Geomatics Science and Technology 测绘科学技术, 2015, 3(4), 67-85
Published Online October 2015 in Hans. http://www.hanspub.org/journal/gst
http://dx.doi.org/10.12677/gst.2015.34010

Hans汉斯

Effect on Application by Mechanism of GDI Rendering Functions and Solutions

Chunmei Chen[1], Qingyuan Li[2]

[1]China University of Mining & Technology, Beijing
[2]Chinese Academy of Surveying and Mapping, Beijing
Email: 815310703@qq.com, liqy@casm.ca.cn

Received: Oct. 2nd, 2015; accepted: Oct. 18th, 2015; published: Oct. 22nd, 2015

Abstract

GDI drawing-functions in Windows API were analyzed and we found that GDI drawing-function mapped the world coordinate system (Cartesian coordinate system) origin (0,0) to the pixel center between (0,0) and (1,1) of the device coordinate system (screen coordinates). It pointed out that in order to maintain the same graphic geometric features and avoiding overprinting, GDI drawing functions compromised with the endpoints and the boundary pixels, which caused some strange phenomenon many programmers have not found and it was difficult to understand. In response to these phenomena, explanations are given. In addition, the effect on some applications is pointed out and solutions are proposed.

Keywords

Graphics Device Interface, Drawing-Function, Primitive Output, Geometric Features

GDI绘图函数机制对应用程序的影响和解决方法

陈春梅[1]，李青元[2]

[1]中国矿业大学(北京)，北京
[2]中国测绘科学研究院，北京
Email: 815310703@qq.com, liqy@casm.ca.cn

文章引用: 陈春梅, 李青元. GDI 绘图函数机制对应用程序的影响和解决方法[J]. 测绘科学技术, 2015, 3(4): 67-85.
http://dx.doi.org/10.12677/gst.2015.34010

收稿日期：2015年10月2日；录用日期：2015年10月18日；发布日期：2015年10月22日

摘　要

对Windows API中的GDI绘图函数进行分析研究，发现了GDI绘图函数将世界坐标系(笛卡尔坐标系)原点(0,0)映射到设备坐标系(屏幕坐标系)的原点(0,0)到(1,1)之间的像素中心处。指出了为了保持图形的几何特征不变和避免叠印，GDI绘图函数会对端点和边界像素进行折衷处理，从而出现一些很多程序员没有发现的难以理解的奇怪现象。针对这些现象给出了解释，指出了其对应用程序的影响，并提出了解决方法。

关键词

图形设备接口，绘图函数，图元输出，几何特征

1. 引言

GDI (Graphical Device Interface)是 Windows 操作系统的子系统，负责在显示设备上显示图形。GDI 可以完成 Windows 操作系统从屏幕窗口显示(GUI)、图形图像渲染到打印机、绘图仪输出等一系列显示工作[1]。目前，大多数图形系统的开发仍然使用 GDI 绘图函数，但是 Windows API 将绘图函数封装起来，程序员在调用 GDI 绘图函数绘制图形时，不了解 GDI 具体的绘制机制，对于一些特殊的应用程序，细心的程序员会发现 GDI 绘图函数及有关的判断函数并不能得到满意的结果[2]。因此有需要深刻理解 GDI 绘图函数的图元输出的方法、存在的一些缺陷和 GDI 绘图函数使用的折衷处理方法，以便程序员在进行几何图元相交、图元距离等相关计算时，清楚其中的误差甚至错误，从而重新设计程序得到更精确的答案。笔者在李青元对 Windows 绘图函数所做的研究[2]的基础上，深入研究 GDI 绘图的机制，从算法上找出 GDI 绘图函数不能得到满意结果的根本原因，并给出相应的解决方法。

2. GDI 屏幕绘图的实质

首先应该清楚，为了描述图形，必须先确定一个合适的二维或三维笛卡尔坐标系(世界坐标系)，然后用世界坐标系中图形的几何描述(坐标位置等)来定义图形中的对象。然后通过将场景信息传送给观察函数、由观察函数识别可见面、将对象映射到视频监视器上来实现对象的显示[3]。在世界坐标系中指定一个图形的几何要素后，输出图元投影到与该输出设备的显示区域相对应的二维平面上，并扫描转换到帧缓存的整数像素位置。

应用程序使用 GDI 绘图函数，通常在计算机屏幕窗口进行显示。计算机屏幕窗口(视屏监视器)上的位置使用与帧缓存中的像素位置相对应的整数屏幕坐标进行描述，称为屏幕坐标系。像素的坐标值由 x 值和 y 值组成，y 代表扫描行号 x 代表列号(扫描行的 x 值)。屏幕刷新等硬件处理一般从屏幕的左上角开始对像素进行编址[3]。从屏幕最上面的扫描行到最下面的扫描行，依次编号为 0，1，2，…，ymax，每一行中从左到右的像素位置从 0 到 xmax 进行编号。如图 1 所示。GDI 绘图函数绘制的图形最终是用离散的整数像素点来表示的。

阅读本文首先应该对常用的画线算法(直线方程、DDA 算法、Bresenham 画线算法)和常用的区域填充算法(逐点判断算法、通用扫描线填充算法、边界填充算法、泛滥填充算法)有所了解。

首先需要清楚GDI绘图函数使用的都是逻辑坐标。常用的GDI绘图函数有：线段LineTo，折线Polyline，

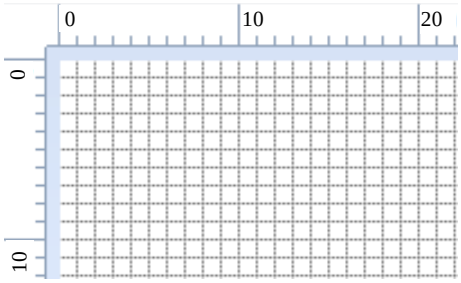


Figure 1. Screen coordinate system
图 1. 屏幕坐标系

多边形区域 Polygon，矩形 RECT 及相应的填充函数 FillRect，椭圆 Ellipse，矩形区域 RGN::CreateRectRgn、多边形区域 RGN::CreatePolygonRgn 及相应的填充函数 FillRgn。本文将在 Windows7 操作系统下，在 Visual Studio 2010 中采用默认的 MM_TEXT 坐标映射方式[4]，依次使用这几种绘图函数绘制几何图形并对线段端点、区域顶点和边界像素的输出显示进行分析。

3. GDI 绘线函数

GDI 提供绘制直线段的函数 BOOL CDC::LineTo (POINT point)和绘制折线段的函数 CDC::Polyline (const POINT *lpPoints, int nCount)。

3.1. LineTo 对端点像素的处理

绘线函数 BOOL CDC::LineTo(POINT point)在 MSDN 中的注释为“The LineTo function draws a line from the current position up to, but not including, the specified point. [5]”(从当前位置到指定点绘制一条线段，但不包括指定点)。

直线段用其两端点坐标位置定义，要在计算机屏幕上显示一条线段，图形系统首先把两端点投影到整数屏幕坐标位置，并用某种方法画线算法(直线方程、DDA 算法、Bresenham 画线算法，使用不同的画线算法两端点间的像素位置可能不完全一致)确定离两端点间的直线路径最近的像素位置。这一过程将一条线段数字化为一组相邻的离散的整数屏幕像素位置(屏幕坐标)。除了水平线段和垂直线段，一般情况下，这些位置是实际线路经(逻辑坐标)的近似。例如，计算出的线位置(7.39, 9.64)转换为像素位置(7, 10)。另外，由于在世界坐标系中，一个点代表数学上一个没有大小的位置，同样一条线在数学上表示抽象的无宽度的线，它的面积为零，但是在屏幕上显示点和线时，至少要占据一个像素的大小或宽度，因此如果图形要保持其几何特性，扫描转换算法必须考虑像素的有限大小[6]。

在 View 类的 OnDraw (CDC* pDC)函数中对同一条线段进行正、反两个方向的绘制，放大 8 倍后结果如图 2，图 3 所示。(注：以下各图均是复制到 Windows 的画图软件后放大 8 倍并选择网格显示的效果。放大 8 倍后能清楚的看到图形在计算机屏幕上显示的实质(像素)。对显示结果进行分析，可见线段的起点像素位置正确，终点像素没有显示。Windows 这么处理有两个好处：一方面保持了线段的几何长度，例如(0,0)到(10,0)的线段，几何长度是 10 个逻辑单位，扫描转换到屏幕上时也是 10 个像素单位；另一方面在同方向绘制首尾相邻的线段时，前一条线段的终点也即后一线段的起点不会重复绘制[7] [8]，提高了效率。这样在笛卡尔坐标系中的同一条线段，线段的两个端点在设备坐标系中的显示结果不一致，这种不一致取决于哪个端点是起点(显示)，哪个端点是终点(不显示) [9]。但是这样处理有一个缺陷，就是如果图形系统中有些线段是正方向绘制，有些是反方向绘制，就是使得相邻线段不相邻，如图 3 所示。这个缺陷可以通过都使用相同方向绘制图元避免。

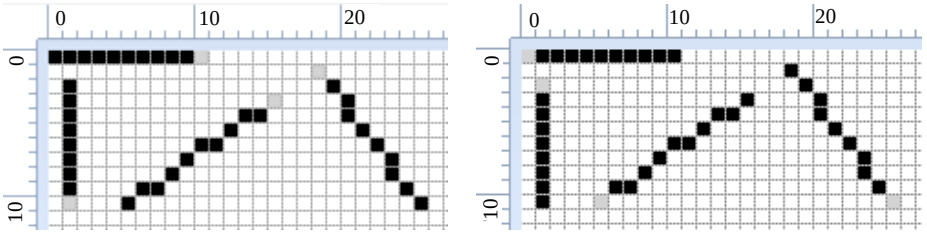


Figure 2. Lines drawing on forward direction (left) and backward direction (right) (pixels in gray color do not exist in fact)
图 2. 正方向(左)和反方向(右)绘制的直线段(注：图中灰色像素点实际上没有)

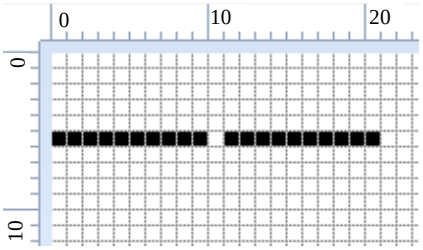


Figure 3. Adjacent lines on opposite directions
图 3. 相反方向绘制的相邻直线段

```
//正方向绘制四条线段，绘制结果如图 2
pDC->MoveTo(0,0);
pDC->LineTo(10,0);
pDC->MoveTo(1,2);
pDC->LineTo(1,10);
pDC->MoveTo(5,10);
pDC->LineTo(15,3);
pDC->MoveTo(25,10);
pDC->LineTo(18,1);
//反方向绘制相同的四条线段，绘制结果如图 2
pDC->MoveTo(10,0);
pDC->LineTo(0,0);
pDC->MoveTo(1,10);
pDC->LineTo(1,2);
pDC->MoveTo(15,3);
pDC->LineTo(5,10);
pDC->MoveTo(18,1);
pDC->LineTo(25,10);
//正方向(图 3)
pDC->MoveTo(0,5);
pDC->LineTo(10,5);
//反方向(图 3)
pDC->MoveTo(20,5);
```

```
pDC->LineTo(10,5);
```

图 3 中几何意义上两条线段在(10,5)处相邻，但是绘制的结果在像素(10,5)处没有邻接。

在应用程序中，如果需要线段在反方向绘制时的结果和正方向绘制的结果相同，并且保持线段的几何特征不变，可以在反方向绘制时用 CDC::SetPixel(POINT startPoint, COLORREF backColor)使起点像素不显示(背景颜色)，用 CDC::SetPixel(POINT endPoint, COLORREF penColor)使终点像素显示(画笔颜色)(图 4 左)。如果不考虑线段的几何特征，单纯地需要线段的端点都显示在正确的屏幕坐标位置，可以在图形显示时用 CDC::SetPixel(POINT endPoint, COLORREF penColor)使终点像素显示出来(图 4 右)。

从图 2 和图 4 可以看出，采用这种方法，线段上所有像素点屏幕位置都一样。在应用程序中，用像素算法判断点到线段的距离时，应考虑画线算法和画线方向的影响。如果要判断点是否在线段上，建议最好不要使用像素算法。实际上，画线算法是将线段数字化为一组离散的整数位置，除了水平和垂直线段外，这些位置是实际线路经的近似[3]。

3.2. Polyline 对端点像素的处理

绘制折线函数 CDC::Polyline (const POINT *lpPoints, int nCount)在 MSDN 中的注释：The Polyline function draws a series of line segments by connecting the points in the specified array [10]。

根据 MSDN 中的注释，折线段通过连接一组点形成一系列相连的直线段而成，预期的结果应该显示所有的点，包括端点(起点和终点)。但是用 Polyline 绘制折线，将正方向和反方向的结果(图 5)进行对比，发现 Polyline 绘制的折线除了最后一个端点之外，起点和中间折点都经过其坐标对应的像素点，并且正反方向绘制的折线，除端点外中间像素屏幕位置一样，但是绘制的折线不符合笛卡尔坐标系中的图形的几何特征。事实上，Polyline 相当于循环调用 LineTo [2]，而且折点像素没有叠印，同时同方向绘制的首尾相邻的折线也不会叠印。

```
//正方向绘制折线，图 5 左
CPoint ps[4];
ps[0].x = 1;    ps[0].y = 10;
ps[1].x = 6;    ps[1].y = 3;
ps[2].x = 20;    ps[2].y = 10;
ps[3].x = 24;    ps[3].y = 5;
pDC->Polyline(ps,4);
//反方向绘制同一折线，图 5 右
ps[0].x = 24; ps[0].y = 5;
ps[1].x = 20; ps[1].y = 10;
ps[2].x = 6;    ps[2].y = 3;
ps[3].x = 1;    ps[3].y = 10;
pDC->Polyline(ps,4);
```

在应用程序中，如果需要将折线终点坐标的像素显示出来，可以使用 CDC::SetPixel(POINT lastPoint, COLORREF penColor)将终点像素着色(图 6)。并且正方向和反方向绘制的折线经过的像素点位置完全一致。

4. GDI 绘制区域的函数

GDI 绘制区域的函数有：多边形区域 CDC::Polygon，矩形 RECT 类及相应的填充函数 FillRect，椭

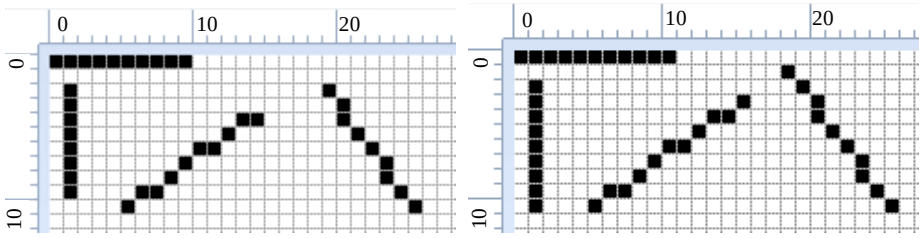


Figure 4. Two solutions to drawing lines
图 4. 绘制直线段的两种解决方法

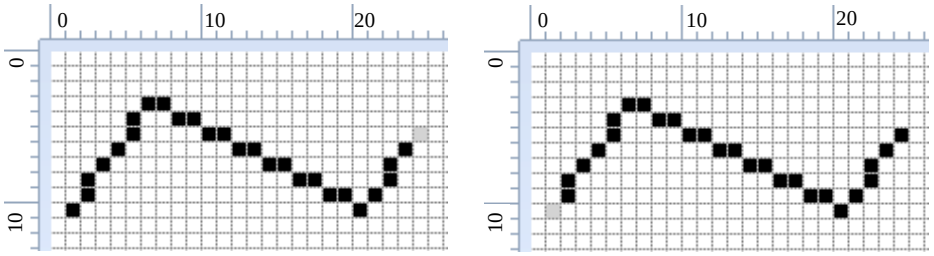


Figure 5. Polylines drawing on forward direction (left) and backward direction (right) (pixels in gray color do not exist in fact)
图 5. 正方向(左)和反方向(右)绘制的折线段(注：图中灰色像素点实际上没有)

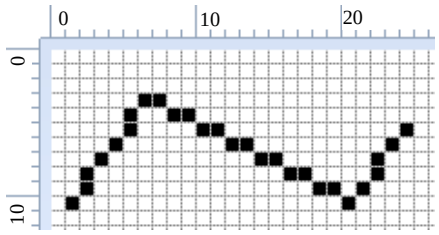


Figure 6. Solution to drawing polylines
图 6. 绘制折线段的解决方法

圆区域 CDC::Ellipse，矩形区域 RGN::CreateRectRgn、多边形区域 RGN::CreatePolygonRgn 及相应的填充函数 FillRgn。下面主要对多边形区域(包括矩形)的填充进行分析。

在计算机图形学中，多边形有两种重要的表示方法：顶点表示和点阵表示。点阵表示是用位于多边形内部的像素的集合来刻画多边形。这种表示方法虽然失去了很多重要的几何信息(如边界、顶点等)，但它是光栅显示系统(如计算机屏幕)显示时所需的表示形式[7]。从多边形顶点表示到点阵表示的转换称为扫面转换多边形，即从多边形的顶点信息出发，求出位于其内部的像素集合，并将其颜色值写入帧缓存中相应的单元。常用的扫描转换多边形的算法有：逐点判断算法、扫描线算法、边缘填充算法和泛滥填充算法。

4.1. Polygon 区域填充函数

MSDN 中对 Polygon 的注释：The Polygon function draws a polygon consisting of two or more vertices connected by straight lines. The polygon is outlined by using the current pen and filled by using the current brush and polygon fill mode [11]。

CDC::Polygon(const POINT *lpPoints, int cCount)使用当前画笔所画的边框围住多边形，并使用当前的画刷填充多边形[9]。Polygon 绘制的多边形边界经过所有指定的顶点像素坐标，但是多边形的几何特性

不一致。例如图 7(左)所绘制的矩形，长 9 个像素单位宽 7 个像素单位，但是用逻辑坐标定义的长为 9 − 1 = 8 宽为 7 − 1 = 6。此外用 Polygon 绘制相邻的多边形时，公共边界会叠印。例如图 7(右)中在绘制完成左矩形后，绘制右矩形，这时左矩形的右边界会被右矩形的左边界覆盖，而导致相同像素位置重复渲染。

```
//用 Polygon 绘制的矩形区域(图 7 左)
CPoint ps[4];
ps[0].x = 1; ps[0].y = 1;
ps[1].x = 1; ps[1].y = 7;
ps[2].x = 9; ps[2].y = 7;
ps[3].x = 9; ps[3].y = 1;
pDC->Polygon(ps,4);
//用 Polygon 绘制相邻的矩形区域(图 7 右)
ps[0].x = 9;   ps[0].y = 1;
ps[1].x = 9;   ps[1].y = 7;
ps[2].x = 20; ps[2].y = 7;
ps[3].x = 20; ps[3].y = 1;
pDC->Polygon(ps,4);
```

4.2. Ellipse 椭圆填充函数

MSDN 中 Ellipse 的注释 The Ellipse function draws an ellipse. The center of the ellipse is the center of the specified bounding rectangle [12]。用当前画笔画此椭圆边界并使用当前画刷填充它的内部区域[9]，椭圆的中心是外界矩形的中心。

从示例中可以看出用 CDC::Ellipse()绘制的椭圆或圆(图 8)保留了几何特性(长半轴、短半轴，直径)。Ellipse 绘制的椭圆也有一些不足(走样)，例如图 8 棕色像素(实际不存在)应该是圆的边界，可知圆的边界像素(黑色)不准确。

```
//长半轴 8，短半轴 6 的椭圆(图 8 左)
pDC->Ellipse(1,1,9,7);
//直径 10 的圆(图 8 右)
pDC->Ellipse(0,0,10,10);
```

4.3. FillRect、FillSolidRect 和 FillRgn 填充区域的方法(扫描线算法)

扫描线算法是扫描转换多边形(矩形)的常用算法，它通过扫描转换整个绘图窗口内的每一条中心扫描线来实现多边形(矩形)的填充问题。一条扫描线上的填充过程可分为下面三个步骤：

1) 求扫描线与多边形各边的交点；
2) 对所求得的交点按 x 坐标从小到大排序；
3) 将交点两两配对，并填充每一区段。

MSDN 没有给出 GDI 绘图函数中多边形的填充方法，但是经过大量的实验得知，FillRect、FillSolidRect 和 FillRgn 在扫描转换边界上的交点时，采用如下取整规则[7]：

假设某非水平边与中心扫描线 y = e (e 为整数)相交，交点的横坐标记作 x，则有如下几种情形：

1) x 为小数，即交点(x, e)落于扫描线 y = e 上非像素中心的位置。若交点在多边形的左边界上，则取边界右端的像素((int)x + 1, e)。若交点在多边形的右边界上，取边界左端的像素((int)x, e)。

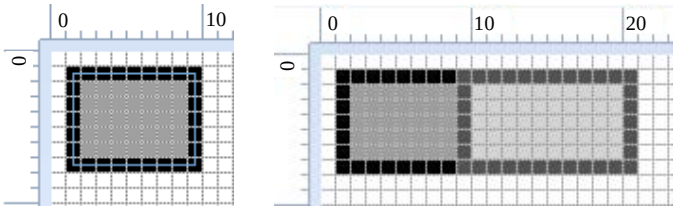


Figure 7. Rectangle (left) and adjacent rectangles (right) drew by polygon-function
图 7. Polygon 绘制的矩形(左)和相邻矩形(右)

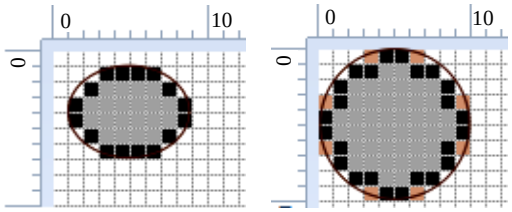


Figure 8. Ellipse (left) and circle (right) drew by ellipse-function
图 8. 用 Ellipse 绘制的椭圆(左)和圆(右)

2) 交点(x, e)正好落在整数像素点上(像素中心位置)，若(x, e)在多边形左边界上，看作属于多边形；若(x, e)在多边形右边界上，则它不属于多边形。

3) 在 2)中，若落在像素中心上的交点(x, e)是多边形的顶点，将多边形的每条边看作下端闭、上端开的，相当于将每条边的上端点处去掉一个像素。

需要注意的是，在 MM_TEXT 映射下，原点在左上角，Y 轴正方向向下，这时候“下闭上开”中的下和上和计算机屏幕中的下和上相反。

另外，由于水平边和扫描线平行，而且实际上水平边在算法中不起任何作用，所以在算法的预处理阶段将它们去掉。

4.3.1. RECT 类及 FillRect 和 Rectangle (LPRECT lprect)

RECT 在 MSDN 中的注释：By convention, the right and bottom edges of the rectangle are normally considered exclusive. In other words, the pixel whose coordinates are (right, bottom) lies immediately outside of the rectangle. For example, when RECT is passed to the FillRect function, the rectangle is filled up to, but not including, the right column and bottom row of pixels. This structure is identical to the RECTL structure [13]。

FillRect 函数填充全部矩形，包括左边和上边边界，但不填充右边和底边边界[9]。

从实验结果以及上述交点取整规则和 MSDN 中的注释可知，CDC::FillRect()和 CDC::FillSolidRect()填充用 CRect 定义的矩形时，右边界和下边界不显示(图 9(a))，这样处理保持了矩形的几何特性(长、宽)不变，同时也解释了用 PtInRect (RECT, POINT)判断点是否在矩形内时，右边界和下边界上的点都被判断为不在矩形内的原因。因为 PtInRect (RECT, POINT)采用的是像素算法，凡是矩形中显示出来的像素都返回真值。

```
CPen pen(PS_SOLID, 1, RGB(0,0,0));
CPen *oldpen = pDC->SelectObject(&pen);
CBrush brush(RGB(100,100,100));
CRect rect(1,1,9,7);
pDC->FillRect(&rect,&brush);//图 9(a)
```

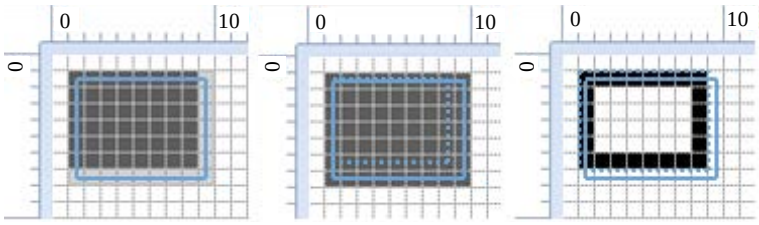


Figure 9. CRect filled by GDI (pixels in gray color do not exist in fact)
图 9. GDI 填充 CRect (注：图中灰色像素点实际上没有)

```
//pDC->Rectangle(&rect);//图 9(c)
pDC->SelectObject(oldpen);
BOOL yn = 0;
CPoint pt;
pt.x = 9; pt.y = 7;
yn = PtInRect(rect,pt);  //yn 返回 0
```

在应用程序中，如果不考虑矩形的几何特性，并且需要 PtInRect()返回理想的结果，可以在显示时将矩形的右下角顶点的 x、y 坐标都增加 1，这时原矩形右边界和下边界(图 9(b)蓝色虚线边框)上的点都被判断为在原矩形内。

如果想要保持矩形的几何特性，又想 PtInRect()返回理想的结果，可以改进 PtInRect()算法，使右边界和下边界点被判断为在矩形内。

```
BOOL PtInRectEx(CRect rect, CPoint point)
{
    if(point.x == rect.right && (point.y<=rect.bottom && point.y>=rect.top))
        return TRUE;
    else if(point.y == rect.bottom && (point.x<=rect.right && point.x>=rect.left))
        return TRUE;
    else
        return rect.PtInRect(point);
}
```

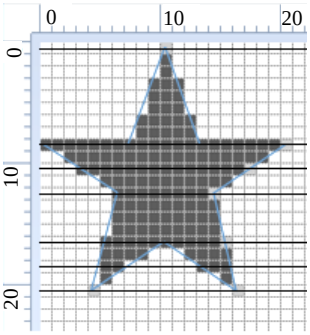
需要注意的是，用 CDC::Rectangle(LPCRECT lprect)绘制的矩形(图 9(c)所示)，右边和下边的黑色像素边框实际上并不是矩形的边界，用 PtInRect 判断右边界和下边界上的点都返回零值，而上边界和左边界都返回真值，MSDN 中对 Rectangle 注释：The rectangle that is drawn excludes the bottom and right edges，也可证实这一点[14]。我们也可以理解为这个函数是将世界坐标映射到像素间的屏幕位置，以使物体边界(图 9(c)蓝色虚线边框)与像素边界对齐(图 9(c)黑色像素边界)，而不是与像素(区域)中心对齐[3]。这样既能显示矩形边界又保持了矩形的几何特性。

4.3.2. CRgn 类及 CRgn::CreatePolygonRgn, CRgn::CreateRectRgn 和 CDC::FillRgn

从示例和上述交点取整规则，用 CRgn::CreateRectRgn(int x1,int y1,int x2,int y2)创建的矩形和用 CRect 定义的矩形显示结果一致，都是右边界和下边界不显示(图 9(a))。同样用 PtInRegion 判断的结果也和用 PtInRect 判断的结果一样。

```
//绘制矩形区域(结果和图 9(a)相同)
```

```
rgn.CreateRectRgn(1,1,9,7);
pDC->FillRgn(&rgn,&brush);
PtInRegion(rgn,9,7);     //返回 0
//绘制五角星形状的多边形区域(图 10(a))
CPoint ps[10];
ps[0].x = 10;    ps[0].y = 0;
ps[1].x = 7;     ps[1].y = 8;
ps[2].x = 0;     ps[2].y = 8;
ps[3].x = 6;     ps[3].y = 12;
ps[4].x = 4;     ps[4].y = 20;
ps[5].x = 10;    ps[5].y = 16;
ps[6].x = 16;    ps[6].y = 20;
ps[7].x = 14;    ps[7].y = 12;
ps[8].x = 20;    ps[8].y = 8;
ps[9].x = 13;    ps[9].y = 8;
rgn.CreatePolygonRgn(ps,10,WINDING);//反方向绘制的结果一样
pDC->FillRgn(&rgn,&brush);
//判断 10 个顶点是否在多边形内(结果如图 10(b)
BOOL yn = 1, yn2=1,yn3=1, yn4=1, yn5=1,        yn6=1,yn7=1, yn8=1,yn9=1, yn10=1;
yn = PtInRegion(rgn,10,0);
yn2 = PtInRegion(rgn,7,8);
yn3 = PtInRegion(rgn,0,8);
yn4 = PtInRegion(rgn,6,12);
yn5 = PtInRegion(rgn,4,20);
yn6 = PtInRegion(rgn,10,16);
yn7 = PtInRegion(rgn,16,20);
yn8 = PtInRegion(rgn,14,12);
yn9 = PtInRegion(rgn,20,8);
yn10 = PtInRegion(rgn,13,8);
```



(a)

监视1

名称	值	类型
yn	0	int
yn2	1	int
yn3	1	int
yn4	1	int
yn5	0	int
yn6	1	int
yn7	0	int
yn8	0	int
yn9	0	int
yn10	1	int

(b)

Figure 10. CRgn filled by GDI
图 10. GDI 填充 CRgn

MSDN 中对 Create<shape>Rgn 的注释:Regions created by the Create<shape>Rgn methods (such as CreateRectRgn and CreatePolygonRgn) only include the interior of the shape; the shape's outline is excluded from the region. This means that any point on a line between two sequential vertices is not included in the region. If you were to call PtInRegion for such a point, it would return zero as the result [15]。

根据 MSDN 的注释所有顶点以及边界上的整数像素点都不在多边形区域内，亦即图 10(b)中的值都应该是 0，但是事实并非如此。

根据上述交点取整规则和水平线的处理，扫描线 y = 0.5 与顶点 ps[0]相交，同时顶点 ps[0]又是右边界上的点，不论将 ps[0]算作一个点还是两个点，顶点 ps[0]都判断为不在多边形内；在处理扫描线 y = 8.5 时，先将两条水平边界删除，根据下闭上开的原则扫描线 y = 8.5 与顶点 ps[2]和顶点 ps[8]相交，因为 ps[2]是左边界上的点而 ps[8]是右边界上的点，所有 ps[2]判断为在多边形内，ps[8]判断为不在多边形内；扫描线 y = 10.5、y = 12.5 和 y = 18.5 和扫描线 y = 8.5 类似；扫描线 y = 16.5 与左边界上点(5，16)，顶点 ps[5]和右边界上点(16，16)相交，其中 ps[5]算作两个点，在填充(5，16)和 ps[5]之间的像素时(5，16)填充画刷颜色，ps[5]不填充，但是在填充 ps[5]和(16,16)之间的像素时 ps[5]填充，(16，16)不填充，所以这三个点的用 PtInRegion 判断将以此返回 1，1，0；由于下闭上开原则，扫描线 y = 20.5 不与 ps[4]、ps[6]相交，所以这两个顶点不在多边形内。

可以看出 MSDN 的注释与用 PtInRegion 进行判断时，边界上的点判断不一致。

在应用程序中，如果想要边界上的整数像素点和所以顶点都显示，建议使用 Polygon()，在判断边界上的点(包括顶点)是否在多边形上(内)，建议程序员根据上述规则自己编写程序。附录中是笔者自己编写的判断点是否在多边形内的函数 BOOL PtInRegionEx (POINT *ps, int num, POINT p)和 BOOL PtInRgn (POINT *ps, int num, POINT p)，这两个函数均可以对区域内和边界上的点返回 TRUE，对区域外的点返回 FALSE。

需要注意的是在用 Polygon()显示相邻多边形时，边界上的像素会相互覆盖而重复绘制(图 7(右)中间边界)，而用 FillRgn()填充的多边形，相邻多边形中的像素不会相互覆盖(图 11)。

5. 结束语

总结上文的实验可以发现，Windows 的 GDI 绘图函数的图元输出遵循以下准则：

1) 整数屏幕位置代表像素区域的中心；

2) GDI 绘线函数 LineTo 起点像素屏幕位置正确，为了保持线段的长度不变，在末尾少一个像素，因此导致正、反方向绘制的线段首尾像素位置不一致；

3) GDI 绘制折线函数 PolyLine 实质上是循环调用 LineTo，因此，同样会在折线末尾少一个像素，从而导致正、反方向绘制的折线首尾像素位置不一致；

4) GDI 填充区域函数 Polygon 相当于先用 PolyLine 绘制边界像素，然后再填充边界内部像素集合，因此绘制出的区域经过所有指定的像素点；

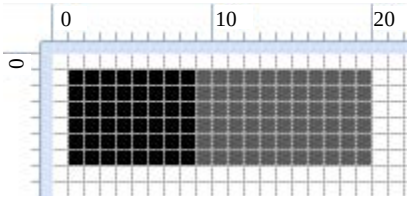


Figure 11. Adjacent rectangles filled by FillRgn-function
图 11. FillRgn 填充的相邻矩形

5) GDI 绘制椭圆函数 Ellipse 能够保持椭圆的长半轴和短半轴不变，但是边界像素不合理；

6) GDI 矩形类 CRect 和区域类 CRgn 及各自的填充函数，根据上文中的边界点取整规则，决定整数边界点是否显示，以及小数边界点如何显示，因此在调用 PtInRect 和 PtInRegion 时边界上的点会得到不一致的结果。

在一般的可视化应用程序中，不需要考虑 GDI 绘图函数是如何处理图形边界的，但是在一些特殊的图形系统中，例如在需要测量图形几何特性，判断点是否在图形上，或者判断点到图形的精确距离等的应用系统中，必须得到正确的判断结果和尽可能精确的栅格化结果，因为屏幕上一个像素的误差可能对应实际地面上几千米的错误，这时候图形软件开发人员和应用人员有必要清楚 GDI 绘图函数的实质和机制，尽量避免得到错误的结果。

基金项目

国家自然科学基金项目(41272367)。

参考文献 (References)

[1] 朱磊, 周彬 (2002) Windows 下的 C/C++高级编程. 人民邮电出版社, 北京, 79-141.

[2] 李青元, 谭海, 王涛 (2011) GDI/GDI+绘图函数缺陷与避免方法研究. *计算机工程与设计*, **12**, 4256-4259.

[3] Donald Hearn, M. Pauline Baker (2010) 计算机图形学. 第三版, 电子工业出版社, 北京, 26-185. (Computer Graphics with OpenGL, 3rd Edition).

[4] 陈建春 (2004) 矢量图形系统开发与编程. 电子工业出版社, 北京, 74-76.

[5] Microsoft (2015) Microsoft developer network [EB/OB]. 2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd145029(v=vs.85)

[6] 倪明田, 吴良芝 (1999) 计算机图形学. 北京大学出版社, 北京, 43-91.

[7] D.F. 罗杰斯 (1987) 计算机图形学的算法基础. 科学出版社, 北京, 30-101.

[8] 项志钢 (2008) 计算机图形学 Computer graphics with OpenGL. 清华大学出版社, 北京, 34-51.

[9] Microsoft 公司 (1993) Microsoft Windows 3.1 程序员参考大全(二)——函数. 91-641.

[10] Microsoft (2015) Microsoft developer network [EB/OB]. 2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd162815(v=vs.85)

[11] Microsoft (2015) Microsoft developer network [EB/OB]. 2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd162814(v=vs.85)

[12] Microsoft (2015) Microsoft developer network [EB/OB]. 2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd162510(v=vs.85)

[13] Microsoft (2015) Microsoft developer network [EB/OB]. 2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd162897(v=vs.85)

[14] Microsoft (2015) Microsoft developer network [EB/OB]. 2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd162898(v=vs.85)

[15] Microsoft (2015) Microsoft developer network [EB/OB]. 2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd183511(v=vs.85)

附　录

```
BOOL PtInRegionEx(POINT *ps, int num, POINT p)
{
    CRgn rgn;
    rgn.CreatePolygonRgn(ps,num,WINDING);
    BOOL YN = FALSE;
    //多边形包围盒
    int minx = 999999999,maxx = -999999999,miny = 999999999,maxy = -999999999;
    for(int i=0; i<num; i++)
    {
        if(ps[i].x <= minx)
            minx = ps[i].x;
        if(ps[i].x >= maxx)
            maxx = ps[i].x;
        if(ps[i].y <= miny)
            miny = ps[i].y;
        if(ps[i].y >= maxy)
            maxy = ps[i].y;
    }
    //包围盒之内的点
    int Dx, Dy, j;
    int minxl, maxxl, minyl, maxyl;    //线段的范围(包围盒)
    if(p.x <= maxx && p.x >= minx && p.y <= maxy && p.y >= miny)
    {
        YN = PtInRegion(rgn, p.x, p.y);
        if( YN == TRUE )
            return YN;
        else
        {
            for(j=0; j<num; j++)
            {
                if(p.x == ps[j].x && p.y == ps[j].y)
                    YN = TRUE;
            }
            for(j=0; j<num; j++)
            {
                if(j == num-1)
                {
                    if(ps[j].x <= ps[0].x)
```

```
                {
                    minxl = ps[j].x;
                    maxxl = ps[0].x;
                }
                else
                {
                    minxl = ps[0].x;
                    maxxl = ps[j].x;
                }
                if(ps[j].y <= ps[0].y)
                {
                    minyl = ps[j].y;
                    maxyl = ps[0].y;
                }
                else
                {
                    minyl = ps[0].y;
                    maxyl = ps[j].y;
                }

                Dx = ps[j].x - ps[0].x;
                Dy = ps[j].y - ps[0].y;
//如果是边界上的点
                if( (p.x<maxxl && p.x>minxl && p.y<maxyl && p.y>minyl) &&
                                    (Dx*(p.y-ps[j].y) + Dy*ps[j].x ) == (Dy*p.x) )
                    YN = TRUE;

            }
            else
            {
                if(ps[j].x <= ps[j+1].x)
                {
                    minxl = ps[j].x;
                    maxxl = ps[j+1].x;
                }
                else
                {
                    minxl = ps[j+1].x;
                    maxxl = ps[j].x;
```

```
                    }
                    if(ps[j].y <= ps[j+1].y)
                    {
                        minyl = ps[j].y;
                        maxyl = ps[j+1].y;
                    }
                    else
                    {
                        minyl = ps[j+1].y;
                        maxyl = ps[j].y;
                    }
                    Dx = ps[j].x - ps[j+1].x;
                    Dy = ps[j].y - ps[j+1].y;
  //如果是边界上的点
                    if( (p.x<maxxl && p.x>minxl && p.y<maxyl && p.y>minyl)
                                    &&(Dx*(p.y-ps[j].y) + Dy*ps[j].x ) == (Dy*p.x) )
                        YN = TRUE;
                }
            }
            return YN;
        }
    }
    //包围盒之外的点
    else
        return YN;
}
const double PI = 3.14159265;
double D2DistanceOfPointToLine(double xx,double yy,double x1,double y1,double x2,double y2)
{
    double a,b,c,ang1,ang2,ang,m;
    double result=0;
    //分别计算三条边的长度
    a = sqrt((x1 - xx) * (x1 - xx) + (y1 - yy) * (y1 - yy));
    if (a == 0)
        return 0;
    b = sqrt((x2 - xx) * (x2 - xx) + (y2 - yy) * (y2 - yy));
    if (b == 0)
        return 0;
    c = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
```

```
  //如果线段时一个点则退出函数并返回距离
if (c == 0)
{
      result = a;
      return result;
}
//如果点(xx,yy)到点(x1,y1)这条边近
if (a < b)
{
      //如果直线段 AB 是水平线，得到直线段 AB 的弧度
      if (y1 == y2)
      {
            if (x1 < x2)
                  ang1 = 0;
            else
                  ang1 = PI;
      }
      else
      {
            m = (x2 - x1) / c;
            if (m - 1 > 0.00001)
                  m = 1;
            ang1 = acos(m);
            if (y1 >y2)
                  ang1 = PI*2    - ang1;//直线(x1,y1)-(x2,y2)与 X 轴正向夹角的弧度
      }
        m = (xx - x1) / a;
      if (m - 1 > 0.00001)
            m = 1;
      ang2 = acos(m);
      if (y1 > yy)
            ang2 = PI * 2 - ang2;//直线(x1,y1)-(xx,yy)与 X 轴正向夹角的弧度

      ang = ang2 - ang1;
      if (ang < 0)
            ang = -ang;
      if (ang > PI)
            ang = PI * 2 - ang;
      //如果是钝角则直接返回距离
```

```
        if (ang > PI / 2)
            return a;
        else
            return a * sin(ang);
    }
    else//如果(xx,yy)到点(x2,y2)这条边较近
    {
        //如果两个点的纵坐标相同，则直接得到直线斜率的弧度
        if (y1 == y2)
            if (x1 < x2)
                ang1 = PI;
            else
                ang1 = 0;
        else
        {
            m = (x1 - x2) / c;
            if (m - 1 > 0.00001)
                m = 1;
            ang1 = acos(m);
            if (y2 > y1)
                ang1 = PI * 2 - ang1;
        }
        m = (xx - x2) / b;
        if (m - 1 > 0.00001)
            m = 1;
        ang2 = acos(m);//直线(x2,y2)-(xx,yy)斜率的弧度
        if (y2 > yy)
            ang2 = PI * 2 - ang2;
        ang = ang2 - ang1;
        if (ang < 0)
            ang = -ang;
        if (ang > PI)
            ang = PI * 2 - ang;//交角的大小
        //如果是对接则直接返回距离
        if (ang > PI / 2)
            return b;
        else
            return b * sin(ang);//如果是锐角，返回计算得到的距离
    }
```

```
}

BOOL PtInRgn(POINT *ps, int num, POINT p)
{
    int minx = 999999999,maxx = -999999999,miny = 999999999,maxy = -999999999;
    for(int i=0; i<num; i++)
    {
        if(ps[i].x <= minx)
            minx = ps[i].x;
        if(ps[i].x >= maxx)
            maxx = ps[i].x;
        if(ps[i].y <= miny)
            miny = ps[i].y;
        if(ps[i].y <= maxy)
            maxy = ps[i].y;
    }
    if(p.x <= maxx && p.x >= minx && p.y <= maxy && p.y >= miny)//对包围盒内的点进行判断
    {
        double mind = 999999999, d1;
        int index=-1, i, j;
        for(j=0; j<num; j++)
        {
            if(j==num-1)
                d1 = D2DistanceOfPointToLine(p.x,p.y,ps[j].x,ps[j].y,ps[0].x,ps[0].y);
            else
                d1 = D2DistanceOfPointToLine(p.x,p.y,ps[j].x,ps[j].y,ps[j+1].x,ps[j+1].y);
            if(d1>=0 && d1<=mind)
            {
                mind = d1;
                index = j;
            }
        }
        double v = 0.0;
        if(index>-1 && index<num-1)
        {
            v = -((ps[index].x - p.x)*(ps[index+1].y - p.y) - (ps[index].y - p.y)*(ps[index+1].x - p.x));
        }
        else if(index == num)
            v = -((ps[index].x - p.x)*(ps[0].y - p.y) - (ps[index].y - p.y)*(ps[0].x - p.x));
        if(v<0)          //点在线段右侧
```

```
            mind = -mind;

            if(mind<0)//多边形外的点返回假
                return FALSE;
            else        //多边形内及边界上的点返回真
                return TRUE;
        }
        else//包围盒外的点返回假
            return FALSE;
    }
```