USING STOCK PENS

The text describes how to use stock pens in Windows programming. Stock pens are predefined pens that Windows provides.

The three stock pens are BLACK_PEN, WHITE_PEN, and NULL_PEN.

- BLACK_PEN draws a solid black line with a width of one pixel.
- WHITE_PEN draws a solid white line with a width of one pixel.
- NULL_PEN is a pen that doesn't draw.

To use a stock pen, you first need to obtain a handle to it using the GetStockObject function. The GetStockObject function takes the name of the stock pen as an argument and returns a handle to the pen. For example, the following code obtains a handle to the WHITE_PEN:

```
HPEN hPen = GetStockObject(WHITE_PEN);
```

Once you have a handle to a pen, you need to select it into the device context using the SelectObject function. The SelectObject function takes two arguments: the device context and the pen handle. The following code selects the WHITE_PEN into the device context:

```
SelectObject(hdc, hPen);
```

Now any lines that you draw will use the WHITE_PEN until you select another pen into the device context or release the device context handle.

To return to using the BLACK_PEN, you can get the handle to that stock object and select it into the device context in one statement:

```
SelectObject(hdc, GetStockObject(BLACK_PEN));
```

The SelectObject function returns the handle to the pen that had been previously selected into the device context. If you start off with a fresh device context and call:

```
HPEN hPen = SelectObject(hdc, GetStockObject(WHITE_PEN));
```

The current pen in the device context will be WHITE_PEN and the variable hPen will be the handle to BLACK_PEN. You can then select BLACK_PEN into the device context by calling:

```
SelectObject(hdc, hPen);
```

Example Code:

```
HPEN hPen;

hPen = SelectObject(hdc, GetStockObject(WHITE_PEN));

// Draw a line using the WHITE_PEN
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);

// Select the BLACK_PEN back into the device context
SelectObject(hdc, hPen);

// Draw a line using the BLACK_PEN
MoveToEx(hdc, 200, 200, NULL);
LineTo(hdc, 300, 300);
```

This code will draw a white line from (100, 100) to (200, 200) and a black line from (200, 200) to (300, 300).

CREATING, SELECTING AND DELETING PENS

Creating Pens

To create a custom pen, you use the CreatePen or CreatePenIndirect functions. These functions take several parameters that define the appearance of the pen, such as the line style, line width, and color. The functions return a handle to the pen, which you can then select into the device context using the SelectObject function.

Selecting Pens

Only one pen can be selected into the device context at a time. To select a pen, you call the SelectObject function with the device context handle and the pen handle as arguments. Once a pen is selected, all lines that you draw will use that pen until you select another pen or release the device context.

Deleting Pens

When you are finished with a pen, you should delete it using the DeleteObject function. This will free up the resources that the pen was using. However, you should not delete a pen while it is still selected into a device context.

GDI Objects

A logical pen is a type of GDI object. GDI objects are resources that are managed by the Graphics Device Interface (GDI). There are six types of GDI objects: brushes, bitmaps, regions, fonts, palettes, and pens. GDI objects are selected into the device context using the SelectObject function.

Rules for Using GDI Objects.

There are three rules for using GDI objects:

- You should eventually delete all GDI objects that you create.
- Don't delete GDI objects while they are selected in a valid device context.
- Don't delete stock objects.

Example Code

The following code shows how to create a pen, select it into the device context, draw a line with the pen, and then delete the pen:

```
HPEN hPen;

hPen = CreatePen(PS_SOLID, 1, RGB(0, 0, 0));
SelectObject(hdc, hPen);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
DeleteObject(hPen);
```

This code will draw a black line from (100, 100) to (200, 200).

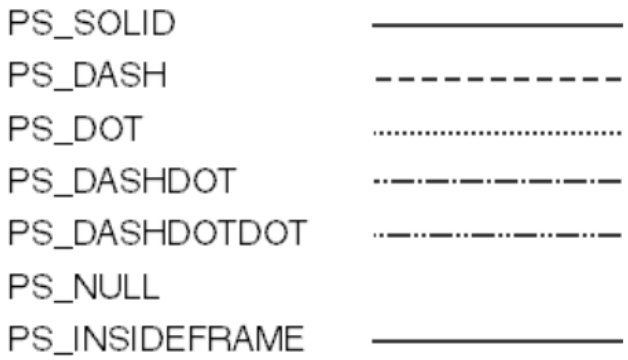


Figure 5–18. *The seven pen styles.*

PS_SOLID

A solid pen draws a solid line with a constant width. The width of the line is specified by the iWidth parameter to the CreatePen function. The following code shows how to draw a solid black line with a width of 2 pixels:

```
HPEN hPen;

hPen = CreatePen(PS_SOLID, 2, RGB(0, 0, 0));
SelectObject(hdc, hPen);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
```

PS_DASH

A dashed pen draws a line that is made up of a series of dashes. The length of the dashes and the spacing between them is specified by the iWidth parameter to the CreatePen function. The following code shows how to draw a dashed black line with a dash length of 10 pixels and a spacing of 5 pixels:

```
HPEN hPen;

hPen = CreatePen(PS_DASH, 10, 5);
SelectObject(hdc, hPen);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
```

PS_DOT

A dotted pen draws a line that is made up of a series of dots. The size of the dots is specified by the iWidth parameter to the CreatePen function. The following code shows how to draw a dotted black line with a dot size of 5 pixels:

```
HPEN hPen;

hPen = CreatePen(PS_DOT, 5, BLACK);
SelectObject(hdc, hPen);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
```

PS_DASHDOT

A dash-dot pen draws a line that is made up of a series of alternating dashes and dots. The length of the dashes and the size of the dots are specified by the iWidth parameter to the CreatePen function. The following code shows how to draw a dash-dot black line with a dash length of 10 pixels, a dot size of 5 pixels, and a spacing of 5 pixels between the dashes and dots:

```
HPEN hPen;

hPen = CreatePen(PS_DASHDOT, 10, 5, 5);
SelectObject(hdc, hPen);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
```

PS_DASHDOTDOT

A dash-dot-dot pen draws a line that is made up of a series of alternating dashes and double dots. The length of the dashes and the size of the dots are specified by the iWidth parameter to the CreatePen function. The following code shows how to draw a dash-dot-dot black line with a dash length of 10 pixels, a dot size of 5 pixels, and a spacing of 5 pixels between the dashes and dots:

```
HPEN hPen;

hPen = CreatePen(PS_DASHDOTDOT, 10, 5, 5);
SelectObject(hdc, hPen);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
```

PS_NULL

A null pen does not draw anything. The following code shows how to draw a null line:

```
HPEN hPen;

hPen = CreatePen(PS_NULL, 0, BLACK);
SelectObject(hdc, hPen);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
```

PS_INSIDEFRAME

The PS_INSIDEFRAME pen style is a special pen style that is used to draw lines that are clipped to the inside of a frame. The frame is specified by the iWidth parameter to the CreatePen function. The following code shows how to draw a solid black line with a width of 2 pixels that is clipped to the inside of a frame with a width of 10 pixels:

```
HPEN hPen;

hPen = CreatePen(PS_INSIDEFRAME, 2, BLACK);
SelectObject(hdc, hPen);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);

DeleteObject(hPen);
```

The provided code snippet is a complete and functional code segment that draws a line using the PS_INSIDEFRAME pen style, a width of 2 pixels, and a black color. It also deletes the pen when it is no longer needed.

Creating and Managing Pens

Pens are used to draw lines and shapes in Windows programming. There are three types of pens: solid pens, dotted pens, and dashed pens.

- *Solid pens draw solid lines, dotted pens draw lines that consist of a series of dots, and dashed pens draw lines that consist of a series of dashes.*

The CreatePen and CreatePenIndirect functions are used to create pens. The CreatePen function takes three arguments: the pen style, the line width, and the color of the pen. The CreatePenIndirect function takes a pointer to a structure of type LOGPEN, which contains the pen style, line width, and color.

Once a pen has been created, it must be selected into the device context before it can be used. The SelectObject function is used to select a pen into the device context. The SelectObject function takes two arguments: the device context and the pen handle.

When a pen is no longer needed, it should be deleted using the DeleteObject function. The DeleteObject function takes a pen handle as an argument.

Line Width

The line width is the width of the line that the pen draws. The line width is specified by the iWidth parameter to the CreatePen function. The iWidth parameter can be a positive integer or zero. A positive integer specifies the width of the line in pixels. Zero specifies a line width of one pixel.

Color

The color of the pen is the color of the line that the pen draws. The color is specified by the crColor parameter to the CreatePen function. The crColor parameter is a COLORREF value, which is a 32-bit value that contains the red, green, and blue components of the color.

Dithered Colors

The PS_INSIDEFRAME pen style is the only pen style that can use a dithered color. A dithered color is a color that is made up of a pattern of other colors. Dithered colors are used to simulate colors that cannot be displayed by the device.

Creating Pens at Initialization

If your program uses a lot of different pens that you initialize in your source code, you can create the pens using the CreatePenIndirect function and store the pen handles in static variables. This can be more efficient than creating the pens each time you need to use them.

Selecting and Deleting Pens

To select a pen into the device context, you use the SelectObject function. The SelectObject function takes two arguments: the device context and the pen handle. To delete a pen, you use the DeleteObject function. The DeleteObject function takes a pen handle as an argument.

Example Code

The following code shows how to create, select, and delete pens:

```
HPEN hPen1, hPen2, hPen3;  // Declare three pen handles

// Create three pens with different styles and attributes
hPen1 = CreatePen(PS_SOLID, 1, RGB(0, 0, 0));  // Solid black pen with width 1 pixel
hPen2 = CreatePen(PS_SOLID, 3, RGB(255, 0, 0));  // Solid red pen with width 3 pixels
hPen3 = CreatePen(PS_DOT, 0, RGB(0, 0, 0));  // Dotted black pen with default width 0 pixels

// Select the second pen into the device context (hPen2) for drawing
SelectObject(hdc, hPen2);
// Draw lines using the selected pen (hPen2)
[Line-drawing functions]

// Select the first pen into the device context (hPen1) for drawing
SelectObject(hdc, hPen1);
// Draw lines using the selected pen (hPen1)
[Line-drawing functions]

// Release the device context before deleting the pens
ReleaseDC(hWnd, hdc);

// Delete the pens that are no longer needed
DeleteObject(hPen1);
DeleteObject(hPen2);
DeleteObject(hPen3);

// Destroy the window
DestroyWindow(hWnd);
```

Deleting Pens

There are two main methods for deleting pens:

Delete the pens during WM_DESTROY processing: This method involves deleting the pens when the window is destroyed. This is the most straightforward approach, but it requires that your program knows which pens will be needed beforehand.

```
DeleteObject(hPen1);
DeleteObject(hPen2);
DeleteObject(hPen3);
```

Create and delete pens during WM_PAINT processing: This method involves creating the pens during each WM_PAINT message and deleting them after calling EndPaint. This approach is more flexible, as it allows you to create pens as needed, but it requires careful handling to avoid deleting the pen currently selected in the device context.

```
83  SelectObject(hdc, CreatePen(PS_DASH, 0, RGB(255, 0, 0)));
84
85  // Draw lines using the selected pen
86  //[Line-drawing functions]
87  //...
88
89  DeleteObject(SelectObject(hdc, GetStockObject(BLACK_PEN)));
```

Combining CreatePen and SelectObject

You can combine the CreatePen and SelectObject calls into a single statement to create a pen on the fly and select it into the device context. This is a concise and efficient approach, but it requires you to handle the deletion of the pen carefully.

```
 96  SelectObject(hdc, CreatePen(PS_DASH, 0, RGB(255, 0, 0)));
 97
 98  // Draw lines using the selected pen
 99  //[Line-drawing functions]
100  //...
101
102  DeleteObject(SelectObject(hdc, GetStockObject(BLACK_PEN)));
```

Retrieving Pen Information

You can use the GetObject function to obtain the values of the LOGPEN structure fields for a given pen handle. This can be useful for inspecting the properties of a pen.

```
GetObject(hPen, sizeof(LOGPEN), (LPVOID) &logpen);
```

Getting the Currently Selected Pen

You can use the GetCurrentObject function to retrieve the handle to the pen that is currently selected into the device context. This can be useful when you need to switch between pens without explicitly saving their handles.

```
hPen = GetCurrentObject(hdc, OBJ_PEN);
```

ExtCreatePen

The ExtCreatePen function is another method for creating pens. It provides more flexibility than the CreatePen function, allowing you to specify additional pen attributes such as the join style, end cap style, and miter limit. This function is discussed in more detail in Chapter 17 of the reference material.

Filling in the Gaps with Dotted and Dashed Pens

When using dotted or dashed pens, a question arises: what happens to the gaps between the dots or dashes?

By default, Windows fills in these gaps with the background color of the device context. This behavior aligns with the common practice of using a white background brush to erase the window background.

Background Mode and Background Color

The background mode and background color of the device context determine how the gaps between dots and dashes are treated.

Background Mode:

- OPAQUE: The default mode, where Windows fills in the gaps with the background color.
- TRANSPARENT: Windows ignores the background color and leaves the gaps transparent.

Background Color: The color that Windows uses to fill in the gaps when the background mode is OPAQUE.

Modifying Background Color and Mode

You can modify the background color and mode using the following functions:

- SetBkColor(hdc, crColor): Sets the background color of the device context.
- GetBkColor(hdc): Retrieves the current background color of the device context.
- SetBkMode(hdc, TRANSPARENT): Sets the background mode to TRANSPARENT, preventing Windows from filling in the gaps.
- GetBkMode(hdc): Retrieves the current background mode of the device context.

Example

The following code snippet sets the background color to red and the background mode to TRANSPARENT:

```
SetBkColor(hdc, RGB(255, 0, 0));
SetBkMode(hdc, TRANSPARENT);
```

DRAWING MODES AND RASTER OPERATIONS(ROPs)

In Windows graphics programming, the appearance of lines drawn on the screen is influenced by the drawing mode defined in the device context.

The drawing mode determines how the color of the pen interacts with the color of the underlying display surface. This allows for a variety of effects, such as drawing lines that appear to change color based on the background color.

Bitwise Boolean Operations for Raster Operations

When drawing a line, Windows performs a bitwise Boolean operation between the pixels of the pen and the pixels of the destination display surface.

Each pixel has a value that represents its color, and the Boolean operation determines the new color value for each pixel based on the values of the corresponding pen pixel and destination pixel.

Binary Raster Operations (ROP2) for Line Drawing

Since line drawing involves only two pixel patterns (the pen and the destination), the Boolean operation used is called a "binary raster operation" or "ROP2." Windows provides 16 different ROP2 codes that specify how to combine the pen pixels and the destination pixels.

Default Drawing Mode: R2_COPYPEN

The default drawing mode in the device context is R2_COPYPEN. This mode simply copies the pixels of the pen to the destination, resulting in the expected behavior of drawing lines using the pen's color.

Other ROP2 Codes and Their Effects

The 15 other ROP2 codes provide various ways to combine pen and destination colors. Some examples include:

The table below summarizes the 16 ROP2 drawing modes and their corresponding Boolean operations:

ROP2 Code	Boolean Operation	Description
R2_BLACK	D & 0	Always draws black.
R2_WHITE	D & 1	Always draws white.
R2_NOTMERGEPEN	~ (D & P)	Inverts the pen color.
R2_MERGEPENDP	D	P
R2_NOTCOPYPEN	~P	Inverts the pen color.
R2_COPYPEN	P	Draws using the pen color.
R2_NOXORPEN	0	Does not draw.
R2_XORPEN	D ^ P	Performs an exclusive-OR operation.
R2_MASKPEN	D & P	Draws using the pen color, but only where the destination color is black.
R2_INVERT	~D	Inverts the destination color.
R2_NOMIRRORPEN	D & ~P	Draws using the pen color, but excludes the pen color where the destination color is black.
R2_MERGECOPYPEN	D & P	Draws using the pen color, but only where the destination color is white.
R2_MIRRORPEN	~P	Inverts the pen color.

Explanation:

ROP2 Code: The numerical code used to represent the raster operation.

Boolean Operation: The boolean operation or combination of operations represented by the ROP2 code.

Description: A brief description of the effect or operation achieved by the specified boolean operation.

Please note that the symbols used in the Boolean operations are as follows:

D: Destination color.

P: Pen color.

~: NOT operation.

&: AND operation.

|: OR operation.

^: XOR operation.

Setting and Getting Drawing Modes

The drawing mode of a device context determines how the color of the pen interacts with the color of the underlying display surface when drawing lines or shapes. To set the drawing mode, use the SetROP2 function:

```
SetROP2(hdc, iDrawMode);
```

The iDrawMode argument specifies the desired drawing mode, which is one of the 16 ROP2 codes defined by Windows. To retrieve the current drawing mode, use the GetROP2 function:

```
iDrawMode = GetROP2(hdc);
```

Default Drawing Mode (R2_COPYPEN)

The default drawing mode is R2_COPYPEN, which simply copies the pen color to the destination. This means that lines or shapes drawn using this mode will appear in the same color as the pen.

R2_NOTCOPYPEN Mode

The R2_NOTCOPYPEN mode inverts the color of the pen before drawing. As a result, lines or shapes drawn with a black pen will appear as white, and lines or shapes drawn with a white pen will appear as black.

R2_BLACK and R2_WHITE Modes

The R2_BLACK mode always draws lines or shapes as black, regardless of the pen color or background color. Similarly, the R2_WHITE mode always draws lines or shapes as white.

R2_NOP Mode

The R2_NOP mode, also known as the "no operation" mode, does not draw anything. It leaves the destination unchanged, essentially erasing any previous drawing.

ROP2 Codes on Color Systems

While the previous examples focused on monochrome systems, most modern systems use color displays.

On color systems, Windows applies the bitwise operation defined by the ROP2 code to each color bit of the pen and destination pixels.

This allows for a wider range of visual effects when drawing lines or shapes.

R2_NOT Drawing Mode

The R2_NOT drawing mode inverts the destination color to determine the color of the line, regardless of the pen color. For instance, drawing a line with a black pen on a cyan destination will result in a magenta line. This mode always produces a visible line except when drawing with a black pen on a medium gray background.

Practical Applications of ROP2 Codes

The ROP2 codes provide various ways to control the appearance of lines and shapes drawn using pens.

For example, the R2_NOT mode can be used to create contrasting lines on colored backgrounds, while the R2_MERGEPEN mode can be used to blend lines with the background.

These drawing modes and their corresponding ROP2 codes offer flexibility in creating various visual effects and enhancing the appearance of graphical elements in Windows applications.

DRAWING FILLED AREAS

Drawing Filled Areas with Borders

Drawing filled areas with borders involves utilizing a combination of pens and brushes to define the outline and interior of the desired shape.

Windows provides several functions for drawing various filled shapes, including rectangles, ellipses, rounded rectangles, chords, pies, polygons, and poly-polygons.

Functions for Drawing Filled Shapes

The following table summarizes the functions for drawing filled shapes:

Function	Description
FillRect	Fills a rectangular area with the current brush
Ellipse	Draws an ellipse and fills the interior with the current brush
RoundRect	Draws a rectangle with rounded corners and fills the interior with the current brush
Chord	Draws an arc on the circumference of an ellipse and fills the interior with the current brush
Pie	Draws a pie wedge defined by the circumference of an ellipse and fills the interior with the current brush
Polygon	Draws a polygon and fills the interior with the current brush
PolyPolygon	Draws multiple polygons and fills the interior of each polygon with the current brush

Using Pens and Brushes

The outline of the filled shape is drawn using the current pen selected in the device context. The selected pen determines the color, width, and style of the outline. To select a pen, use the SelectObject function:

```
SelectObject(hdc, hPen);
```

where hdc is the device context and hPen is the handle to the pen.

The interior of the filled shape is filled using the current brush selected in the device context. The selected brush determines the color and pattern of the interior. To select a brush, use the SelectObject function:

```
SelectObject(hdc, hBrush);
```

where hdc is the device context and hBrush is the handle to the brush.

Default Brushes

By default, Windows uses the WHITE_BRUSH for filling the interior of shapes. You can change the default brush by selecting a different brush into the device context.

Windows defines several stock brushes, such as LTGRAY_BRUSH, GRAY_BRUSH, DKGRAY_BRUSH, BLACK_BRUSH, and NULL_BRUSH. To select a stock brush, use the GetStockObject function:

```
HBRUSH hBrush = GetStockObject(GRAY_BRUSH);
SelectObject(hdc, hBrush);
```

Drawing without a Border

To draw a filled shape without a border, select the NULL_PEN into the device context:

```
SelectObject(hdc, GetStockObject(NULL_PEN));
```

This will prevent Windows from drawing the outline of the shape.

Drawing the Outline without Filling

To draw the outline of a shape without filling the interior, select the NULL_BRUSH into the device context:

```
SelectObject(hdc, GetStockObject(NULL_BRUSH));
```

This will draw the outline of the shape using the current pen color, but the interior will remain transparent.

Customizing Brushes

In addition to using stock brushes, you can also create customized brushes. This allows you to define more complex brush patterns. Creating customized brushes is covered in more detail in subsequent chapters.

Code Examples

Here are some code examples for drawing filled areas with borders:

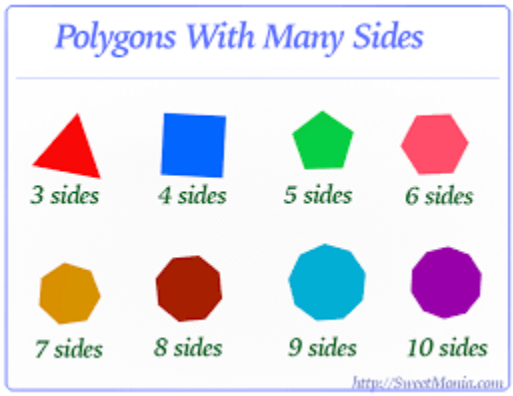
```
// Draw a filled rectangle with a black outline
HBRUSH hBrush = GetStockObject(RED_BRUSH);
SelectObject(hdc, hBrush);

HPEN hPen = CreatePen(PS_SOLID, 2, RGB(0, 0, 0));
SelectObject(hdc, hPen);

FillRect(hdc, 100, 100, 200, 200);

// Draw a filled ellipse with a blue outline
hBrush = GetStockObject(GREEN_BRUSH);
SelectObject(hdc, hBrush);

hPen = CreatePen(PS_SOLID, 3, RGB(0, 0, 255));
SelectObject(hdc, hPen);

Ellipse(hdc, 300, 300, 400, 400);
```

These examples demonstrate the use of FillRect and Ellipse to draw filled shapes with borders. The code first selects the desired brush and pen into the device context, and then calls the respective drawing function to create the shape.

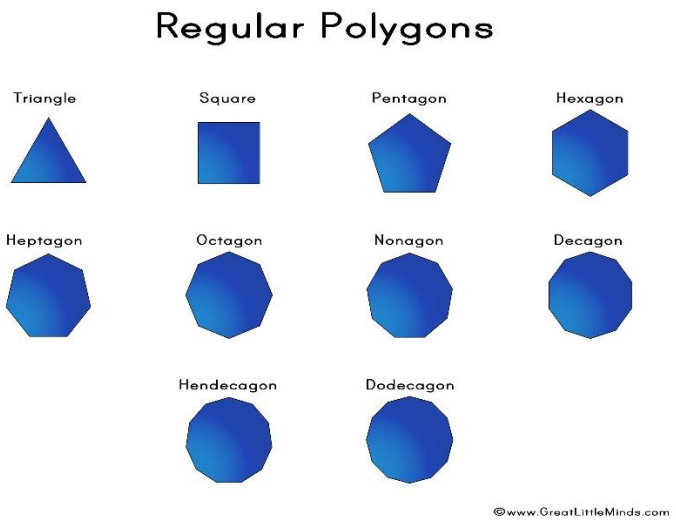
```
// Draw a filled rectangle with a black outline
HBRUSH hBrush = CreateSolidBrush(RGB(255, 0, 0)); // Create a solid red brush
SelectObject(hdc, hBrush); // Select the red brush into the device context

HPEN hPen = CreatePen(PS_SOLID, 2, RGB(0, 0, 0)); // Create a black pen with a width of 2 pixels
SelectObject(hdc, hPen); // Select the black pen into the device context

// Draw a filled rectangle with coordinates (100, 100), (200, 100), (200, 200), and (100, 200)
Rectangle(hdc, 100, 100, 200, 200);

// Draw a filled ellipse with a blue outline
DeleteObject(hBrush); // Delete the red brush to free up resources

hBrush = CreateSolidBrush(RGB(0, 255, 0)); // Create a solid green brush
SelectObject(hdc, hBrush); // Select the green brush into the device context

DeleteObject(hPen); // Delete the black pen to free up resources

hPen = CreatePen(PS_SOLID, 3, RGB(0, 0, 255)); // Create a blue pen with a width of 3 pixels
SelectObject(hdc, hPen); // Select the blue pen into the device context

// Draw a filled ellipse with coordinates (300, 300), (400, 300), (400, 400), and (300, 400)
Ellipse(hdc, 300, 300, 400, 400);

// Clean up resources
DeleteObject(hBrush); // Delete the green brush to avoid memory leaks
DeleteObject(hPen); // Delete the blue pen to avoid memory leaks
```

DRAWING POLYGONS

In-depth explanation of the Polygon function, the Polygon-filling mode, and the SetPolyFillMode function in Windows graphics programming.



The Polygon and PolyPolygon functions are used to draw polygons, which are multi-sided shapes composed of connected line segments. The Polygon function draws a single polygon, while the PolyPolygon function draws multiple polygons.



Polygon Function

The Polygon function takes three arguments:

- **hdc:** The device context in which to draw the polygon
- **apt:** An array of POINT structures that define the vertices of the polygon
- **iCount:** The number of vertices in the polygon

The POINT structure is defined as:

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
} POINT;
```

The Polygon function connects the specified vertices with lines and fills the enclosed area with the current brush. If the last vertex in the apt array is different from the first vertex, Windows automatically connects the last vertex to the first vertex, closing the polygon.

PolyPolygon Function

The PolyPolygon function takes four arguments:

- **hdc:** The device context in which to draw the polygons
- **apt:** An array of POINT structures that define the vertices of all polygons
- **aiCounts:** An array of integers that specify the number of vertices in each polygon
- **iPolyCount:** The number of polygons to draw

The PolyPolygon function draws multiple polygons based on the provided vertex data and vertex counts. It fills the enclosed areas of each polygon with the current brush, just like the Polygon function.

Polygon-Filling Mode

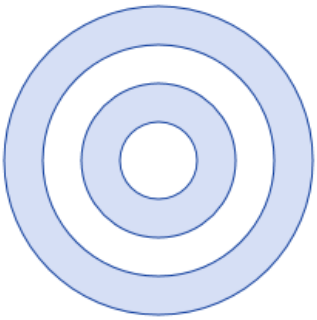
The polygon-filling mode determines how the interior of a polygon is filled. The default polygon-filling mode is ALTERNATE, but you can set it to WINDING using the SetPolyFillMode function:

```
SetPolyFillMode(hdc, iMode);
```

The iMode parameter can be either ALTERNATE or WINDING.

Alternate Mode

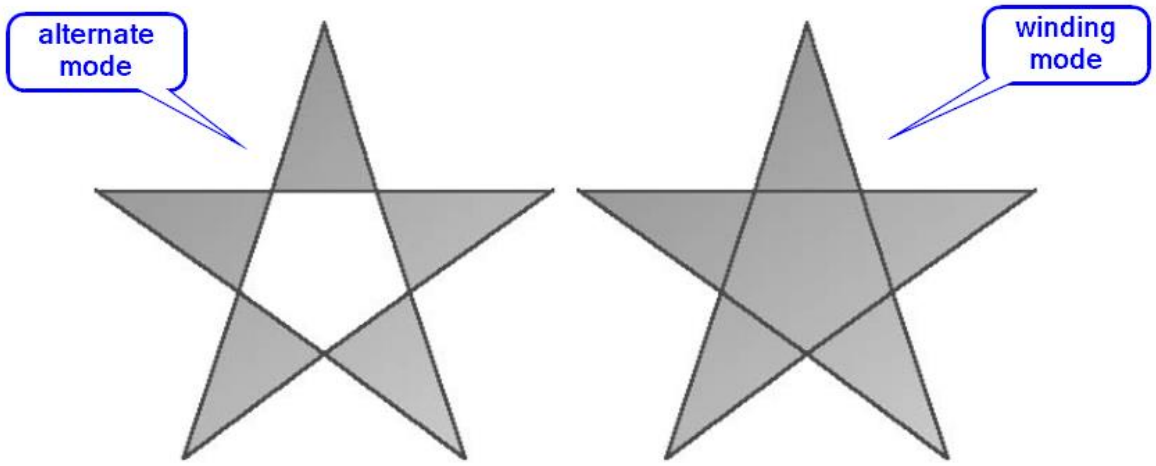
In alternate mode, an imaginary line is drawn from a point inside the enclosed area to infinity. The enclosed area is filled only if the imaginary line crosses an odd number of polygon boundary lines. This is why the points of the star in Figure 5-19 are filled but the center is not.



D2D1_FILL_MODE_ALTERNATE

Winding Mode

Winding mode is more complex than alternate mode and is generally considered to be more robust. It calculates a winding number for each point inside the polygon based on the number of times the polygon boundary lines wrap around the point. The point is considered inside the polygon if the winding number is non-zero.



Choosing the Polygon-Filling Mode

In most cases, winding mode will fill all enclosed areas of a single polygon. However, for complex polygons with self-intersections or holes, alternate mode may be more appropriate.

Winding Mode for Filling Enclosed Areas

While winding mode generally fills all enclosed areas of a single polygon, there are exceptions. To determine whether an enclosed area is filled in winding mode, follow these steps:

- Imagine a line drawn from a point inside the enclosed area to infinity.
- Count the number of times the polygon boundary lines cross this imaginary line.
- If the number of boundary line crossings is odd, the area is filled.
- If the number of boundary line crossings is even, the area is filled if the number of boundary lines going in one direction is not equal to the number of boundary lines going in the other direction.

Filling Enclosed Areas in Figure 5-20

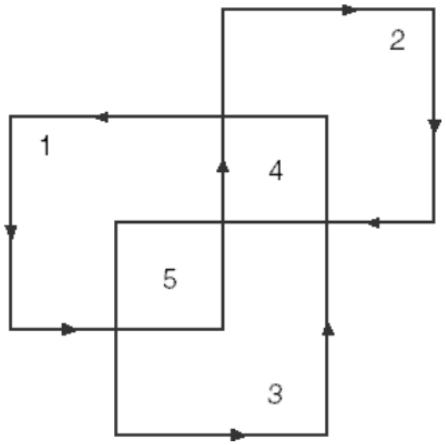


Figure 5–20. *A figure in which winding mode does not fill all interior areas.*

Applying these rules to the figure in Figure 5-20, we get the following results:

Enclosed areas 1, 2, and 3: Both winding mode and alternate mode will fill these areas.

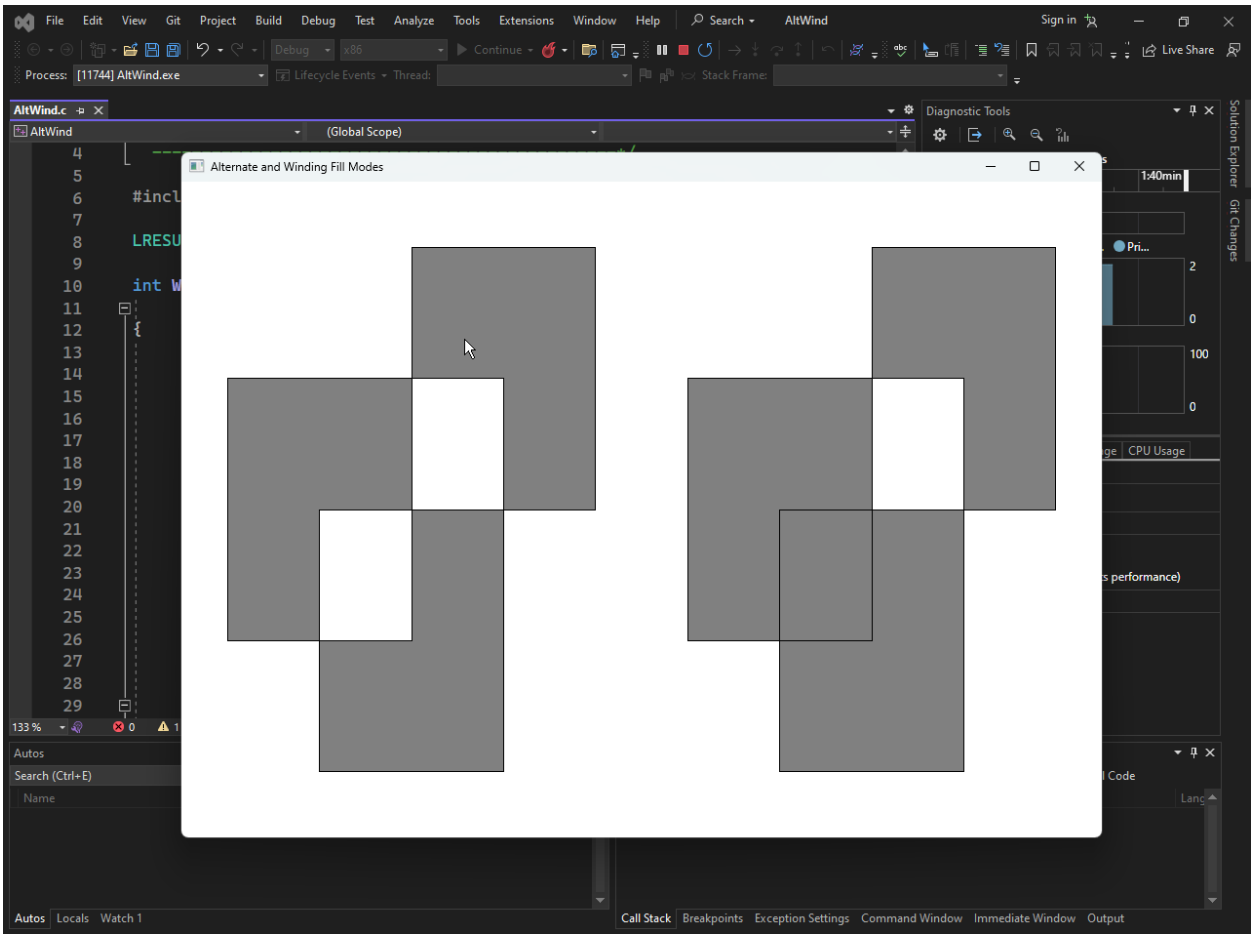
Enclosed area 4: Winding mode will not fill this area. While the number of boundary line crossings is even (two), the two lines go in opposite directions, resulting in a winding number of zero.

Enclosed area 5: Winding mode will fill this area. The number of boundary line crossings is even (two), but both lines go in the same direction, resulting in a winding number of two.

ALTWIND Program Demonstration

The ALTWIND program in Figure 5-21 demonstrates that Windows correctly handles winding mode for filling enclosed areas. The program draws a complex figure with multiple enclosed areas and displays the winding number for each area. As expected, only the areas with non-zero winding numbers are filled.

Program source code in Chapter 5, altwind folder.



The program is structured as a typical Windows application and consists of two main parts: the WinMain function, which serves as the entry point for the application, and the WndProc function, which handles messages for the application's window.

In the WinMain function, the program initiates the window class, creates a window, and enters the message loop. The window class is registered with specific attributes, such as the window procedure (WndProc), the instance handle, and the application name ("AltWind"). If the class registration fails, an error message is displayed, and the program exits.

The window is created using CreateWindow, and the application window is displayed and updated. The message loop is responsible for processing messages, including user input and system events.

The WndProc function is the window procedure that handles messages related to the window. It includes the handling of three main messages: WM_SIZE, WM_PAINT, and WM_DESTROY.

- When the window is resized, WM_SIZE is triggered. The client area dimensions are updated in response to this message.
- This message is responsible for painting or repainting the window. Inside the WM_PAINT block:
- The program begins painting with BeginPaint, sets the brush color to gray, and calculates scaled coordinates for the figure based on the client area size.
- The figure is drawn using Polygon with the fill mode set to ALTERNATE.
- The x-coordinates of the figure are then shifted, and the figure is redrawn with the fill mode set to WINDING.
- Finally, EndPaint is called to end the painting process.
- When the user closes the window, WM_DESTROY is triggered, leading to the termination of the application.

The figure to be drawn is defined by an array of POINT structures named aptFigure, representing the (x, y) coordinates of the vertices. The program uses GDI (Graphics Device Interface) functions to draw polygons and handle window-related tasks.

In summary, this code demonstrates a Windows application that draws a geometric figure twice within a window, each time using a different fill mode (ALTERNATE and WINDING). It provides insights into handling window messages and utilizing GDI functions for graphics rendering in a Windows environment.

BRUSHING THE INTERIORS OF SHAPES

The interiors of various shapes, including rectangles, rounded rectangles, ellipses, chords, pies, polygons, and poly-polygons, are filled with the current brush selected in the device context.

This brush defines the pattern or texture used to fill the interior of the shape.

Brush Characteristics

A brush is essentially a small 8-pixel-by-8-pixel bitmap that is repeated horizontally and vertically to fill the area. When the display capabilities are limited, Windows utilizes dithering to create the illusion of more colors. Dithering involves strategically placing black and white pixels to simulate various shades of gray or other colors.

Types of Brushes

Windows provides five functions for creating logical brushes:

CreateSolidBrush:

This function creates a solid-color brush, where the interior of the shape is filled with a uniform color. Windows may create a dithered bitmap based on the specified color depending on the display capabilities.

```
HBRUSH CreateSolidBrush(COLORREF crColor);
```

The CreateSolidBrush function creates a solid-color brush, where the interior of the shape is filled with a uniform color. The color is specified using the crColor argument, which is a COLORREF value. Windows may convert this color to the nearest available pure color depending on the display capabilities.

CreateHatchBrush:

This function creates a hatch brush, where the interior of the shape is filled with a pattern of diagonal, horizontal, or vertical lines. The iHatchStyle argument specifies the type of hatch pattern, and crColor specifies the color of the hatch lines. The spaces between the hatch lines are filled based on the current background mode and background color.

```
HBRUSH CreateHatchBrush(INT iHatchStyle, COLORREF crColor);
```

This function creates a hatch brush with the specified hatch style iHatchStyle and color crColor. The available hatch styles are:

- HS_HORIZONTAL: Horizontal lines
- HS_VERTICAL: Vertical lines
- HS_FDIAG: Upward diagonal lines
- HS_BDIAG: Downward diagonal lines
- HS_CROSS: Crosshatch pattern
- HS_DIAGCROSS: Crosshatch pattern at 45 degrees

The CreateHatchBrush function creates a hatch brush, where the interior of the shape is filled with a pattern of diagonal, horizontal, or vertical lines. The iHatchStyle argument specifies the type of hatch pattern, and crColor specifies the color of the hatch lines. The spaces between the hatch lines are filled based on the current background mode and background color.

CreatePatternBrush:

This function creates a pattern brush based on an 8-bit-per-pixel bitmap. The bitmap pattern is repeated to fill the interior of the shape.

```
HBRUSH CreatePatternBrush(HBITMAP hbm);
```

This function creates a pattern brush based on the specified monochrome bitmap hbm. The bitmap pattern is repeated to fill the interior of the shape.

The CreatePatternBrush function creates a pattern brush based on an 8-bit-per-pixel monochrome bitmap. The bitmap pattern is repeated to fill the interior of the shape. The hbm argument is the handle to the bitmap.

CreateDIBPatternBrushPt:

This function creates a pattern brush based on a 24-bit-per-pixel bitmap. The bitmap pattern is repeated to fill the interior of the shape.

```
HBRUSH CreateDIBPatternBrushPt(LPBYTE lpDIBPattern, DWORD dwMask, DWORD dwColor);
```

This function creates a pattern brush based on the specified 24-bit-per-pixel bitmap lpDIBPattern. The dwMask parameter specifies a mask that defines which areas of the bitmap are transparent. The dwColor parameter specifies the color to use for non-transparent areas.

CreateBrushIndirect:

This function creates a brush based on a LOGBRUSH structure, which provides detailed information about the brush's characteristics, including color, pattern style, and width.

```
HBRUSH CreateBrushIndirect(const LOGBRUSH *lpLogBrush);
```

This function creates a brush based on the specified LOGBRUSH structure, which provides detailed information about the brush's characteristics.

The CreateBrushIndirect function is the most versatile brush creation function, as it allows you to specify all of the brush's characteristics.

The CreateBrushIndirect function creates a brush based on a LOGBRUSH structure, which provides detailed information about the brush's characteristics. The lpLogBrush argument points to the LOGBRUSH structure.

The LOGBRUSH structure defines the properties of a logical brush. It includes the following fields:

- *lbStyle: Specifies the brush style, which can be BS_SOLID, BS_HOLLOW, BS_PATTERN, BS_HATCHED, BS_DIBPATTERN, or BS_DIBPATTERNPT.*
- *lbColor: Specifies the brush color.*
- *lbHatch: Specifies the hatch style for hatch brushes.*
- *lbReserved: Reserved for future use.*

Selecting the Brush

Once a brush is created, it is selected into the device context using the SelectObject function:

```
SelectObject(hdc, hBrush);
```

This makes the selected brush the current brush for filling shapes.

Brush Management

As with pens, logical brushes are GDI objects and require proper management to avoid memory leaks. Each brush that is created must be explicitly deleted using the DeleteObject function:

```
DeleteObject(hBrush);
```

However, a brush should not be deleted while it is still selected into the device context.

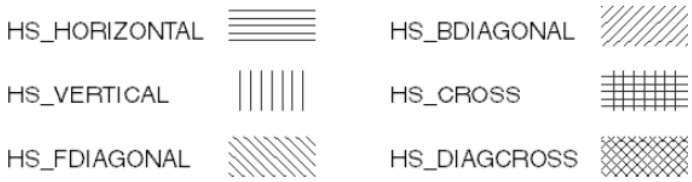


Figure 5–23. *The six hatch brush styles.*

To obtain information about a brush, use the GetObject function.

```
GetObject(hBrush, sizeof(LOGBRUSH), (LPVOID) &logbrush);
```

where logbrush is a LOGBRUSH structure.

Understanding the different brush types and their creation functions is essential for effective graphics programming in Windows. Remember to properly manage brush objects to avoid memory leaks.

GDI MAPPING MODE

The GDI mapping mode is a crucial aspect of graphics programming in Windows, as it determines how logical coordinates are translated into device coordinates (pixels) when drawing on the client area.

By default, all drawing occurs in units of pixels relative to the upper-left corner of the client area, but the mapping mode allows for more flexible coordinate systems.

Logical Units vs. Device Units

GDI drawing functions typically operate on logical units, which represent abstract coordinates that are not directly tied to the physical display.

The mapping mode translation process converts these logical units into device units, which are the actual pixel coordinates on the display.

This translation ensures that the drawn elements are scaled correctly and positioned appropriately regardless of the display resolution or scaling settings.

Mapping Mode Options

Windows provides eight mapping modes, each offering a different way to map logical coordinates to device coordinates.

These mapping modes are listed in the following table using the identifiers defined in WINGDI.H:

Mapping Mode	Description
MM_ANISOTROPIC	Allows for anisotropic scaling, where the x and y axes can be scaled independently.
MM_ISOTROPIC	Scales the x and y axes uniformly, preserving aspect ratios.
MM_HIMETRIC	Uses millimeters as logical units.
MM_LOMETRIC	Uses hundredths of a millimeter as logical units.
MM_TEXT	Uses characters as logical units, where each character width is considered one unit.
MM_TWIPS	Uses twips (1/1440 inch) as logical units.
MM_LOENGLISH	Uses tenths of an inch as logical units.
MM_HIENGLISH	Uses inches as logical units.

Mapping Mode Parameters

In addition to the mapping mode itself, several other parameters influence the mapping process:

Window Origin: Specifies the logical coordinates corresponding to the upper-left corner of the device context's clipping region.

Viewport Origin: Defines the device coordinates corresponding to the upper-left corner of the viewport, which is the portion of the clipping region that is actually displayed.

Window Extents: Represents the width and height of the logical coordinate system, measured in logical units.

Viewport Extents: Represents the width and height of the viewport, measured in device units.

Mapping Mode	Logical Unit	x-axis Orientation	y-axis Orientation
MM_TEXT	Characters	Right	Down
MM_LOMETRIC	Hundredths of a millimeter (0.1 mm)	Right	Up
MM_HIMETRIC	Tenths of a millimeter (0.01 mm)	Right	Up
MM_LOENGLISH	Tenths of an inch (0.01 in.)	Right	Up
MM_HIENGLISH	Hundredths of an inch (0.001 in.)	Right	Up
MM_TWIPS	Twip (1/1440 inch)	Right	Up
MM_ISOTROPIC	Arbitrary (equal scaling)	Selectable	Selectable
MM_ANISOTROPIC	Arbitrary (unequal scaling)	Selectable	Selectable

Mapping Mode Implications

The mapping mode also influences the orientation of the x and y axes. In most cases, the x-axis increases from left to right, and the y-axis increases from top to bottom.

However, certain mapping modes, such as MM_TEXT, may have different orientations depending on the system's text directionality settings.

Setting and Retrieving Mapping Mode

The SetMapMode function sets the mapping mode for a device context, while the GetMapMode function retrieves the current mapping mode.

The mapping mode determines how logical coordinates specified in GDI functions are translated into device coordinates (pixels) when drawing on the client area.

```
SetMapMode(hdc, iMapMode);
```

hdc: The handle to the device context.

iMapMode: The identifier of the mapping mode to set. The possible values are:

MM_TEXT: Logical units are characters.

MM_LOMETRIC: Logical units are hundredths of a millimeter.

MM_HIMETRIC: Logical units are tenths of a millimeter.

MM_LOENGLISH: Logical units are tenths of an inch.

MM_HIENGLISH: Logical units are hundredths of an inch.

MM_TWIPS: Logical units are twips (1/1440 inch).

MM_ISOTROPIC: Logical units are scaled equally in the x and y directions.

MM_ANISOTROPIC: Logical units can be scaled differently in the x and y directions.

Retrieving Mapping Mode

```
iMapMode = GetMapMode(hdc);
```

hdc: The handle to the device context.

iMapMode: Receives the current mapping mode identifier.

Default Mapping Mode

The default mapping mode is MM_TEXT, which means logical units are directly equivalent to physical pixels.

This simplifies drawing operations, as coordinates can be specified directly in pixel units.

Example TextOut Calls with Different Mapping Modes

MM_TEXT:

```
TextOut(hdc, 8, 16, TEXT("Hello"), 5);
```

The text starts 8 pixels from the left edge and 16 pixels from the top of the client area.

MM_LOENGLISH:

```
SetMapMode(hdc, MM_LOENGLISH);
TextOut(hdc, 50, -100, TEXT("Hello"), 5);
```

Set mapping mode to MM_LOENGLISH, where logical units are hundredths of an inch.

The text starts 0.5 inch from the left edge and 1 inch from the top of the client area. The negative sign in the y-coordinate indicates a downward direction.

Alternative Scaling Approaches

If you prefer working with pixel units, the default MM_TEXT mode is sufficient. However, if you need to display elements in specific dimensions like inches or millimeters, you can:

- Use GetDeviceCaps to obtain the necessary conversion factors.
- Perform your own scaling calculations to convert logical units to device units.
- Utilize alternative mapping modes that directly handle inch or millimeter measurements.

Coordinate Limitations in Windows 98 vs Windows 10|11

While GDI functions allow 32-bit coordinates, Windows 98 limits coordinates to 16 bits, ranging from -32,768 to 32,767.

This limitation affects functions that use coordinates for both starting and ending points, as well as the width and height of rectangles, which should also be within the 16-bit range.

In Windows 10 and 11, the coordinate limitations discussed for Windows 98 no longer apply. GDI functions now fully utilize 32-bit coordinates, allowing for a wider range of values and supporting larger graphics and drawing operations.

This enhancement eliminates the need for programmers to manually handle coordinate conversions or restrict drawing dimensions within the 16-bit range.

The ability to handle 32-bit coordinates provides several benefits:

Expanded Drawing Area: Developers can create graphics and user interfaces that span a broader area of the screen without encountering coordinate limitations.

High-Resolution Support: With 32-bit coordinates, GDI can effectively manage drawing on high-resolution displays, ensuring accurate positioning and scaling of graphical elements.

Simplified Programming: Programmers can focus on the design and implementation of their graphics without being constrained by coordinate limitations.

Future-Proof Development: This enhancement aligns with the increasing use of high-resolution displays and the demand for larger, more immersive graphics in various applications.

Device Coordinates and Logical Coordinates

In Windows graphics programming, device coordinates represent the physical pixels on the display, while logical coordinates are abstract units used to specify positions and sizes within the graphical environment.

Logical coordinates provide a more flexible and scalable way to define graphical elements, while device coordinates ensure accurate rendering on the specific display hardware.

Mapping Modes

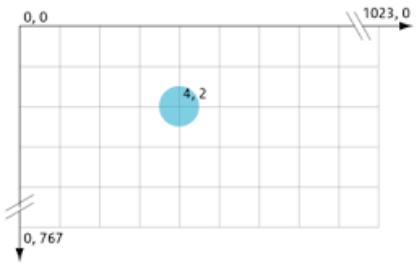
The mapping mode determines how logical coordinates are translated into device coordinates. Windows provides eight different mapping modes, each with its own characteristics.

The default mapping mode is MM_TEXT, which uses pixels as logical units. Other mapping modes allow for specifying coordinates in terms of inches, millimeters, or twips (1/1440 inch).

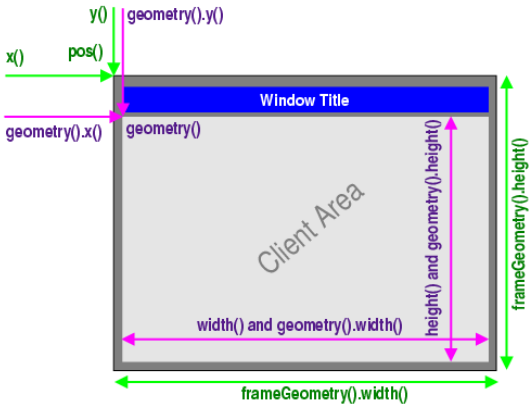
Device Coordinate Systems

Windows defines three device coordinate systems:

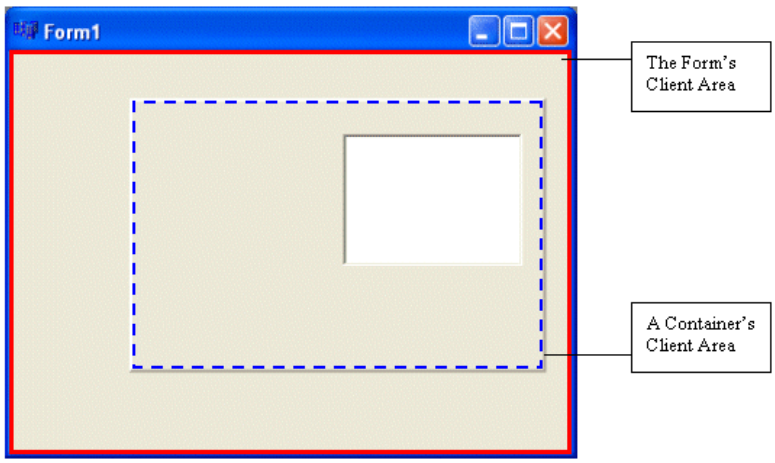
Screen Coordinates: These coordinates encompass the entire display area, with the upper-left corner being the origin (0, 0). Screen coordinates are used for functions like CreateWindow, MoveWindow, GetCursorPos, and GetWindowRect.



Whole-Window Coordinates: These coordinates represent the entire application window, including the title bar, menu, scroll bars, and border. The upper-left corner of the sizing border is the origin (0, 0). Whole-window coordinates are less commonly used but can be obtained through GetWindowDC.



Client-Area Coordinates: These coordinates are specific to the client area of the window, which is the area within the window's border that excludes the title bar, menu, scroll bars, and sizing border. The upper-left corner of the client area is the origin (0, 0). Client-area coordinates are used for most GDI drawing functions.



Coordinate Conversion Functions

Windows provides functions to convert between different device coordinate systems:

- ClientToScreen and ScreenToClient: These functions convert between client-area and screen coordinates.
- GetWindowRect: This function retrieves the position and size of the entire window in screen coordinates.

These functions allow programmers to seamlessly transition between different coordinate systems when necessary.

ViewPort and Window

In Windows graphics programming, the viewport and the window are crucial concepts related to the mapping of logical coordinates to device coordinates.

The viewport defines the area on the display where graphical elements will be rendered, while the window represents the logical coordinate space from which these elements are drawn.

Viewport

The viewport is a rectangular region within the device coordinate system (pixels) that determines the visible portion of the graphical output.

It is specified using the SetWindowExtEx and SetViewportExtEx functions, which define the extent of the viewport in both logical and device coordinates.

The origin of the viewport is typically located at the upper-left corner of the client area, but it can also be set to other positions using the SetWindowOrgEx function.

Window

The window, on the other hand, defines the logical coordinate space from which graphical elements are drawn.

It is specified using the SetWindowExtEx function, which sets the extent of the window in logical coordinates.

The origin of the window is typically located at the upper-left corner of the logical coordinate space, but it can also be set to other positions using the SetWindowOrgEx function.

Coordinate Transformation

When rendering graphical elements, Windows transforms logical coordinates from the window to device coordinates within the viewport. This transformation involves scaling and translation according to the following formulas:

```
xViewport = (xWindow - xWinOrg) * xViewExt / xWinExt + xViewOrg;
yViewport = (yWindow - yWinOrg) * yViewExt / yWinExt + yViewOrg;
```

$$x_{\text{Viewport}} = (x_{\text{Window}} - x_{\text{WinOrg}}) \times \frac{x_{\text{ViewExt}}}{x_{\text{WinExt}}} + x_{\text{ViewOrg}}$$

$$y_{\text{Viewport}} = (y_{\text{Window}} - y_{\text{WinOrg}}) \times \frac{y_{\text{ViewExt}}}{y_{\text{WinExt}}} + y_{\text{ViewOrg}}$$

where:

- (xWindow, yWindow) is the logical coordinate to be transformed.
- (xViewport, yViewport) is the corresponding device coordinate.
- (xWinOrg, yWinOrg) is the origin of the window in logical coordinates.
- (xViewOrg, yViewOrg) is the origin of the viewport in device coordinates.
- xViewExt and yViewExt are the extent of the viewport in device coordinates along the x and y axes, respectively.
- xWinExt and yWinExt are the extent of the window in logical coordinates along the x and y axes, respectively.

Code Implementation

The following code snippet demonstrates how to set the viewport and window using the SetWindowExtEx, SetViewportExtEx, and SetWindowOrgEx functions:

```
HDC hdc = GetDC(hWnd); // Get device context
RECT rc;
GetClientRect(hWnd, &rc); // Get client area rectangle

// Set viewport to client area
SetViewportExtEx(hdc, rc.right, rc.bottom, 0, 0);
SetWindowOrgEx(hdc, 0, 0, NULL);

// Set window extents to 100 units in both directions
SetWindowExtEx(hdc, 100, 100, NULL, NULL);
```

In this example, the viewport is set to the entire client area, and the window extents are set to 100 units in both directions. This means that one logical unit corresponds to one pixel on the display.

Viewport Coordinate Transformation Formulas

In Windows graphics programming, the viewport and the window play crucial roles in positioning and rendering graphical elements on the display.

The viewport defines the visible area on the screen where graphical elements will be shown, while the window represents the logical coordinate space from which these elements are drawn.

Coordinate Transformation

When rendering a graphical element, Windows transforms its logical coordinates (specified in the window) into device coordinates (pixels) within the viewport.

This transformation involves scaling and translation according to the following formulas:

For xViewport:

$$x_{\text{Viewport}} = \frac{(x_{\text{Window}} - x_{\text{WinOrg}}) \cdot x_{\text{ViewExt}}}{x_{\text{WinExt}}} + x_{\text{ViewOrg}}$$

where:

- xViewport: The x-coordinate in the viewport (device coordinates).
- xWindow: The x-coordinate in the logical window.
- xWinOrg: The x-coordinate of the logical window origin.
- xViewExt: The width of the viewport in device coordinates (pixels).
- xWinExt: The width of the logical window.
- xViewOrg: The x-coordinate of the viewport origin.

$$y_{\text{Viewport}} = \frac{(y_{\text{Window}} - y_{\text{WinOrg}}) \cdot y_{\text{ViewExt}}}{y_{\text{WinExt}}} + y_{\text{ViewOrg}}$$

where:

- yViewport: The y-coordinate in the viewport (device coordinates).
- yWindow: The y-coordinate in the logical window.
- yWinOrg: The y-coordinate of the logical window origin.
- yViewExt: The height of the viewport in device coordinates (pixels).
- yWinExt: The height of the logical window.

- yViewOrg: The y-coordinate of the viewport origin.

Window and Viewport Origins

The formulas utilize two points that specify an "origin" of the window and the viewport:

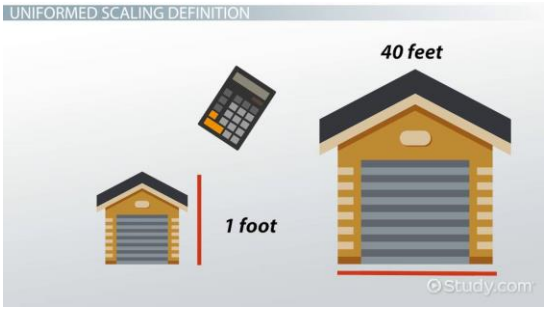
- (xWinOrg, yWinOrg): The window origin in logical coordinates.
- (xViewOrg, yViewOrg): The viewport origin in device coordinates.

By default, these origins are set to (0, 0), but they can be modified using the SetWindowOrgEx and SetViewportOrgEx functions. The formulas ensure that the logical point (xWinOrg, yWinOrg) is always mapped to the device point (xViewOrg, yViewOrg).

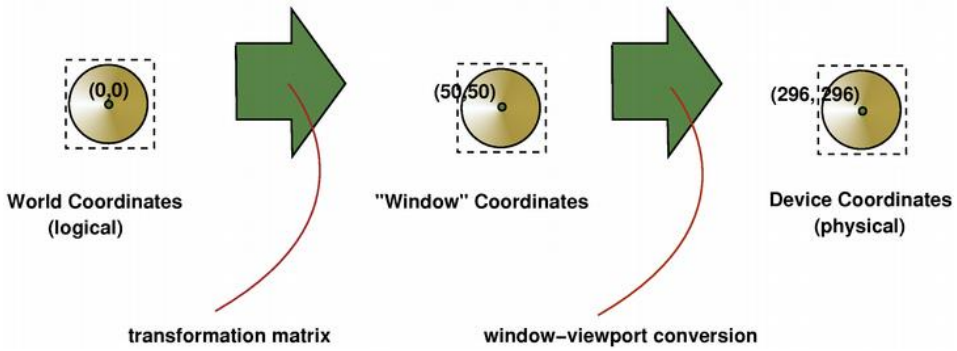
Scaling and Translation

The formulas perform two primary operations: scaling and translation.

Scaling: The scaling factor is determined by the ratio of the viewport extent to the window extent. This scaling ensures that the graphical elements are appropriately sized within the visible viewport area.



Translation: The subtraction of the window origin and the addition of the viewport origin effectively translate the logical coordinates from the window coordinate system to the viewport coordinate system, aligning the logical window with the visible viewport.



Window and Viewport Extents

The formulas also incorporate two points that specify "extents":

- (xWinExt, yWinExt): The window extent in logical coordinates.
- (xViewExt, yViewExt): The viewport extent in device coordinates.

In most mapping modes, the extents are determined by the mapping mode and cannot be directly changed.

Each extent alone has no significance, but the ratio of the viewport extent to the window extent serves as a scaling factor for converting logical units to device units.

Mapping Mode and Extents

For instance, when using the MM_LOENGLISH mapping mode, Windows sets xViewExt to a specific number of pixels and xWinExt to the length in hundredths of an inch occupied by xViewExt pixels.

This ratio provides the pixels per hundredths of an inch conversion factor. The scaling factors are expressed as ratios of integers rather than floating-point values for performance optimization.

Negative Extents

The extents can have negative values. This implies that values on the logical x-axis don't necessarily have to increase to the right, and values on the logical y-axis don't necessarily have to increase going down.

Inverse Transformation

Windows can also transform from viewport (device) coordinates to window (logical) coordinates:

For xWindow:

$$x_{\text{Window}} = \frac{(x_{\text{Viewport}} - x_{\text{ViewOrg}}) \cdot x_{\text{WinExt}}}{x_{\text{ViewExt}}} + x_{\text{WinOrg}}$$

This formula is essentially the inverse of the first one. It calculates the x-coordinate in the window space based on the x-coordinate in the viewport space.

This type of transformation is often used in computer graphics or windowing systems to convert coordinates between different coordinate spaces, such as transforming from a viewport space back to a window space.

For yWindow:

$$y_{\text{Window}} = \frac{(y_{\text{Viewport}} - y_{\text{ViewOrg}}) \cdot y_{\text{WinExt}}}{y_{\text{ViewExt}}} + y_{\text{WinOrg}}$$

This formula is analogous to the previous one. It calculates the y-coordinate in the window space based on the y-coordinate in the viewport space.

Like the x-coordinate transformation, it's a way to convert coordinates between different coordinate spaces, such as transforming from a viewport space back to a window space.

CO-ORDINATE CONVERSION FUNCTIONS

Windows provides two functions for converting between device points (pixels) and logical points (units defined by the mapping mode) in a program:

DPtoLP (Device Point to Logical Point)

The DPtoLP function converts an array of device coordinates (POINT structures) to logical coordinates. The syntax is as follows:

```
BOOL DPtoLP(HDC hdc, LPPOINT lpPoints, int cPoints);
```

Parameters:

- hdc: Handle to the device context containing the mapping mode and viewport and window extents.
- lpPoints: Pointer to an array of POINT structures containing the device coordinates to be converted.
- cPoints: Number of points in the array.
- Return value: TRUE if the conversion was successful, FALSE otherwise.

LPtoDP (Logical Point to Device Point)

The LPtoDP function converts an array of logical coordinates (POINT structures) to device coordinates (pixels). The syntax is as follows:

```
BOOL LPtoDP(HDC hdc, LPPOINT lpPoints, int cPoints);
```

Parameters:

- hdc: Handle to the device context containing the mapping mode and viewport and window extents.
- lpPoints: Pointer to an array of POINT structures containing the logical coordinates to be converted.
- cPoints: Number of points in the array.
- Return value: TRUE if the conversion was successful, FALSE otherwise.

Applications of Coordinate Conversion Functions

These coordinate conversion functions are useful for various purposes, including:

Converting client area dimensions to logical units: The DPtoLP function can be used to convert the client area dimensions obtained from GetClientRect (which are always in device units) to logical coordinates.

Scaling and positioning graphical elements: Both DPtoLP and LPtoDP functions can be used to scale and position graphical elements according to the mapping mode and viewport extents.

Converting mouse coordinates to logical units: The DPtoLP function can be used to convert mouse coordinates obtained from GetMessagePos or GetCursorPos (which are in device units) to logical units, allowing for precise mouse interaction with graphical elements.

In summary, the coordinate conversion functions DPtoLP and LPtoDP play a crucial role in translating between device coordinates and logical coordinates, enabling developers to effectively position and scale graphical elements within the Windows graphics programming environment.

WORKING WITH MM_TEXT

MM_TEXT Mapping Mode

The MM_TEXT mapping mode is a specialized mapping mode in Windows graphics programming designed for rendering text and other graphical elements without scaling.

In this mode, logical coordinates directly correspond to device coordinates (pixels), and no scaling is applied during coordinate conversion.

This makes MM_TEXT ideal for applications that require precise pixel-level control over their graphical output, such as text rendering and bitmap manipulation.

Default Origins and Extents

MM_TEXT mapping mode has the following default origins and extents:

Origin	Default Value	Changeable
Window Origin	(0, 0)	Yes
Viewport Origin	(0, 0)	Yes
Window Extent	(1, 1)	No
Viewport Extent	(1, 1)	No

The window origin and viewport origin can be modified using the SetWindowOrgEx and SetViewportOrgEx functions, respectively.

These functions essentially shift the coordinate axes, allowing for repositioning of the graphical elements within the display area. The window and viewport extents, however, cannot be changed in MM_TEXT mode.

Coordinate Transformation

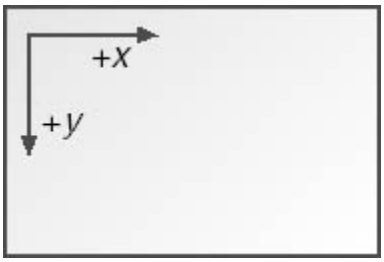
In MM_TEXT mapping mode, the formulas for converting between window coordinates and viewport coordinates simplify to:

```
xViewport = xWindow - xWinOrg + xViewOrg
yViewport = yWindow - yWinOrg + yViewOrg
```

These simplified formulas reflect the absence of scaling in MM_TEXT mode. The transformation solely involves adjusting the coordinates based on the window and viewport origins.

Text Considerations

The MM_TEXT mapping mode is not specifically designed for text rendering; rather, it is the orientation of the axes that aligns with the natural reading direction of text in most languages (left to right, top to bottom).



The axes in MM_TEXT mode increase in the same direction as text is read, making it a convenient choice for text-based applications.

Window and Viewport Origins

The SetWindowOrgEx and SetViewportOrgEx functions are primarily used to shift the coordinate axes and reposition graphical elements. Generally, only one of these functions is used at a time, as they have opposite effects.

SetWindowOrgEx: Shifting the window origin causes the logical point (xWinOrg, yWinOrg) to be mapped to the device point (0, 0), effectively moving the logical coordinate system relative to the display area.

SetViewportOrgEx: Shifting the viewport origin causes the logical point (0, 0) to be mapped to the device point (xViewOrg, yViewOrg), effectively moving the viewport within the client area.

Positioning the Logical Origin

The MM_TEXT mapping mode simplifies coordinate transformation by directly mapping logical coordinates to device coordinates without scaling.

However, this can sometimes make it challenging to position elements relative to specific points on the screen.

The SetViewportOrgEx and SetWindowOrgEx functions provide a way to adjust the logical origin, allowing you to reposition the coordinate system.

Centering the Client Area

In the first code example, SetViewportOrgEx is used to center the logical origin within the client area.

This effectively shifts the coordinate system so that the logical point (0, 0) corresponds to the center of the client area.

This can be useful for applications that need to display text or other elements around the center of the screen.

Positioning Text at the Upper Left Corner

Since the logical origin is now centered, displaying text at the upper left corner requires using negative coordinates.

The TextOut function takes the x and y coordinates of the starting point for the text,

so -cxClient/2 and -cyClient/2 are used to position the text at the upper left corner of the client area.

```
// Center the logical origin within the client area
SetViewportOrgEx(hdc, cxClient / 2, cyClient / 2, NULL);

// Display text at the upper left corner of the client area
TextOut(hdc, -cxClient / 2, -cyClient / 2, "Hello", 5);

// Alternative approach to centering the logical origin using SetWindowOrgEx
SetWindowOrgEx(hdc, -cxClient / 2, -cyClient / 2, NULL);

// Retrieve the current viewport origin
POINT viewportOrg;
GetViewportOrgEx(hdc, &viewportOrg);

// Retrieve the current window origin
POINT windowOrg;
GetWindowOrgEx(hdc, &windowOrg);
```

Alternative Approach Using SetWindowOrgEx

The second code example demonstrates an alternative approach to centering the logical origin using SetWindowOrgEx.

Instead of shifting the viewport, SetWindowOrgEx moves the window origin to the desired position.

This achieves the same effect as using SetViewportOrgEx, but it does so by adjusting the logical coordinate system rather than the viewport.

Using Both SetViewportOrgEx and SetWindowOrgEx

The provided warning against using both SetViewportOrgEx and SetWindowOrgEx together highlights the potential conflicts that can arise when modifying both the window and viewport origins simultaneously.

While each function has its specific purpose for adjusting the coordinate system, using them together can lead to unpredictable results.

Retrieving Current Origins

The GetViewportOrgEx and GetWindowOrgEx functions are used to obtain the current values of the viewport and window origins, respectively.

These functions return the origins in device coordinates for GetViewportOrgEx and logical coordinates for GetWindowOrgEx. This allows you to check the current origin values and make adjustments as needed.

Shifting Display Output with SetWindowOrgEx and SetViewportOrgEx

The SetWindowOrgEx and SetViewportOrgEx functions allow you to adjust the coordinate system by shifting either the window origin or the viewport origin.

This can be useful for repositioning the display output within the client area of your window, such as in response to scroll bar input or other user actions.

Example: Scrolling with Scroll Bar

The provided code snippet demonstrates two approaches to implementing scrolling behavior based on the vertical scroll bar position:

```
144     case WM_PAINT:
145         hdc = BeginPaint(hwnd, &ps);
146         SetWindowOrgEx(hdc, 0, cyChar * iVscrollPos);
147
148         for (i = 0; i < NUMLINES; i++)
149         {
150             y = cyChar * i;
151             // [display text]
152         }
153
154         EndPaint(hwnd, &ps);
155         return 0;
```

The provided code snippet and explanation discuss the adjustment of the window origin for display output within the client area of a window.

In this specific case, it shows how to shift the display output vertically in response to scroll bar input from the user.

In the context of a Windows message handling procedure (WndProc), particularly when handling the WM_PAINT message, this code demonstrates how to adjust the display output vertically based on the iVscrollPos value. Here's the breakdown:

BeginPaint:

Initiates the painting process and retrieves a device context (hdc) for the client area of the specified window (hwnd).

SetWindowOrgEx:

Adjusts the window origin in the device context (hdc) to shift the display vertically.

The cyChar * iVscrollPos represents the vertical offset based on the current position of the vertical scroll bar (iVscrollPos).

Loop for Display Output:

A loop (for loop) iterates through the lines to be displayed (NUMLINES times).

The variable y is calculated as cyChar * i, where cyChar is the height of a character cell and i is the loop index.

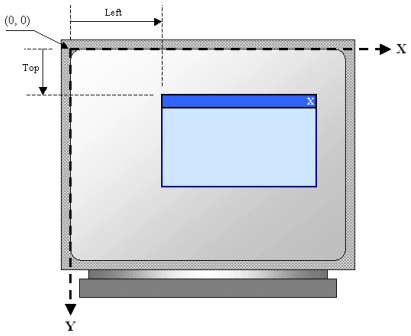
EndPaint:

Signals the end of the painting process and releases the device context.

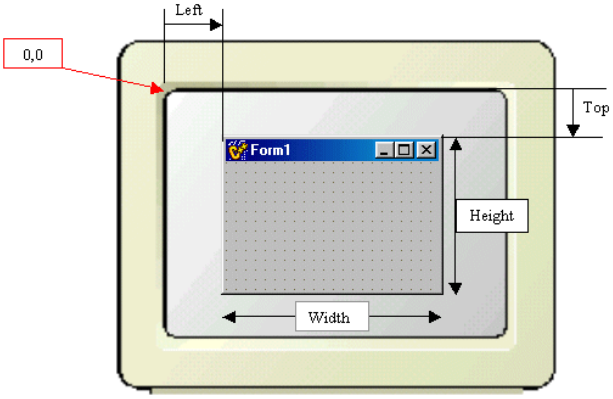
Returns 0 to indicate that the message has been processed.

This code snippet achieves the same scrolling effect using SetWindowOrgEx. It shifts the entire coordinate system based on iVscrollPos by modifying the window origin. This eliminates the need to adjust individual y-coordinates for each line.

Adjusting y-coordinates for each text line: This method involves modifying the y-coordinate for each line of text based on the scroll bar position. While it works effectively, it requires passing the scroll bar position to the text rendering function.



Shifting the window origin: Using SetWindowOrgEx, you can adjust the entire coordinate system based on the scroll bar position. This eliminates the need to pass the scroll bar position to the text rendering function, as the display is adjusted by modifying the window origin.



Logical Origin and Coordinate System Orientation

In MM_TEXT mapping mode, the y-axis increases as you move down the axis, which is an unconventional orientation compared to the standard Cartesian coordinate system where the y-axis increases as you move up the axis.

This difference in orientation can lead to unexpected results when positioning text or other graphical elements.

Alternative Mapping Modes

The next five mapping modes (MM_HIENGLISH, MM_LOENGLISH, MM_HIMETRIC, MM_LOMETRIC, and MM_TWIPS) adhere to the standard Cartesian coordinate system orientation, with the y-axis increasing as you move up the axis.

These modes may be more suitable for applications that require a consistent coordinate system orientation.

FIVE METRIC MAPPING MODES

Windows provides five mapping modes specifically designed for expressing logical coordinates in physical measurements, ensuring consistent graphical output regardless of device resolution. These mapping modes are:

- MM_LOENGLISH: This mode uses 0.01 inches as the logical unit, corresponding to approximately 0.254 millimeters.
- MM_LOMETRIC: This mode utilizes 0.1 millimeters as the logical unit, equivalent to approximately 0.00394 inches.
- MM_HIENGLISH: This mode employs 0.001 inches as the logical unit, corresponding to approximately 0.0254 millimeters.
- MM_TWIPS: This mode employs 1/1400 inches, also known as a twip, as the logical unit, corresponding to approximately 0.000694 inches or 0.0176 millimeters.
- MM_HIMETRIC: This mode employs 0.01 millimeters as the logical unit, equivalent to approximately 0.000394 inches or 0.01 millimeters.

Mapping Mode	Logical Unit	Inch	Millimeter
MM_LOENGLISH	0.01 in.	0.010	0.254
MM_LOMETRIC	0.1 mm.	0.0039	0.1
MM_HIENGLISH	0.001 in.	0.001	0.0254
MM_TWIPS	1/1400 in.	0.0007	0.0176
MM_HIMETRIC	0.01 mm.	0.0004	0.01

Default Origins and Extents

By default, the window and viewport origins are set to (0, 0) and cannot be changed. The window and viewport extents are initially undefined and are calculated based on the mapping mode and device resolution. These extents represent the physical dimensions of the logical unit in device coordinates.

Coordinate Transformation Formulas

For the metric mapping modes, the coordinate transformation formulas are slightly different from those used in the MM_TEXT mode. The viewport coordinates are calculated as follows:

1. **For x-coordinate in the viewport:**

$$x_{\text{Viewport}} = (x_{\text{Window}} - x_{\text{WinOrg}}) \times \frac{x_{\text{ViewExt}}}{x_{\text{WinExt}}} + x_{\text{ViewOrg}}$$

2. **For y-coordinate in the viewport:**

$$y_{\text{Viewport}} = (y_{\text{Window}} - y_{\text{WinOrg}}) \times \frac{y_{\text{ViewExt}}}{y_{\text{WinExt}}} + y_{\text{ViewOrg}}$$

These formulas take into account the extents of the window and viewport, ensuring that the logical coordinates are accurately mapped to their corresponding device coordinates.

Example: MM_LOENGLISH

In MM_LOENGLISH mode, the extents are calculated based on the number of horizontal and vertical pixels corresponding to 0.01 inches. For instance, if the display resolution is 1024x768 pixels, the extents would be:

$$\frac{x_{\text{ViewExt}}}{x_{\text{WinExt}}} = \frac{1024 \text{ pixels}}{0.01 \text{ inches}} = 102,400 \text{ pixels/inch}$$

$$\frac{-y_{\text{ViewExt}}}{y_{\text{WinExt}}} = \frac{768 \text{ pixels}}{0.01 \text{ inches}} = -76,800 \text{ pixels/inch}$$

Using these extents, the coordinate transformation formulas accurately map logical coordinates to device coordinates, ensuring consistent graphical output regardless of device resolution.

Windows 98 Approach

In Windows 98, the viewport and window extents are calculated based on the system font size and the default logical unit size for the chosen mapping mode.

For instance, if you have selected a 96 dpi system font, the viewport extents will be set to:

(96, 96) for all metric mapping modes.

This means that one logical unit corresponds to 96 pixels in both the x and y directions.

The window extents are then calculated based on the logical unit size.

For example, in MM_LOENGLISH, the logical unit is 0.01 inches, so the window extents are set to (100, -100), representing the physical dimensions of the display in logical units.

The negative value for the y-coordinate indicates that the y-axis increases as you move down the axis, which is consistent with the MM_TEXT mapping mode.

Windows NT Approach

Windows NT adopts a different approach, using the pixel dimensions of the screen and the assumed size of the display to determine the viewport and window extents.

The viewport extents are set to the actual pixel dimensions of the screen, obtained from the HORZRES and VERTRES indexes of GetDeviceCaps. For instance, if the display resolution is 1024x768, the viewport extents will be set to (1024, -768).

The window extents are based on the assumed size of the display, retrieved from the HORZSIZE and VERTSIZE indexes of GetDeviceCaps. These values are typically 320 and 240 millimeters, representing the physical dimensions of a standard monitor.

Comparing the Approaches

The Windows 98 approach ensures that the logical unit size is consistent across different devices, regardless of the display resolution. This can be useful for applications that require consistent scaling of graphical elements.

The Windows NT approach, on the other hand, takes into account the actual pixel dimensions of the screen, allowing for more precise placement of graphical elements. This can be beneficial for applications that require precise alignment with screen elements or for creating graphics that span the entire screen.

Mapping Mode	Viewport Extents (x, y)	Window Extents (x, y)
MM_LOMETRIC	(96, 96)	(254, -254)
MM_HIMETRIC	(96, 96)	(2540, -2540)
MM_LOENGLISH	(96, 96)	(100, -100)
MM_HIENGLISH	(96, 96)	(1000, -1000)
MM_TWIPS	(96, 96)	(1440, -1440)

Implications for Coordinate Transformation

- The different approaches used by Windows 98 and Windows NT affect the calculation of the viewport coordinates from the logical coordinates.
- The coordinate transformation formulas remain the same, but the extents used in the formulas will differ depending on the operating system and the mapping mode.
- In Windows 98, the extents represent the number of pixels corresponding to the logical unit size. For instance, in MM_LOENGLISH, the xViewExt/xWinExt ratio is 96 pixels/inch, indicating that one logical unit corresponds to 96 pixels.
- In Windows NT, the extents represent the physical dimensions of the display in logical units. For example, in MM_LOENGLISH, the xViewExt/xWinExt ratio is 1024/1260, indicating that one logical unit corresponds to approximately 0.81 pixels.
- Windows 10 and 11 follow a similar approach to Windows NT in setting the viewport and window extents for the metric mapping modes.

- They utilize the actual pixel dimensions of the display, obtained from the HORZRES and VERTRES indexes of GetDeviceCaps, to determine the viewport extents. The window extents, however, are calculated using a slightly different method.

- Instead of relying solely on the assumed size of the display, Windows 10 and 11 consider both the display resolution and the logical unit size to determine the window extents.

- This approach ensures that the physical dimensions of the display are accurately represented in logical units, maintaining consistency across different mapping modes and display resolutions.

Comparing their approaches:

Operating System	Approach	Viewport Extents	Window Extents
Windows 98	System font size and logical unit size	(96, 96)	Consistent across devices
Windows NT	Pixel dimensions of screen and assumed display size	Actual pixel dimensions	Physical dimensions in logical units
Windows 10 and 11	Pixel dimensions of screen and logical unit size	Actual pixel dimensions	Physical dimensions in logical units, considering display resolution

As you can see, the primary difference lies in how the window extents are calculated. Windows 98 prioritizes consistency across devices, while Windows NT and later versions focus on precise representation of the physical display dimensions.

Negative Y-Axis Orientation in Metric Mapping Modes

The five metric mapping modes (MM_LOENGLISH, MM_LOMETRIC, MM_HIENGLISH, MM_TWIPS, and MM_HIMETRIC) have an unconventional orientation of the y-axis: values increase as you move up the device.

This differs from the standard Cartesian coordinate system, where the y-axis increases as you move down the axis.

This peculiarity arises from how these mapping modes interpret the HORZRES and VERTRES values obtained from GetDeviceCaps.

These values represent the pixel dimensions of the screen, but the mapping modes treat them as the vertical and horizontal dimensions, respectively.

Default Window and Viewport Origins

The default window and viewport origins for all mapping modes, including metric mapping modes, are set to (0, 0).

This means that the logical (0, 0) point coincides with the pixel (0, 0) in the top-left corner of the device.

With the negative y-axis orientation, the default coordinate system for metric mapping modes appears inverted.

The y-axis increases as you move up the device, and the x-axis increases as you move to the right.

Consequences of Negative Y-Axis Orientation

This unconventional orientation can lead to unexpected behavior when drawing graphics.

For instance, attempting to display text using the default coordinate system would place it below the top edge of the client area.

To avoid such issues, it's necessary to adjust the coordinate system to align with the expectations of a standard Cartesian coordinate system.

This can be achieved using either SetViewportOrgEx or SetWindowOrgEx.

Adjusting Coordinate System with SetViewportOrgEx

SetViewportOrgEx allows you to shift the logical origin of the coordinate system, effectively moving the entire coordinate system within the client area.

By adjusting the y-coordinate argument, you can counteract the negative y-axis orientation and establish a standard coordinate system.

The two common approaches to adjusting the coordinate system using SetViewportOrgEx:

Shifting the Logical Origin to the Lower-Left Corner

To place the logical origin at the lower-left corner of the client area, you can use the following code:

```
SetViewportOrgEx(hdc, 0, cyClient, NULL);
```

Here, hdc is the device context, cyClient is the height of the client area in pixels, and NULL indicates that the viewport extents should remain unchanged.

With this adjustment, the coordinate system will appear as the upper right quadrant of a rectangular coordinate system.

This configuration is useful for applications that require consistent scaling and alignment with the client area's boundaries.

Shifting the Logical Origin to the Center of the Client Area

To center the logical origin within the client area, you can use the following code:

```
SetViewportOrgEx(hdc, cxClient / 2, cyClient / 2, NULL);
```

Here, cxClient is the width of the client area in pixels, and cyClient is the height of the client area in pixels. Again, NULL indicates that the viewport extents should remain unchanged.

This adjustment creates a true four-quadrant Cartesian coordinate system, where equal logical units represent equal physical distances on both the x and y axes.

This configuration is suitable for applications that require precise positioning and scaling relative to the center of the display.

Comparing the two approaches:

- Shifting the origin to the lower-left corner provides a consistent starting point for drawing graphics and aligns with the boundaries of the client area.
- Centering the origin offers a symmetrical coordinate system, simplifying positioning and scaling calculations.
- The choice between these approaches depends on the specific requirements of the application.

Adjusting Coordinate System with SetWindowOrgEx

While SetViewportOrgEx shifts the viewport's origin, SetWindowOrgEx shifts the window's origin. This means that the entire coordinate system, including both the viewport and window origins, is moved within the client area.

Using SetWindowOrgEx to adjust the coordinate system requires converting the desired logical origin (cxClient, cyClient) to device coordinates using the DPtoLP function. This is because SetWindowOrgEx takes its arguments in device coordinates.

Here's an example of how to set the logical origin to the center of the client area using SetWindowOrgEx:

```
POINT pt = {cxClient, cyClient};
DPtoLP(hdc, &pt, 1);
SetWindowOrgEx(hdc, -pt.x / 2, -pt.y / 2, NULL);
```

This code converts the logical point (cxClient, cyClient) to device coordinates using DPtoLP and then sets the window origin to the negative half of that point. This effectively centers the logical origin within the client area.

In summary, both SetViewportOrgEx and SetWindowOrgEx allow you to adjust the coordinate system to align with your application's requirements. SetViewportOrgEx is simpler to use, while SetWindowOrgEx offers more flexibility but requires additional calculations.

ROLL YOUR OWN MAPPING MODES

- *In the world of Windows programming, we have two mapping modes known as MM_ISOTROPIC and MM_ANISOTROPIC.*

- *These two modes are a bit special because they allow you to tweak how Windows interprets and scales coordinates.*

- *Firstly, let's tackle MM_ISOTROPIC. The term "isotropic" essentially means equal in all directions. So, in MM_ISOTROPIC, the x and y axes are scaled equally.*

- *This is handy when you want to create images that maintain their proper shape, regardless of the shape of the screen. If you've got a clock, for example, it will resize appropriately as you adjust the window size.*

- *Now, what makes MM_ISOTROPIC different from the other mapping modes we've talked about before is that it gives you control over the physical size of the logical unit.*

- *Logical units are what you use to draw things on the screen. With MM_ISOTROPIC, you can adjust these logical units based on the size of the window.*

- *So, your images can always fit nicely within the window, growing or shrinking as needed. On the other hand, we have MM_ANISOTROPIC.*

- *Here, the term "anisotropic" means not equal. In this mode, you can change the window and viewport extents freely.*

- *Unlike MM_ISOTROPIC, Windows won't automatically adjust these values to make x and y logical units represent the same physical dimensions.*

- *It's like an unconstrained playground for adjusting your mapping. In a nutshell, MM_ISOTROPIC is a bit flexible.*

- *You can adjust the size of logical units, and it's good for maintaining the aspect ratio of your images.*

- *Meanwhile, MM_ANISOTROPIC gives you full control over the extents, allowing for more freedom but without the automatic adjustments you get with MM_ISOTROPIC.*

So, when you're working with WINAPI and dealing with these mapping modes, MM_ISOTROPIC and MM_ANISOTROPIC let you play with how your drawings scale and fit within the window. It's like having different tools in your toolkit for handling different scenarios in your Windows programs. *Let's now write these notes in- depth:*

MM_ISOTROPIC and MM_ANISOTROPIC Mapping Modes

In addition to the standard mapping modes, there are two other mapping modes that provide more control over how logical coordinates are mapped to device coordinates: MM_ISOTROPIC and MM_ANISOTROPIC.

MM_ISOTROPIC

- MM_ISOTROPIC is a mapping mode that ensures that one logical unit corresponds to the same physical distance on both the x-axis and the y-axis.
- This means that objects drawn in MM_ISOTROPIC will always maintain their correct aspect ratio, regardless of the aspect ratio of the display device.
- To use MM_ISOTROPIC, you must call the SetWindowExtEx and SetViewportExtEx functions to specify the extents of the window and viewport.
- The extents are specified in logical units, and Windows will scale them so that one logical unit corresponds to the same physical distance on both the x-axis and the y-axis.

MM_ANISOTROPIC

- MM_ANISOTROPIC is a mapping mode that allows you to independently specify the scaling factors for the x-axis and the y-axis.
- This means that you can stretch or compress objects along either axis without affecting the other axis.
- To use MM_ANISOTROPIC, you must also call the SetWindowExtEx and SetViewportExtEx functions to specify the extents of the window and viewport.
- The extents are specified in logical units, and Windows will use the scaling factors that you specified to map logical units to device coordinates.

Comparison of MM_ISOTROPIC and MM_ANISOTROPIC

The following table summarizes the key differences between MM_ISOTROPIC and MM_ANISOTROPIC:

Feature	MM_ISOTROPIC	MM_ANISOTROPIC
Aspect ratio	Maintained	Can be stretched or compressed
Scaling factors	Automatically adjusted to maintain aspect ratio	Specified by the programmer

When to Use MM_ISOTROPIC and MM_ANISOTROPIC

- MM_ISOTROPIC is a good choice for applications that need to draw objects with a consistent aspect ratio, regardless of the display device. For example, MM_ISOTROPIC would be a good choice for drawing a circle or a square.

- MM_ANISOTROPIC is a good choice for applications that need to stretch or compress objects along one axis without affecting the other axis. For example, MM_ANISOTROPIC would be a good choice for drawing a horizontal or vertical line.

MM_ISOTROPIC Mapping Mode Overview

The MM_ISOTROPIC mapping mode is a specialized mapping mode in the Windows Graphics Device Interface (GDI) that ensures equal logical units on both the x and y axes.

This means that rectangles with equal logical widths and heights are displayed as squares, and ellipses with equal logical widths and heights are displayed as circles.

This mapping mode is particularly useful for applications that require precise scaling of graphical elements.

Key Features of MM_ISOTROPIC Mapping Mode

Preserves Equal Logical Units: The MM_ISOTROPIC mapping mode maintains the same physical distance for each logical unit on both the x and y axes, ensuring consistent scaling regardless of the device's aspect ratio.

Customizable Extents: Unlike other predefined mapping modes, MM_ISOTROPIC allows you to define the window and viewport extents using the SetWindowExtEx and

SetViewportExtEx functions. This flexibility enables you to control the scale and positioning of the logical window within the physical viewport.

Aspect Ratio Adjustment: Windows automatically adjusts the extents when using MM_ISOTROPIC to maintain equal logical units on both axes. This ensures that graphical elements are displayed correctly even on devices with varying aspect ratios.

Setting Up MM_ISOTROPIC Mapping Mode

To enable MM_ISOTROPIC mapping mode, you can use the SetMapMode function with the MM_ISOTROPIC flag.

Once the mapping mode is set, you can define the window and viewport extents using the SetWindowExtEx and SetViewportExtEx functions.

Example Usage

Consider a scenario where you want to create a one-quadrant virtual coordinate system with the origin (0, 0) at the lower-left corner of the client area and logical units ranging from 0 to 32,767. To achieve this, you would use the following code:

```
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 32767, 32767, NULL);
SetViewportExtEx(hdc, cxClient, -cyClient, NULL);
SetViewportOrgEx(hdc, 0, cyClient, NULL);
```

In this example, hdc is the handle to the device context, cxClient and cyClient are the width and height of the client area, respectively.

Additional Considerations

Window Extents vs. Viewport Extents: Window extents define the logical size of the virtual coordinate system, while viewport extents define the physical area on the device where the logical window is displayed.

- **Aspect Ratio Adjustment Implications:** When the client area is wider than it is high, Windows adjusts the x extents, potentially causing a portion of the client area to fall outside the logical window. Conversely, when the client area is higher than it is wide, Windows adjusts the y extents, potentially leaving a portion of the client area unused.
- **mm_Isotropic vs. mm_Anisotropic:** MM_ISOTROPIC maintains equal logical units on both axes, while MM_ANISOTROPIC allows independent scaling of x and y axes.

Going deeper….

How to Use MM_ISOTROPIC:

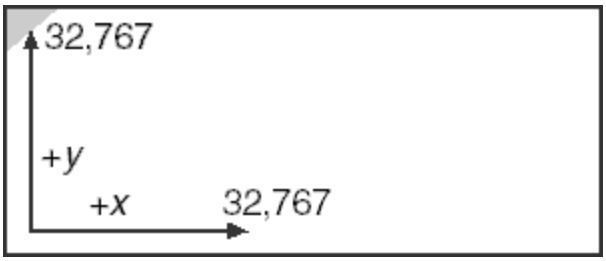
- Set the mapping mode to MM_ISOTROPIC using the SetMapMode function.
- Set the window extents using the SetWindowExtEx function. The window extents specify the size of the logical window in logical units.
- Set the viewport extents using the SetViewportExtEx function. The viewport extents specify the size of the client area in device units.
- Call the SetWindowOrgEx and SetViewportOrgEx functions to set the origins of the window and viewport.

Example of Using MM_ISOTROPIC:

Here is an example of how to use MM_ISOTROPIC to create a traditional one-quadrant virtual coordinate system where (0, 0) is at the lower left corner of the client area and the logical width and height ranges from 0 to 32,767:

```
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 32767, 32767, NULL);
SetViewportExtEx(hdc, cxClient, -cyClient, NULL);
SetViewportOrgEx(hdc, 0, cyClient, NULL);
```

- *When using MM_ISOTROPIC, Windows may adjust the window and viewport extents to maintain the aspect ratio of the display device.*
- *You should call SetWindowExtEx before calling SetViewportExtEx to make the most efficient use of space in the client area.*
- *If you need to stretch or compress objects along one axis without affecting the other axis, you should use the MM_ANISOTROPIC mapping mode.*



Program code can be found in the 7 … Chapter 5 folder called mm_isotrophic.c and mm_anisotrophic.c. The programs cover these two topics in depth.

After going through those, you can go back and understand the Book's code, chapter 5 whatsize program folder. Explanation:

Mapping Modes and Show Function:

The Show function takes care of displaying the dimensions of the window's client area in different metric mapping modes. It utilizes the SetMapMode function to set the desired mapping mode, and then it retrieves the client area dimensions using GetClientRect. The DPtoLP function is used to convert device coordinates to logical coordinates. The function then displays the information using TextOut.

WM_CREATE Handling:

The WM_CREATE case in the WndProc function is responsible for initializing the device context (hdc) and obtaining information about the system's fixed font using GetTextMetrics. This information, particularly the average character width (cxChar) and character height (cyChar), is crucial for later calculations and display.

Dynamic Text Drawing in WM_PAINT:

In the WM_PAINT case of the WndProc function, dynamic text is drawn to display information about the window's client area in various mapping modes. The TextOut function is used to output the heading and underline, and the Show function is called for each mapping mode to display the corresponding dimensions.

MM_ANISOTROPIC Setup:

Within the WM_PAINT case, the code sets up the MM_ANISOTROPIC mapping mode using SetMapMode and defines the logical and viewport extents with SetWindowExtEx and

SetViewportExtEx. This particular mapping mode allows for arbitrary scaling in both x and y directions.

Logical and Device Coordinates Conversion:

The DPtoLP function is used to convert device coordinates to logical coordinates. This is crucial for accurately representing the logical dimensions of the client area, especially when dealing with mapping modes that may have different units or scaling factors.

Text Drawing with Different Mapping Modes:

The code showcases how to use various metric mapping modes (MM_TEXT, MM_LOMETRIC, MM_HIMETRIC, MM_LOENGLISH, MM_HIENGLISH, MM_TWIPS) and dynamically display the corresponding dimensions of the client area. This requires an understanding of how mapping modes affect the interpretation of logical units.

The main points highlighted in the provided passage are:

Font Selection for Text Display:

The code in WHATSIZE ensures ease in displaying information by utilizing a fixed-pitch font. This is achieved by using the SelectObject function to set the current font of the device context (hdc) to the system's fixed-pitch font obtained through GetStockObject (SYSTEM_FIXED_FONT). A fixed-pitch font ensures that characters have a uniform width, simplifying the layout and alignment of displayed information.

Use of MM_ANISOTROPIC Mapping Mode:

WHATSIZE employs the MM_ANISOTROPIC mapping mode with logical units set to character dimensions. This mapping mode allows for arbitrary scaling in both x and y directions, and by setting logical units to character dimensions, the program can accurately represent and measure the dimensions of the client area in terms of characters.

Dynamic Mapping Mode Switching:

To obtain the size of the client area for one of the six mapping modes, WHATSIZE dynamically switches the mapping mode within its Show function. The steps involve saving the current device context (SaveDC), setting the desired mapping mode (SetMapMode), obtaining the client area coordinates (GetClientRect), converting them to logical coordinates using DPtoLP, and finally restoring the original mapping mode (RestoreDC). This process ensures accurate and consistent representation of dimensions across different mapping modes.

Presentation of Information:

The Show function in WHATSIZE is responsible for presenting information about the client area for each mapping mode. It dynamically adjusts the mapping mode, obtains the relevant dimensions, and then displays the information in a structured format.

Visual Output Example:

Figure below is referenced as a typical display from WHATSIZE, showcasing how the program visually presents information about the client area under different mapping modes. The display includes details such as the mapping mode, left, right, top, and bottom dimensions, providing a comprehensive overview of the window's characteristics.