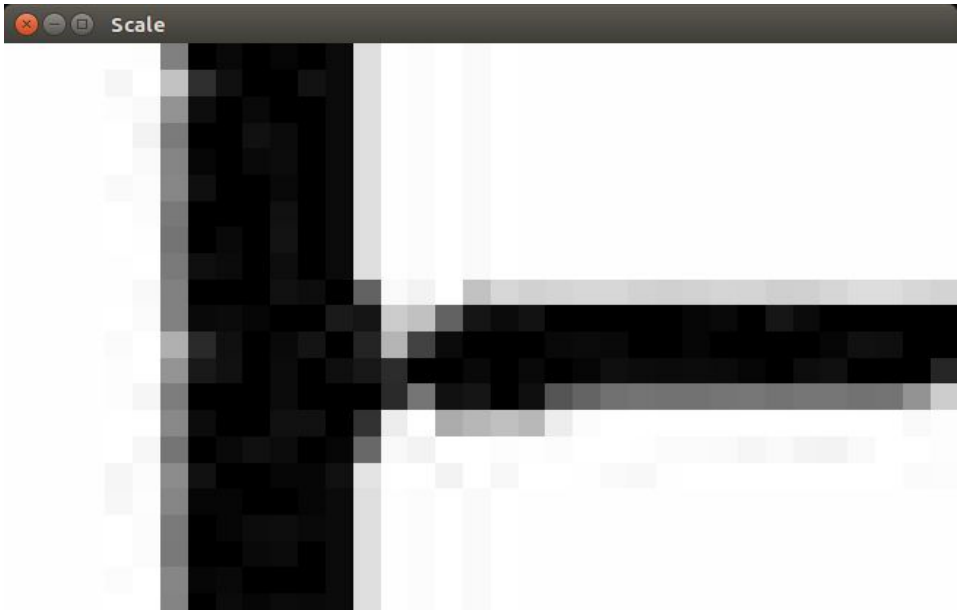
最邻近插值：



最后结果 2（image2.bmp 放大 3 倍的边界效果图）

双线性插值：

最邻近插值：



Task2：旋转

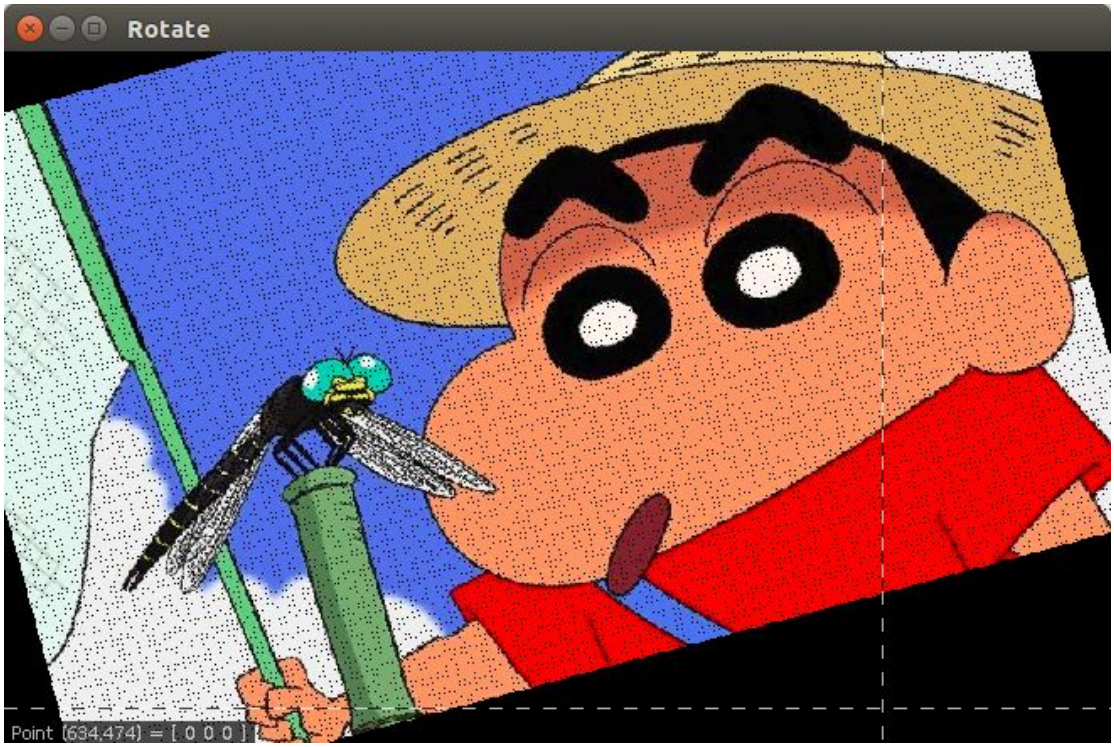
函数原型：

```
CImg<unsigned char> myRotate(CImg<unsigned char>& inputImage, const int positionX, const int positionY, const float angle, bool boundary)
```

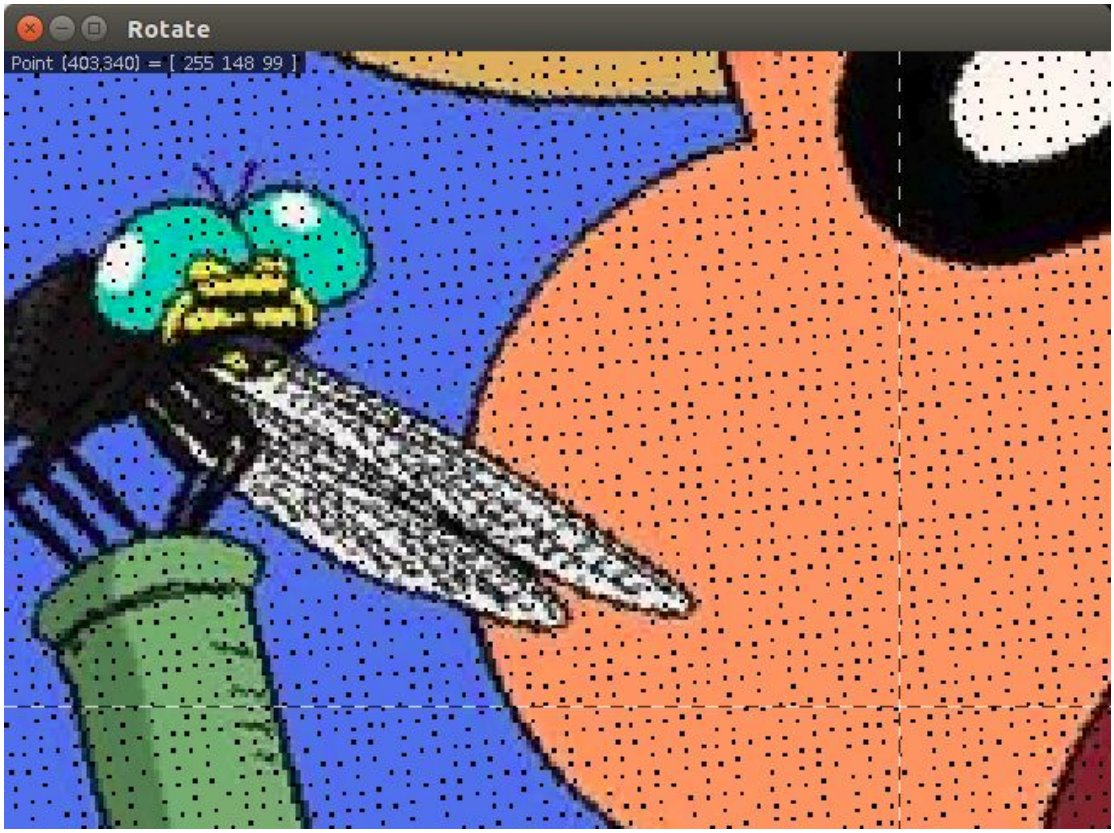
inputImage 为原图，positionX、positionY 分别为指定旋转中心的横纵坐标，angle 为旋转角度（0~360），boundary 表示是否需要用零填充边界以完整显示。

实验结果：

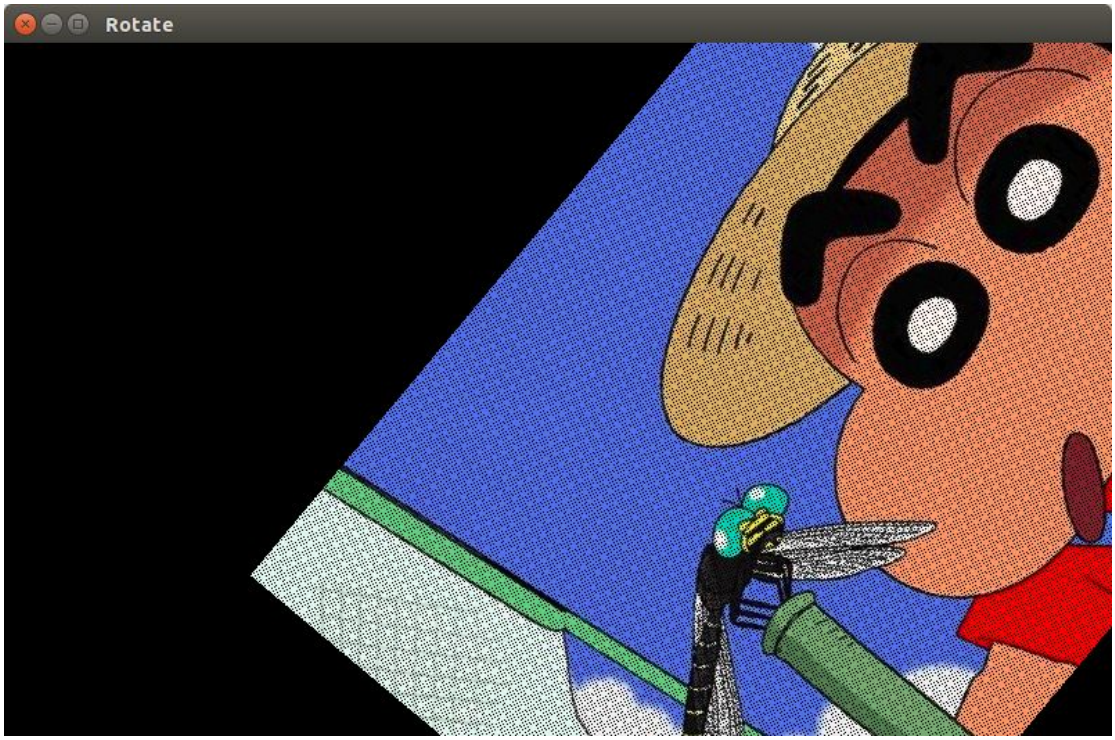
Test1（image3.bmp， 中心为（200， 300）， 旋转 15°，boundart=0）



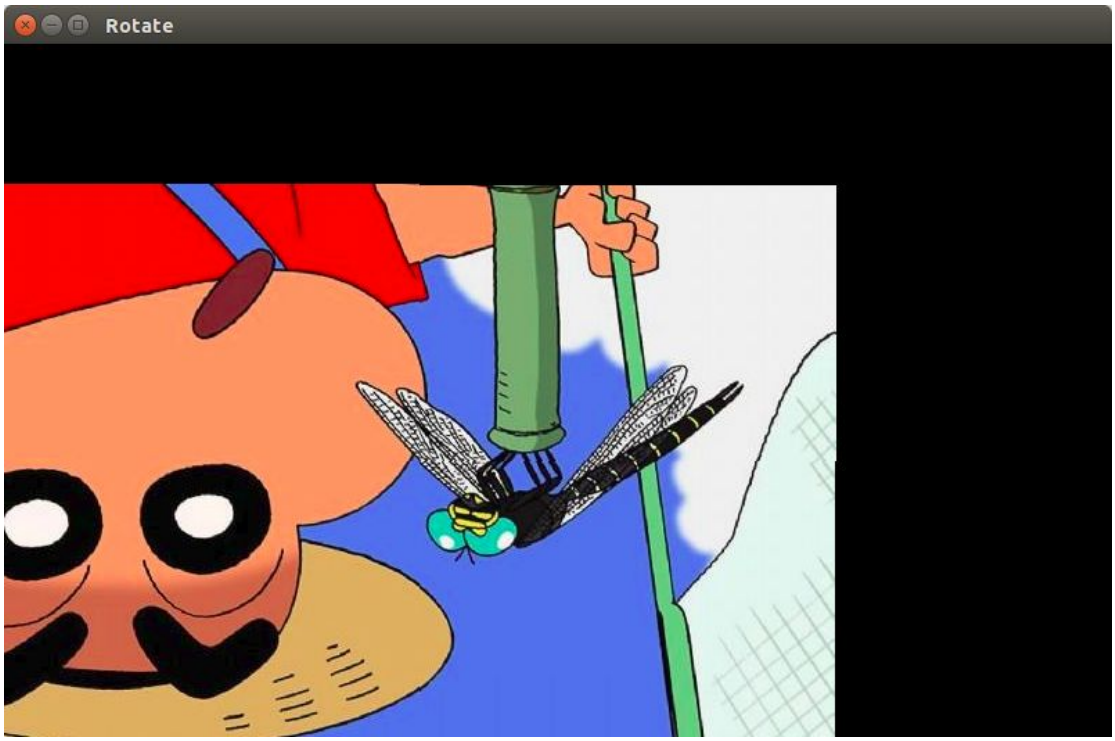
放大之后发现有很多洞洞：



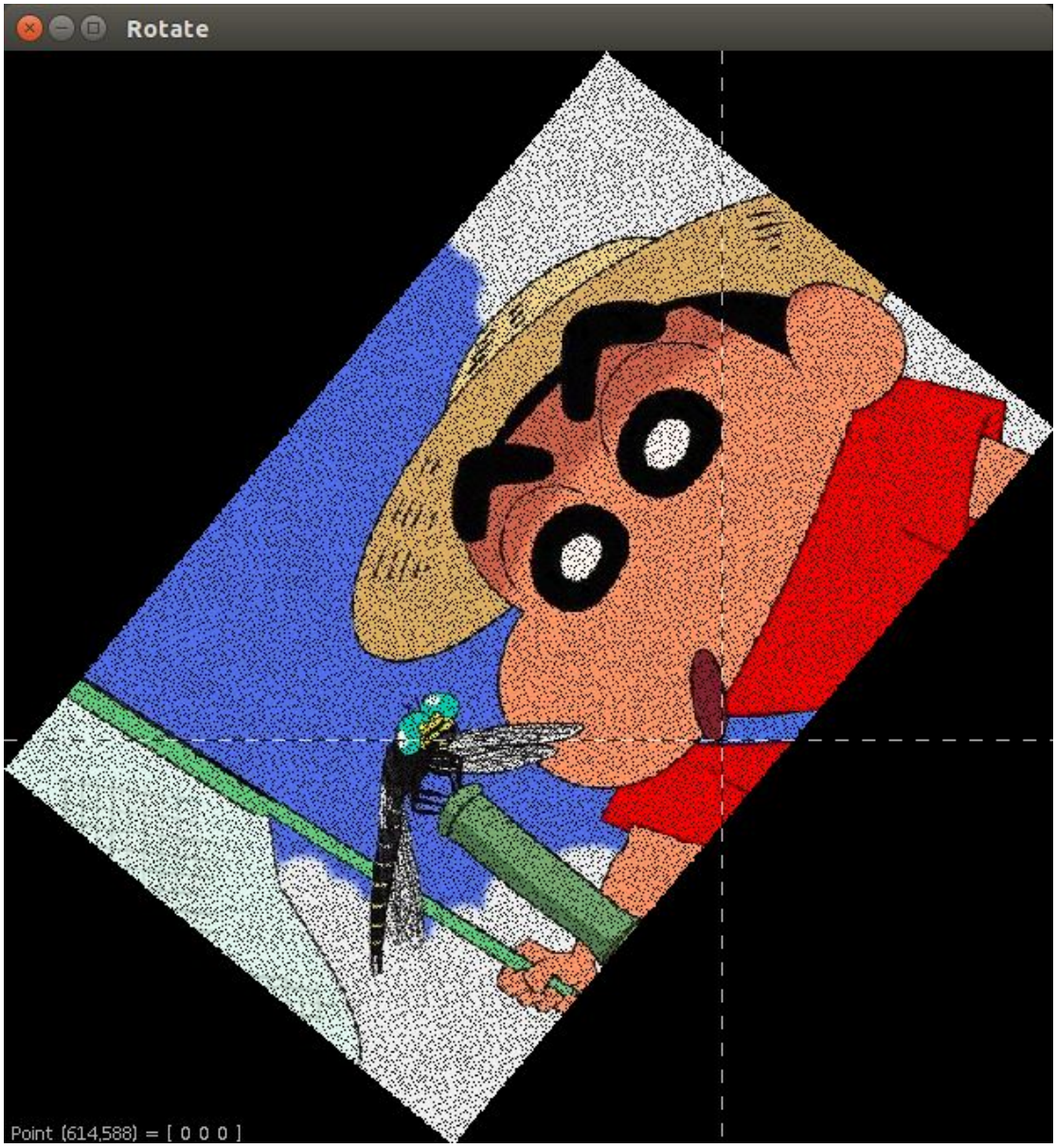
Test2（image3.bmp， 中心为（500， 0）， 旋转 50°，boundary=0）



Test3（image3.bmp， 中心为（300， 300）， 旋转 180°，boundary=0）



Test4（image.bmp，中心为（300,300）， angle=50， boundary=1）



抗锯齿效果的改进处理：

原来的浮点数计算，都是向下取整：

```
int newPointX = positionX + static_cast<int>(floor(cos(angPoint) * distPoint));
int newPointY = positionY - static_cast<int>(floor(sin(angPoint) * distPoint));

if(boundary == 0) {
        if (newPointY < height && newPointY >= 0 && newPointX < width && newPointX >=0) {
                for (int zz = 0; zz < spectrum; zz ++) {
                        img.atXYZC(newPointX, newPointY, 1, zz) = inputImage.atXYZC(xx, yy, 1, zz);
                }
        }
}
else {
        if (newPointY < maxY && newPointY >= minY && newPointX < maxX && newPointX >= minX) {
                for (int zz = 0; zz < spectrum; zz ++) {
                        imgBoundary.atXYZC(newPointX - minX, newPointY - minY, 1, zz) = inputImage.atXYZC(xx, yy, 1, zz);
                }
        }
}
```

改进后，也向上取整，对于新获取的点 newPointX1\newPointY1\newPointX2\

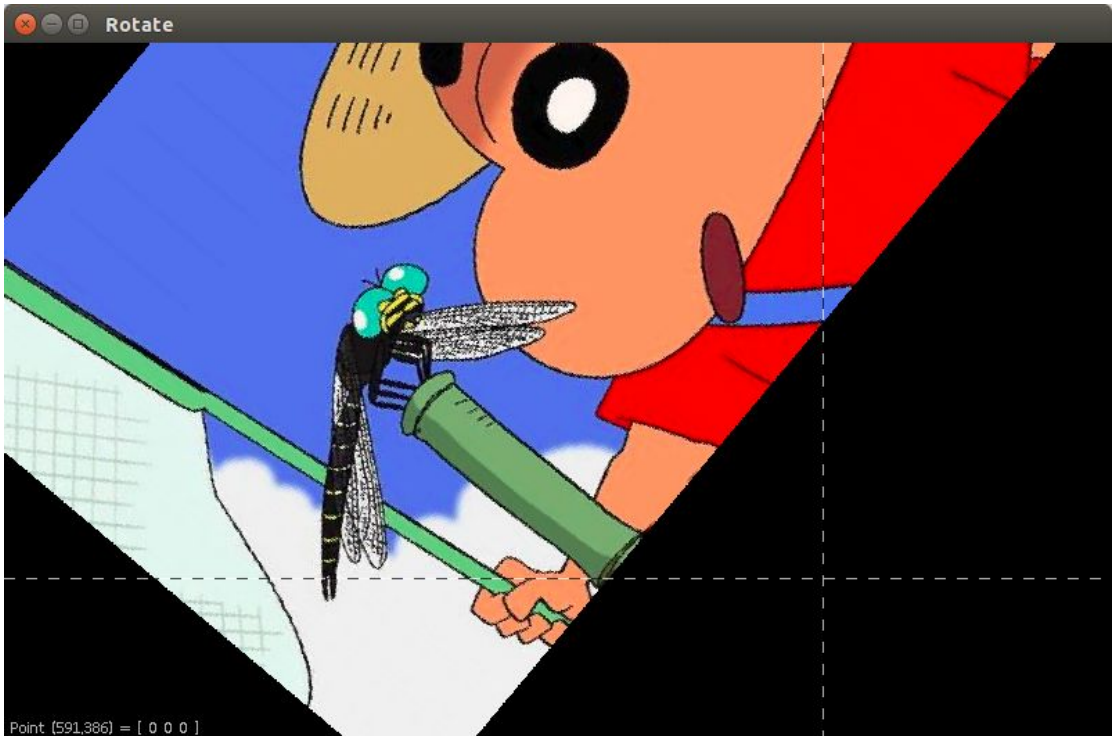
newPointY2 可以组合成四个毗邻的点，再填充这四个点

如下图：

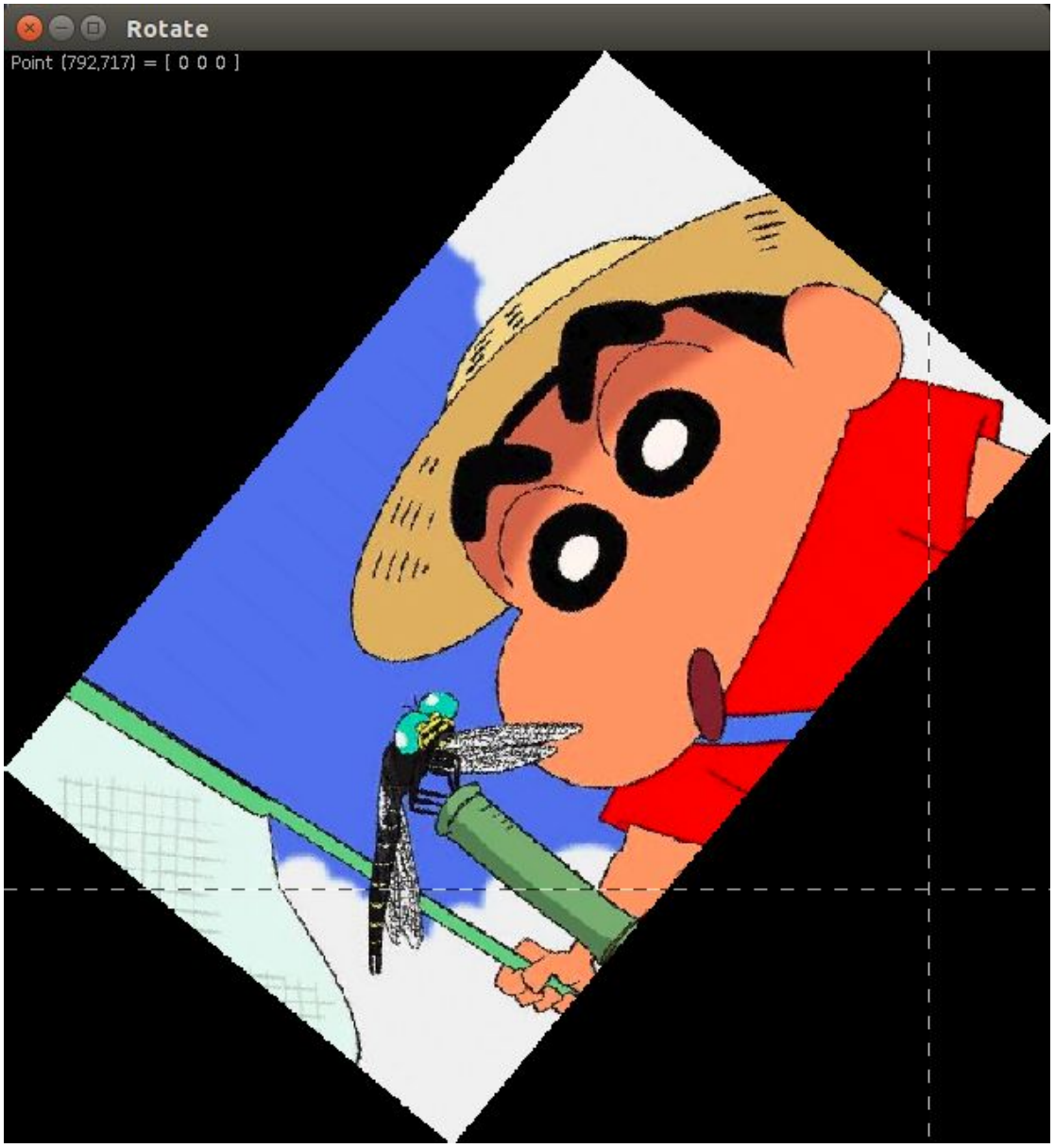
```
int newPointX1 = positionX + static_cast<int>(floor(cos(angPoint) * distPoint));
int newPointY1 = positionY - static_cast<int>(floor(sin(angPoint) * distPoint));
int newPointX2 = positionX + static_cast<int>(ceil(cos(angPoint) * distPoint));
int newPointY2 = positionY - static_cast<int>(ceil(sin(angPoint) * distPoint));

if(boundary == 0) {
        if (newPointY1 < height && newPointY1>= 0 && newPointX1 < width && newPointX1>=0
        && newPointY2 < height && newPointY2>= 0 && newPointX2 < width && newPointX2>=0) {
                for (int zz = 0; zz < spectrum; zz ++) {
                        img.atXYZC(newPointX1, newPointY1, 1, zz) = inputImage.atXYZC(xx, yy, 1, zz);
                        img.atXYZC(newPointX1, newPointY2, 1, zz) = inputImage.atXYZC(xx, yy, 1, zz);
                        img.atXYZC(newPointX2, newPointY1, 1, zz) = inputImage.atXYZC(xx, yy, 1, zz);
                        img.atXYZC(newPointX2, newPointY2, 1, zz) = inputImage.atXYZC(xx, yy, 1, zz);
                }
        }
}
else {
        if (newPointY1 < maxY && newPointY1 >= minY && newPointX1 < maxX && newPointX1 >= minX
        && newPointY2 < maxY && newPointY2 >= minY && newPointX2 < maxX && newPointX2 >= minX) {
                for (int zz = 0; zz < spectrum; zz ++) {
                        imgBoundary.atXYZC(newPointX1 - minX, newPointY1 - minY, 1, zz) = inputImage.atXYZC(xx, yy, 1, zz);
                        imgBoundary.atXYZC(newPointX1 - minX, newPointY2 - minY, 1, zz) = inputImage.atXYZC(xx, yy, 1, zz);
                        imgBoundary.atXYZC(newPointX2 - minX, newPointY1 - minY, 1, zz) = inputImage.atXYZC(xx, yy, 1, zz);
                        imgBoundary.atXYZC(newPointX2 - minX, newPointY2 - minY, 1, zz) = inputImage.atXYZC(xx, yy, 1, zz);
                }
        }
}
```

效果如图：

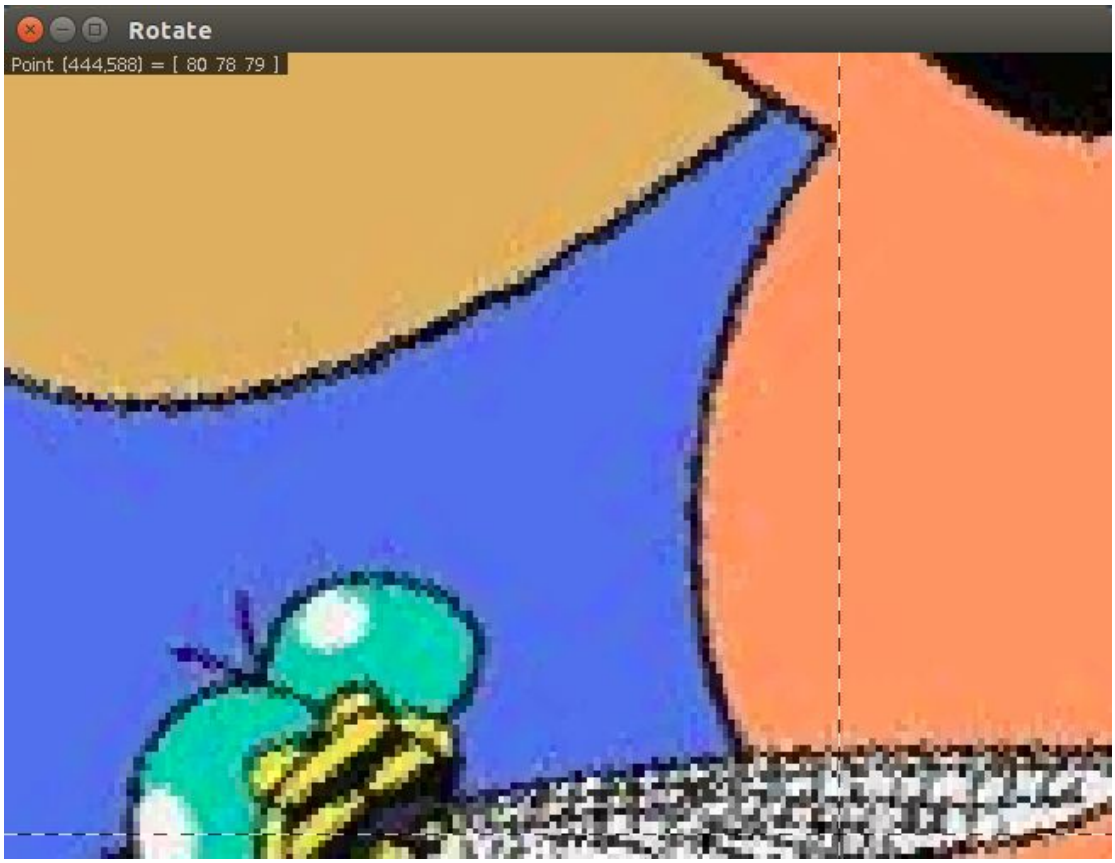
Test5（image3.bmp，中心（200,200），angle=50°，boundary=0）



Test6（image3.bmp，中心（200,200），angle=50°，boundary=1）



放大之后也没有锯齿坑：



Task3：绘制

1. 矩形绘制

i. 原型

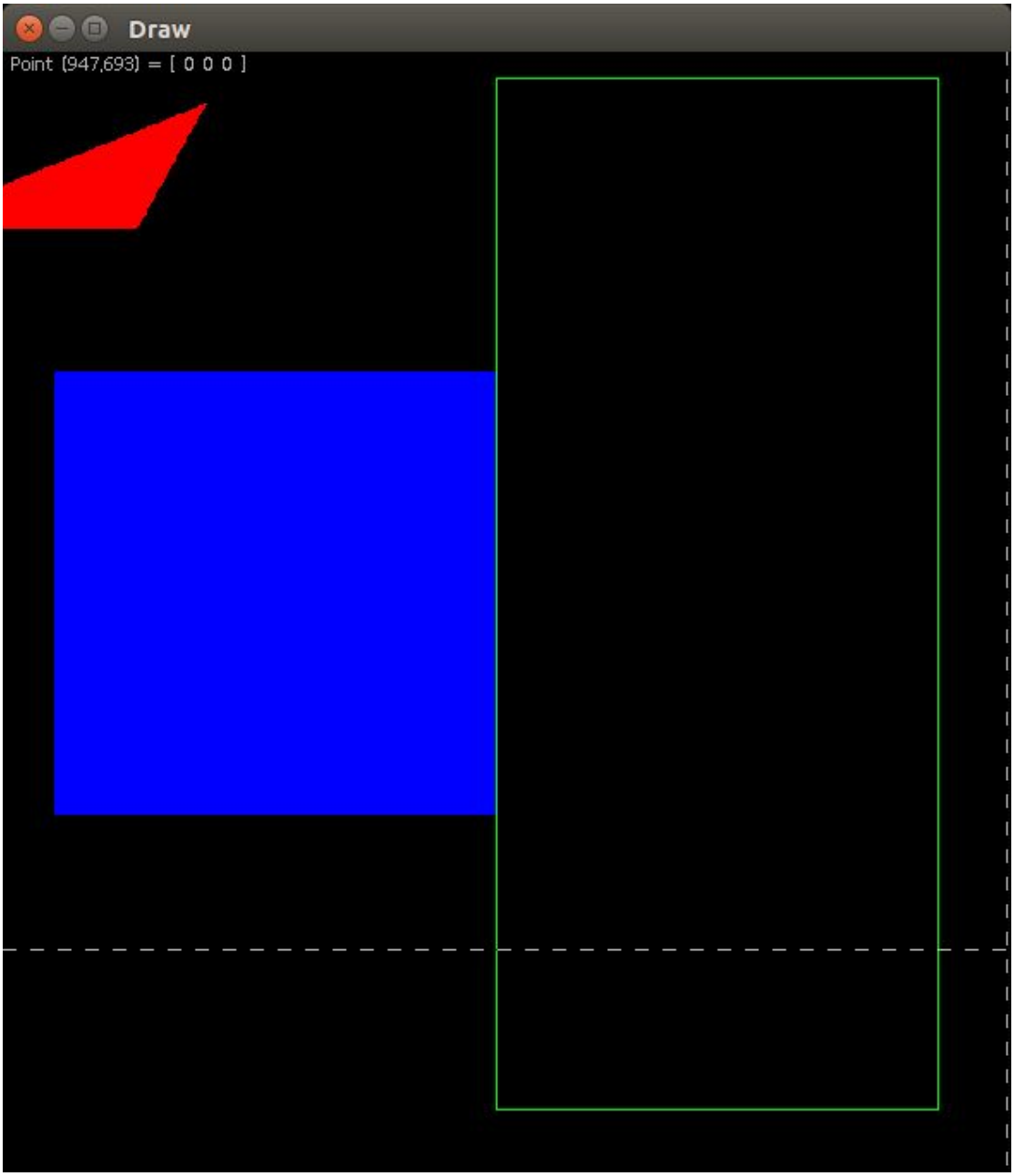
```
void drawRec(CImg<unsigned char>& inputImage, vector<int>& position, vector<int>& color, bool solid)
```

inputImage 为原图(1024*1024),position 为存储了对角坐标的 XY 值 ,color 存储颜色值，solid=1 为实心，solid=0 为空心

ii. 效果如图

Test1（rectangle， 顶点为（300,300）、（600,600），颜色为（0,0,255）， solid=1）

Test2（rectangle， 顶点为（600,800）、（900,100），颜色为（0,255,0）， solid=0）



2．三角形

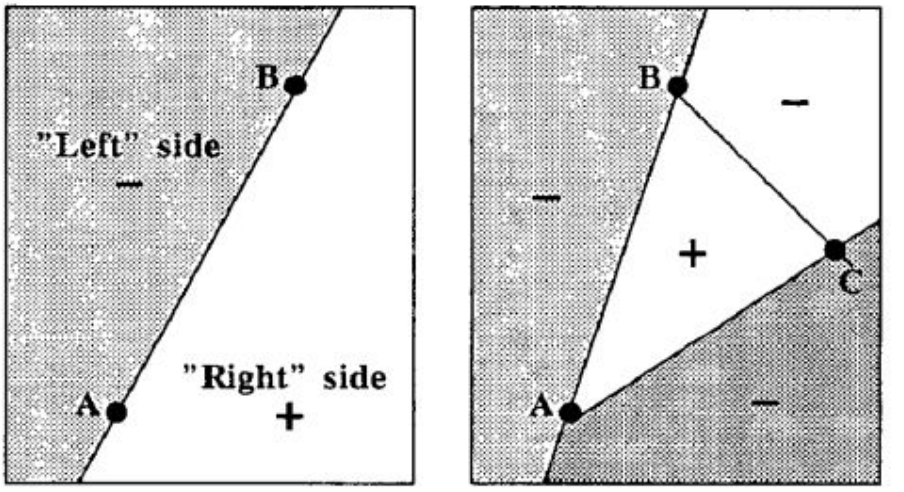
i. 原型

```
void drawTri(CImg<unsigned char>& inputImage,  vector<int>& position, vector<int>& color, bool solid )
```

inputImage 为原图（1024*1024），position 为存储了三个顶点坐标的 XY 值，color 存储颜色值，solid=1 为实心，solid=0 为空心

原理：(判断某一点是否在三角形内需要判断该点是否在重叠的“+”区域)

把三角形3条边界定义为边函数 $E_i(x,y)$ ，填充像素需满足三个 $E_i(x,y) \geq 0$ ，那么相邻的三角形也不会重绘或有空隙。详情请参考[1]。

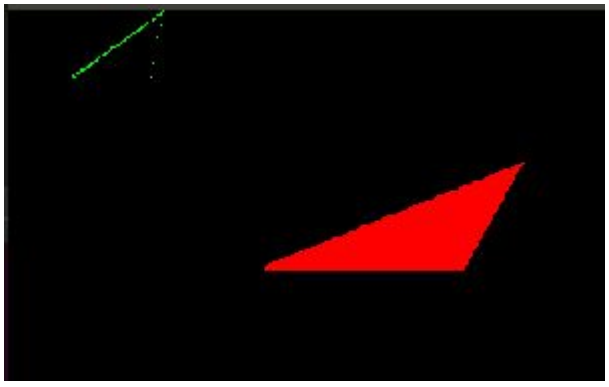


Subdivision of plane by line through points A and B

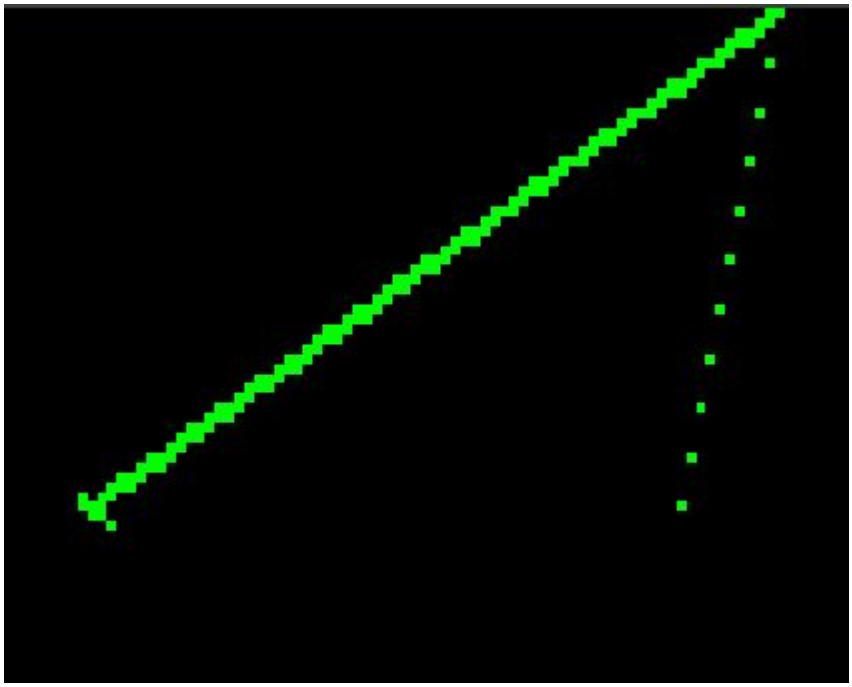
Triangle formed by union of right sides of AB, BC and CA

Figure 1. A Triangle Can be Formed by Combination of Edges

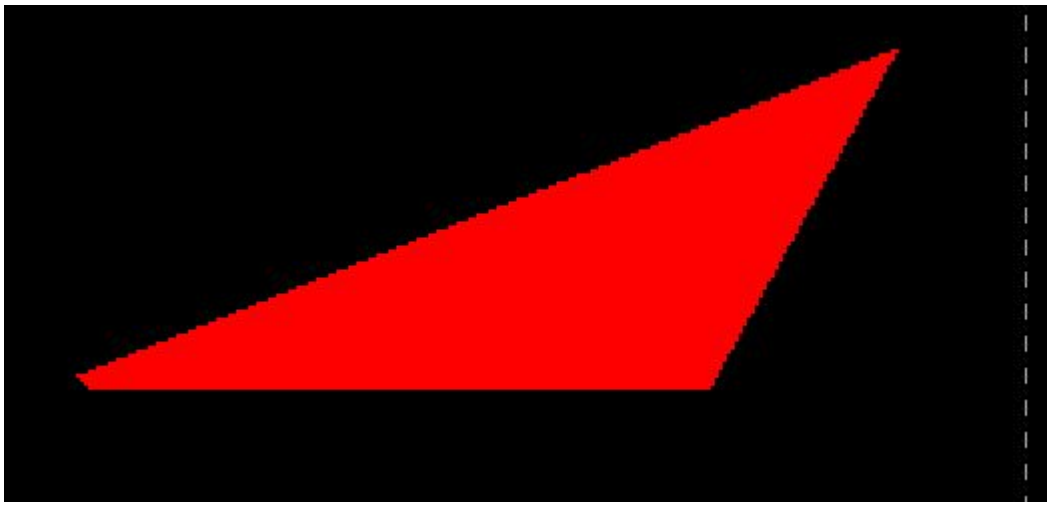
ii. 效果



Test1 (triangle , 顶点为 (50 , 50)(100,100)(120,0), 颜色 (0,255,0), solid=0)



Test2（triangle，顶点为（200，200）（300,300）（400,120），颜色（255,0,0），solid=1）

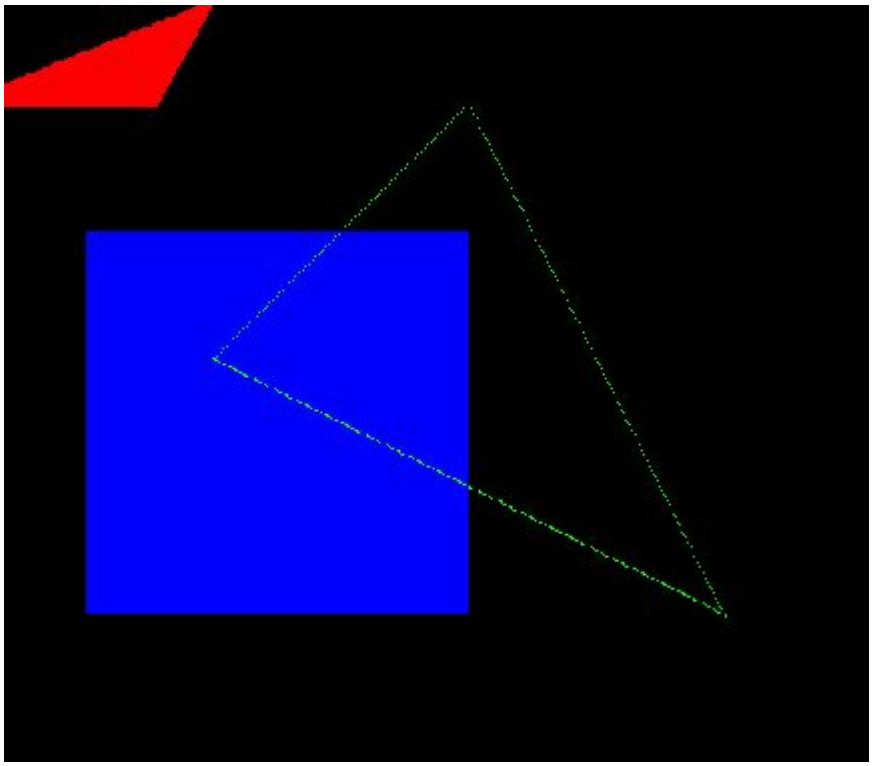


iii. 结论

空心的轮廓可能是由于边界判定的条件过于严格，导致很多范围内的点都不符合要求，出现了空白的区域。

当我把坐标之间的区域调大之后效果会好一点。

（下例，（600,200）（400,400）（800,600），颜色（0,255,0），solid= 0）



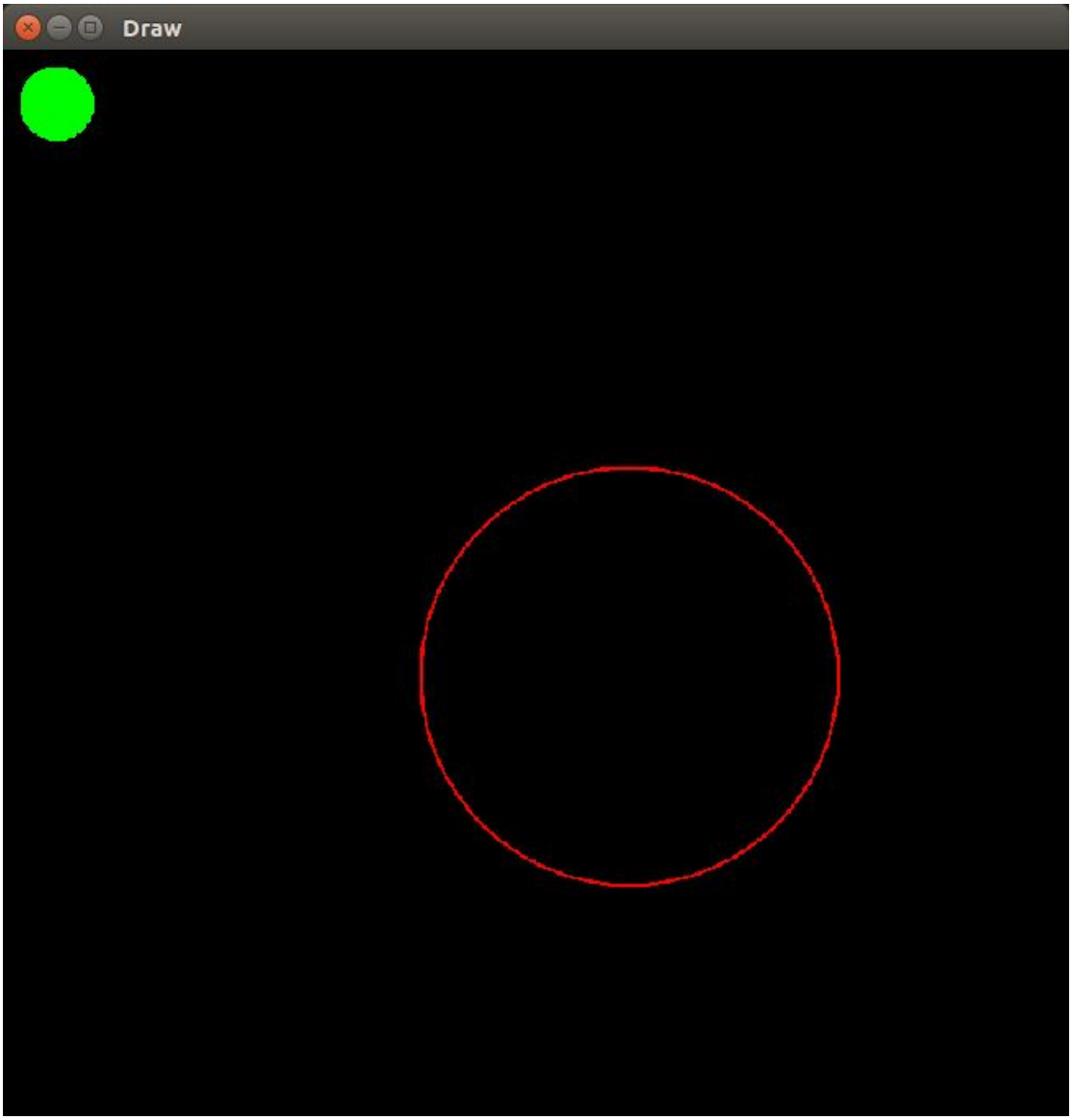
3．圆形

i. 原型

ii. 效果

Test1（circle，(50,50)， R=35， 颜色（0,255,0）, solid=1）

Test2（circle, (600, 600), R=200, 颜色（255,0,0）, solid=0）


Draw