International Journal of Geology and Earth Sciences

ISSN 2395-647X www.ijges.com
Vol. 1, No. 3, December 2015

Research Paper

EFFECT ON APPLICATION BY MECHANISM OF GDI RENDERING FUNCTIONS AND SOLUTIONS

Chunmei Chen[1]*, Qingyuan Li[2] and Huiling Jia[1]

*Corresponding Author: ***Chunmei Chen*** ✉ 815310703@qq.com

GDI drawing-functions in Windows API were analyzed and 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 characteristics 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 Characteristics

INTRODUCTION

GDI (Graphical Device Interface) , one of the subsystems of Windows operating system, is responsible for displaying graphics on the display device. GID can complete series of display work from Graphical User Interface(GUI) and graphics rendering to printer and plotter output (Lie, 2002). At present, most of graphics system development is still using GDI drawing functions packaged in the Windows API. When the programmers call the GDI drawing functions, they do not understand the specific drawing mechanism. For some specific applications, the programmers may find the GDI drawing functions and some related functions can not get correct results (Li, 2011). Therefor, it is necessary to understand the output methods, defects and he compromise processing methods of GDI drawing functions deeply and clearly. So that programmers can know the tolerance or ever error and get more accurate answers from the relevant calculation, such as the intersection and distance of geometric primitives, by reprogramming the applications. On the basis of the research of Windows drawing functions by LI Qing-yuan, the author studied the mechanism of GDI drawing functions, figured out the fundamental reason for the tolerance and error of the GDI drawing functions and gave the corresponding solutions.

ESSENCE OF GDI DRAWING FUNCTIONS

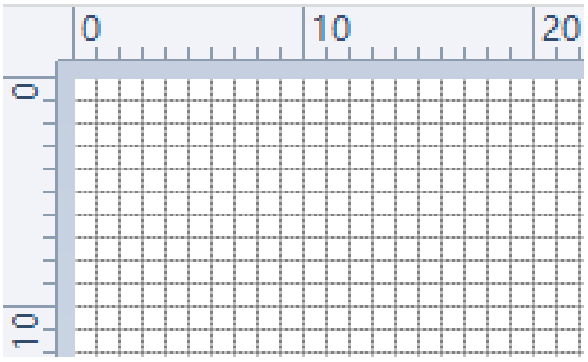
First of all, it is clear that in order to describe graphics, a suitable two-dimensional or three-

[1] China University of Mining & Technology, Beijing, China.
[2] Chinese Academy of Surveying and Mapping, Beijing, China.

dimensional Descartes coordinate system(world coordinate system) must be determined, and then use the geometric description (coordinate position, etc.) of the graphics in the world coordinate system to define the graphics objects. Then the display of the object can be achieved by transmitting the scene information to the observation function, identifying the visible face through observation function and mapping the objects to a video monitor (Donlad, 2010). After Specifying a graphic geometric elements in the world coordinate system, the output element will be projected onto the display area of the two-dimensional plane corresponding to the output device and then be scanned and converted to integer pixel position of the frame buffer.

The application uses GDI drawing functions and usually shows on a computer screen window. The location in the computer screen window (video monitor) is described by the screen coordinates corresponding to the integer pixel position of the frame buffer , and it is called a screen coordinate system. The coordinate value of a pixel include the x and y values, the y represents the number of scanning lines, the x represents the column number (the x value of scanning lines). Screen refresh and other hardware processing generally start to address the screen pixels from the upper left corner of the screen (Donlad, 2010)]. As shown in Figure 1, from the top scan line of the screen to the bottom scan line,they are numbered by 0,1,2, ..., ymax, the pixel location in each row are numbered from 0 to xmax from left to right. The graphics drawn through GDI functions ultimately represented by discrete integer pixels.

Figure 1: Screen Coordinate System



Readers of this article should first understand the common line drawing algorithm (linear equation, DDA algorithm, Bresenham line drawing algorithm) and the common area filling algorithm (Pixel judgment algorithm, universal scan line fill algorithm, boundary fill algorithm, flood fill algorithm).

GDI drawing functions use the logical coordinates. Common GDI drawing functions are: LineTo for drawing line, Polyline for drawing polyline, Polygon for drawing polygon, RECT and the corresponding function FillRect for rendering rectangle, Ellipse for drawing ellipse, RGN :: CreateRectRgn, RGN :: CreatePolygonRgn and the corresponding fill function-FillRgn for rendering region. This article will apply to the default mapping coordinates MM_TEXT[4] to draw these geometries using these drawing functions in Visual Studio 2010 under Windows7 operating system, and then analyze the display of the line endpoints, area vertexes and boundary pixels.

GDI DRAWING LINE FUNCTION

GDI provides the function that draw a straight line segment, BOOL CDC :: LineTo (POINT point) and the functions that draw a line segments, CDC :: Polyline (const POINT * lpPoints, int nCount).

Processing of Endpoint Pixels by Line to Function

Line-draw function, BOOL CDC::LineTo (POINT point), is annotated in the MSDN by "The LineTo function draws a line from the current position up to, but not including, the specified point. " (Jianchun, 2004)

A straight segment is defined by its start and end point coordinates . To show a line on the computer screen, the graphics system first projects the two end points to the location of integer screen coordinates, and then uses one line drawing algorithm (linear equation, DDA algorithm, Bresenham line drawing algorithm ,the location of the pixels between two end points may not be exactly same) to determine the pixel location that make the linear path between two end points nearest. This process will digitize a line segment to a group of adjacent and discrete integer screen pixel positions (screen coordinate). In general case , these locations are approximate to the actual line path (logical coordinates) except the horizontal line and the vertical line. For example, The real position of the line (7.39, 9.64) is convert to the pixel position (7,10). In addition, in the world coordinate system, a point presents a position without size in math, a line express a line without width in math as well, its area is zero, but when they are shown on the screen, they should occupy one pixel size or width at least. So, scan conversion algorithm must take into account the limited size of the pixels to maintain the geometric characteristics of the graphics (Mingtian, 1999).

In the function OnDraw (CDC * pDC) in the View class, a line is drawn from two directions of the positive and negative. After the amplification of eight times, the results are shown in Figure 2 and 3. (The following figures are the result of the graphics which are copied to the Windows drawing software and magnified eight times and shown with grid lines, the essence of graphics on the computer screen (pixel) can be clearly seen) The results show that the starting pixel position of the segment is correct, while the end pixel do not exist.

There are two advantages for Windows to do this: on the one hand it keeps the geometry length of a line, for example the line drawing from (0,0) to (10,0) whose geometrical length is 10 logical units has 10 pixels units as well after converting to the screen through scanning; On the other hand ,when draw lines end to end in the same direction, the end of the first segment, that is the starting point of the second segment, will not repeat to draw (David, 1987; Zhigang, 2008). The drawing line function will improve efficiency by this way. The same line in a Cartesian coordinate system displays different when draw in different directions in the device coordinate system, and this inconsistency depending on which extreme point is the starting point (shown), which is the end point (not shown) (microsoft Company). However, this process way has a defect, if some lines are drawn in a positive direction and some are drawn in the opposite direction in the graphics system, that will make the adjacent segments separate, as shown in Figure 3. This defect can be avoided by drawing the primitives in the same direction.

//The results to draw four segments in the positive direction are shown in Figure 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);

//The results to draw four segments in the opposite direction are shown in Figure 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);

//draw in the positive direction(Figure 3)

pDC->MoveTo(0,5);

pDC->LineTo(10,5);

//draw in the opposite direction(Figure 3)

pDC->MoveTo(20,5);

pDC->LineTo(10,5);

In Figure 3, the two segments are adjacent at (10,5) in the geometric sense, while they are separate on the pixel (10,5) in the drawing result.

In the application, if it is necessary to get the same result to draw the line segment in the positive direction and opposite direction, and remain the geometric characteristics at the same time, then the function CDC::SetPixel (POINT startPoint, COLORREF backColor) can be used to draw in the opposite direction to make the starting pixel not display (background color), and the function CDC::SetPixel (POINT endPoint,

Figure 2: Lines Drawing in Forward Direction(left) and Backward Direction(right) (Pixels in Gray Color are not Exist in Fact)

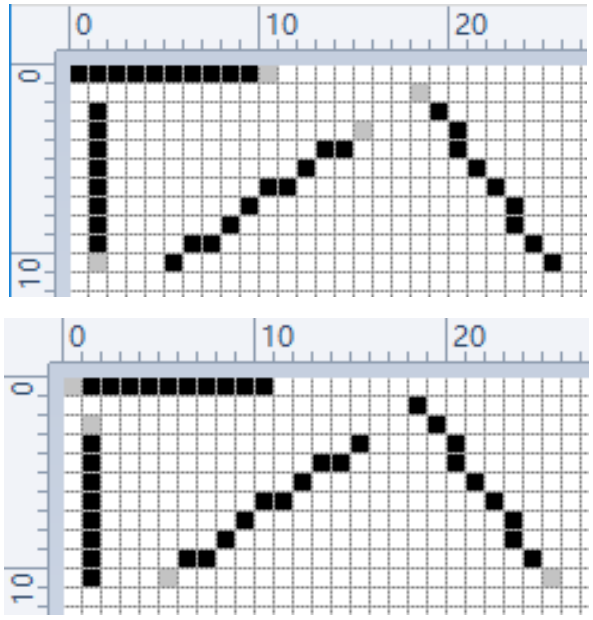
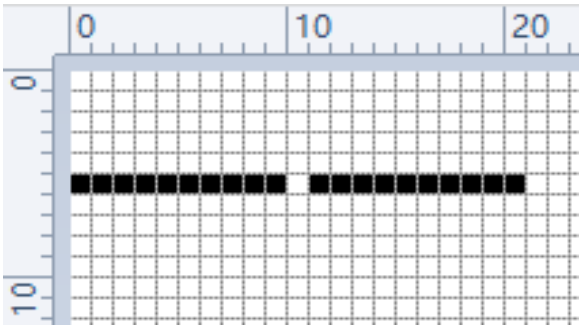


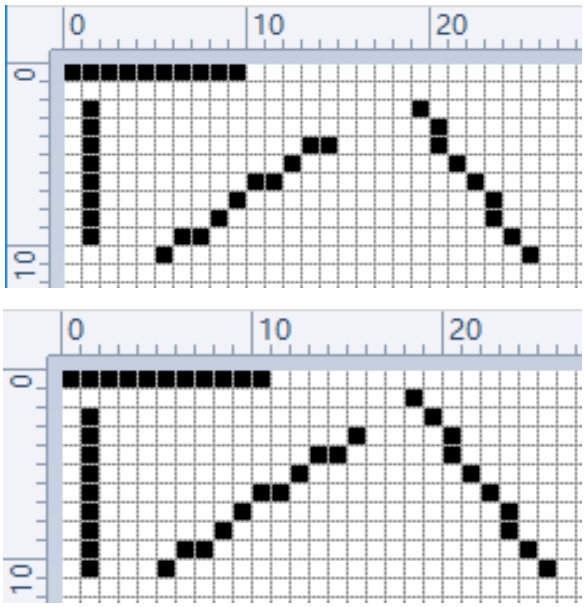
Figure 3: Adjacent Lines in Opposite Directions



COLORREF penColor) can be used to make the end-pixel display (pen color) (Figure 4 left). If it is simply necessary to draw the extreme points in the correct screen coordinates but do not consider the geometric characteristics of segments, the function CDC::SetPixel (POINT endPoint, COLORREF penColor) can be used to make the end pixel displayed (Figure 4 right).

As shown as Figure 2 and 4, using this method, all the pixels on the line have the same screen

Figure 4: Two Solutions to Drawing Lines



location. In the application, the impact of line drawing algorithm and drawing direction should be considered to calculate the distance between the point and the segment using pixel algorithms. If you want to judge whether the point is on the line segment, it would be best not to use pixel algorithms. In fact, the line drawing algorithm is to digitize the segment into a set of discrete integer position, these positions are approximate to the actual line except the horizontal and vertical segments (Donlad, 2010).

Processing of Endpoint Pixels by Polyline Function

The Polyline drawing function CDC::Polyline (const POINT * lpPoints, int nCount) comments in the MSDN : The Polyline function draws a series of line segments by connecting the points in the specified array (Microsoft Company) .

According to MSDN, polyline is formed by connecting a group of points to a series of connected line segments, and the expected results should show all the discrete points, including the extreme point (start point and endpoint).

But compare the result of drawing in the positive direction and the opposite direction by the function- Polyline (Figure 5), it will be found that the beginning and intermediate vertex coordinates have corresponding pixel points but the last point, and the points except the endpoints of the lines drawing from positive and negative directions have the same pixel screen positions, but the lines does not meet the geometric characteristics of the graphics in the Cartesian coordinate system. In fact, The function Polyline call the function LineTo circularly (Li, 2011), and the vertex pixels have no overprinted, the end to end adjacent lines drawn in the same direction have no overprint at the same time.

//Draw a polyline in the positive direction, Figure 5-left

CPoint ps (Chen, 2004);

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);

//Draw a polyline in the opposite direction, figure 5-right

ps[0].x = 24; ps[0].y = 5;

ps[1].x = 20; ps[1].y = 10;

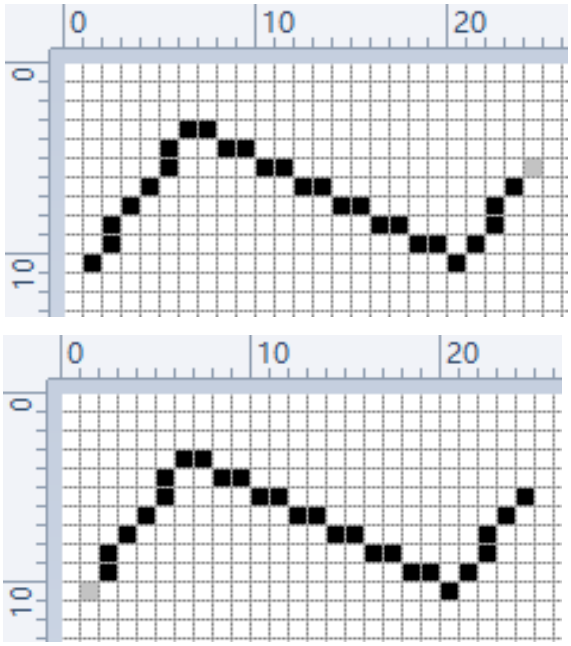
ps[2].x = 6; ps[2].y = 3;

ps[3].x = 1; ps[3].y = 10;

pDC->Polyline(ps,4);

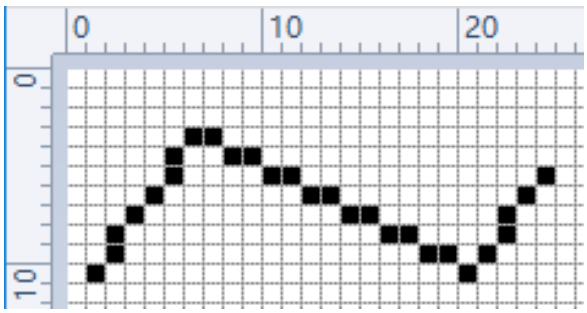
In the application, if it is necessary to display

Figure 5: Polylines Drawing in Forward Direction(left) and Backward Direction(right) (Pixels in Gray Color are not Exist in Fact)



the pixel of the end of the lines, the function CDC::SetPixel (POINT lastPoint, COLORREF penColor) can be used (Figure 6). And the line

Figure 6: Solution to Drawing Polylines



drawn in the positive direction has the same pixel positions with the line drawn in the opposite direction.

GDI DRAWING REGION FUNCTION

The functions of GDI to draw area include: CDC::Polygon for drawing polygon, Rectangular RECT class and the corresponding fill function-FillRect, CDC::Ellipse for drawing ellipse, RGN::CreateRectRgn for create rectangular region, RGN::CreatePolygonRgn for create polygon region and the corresponding filling function-FillRgn. Then it will mainly analyze the filling of polygon area (including rectangle).

In Computer Graphics, there are two kinds of methods to represent the polygon: vertex representation and lattice representation. Lattice representation describes a polygon using the set of pixels inside the polygon, it loses lots of important geometry information (Such as boundary, vertex, etc.) But it is the representation necessary for the raster display system (such as a computer screen) to display (David, 1987). The conversation from vertex representation to lattice representation called polygon scan-conversion. It firstly start from the vertices information of the polygon, calculate the set of pixels inside the polygon, and then writes the value of its color into the corresponding unit of the frame buffer. The algorithm of polygon scan-conversion that are commonly used include: point by point judgment algorithm, scan line algorithm, edge fill algorithm and flood fill algorithm.

Region Filling Function Polygon

The annotation of Polygon in MSDN: 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.

The function CDC::Polygon (const POINT * lpPoints, int nCount) use the boundary drawn by current brush to surround the polygons, and use the current brush to fill a polygon. The polygon

boundary drawn by the function of Polygon includes all the specified vertices pixel coordinates, but the geometry characteristics of the polygon are inconsistent. Such as the rectangle shown in Figure 7 (left), nine pixels long and 7 pixels wide, but it is 9-1 = 8 long, and 7-1 = 6 wide defined by logical coordinates. Additionally, the common boundary will overprint when the adjacent polygons are drawn by the function of Polygon. As shown in figure 7 (right), the right rectangle is drawn after the left rectangle, then the right edge of the left rectangle will be overwritten by the left edge of the right rectangle, which led to the repeat rendering in the same pixel position.

```
//Rectangle drawn by polygon-function (Figure 7 left)
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);
//Adjacent rectangles drawn by polygon-function(figure7 right)
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);
```

Ellipse Filling Function Ellipse

The annotation of Ellipse in MSDN : The Ellipse function draws an ellipse. The center of the ellipse is the center of the specified bounding rectangle. The ellipse boundary is drawn by current pen and interior region is filled by the current brush.

It can be seen from the example that the ellipse or circle (Figure 8) drawn by the function of CDC::Ellipse retain the geometrical characteristics (long axle, short axle, diameter). The ellipse drawn by the function of Ellipse also has some drawbacks (aliasing), such as the brown pixels shown in Figure 8 (actually does not exist) should be the boundary of the circle. This shows that the the boundary pixels (black) of the circle is not accurate.

```
//The ellipse has semi-major axis of 8 pixels and semi-minor axis of 6 pixels (Figure 8 left)
pDC->Ellipse(1,1,9,7);
//The circle has a diameter of 10 pixels (Figure 8 right)
pDC->Ellipse(0,0,10,10);
```

FillRect0FillSolidRect and FillRgn

Scan line algorithm is commonly used to convert

Figure 7: Rectangle(left) and Adjacent Rectangles(right) Drew by Polygon-function

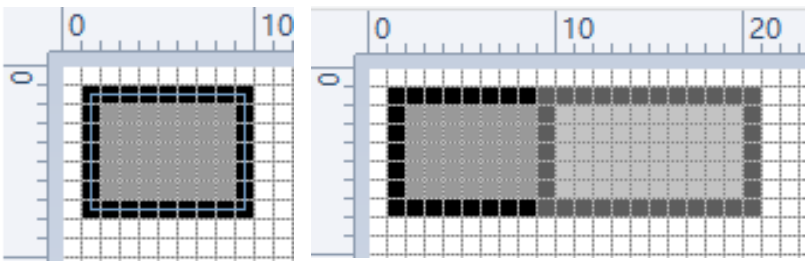
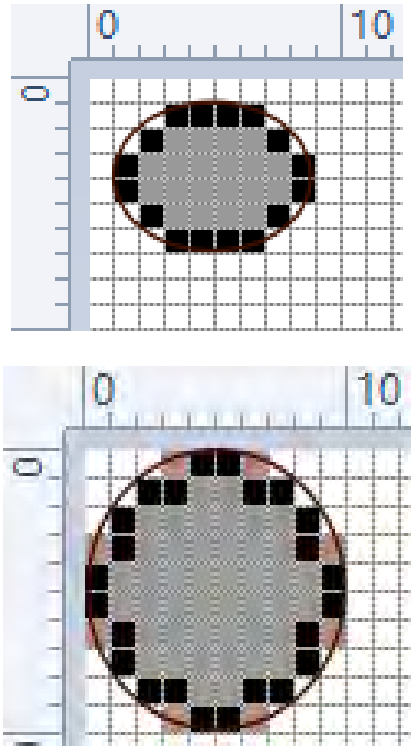


Figure 8: Ellipse (left) and Circle(right) Drew by Ellipse-function



the polygon (rectangle). The algorithm fill the polygon (rectangle) by scan-converting each central scan line across the entire drawing window. The filling process of a scan line can be divided into the following three steps:

(1) Get the intersection points of each scanning line and polygon edges;

(2) Sort the intersection points by the x coordinates from small to large;

(3) Pair the intersection points and fill each section.

MSDN did not give the method to fill polygons with GDI drawing functions, but lots of experiments show that, FillRect, FillSolidRect and FillRgn use the following rounding rules in the scan conversion of the intersection points on the boundary.

Assuming that a non-horizontal edge intersect with the central scanning line y=e (e is an integer), and the abscissa of the intersection point is denoted by x, then there are several conditions:

(1) x is a decimal, that is the intersection point (x, e) is located in the non-pixel center position of scan line y = e. If the intersection point is on the left edge of the polygon, then choose the right boundary pixel ((int) x + 1, e). If the intersection point is on the right side of the polygon boundary, choose the left boundary pixel ((int) x, e).

(2) If the intersection point (x, e) is right located on the integer pixel (pixel center), if (x, e) is on the left edge of a polygon, regarded it as belonging to polygons; if (x, e) is on the right border of a polygon, then it does not belong to the polygon.

(3) In (2), if the intersection point (x, e) falling on the center pixel is the vertex of a polygon, each edge of the polygon will be regarded as bottom–closed and top-open, which is equivalent to remove the pixel on the up endpoint of each edge.

It should be noted that, in MM_TEXT mapping, the origin is in the left corner , the positive direction of Y-axis is downward. The bottom and top in “bottom–closed and top-open” is opposite with the bottom and top of the computer screen.

In addition, since the horizontal edges are parallel with the scan lines, and in fact the horizontal edges has no effect in the scan conversion algorithm, so they can be removed in the pretreatment of the algorithm.

RECT class and FillRect and Rectangle (LPRECT lprect)

The annotation of RECT in MSDN: By convention, the right and bottom edges of the rectangle are

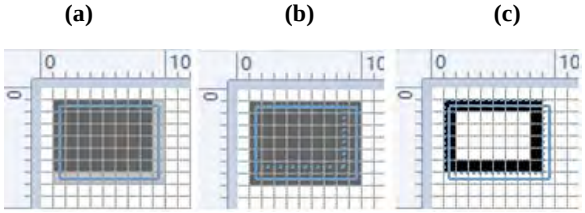
normally considered exclusive. In other words, the pixel whose coordinates are (right, bottom) lies immediately outside of the rectangle. For example, when RECT is passed to the FillRect function, the rectangle is filled up to, but not including, the right column and bottom row of pixels. This structure is identical to the RECTL structure.

The function of FillRect fills the entire rectangle, including the left and top boundary, but does not fill the right and bottom boundaries.

It can be found from the experimental results and the intersection point rounding rules and MSDN annotations that: the function of CDC::FillRect and CDC::FillSolidRect fill the rectangle defined by CRect with the right boundary and the bottom boundary not displayed (Figure 9a). This treatment method maintains the geometric characteristics (length and width) unchanged, and explains the reason why the points on the right boundary and the bottom boundary are judged to be outside the rectangle using PtInRect (RECT, POINT) to determine whether the point is within the rectangle. Because the function of PtInRect (RECT, POINT) algorithm uses pixel algorithm, all pixels that are displayed in the rectangle will return the true value.

```
)
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);//Figure 9(a)
//pDC->Rectangle(&rect);//Figure 9(c)
pDC->SelectObject(oldpen);
BOOL yn = 0;
```

Figure 9: CRect Filled by GDI (pixels in gray color are not exist in fact)




```
CPoint pt;
pt.x = 9; pt.y = 7;
yn = PtInRect(rect,pt);        //return 0
//return 0
```

In the application, if you do not consider the geometric characteristics of the rectangle, and need PtInRect to return the desired results, this can be done to add 1 to the x and y coordinate values of the vertices in the bottom-right corner of the rectangle when to display, then the right boundary and lower boundary of the original rectangle (Figure 9 (b) blue dotted boundary) will be judged to be in the original rectangle.

If you want to keep the geometric characteristics of the rectangle, and want PtInRect returns the desired results, it can be done to improve the PtInRect algorithm as following, so that the points on the right boundary and the bottom boundary are judged to be within the rectangle.

```
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);

}

It should be noted that the rectangle drawn by the function of CDC::Rectangle (LPCRECT lprect) (Figure 9 (c)), the black boundary pixels in the right and bottom are not the real boundaries of the rectangle. It returns zero when to pass the points on the right boundary and bottom boundary to the PtInRect function. It returns the true value when to determine the points on the up boundary and the left boundary. The annotation of Rectangle in MSDN can confirm this too: The rectangle that is drawn excludes the bottom and right edges. It also can be understood that the function is to map world coordinates into the screen position between pixels, so the object boundary (Figure 9 (c) blue dashed boundary) is aligned with the pixel boundaries (Figure 9 (c) black pixel boundaries), but not aligned with the pixel (area) center [3]. It not only displays the rectangle boundaries but also maintains the geometrical characteristics of the rectangle.

CRgn class and CRgn::CreatePolygonRgn, CRgn::CreateRectRgn and CDC::FillRgn

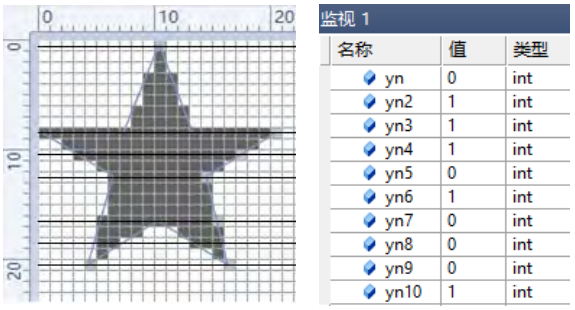
As show as the examples and the intersection point rounding rules, the rectangle created with CRgn::CreateRectRgn (int x1, int y1, int x2, int y2) and the rectangle defined by CRect have the consistent results that the right boundary and the bottom boundary are not displayed (Figure 9 (a)). The results of the judgment with PtInRegion and PtInRect are same too.

// Draw a rectangular area (the result is same as Figure 9 (a))

rgn.CreateRectRgn(1,1,9,7);

Figure 10: CRgn Filled by GDI

(a) **(b)**



pDC->FillRgn(&rgn,&brush);

PtInRegion(rgn,9,7); //return 0

//Draw the polygon area of pentagram (Figure. 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);//the results are same drawing in both directions

pDC->FillRgn(&rgn,&brush);

//determine whether the 10 vertices are within the polygon (results shown in Figure 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);
```

The annotation of Create<shape>Rgn in the MSDN: 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].

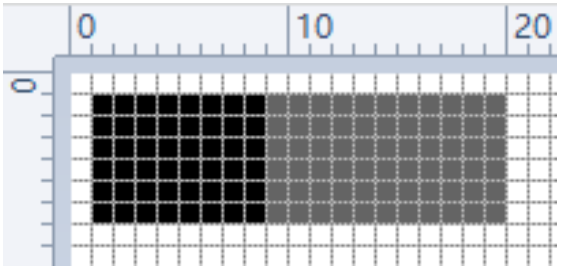
According to the annotation in MSDN, all the vertices and the integer pixel points of the boundaries are not in the polygonal area. That is all the values in the Figure 10(b) should be zero, but that is not the case.

According to the intersection point rounding rules and the processing of horizontal line, the vertex ps[0], intersecting with the scanning line y=0 on the right boundary, is determined not to be within the polygon, no matter whether to regard the ps[0] as one point or two points; During the processing of scan line y= 8, the two horizontal boundary are deleted firstly. According the principle of "bottom–closed and top-open", the vertex ps[2] is determined to be within the polygon and the vertex ps[8] is not, both vertexes are intersecting with the scanning line y=8, because ps[2] is a point on the left boundary while ps[8] is a point on the right boundary; The scanning line y=10, y=12 and y=18 is similar to the scanning line y=8; The scanning line y=16 and the point (5, 16) on the left boundary, the vertex ps[5] and the point (16, 16) on the right boundary are intersecting. The ps[5] is counted as two points, when to fill the pixels on the scanning line. The (5,16) is filled with brush color, the ps[5] is not filled when to fill the pixels between the (5, 16) and ps[5], but when to fill the pixel between (16, 16) and ps[5], ps[5] will fill with brush color and (16, 16) will not be filled. So it will return 1,1 and 0 to judge the three points with the function of PtInRegion respectively; According the principle of "bottom–closed and top-open", the scanning line y=20 does not intersect with ps[4] or ps[6], so the two points are not inside the polygon.

According to the above analysis, it can be seen that the judgment of the points on the boundary using the function of PtInRegion is inconsistent with the annotation in the MSDN.

In the application, if you want to display the integer pixels on the boundary and all of the vertices of the region, the function Polygon is recommend. Program a new judgment function based on the boundary point rounding rules above when you need to decide whether the point is in the region or on the boundary. The author have wrote two functions to judge whether the point is inside the polygon: BOOL PtInRegionEx (POINT * ps, int num, POINT p) and BOOL PtInRgn (POINT * ps, int num, POINT p) that have been shown in the appendix. These two functions can return TRUE with the point within the region and the point on the boundary, and return FALSE with the point outside the region.

Figure 11: Adjacent Rectangles Filled by FillRgn-function



It should be noticed when to display the adjacent polygons with Polygon, the pixels of the boundary will overwrite leading to repeating drawing (Fig. 7 (right) intermediate boundary), while the pixels of the adjacent polygons will not overwrite if the polygons are filled with FillRgn (Figure 11).

CONCLUSION

To sum up the experiments above, we can find that the output of the GDI drawing functions of the Windows follows the following criteria:

(1) Integer screen position represents the center of the pixel area;

(2) The start pixel of the LineTo function, GDI drawing line function, is correct. In order to keep a fixed length of the line segments, the LineTo function reduces a pixel at the end of the line segment, resulting line pixels drawing in different directions at the start and end points are not consistent;

(3) The GDI drawing polyline function Polyline is essentially a loop call for LineTo, therefor, there is one less pixel at the end of polyline. Pixels at the start and end points of polyline are not consistent;

(4) GDI region filling function Polygon is equivalent to draw the boundary pixels with Polyline, then fill the internal pixel collection inside the boundary, so the region through all the specified pixels;

(5) GDI drawing ellipse function Ellipse can keep the long axis and the short axis length of the ellipse, but the boundary pixels are not reasonable;

(6) GDI rectangle class CRect and region class CRgn and respective filled functions, according to the above rounding rule of the boundary points, decide integer boundary points are displayed or not, and how to display fractional boundary points. So the call of PtInRect and PtInRegion will receive inconsistent results at boundary points.

In general visualization applications, it is not necessary to consider how the GDI drawing functions work. But in some special graphics systems, such as when you need to measure the geometry of the graphics, determine whether the point is in the graphics or calculate the precise distance from point to graphic in the application system, we must get correct judging result and accurate drawing result. Because the error of one pixel on the screen may correspond to thousands of meters.

ACKNOWLEDGEMENT

This work is supported by National Nature Science Foundation of China project “Theory and method of anisotropic property field inner geology body based on volume function” (Projec number 41272367).

REFERENCES

1. [America] David E RogersProcedural Elements for Computer Graphics[M] Science PressBeijing198730-101.

2. [America] Donald Hearn,M.Pauline BakerComputer Graphics with OpenGL Third Edition[M]PUBLISHING HOUSE OF ELECTRONICS INDUSTRYBeijing201026-185.
3. [America] Microsoft CompanyMicrosoft Windows 3.1 Programmer Reference (Second)[M]TSINGHUA UNIVERSITY PRESSBeijing199391-641.
4. [America] Zhigang XiangComputer Graphics with OpenGL[M]PEKING UNIVERSITY PRESSBeijing200834-51.
5. Jianchun ChenVector Graphics System Development and Programming [M]Beijing200474-76.
6. LI Qing-yuanTAN HaiWANG TaoStudy on defects in GDI/GDI+ rendering functions and solutionsComputer Engineering and DesignBeijing201132(12)4256-4259
7. Lie ZhuBin ZhouC/C++ Advanced Programming under Windows[M]POSTS & TELECOM PRESSBeijing200279-141
8. Microsoft .Microsoft Developer Network [EB/OB], 2015/2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd162814(v=vs.85)
9. Microsoft .Microsoft Developer Network [EB/OB], 2015/2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd162510(v=vs.85)
10. Microsoft .Microsoft Developer Network [EB/OB], 2015/2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd162897(v=vs.85)
11. Microsoft .Microsoft Developer Network [EB/OB], 2015/2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd162898(v=vs.85)
12. Microsoft .Microsoft Developer Network [EB/OB], 2015/2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd183511(v=vs.85)
13. Microsoft. Microsoft Developer Network [EB/OB],2015/2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd145029(v=vs.85)
14. Microsoft.Microsoft Developer Network [EB/OB], 2015/2015-4-29. https://msdn.microsoft.com/en-us/library/windows/desktop/dd162815(v=vs.85)
15. Mingtian NiLiangzhi WuComputer Graphics[M]PEKING UNIVERSITY PRESSBeijing199943-91

APPENDIX

```
BOOL PtInRegionEx(POINT *ps, int num, POINT p)
{
    CRgn rgn;
    rgn.CreatePolygonRgn(ps,num,WINDING);
    BOOL YN = FALSE;
    //bounding box of polygon
    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;
    }
    //points in the bounding box
    int Dx, Dy, j;
    int minxl, maxxl, minyl, maxyl;//range of segments(bounding box)
    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++)
            {
```

APPENDIX

```
            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;
```

APPENDIX

```
    //if the point is in the boundary
                                    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;
```

APPENDIX

```
    //if the point is in the boundary
                                        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;
                }
        }
        //points outside the bounding box
        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;
        //caculate length of each edge respectively
        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));
      //if the segment is one point,exit and return the length
        if (c == 0)
        {
```

APPENDIX

```
        result = a;
        return result;
}
//if point(xx,yy) and point(x1,y1) are nearer
if (a < b)
{
        //if the line AB is horizontal, calculate the radian of line AB
        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;//radian of X-axis normal direction and line from (x1,y1) to (x2,y2)
        }
 m = (xx - x1) / a;
        if (m - 1 > 0.00001)
                m = 1;
        ang2 = acos(m);
        if (y1 > yy)
           ang2 = PI * 2 - ang2;//radian of X-axis normal direction and line from (x1,y1) to (xx,yy)
```

APPENDIX

```
            ang = ang2 - ang1;
            if (ang < 0)
                  ang = -ang;
            if (ang > PI)
                  ang = PI * 2 - ang;
            //if it is obtuse angle,return the length
            if (ang > PI / 2)
                  return a;
            else
                  return a * sin(ang);
      }
      else//,//if point(xx,yy) and point(x2,y2) are nearer
      {
            //if the y-coordinates of the two points are same
            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)
```

APPENDIX

```
                m = 1;
        ang2 = acos(m);//the radian of the slope of line (x2,y2)-(xx,yy)
        if (y2 > yy)
                ang2 = PI * 2 - ang2;
        ang = ang2 - ang1;
        if (ang < 0)
                ang = -ang;
        if (ang > PI)
                ang = PI * 2 - ang;//angle of intersection
        //if it is obtuse angle,return the length
        if (ang > PI / 2)
                return b;
        else
                //if it is acute angle,return the length through calculation
    }
}

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;
```

APPENDIX

```
        }
        if(p.x <= maxx && p.x >= minx && p.y <= maxy && p.y >= miny)//judge the points in the
bounding box
        {
                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)         //if the point is on the right of the line
                mind = -mind;
                //return false with the points outside the polygon
                if(mind<0)
```

APPENDIX

```
                return FALSE;
        else            //return true with the points inside the polygon and in the boundary
                return TRUE;
    }
    else//return false with the points outside the bounding box
        return FALSE;
}
```