DRAWING LINES AND DOTS

Chapter 5 Part 2

SetPixel Function

The SetPixel function is a fundamental building block for creating graphics in Windows GDI. It allows you to set the color of a specific pixel at a specified x and y coordinate. The syntax is as follows:

```
SetPixel(hdc, x, y, crColor);
```

hdc: A handle to the device context (DC) that represents the drawing surface.

x: The x-coordinate of the pixel to be set.

y: The y-coordinate of the pixel to be set.

crColor: A COLORREF value representing the desired color of the pixel.

When called, SetPixel sets the specified pixel to the given color.

If the specified color cannot be represented on the video display, the function sets the pixel to the nearest pure non-dithered color and returns that value.

GetPixel Function

The GetPixel function is another essential tool for working with pixels in Windows GDI.

It retrieves the color of a specific pixel at a specified x and y coordinate. The syntax is as follows:

```
crColor = GetPixel(hdc, x, y);
```

hdc: A handle to the DC that represents the drawing surface.

x: The x-coordinate of the pixel to retrieve the color from.

y: The y-coordinate of the pixel to retrieve the color from.

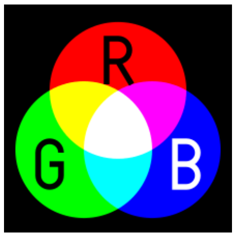
The GetPixel function returns a COLORREF value representing the color of the specified pixel.

Limitations of SetPixel and GetPixel

While SetPixel and GetPixel provide direct access to individual pixels, they are not commonly used for complex graphics operations. This is primarily due to performance considerations.

Performance Overhead: Drawing complex shapes using SetPixel involves calling the function repeatedly for each pixel, which can be inefficient. Higher-level GDI functions, such as LineTo and Polyline, are optimized for efficient line drawing and utilize specialized hardware acceleration when available.

Device-Dependent Colors: COLORREF values represent colors in a device-dependent manner. Using SetPixel and GetPixel directly can lead to color discrepancies between different display devices.



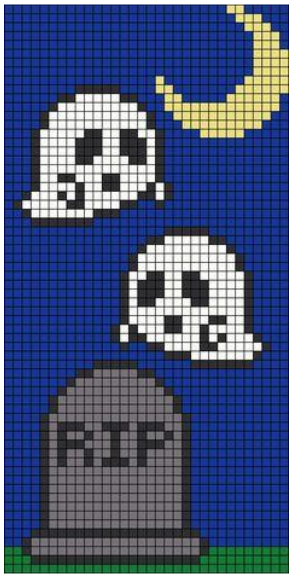
Alternative Graphics Approaches

In general, it is recommended to use higher-level GDI functions whenever possible.

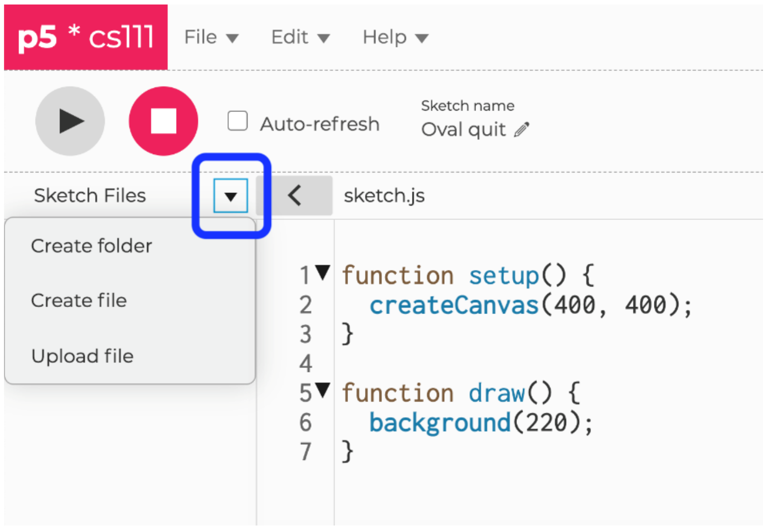
These functions provide better performance, hardware acceleration, and device-independent color handling.

For specialized cases where direct pixel manipulation is required, there are alternative approaches that offer more efficiency and flexibility.

Raster Operations: GDI provides raster operations (Rops) that allow for efficient manipulation of pixel patterns. These operations can be combined with SetPixel and GetPixel to achieve more complex graphics effects.



Custom Drawing Functions: Developers can create their own drawing functions that employ optimized algorithms and utilize hardware acceleration when available. This approach can be particularly beneficial for specialized graphics tasks.



```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
}
```

Line Drawing Functions in Windows GDI(we're here now… 5.2

Windows GDI provides a variety of functions for drawing straight lines. These functions offer different levels of flexibility and control over the line drawing process.

LineTo Function

The LineTo function draws a single straight line from the current pen position to the specified endpoint. The syntax is as follows:

```
LineTo(hdc, x, y);
```

hdc: A handle to the device context (DC) that represents the drawing surface.

x: The x-coordinate of the endpoint of the line.

y: The y-coordinate of the endpoint of the line.

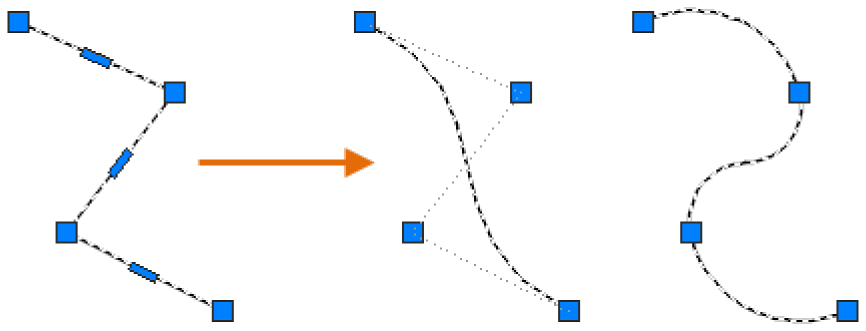
After calling LineTo, the current pen position is updated to the endpoint of the drawn line.

Polyline and PolylineTo Functions

Polyline provides a more efficient approach for drawing a series of connected lines.

It takes an array of POINT structures, each representing a vertex of the polyline, and the number of points in the array.

Polyline draws lines connecting the specified points, effectively creating a continuous polyline.



The Polyline and PolylineTo functions draw a series of connected straight lines. Polyline defines an open polyline, while PolylineTo defines a closed polyline. The syntax for both functions is as follows:

```
Polyline(hdc, lpPoints, cCount);
PolylineTo(hdc, lpPoints, cCount);
```

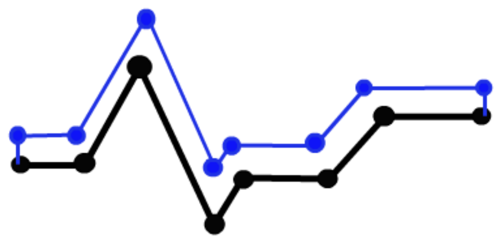
hdc: A handle to the DC that represents the drawing surface.

lpPoints: A pointer to an array of POINT structures, where each POINT structure specifies an x-coordinate and a y-coordinate for a vertex of the polyline.

cCount: The number of vertices in the polyline.

PolylineTo is similar to Polyline, but it utilizes the current position of the DC as the starting point of the polyline. It then draws lines connecting the remaining points in the provided array, updating the current position to the endpoint of the last line drawn.

PolyPolyline Function



The PolyPolyline function draws multiple polylines. The syntax is as follows:

```
PolyPolyline(hdc, polyPoints, nCount);
```

hdc: A handle to the DC that represents the drawing surface.

polyPoints: A pointer to an array of POLYLINE structures, where each POLYLINE structure specifies a polyline using its lpPoints member and its cCount member.

nCount: The number of polylines in the array.

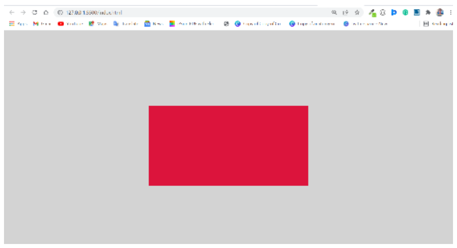
Factors Affecting Line Appearance

Five device context attributes influence the appearance of lines drawn using these functions:

Current pen position: This attribute applies to LineTo, PolylineTo, PolyBezierTo, and ArcTo functions. It determines the starting point of the line.

Pen: The pen defines the style and attributes of the line, including its width, color, and pattern.

Background mode: This attribute determines how the background color is handled when drawing lines. OPAQUE mode fills the background with the specified color, while TRANSPARENT mode allows the underlying background to show through.



Background color: This attribute specifies the color used to fill the background in OPAQUE mode.

Drawing mode: This attribute determines how source pixels are combined with destination pixels when drawing lines. It affects how the line color is blended with the background color.

DRAWING STRAIGHT LINES WITH MOVETOEX AND LINETO

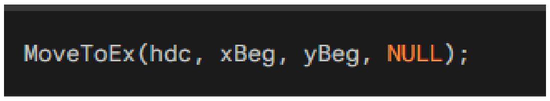
Drawing straight lines in Windows GDI involves two primary functions: MoveToEx and LineTo.

These functions work together to define the starting point and endpoint of the line.

MoveToEx Function

The MoveToEx function sets the current position of the device context (DC).

This position serves as the starting point for subsequent drawing operations, including line drawing. The syntax is as follows:



```
MoveToEx(hdc, xBeg, yBeg, NULL);
```

hdc: A handle to the DC that represents the drawing surface.

xBeg: The x-coordinate of the starting point.

yBeg: The y-coordinate of the starting point.

NULL: A placeholder for the previous current position value, which is not used in most cases.

The MoveToEx function doesn't actually draw anything; it simply updates the DC's current position.

This position remains the starting point until it is explicitly changed by another MoveToEx call or by certain other GDI functions.

MoveToEx and LineTo work together to draw individual lines.

MoveToEx sets the current position of the device context (DC), which defines the starting point of the line.

LineTo then draws a straight line from the current position to the specified endpoint. The current position is updated to the endpoint after drawing the line.

LineTo Function

The LineTo function draws a straight line from the current position of the DC to the specified endpoint. The syntax is as follows:

```
LineTo(hdc, xEnd, yEnd);
```

hdc: A handle to the DC that represents the drawing surface.

xEnd: The x-coordinate of the endpoint.

yEnd: The y-coordinate of the endpoint.

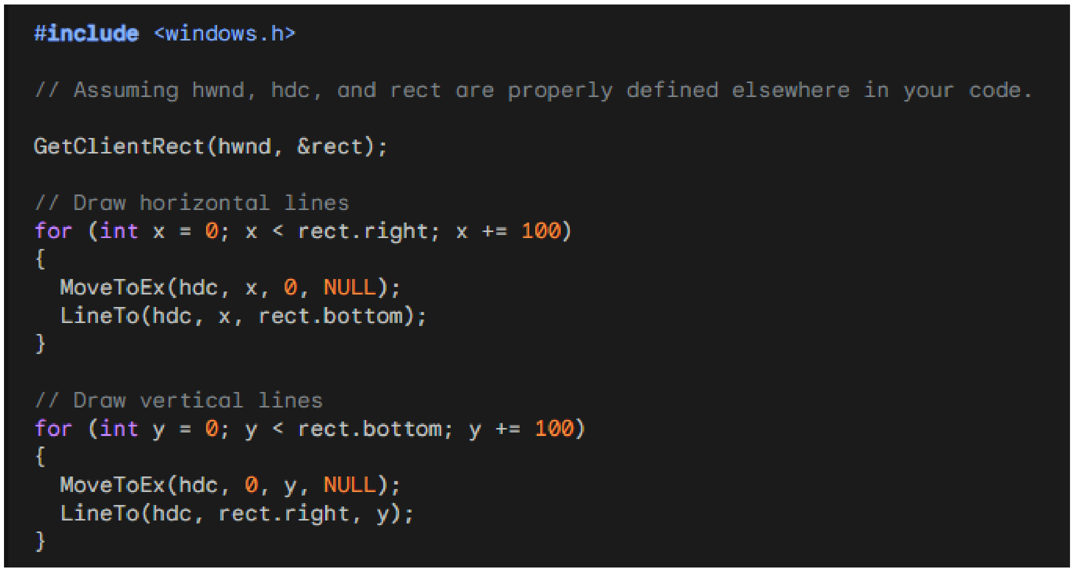
The LineTo function draws a line from the current position, which was previously set by the MoveToEx call, to the specified endpoint.

Once the line is drawn, the current position is updated to the endpoint.

Drawing a Grid with MoveToEx and LineTo

The provided code demonstrates how to draw a grid in the client area of a window using MoveToEx and LineTo.

It iterates through the client area coordinates, alternately calling MoveToEx to set the current position and then calling LineTo to draw a line.



```
#include <windows.h>

// Assuming hwnd, hdc, and rect are properly defined elsewhere in your code.

GetClientRect(hwnd, &rect);

// Draw horizontal lines
for (int x = 0; x < rect.right; x += 100)
{
  MoveToEx(hdc, x, 0, NULL);
  LineTo(hdc, x, rect.bottom);
}

// Draw vertical lines
for (int y = 0; y < rect.bottom; y += 100)
{
  MoveToEx(hdc, 0, y, NULL);
  LineTo(hdc, rect.right, y);
}
```

This code snippet effectively draws a series of vertical and horizontal lines spaced 100 pixels apart within the client area of the window.

Windows 10 and 11 both use the full 32-bit values for coordinates, providing a wider range of representable positions within the drawing surface. This allows for more precise positioning of graphical elements and enables the creation of larger and more detailed graphics.

With the transition to 32-bit and later 64-bit operating systems, the coordinate range has expanded significantly, providing greater flexibility for graphics programming.

This broader range of coordinates is particularly beneficial for applications that require precise positioning of graphical elements, such as CAD software, architectural design tools, and high-resolution image editing programs.

It also allows for the creation of larger and more complex graphics without encountering limitations due to coordinate values.

Both Windows 10 and 11 fully utilize the 32-bit range for coordinate values, eliminating the limitations faced in earlier versions of the operating system. This provides a more versatile and expansive canvas for graphics applications.

Retrieving the Current Position

If you need to retrieve the current position of the DC, you can use the GetCurrentPositionEx function:

```
GetCurrentPositionEx(hdc, &pt);
```

This function stores the current position in the provided POINT structure.

Choosing the Right Function

The choice between MoveToEx, LineTo, Polyline, and PolylineTo depends on the specific drawing requirements:

Individual Lines: Use MoveToEx followed by LineTo for drawing a single line.

Series of Connected Lines: Use Polyline for drawing a sequence of connected lines, especially when dealing with a large number of points.

Starting from Current Position: Use PolylineTo to draw a polyline starting from the current position of the DC.

Application in Drawing a Sine Wave

The SINEWAVE program demonstrates the use of Polyline to draw a sine wave.

It calculates a series of points representing the sine function and then invokes Polyline to connect these points, creating the smooth curve of the sine wave.

Code found in chapter 5 (sinwave folder)

Preprocessor Directives and Constants

The code begins by including the necessary header files: windows.h for Windows API functions and math.h for mathematical operations.

It then defines two constants: NUM, which represents the number of points in the sine wave (1000 in this case), and TWOPI, which represents twice the value of pi (used for calculating sine wave values).

WinMain Function

The WinMain function serves as the entry point for the application. It performs the following tasks:

Registers the Window Class: It defines the window class properties using WNDCLASS structure, including its name, style, and associated functions. This class determines the behavior and appearance of the application's window.

Creates the Window: It creates the application window using the CreateWindow function, specifying its name, class, initial position and size, and instance handle.

Shows the Window: It displays the created window using the ShowWindow function, making it visible to the user.

Updates the Window: It updates the window's contents using UpdateWindow, ensuring the sine wave is drawn correctly.

Message Loop: It enters a message loop using GetMessage and DispatchMessage functions. This loop continuously processes messages from the system and directs them to the appropriate handler functions.

Returns Exit Code: Finally, it returns the exit code from the message loop, indicating the application's termination status.

WndProc Function

The WndProc function handles various messages sent to the application window. It processes the following messages:

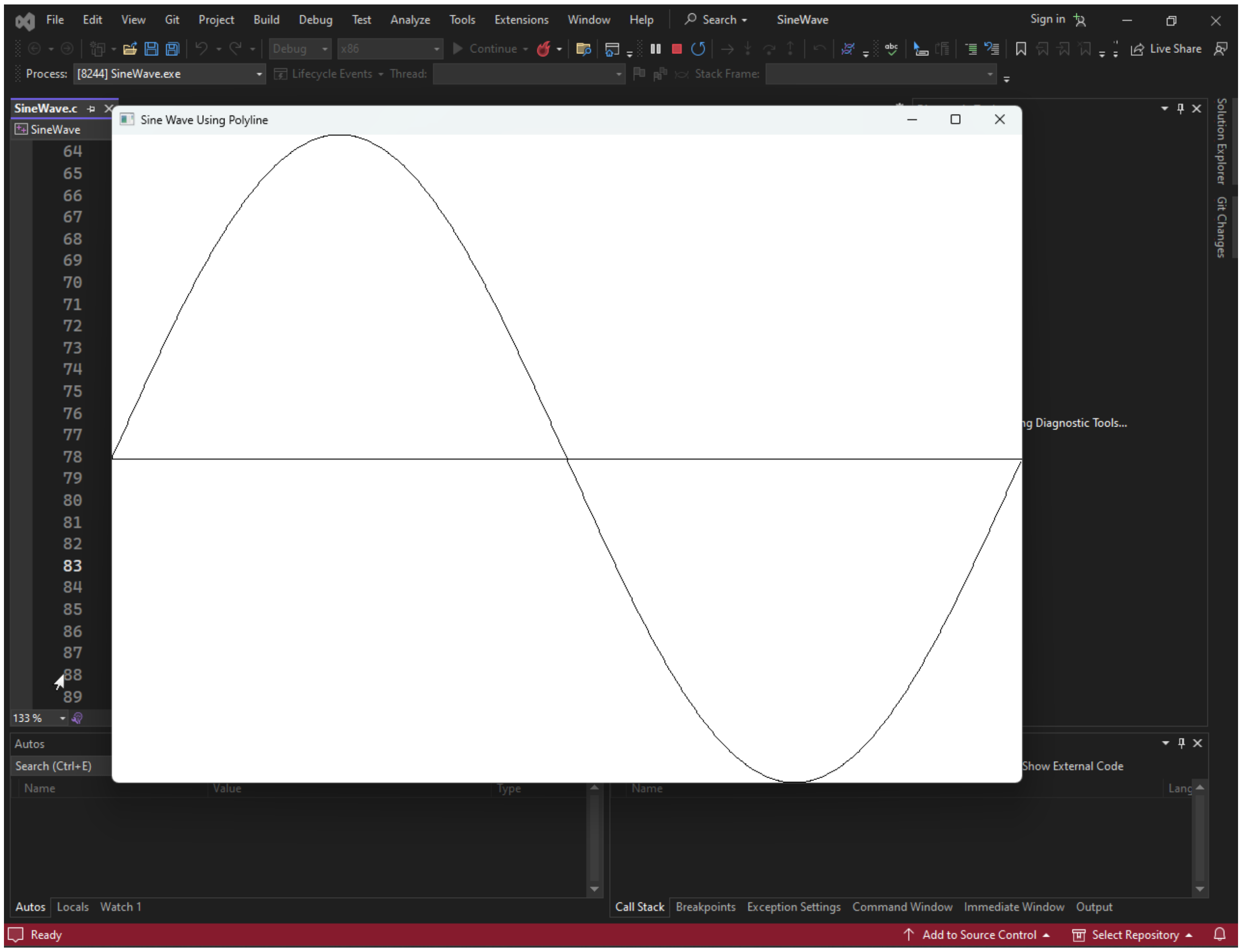
WM_SIZE: This message is received when the window's size changes. The function stores the new client area width and height in cxClient and cyClient variables for later use.

WM_PAINT: This message is received when the window needs to be repainted. The function performs the following tasks:

- **Obtains Device Context:** It retrieves the device context (DC) using BeginPaint.
- **Draws Axis Line:** It draws a horizontal line across the middle of the client area using MoveToEx and LineTo functions.
- **Calculates Sine Wave Values:** It iterates over the NUM points, calculating the sine wave value (scaled to fit the client area's height) for each point.
- **Sets Point Coordinates:** For each point, it sets the x-coordinate to its corresponding position along the client area's width and the y-coordinate to the calculated sine wave value.
- **Draws Sine Wave:** It draws the entire sine wave using a single Polyline call, providing the array of POINT structures containing the calculated coordinates.

WM_DESTROY: This message is received when the window is destroyed. The function posts a quit message to terminate the application.

Default Window Procedure: For any unhandled messages, it calls the DefWindowProc function, allowing the default message handling mechanism to take over.



Conclusion

The provided code effectively demonstrates the use of Polyline to draw a smooth sine wave within the client area of a window.

It highlights the efficiency of Polyline compared to calling LineTo multiple times and showcases the application of mathematical functions in graphics programming.

THE RECTANGLE FUNCTION

The Rectangle function is a fundamental graphics drawing tool in Windows GDI.

It draws a solid rectangular shape within a specified bounding box. The function's syntax is as follows:

```
Rectangle(hdc, xLeft, yTop, xRight, yBottom);
```

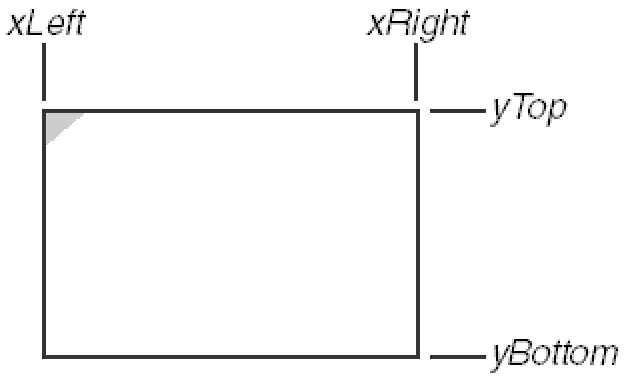
hdc: A handle to the device context (DC) representing the drawing surface.

xLeft: The x-coordinate of the upper left corner of the bounding box.

yTop: The y-coordinate of the upper left corner of the bounding box.

xRight: The x-coordinate of the lower right corner of the bounding box.

yBottom: The y-coordinate of the lower right corner of the bounding box.

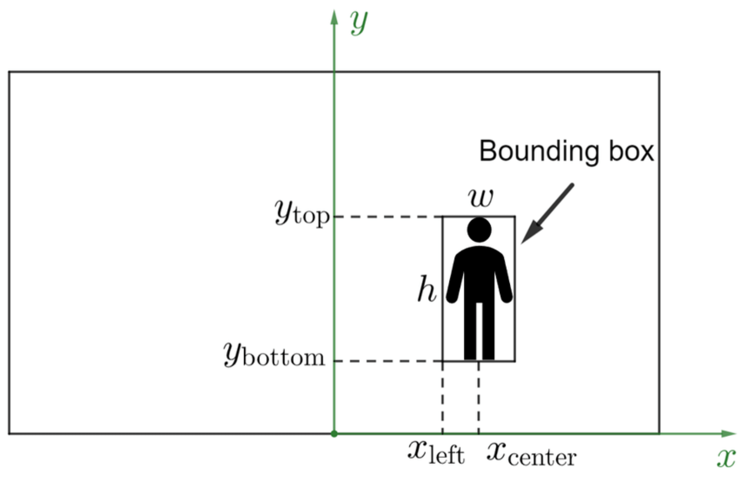


Bounding Box Concept

The bounding box is a rectangular area that defines the extent of a graphical object.

It serves as a reference for positioning and sizing the object within the drawing surface.

In the case of the Rectangle function, the bounding box represents the exact dimensions of the rectangle to be drawn.



Rectangle Function Behavior

The Rectangle function draws a solid rectangle within the specified bounding box, filling the enclosed area with the current area-filling brush.

The default brush is solid white, making it appear as a filled rectangle.

Avoiding Off-by-One Errors

Windows GDI utilizes the "up to (but not including)" approach for drawing objects using the bounding box coordinates.

This means that the specified coordinates define the outer limits of the object, not the points that are actually drawn.

To avoid off-by-one errors, it's crucial to consider this convention when defining the bounding box coordinates.

Ensure that the difference between the specified coordinates represents the desired width and height of the object, not one pixel greater.

Example Code

The following code snippet demonstrates the use of the Rectangle function to draw a rectangle:

```
HDC hdc = GetDC(hwnd);
Rectangle(hdc, 10, 20, 100, 150);
ReleaseDC(hwnd, hdc);
```

This code draws a solid rectangle with its upper left corner at (10, 20) and its lower right corner at (100, 150).

The rectangle will be filled with the current area-filling brush, which is solid white by default.

Off-by-One Errors in Graphics Programming

Off-by-one errors are common pitfalls in graphics programming.

They arise from inconsistencies in how different graphics systems interpret and handle coordinate values.

Some systems draw objects to encompass the specified right and bottom coordinates, while others draw up to but not including those coordinates.

Windows GDI adopts the latter approach, meaning that the specified coordinates define the outer limits of the object, not the points that are actually drawn.

This convention can lead to off-by-one errors if programmers are not careful in their calculations.

Bounding Box and Coordinate Interpretation

To avoid these errors, it's crucial to understand the concept of the bounding box.

A bounding box is an imaginary rectangular area that defines the extent of a graphical object.

It serves as a reference for positioning and sizing the object within the drawing surface.

In the context of Windows GDI and the Rectangle, Ellipse, and RoundRect functions, the bounding box determines the exact dimensions and placement of the respective shapes.

The specified coordinates represent the corners of the bounding box, and the object is drawn within this enclosed area.

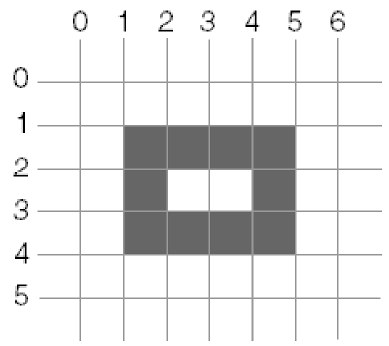
💥💥💥

Rectangle Function and Off-by-One Prevention

Consider the Rectangle function call:

```
Rectangle(hdc, 1, 1, 5, 4);
```

Would be drawn as:



The rectangle itself would occupy the inner area, leaving a one-pixel gap between the rectangle and the edges of the bounding box.

This gap is due to the "up to (but not including)" approach employed by Windows GDI.

To prevent off-by-one errors, programmers should ensure that the difference between the specified coordinates represents the desired width and height of the object, not one pixel greater.

In other words, the coordinates should define the outer edges of the object, not the points that should be drawn.

Ellipse and RoundRect Functions

The Ellipse and RoundRect functions follow the same principles as the Rectangle function.

They utilize the bounding box concept and the "up to (but not including)" approach to draw ellipses and rounded rectangles within specified areas.

The Rectangle function draws a solid rectangle within a specified bounding box.

```
Rectangle(hdc, xLeft, yTop, xRight, yBottom);
```

```
HDC hdc = GetDC(hwnd);
Rectangle(hdc, 10, 20, 100, 150);
ReleaseDC(hwnd, hdc);
```

This code draws a solid rectangle with its upper left corner at (10, 20) and its lower right corner at (100, 150).

The Ellipse function draws an ellipse within a specified bounding box.

```
Ellipse(hdc, xLeft, yTop, xRight, yBottom);
```

```
HDC hdc = GetDC(hwnd);
Ellipse(hdc, 50, 50, 150, 100);
ReleaseDC(hwnd, hdc);
```

This code draws an ellipse with its center at (100, 75) and its horizontal radius of 50 pixels and vertical radius of 25 pixels.

The RoundRect function draws a rectangle with rounded corners within a specified bounding box.

```
RoundRect(hdc, xLeft, yTop, xRight, yBottom, xCornerRadius, yCornerRadius);
```

```
HDC hdc = GetDC(hwnd);
RoundRect(hdc, 30, 30, 120, 90, 15, 20);
ReleaseDC(hwnd, hdc);
```

This code draws a rounded rectangle with its upper left corner at (30, 30), its lower right corner at (120, 90), and corner radius of 15 pixels horizontally and 20 pixels vertically.

Fun Notes to Read

Yes, when it comes to creating graphical applications in C, especially with the Windows API, you often deal with lower-level concepts and have more manual control over the drawing process. In C, you work with device contexts, pixels, and lower-level drawing functions.

For example, when using the Windows API in C to draw on a window, you might deal with concepts like device contexts (HDC), which represent a drawing surface, and use functions like Rectangle, Ellipse, and RoundRect to draw basic shapes. You might also handle bitmaps directly by creating, modifying, and displaying them manually.

Here's why:

Procedural Nature: C is a procedural programming language, and when you work with graphical programming in C, you often directly call functions that correspond to graphical operations. This can provide more control but might also require more manual management.

Direct Memory Manipulation: In C, you have more direct access to memory, which means you can manipulate data structures and perform operations at a lower level. This is evident when dealing with bitmaps or other pixel-based graphics.

Windows API: When programming in C for Windows, you often use the Windows API, which exposes functions for interacting with the operating system and creating graphical user interfaces. This API is designed to be used with the C programming language.

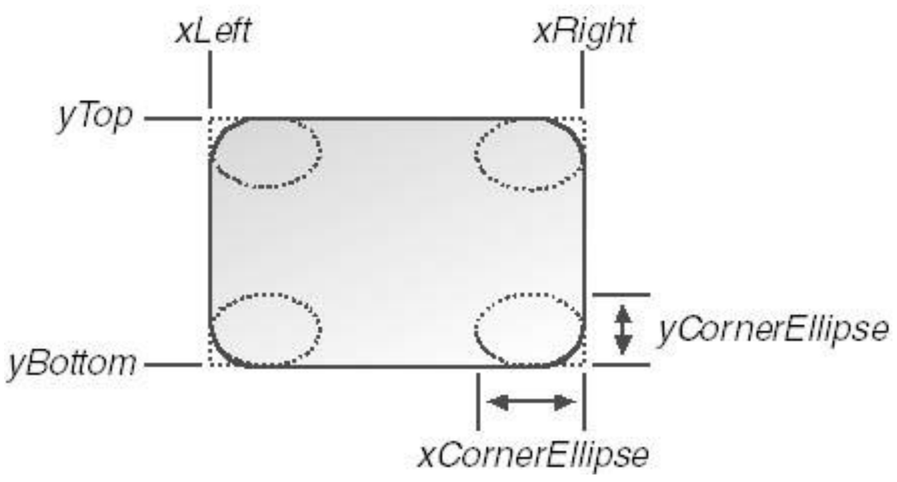
On the other hand, higher-level languages like C# (especially with Windows Forms, WPF, or UWP) abstract away many of these low-level details. They provide more intuitive, object-oriented frameworks for building graphical applications, making it easier to work with graphical elements without having to deal with the nitty-gritty details of device contexts and manual memory manipulation.

In summary, while C provides more control and lower-level access to system resources, it also requires more manual management, especially when working with graphics. Higher-level languages like C# abstract away many of these details, allowing for more rapid development of graphical applications. The choice between them often depends on the specific requirements and the level of control you need.

RoundRect Function

The RoundRect function draws a rectangle with rounded corners within a specified bounding box. It utilizes a small ellipse to define the curvature of the rounded corners.

The width of this ellipse is represented by xCornerEllipse, and the height is represented by yCornerEllipse.



The default approach to calculating the corner ellipse dimensions uses the formulas:

```
xCornerEllipse = (xRight - xLeft) / 4;
yCornerEllipse = (yBottom - yTop) / 4;
```

This method, while straightforward, may result in uneven rounding due to the difference in the rectangle's dimensions.

To achieve more consistent rounding, it's recommended to set xCornerEllipse equal to yCornerEllipse in real dimensions.

Arc, Chord, and Pie Functions

The Arc, Chord, and Pie functions all share the same arguments and are used to draw various elliptical shapes:

Arc: Draws an elliptical arc from a starting point (xStart, yStart) to an ending point (xEnd, yEnd).

Chord: Draws an elliptical arc from a starting point (xStart, yStart) to an ending point (xEnd, yEnd), filling the enclosed area with the current area-filling brush.

Pie: Draws an elliptical pie slice from a starting point (xStart, yStart) to an ending point (xEnd, yEnd), filling the enclosed area with the current area-filling brush.

💥💥💥💥💥💥💥💥

Arc Drawing Mechanism

For the Arc function, Windows draws an arc in a counterclockwise direction around the circumference of the ellipse, starting at the point where an imaginary line connecting (xStart, yStart) intersects the ellipse and ending at the point where an imaginary line connecting (xEnd, yEnd) intersects the ellipse.

Chord and Pie Drawing Mechanism

For the Chord and Pie functions, the drawing process is similar to the Arc function, but the enclosed area is filled with the current area-filling brush.

In the case of Chord, the area between the arc and the connecting lines is filled. For Pie, the entire pie slice is filled.

```
// Draw an arc on the device context (hdc)
Arc(hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd);

// Draw a chord on the device context (hdc)
Chord(hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd);

// Draw a pie slice on the device context (hdc)
Pie(hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd);
```

The Arc function draws an elliptical arc within a specified bounding box. It takes the following arguments:

- **hdc:** A handle to the device context (DC) representing the drawing surface.
- **xLeft, yTop, xRight, yBottom:** The coordinates of the bounding box that encloses the ellipse.
- **xStart, yStart, xEnd, yEnd:** The coordinates of the starting and ending points of the arc.

The Arc function draws an arc in a counterclockwise direction around the circumference of the ellipse, starting at the point where an imaginary line connecting (xStart, yStart) intersects the ellipse and ending at the point where an imaginary line connecting (xEnd, yEnd) intersects the ellipse.

The Chord function is similar to the Arc function, but it also fills the enclosed area between the arc and the connecting lines with the current area-filling brush. It takes the same arguments as the Arc function:

- **hdc:** A handle to the device context (DC) representing the drawing surface.
- **xLeft, yTop, xRight, yBottom:** The coordinates of the bounding box that encloses the ellipse.

- **xStart, yStart, xEnd, yEnd:** The coordinates of the starting and ending points of the chord.

The Pie function draws an elliptical pie slice within a specified bounding box. It also fills the enclosed area with the current area-filling brush. It takes the same arguments as the Arc and Chord functions:

- **hdc:** A handle to the device context (DC) representing the drawing surface.
- **xLeft, yTop, xRight, yBottom:** The coordinates of the bounding box that encloses the ellipse.
- **xStart, yStart, xEnd, yEnd:** The coordinates of the starting and ending points of the pie slice.

```
#include <windows.h>
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            // Draw a rectangle | Coordinates for the rectangle
            Rectangle(hdc, 50, 50, 150, 100);
            // Draw an ellipse | Coordinates for the ellipse
            Ellipse(hdc, 100, 150, 200, 250);
            // Draw a rounded rectangle | Coordinates and corner radii for the rounded rectangle
            RoundRect(hdc, 250, 50, 350, 100, 20, 20);
            // Draw an arc |  Coordinates, starting point, and ending point for the arc
            Arc(hdc, 400, 50, 500, 100, 425, 75, 475, 75);
            // Draw a chord | Coordinates, starting point, and ending point for the chord
            Chord(hdc, 400, 150, 500, 200, 425, 175, 475, 175);
            // Draw a pie slice | Coordinates, starting point, and ending point for the pie slice
            Pie(hdc, 400, 250, 500, 300, 425, 275, 475, 275);
            EndPaint(hwnd, &ps);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}
```

It defines a callback function, WndProc, responsible for handling window messages.

Within the WndProc function, different shapes are drawn on the window in response to the WM_PAINT message. The BeginPaint function is used to prepare the device context (hdc) for painting.

Firstly, a rectangle is drawn using the Rectangle function, specifying its coordinates (50, 50) for the upper-left corner and (150, 100) for the lower-right corner.

Next, an ellipse is drawn using the Ellipse function with coordinates (100, 150) for the upper-left corner and (200, 250) for the lower-right corner. This creates an elliptical shape on the window.

Subsequently, a rounded rectangle is drawn using the RoundRect function. The coordinates (250, 50) and (350, 100) define the bounding rectangle, and corner radii of 20 pixels each give the rectangle rounded corners.

An arc is drawn using the Arc function. The coordinates (400, 50) and (500, 100) define the bounding rectangle, and (425, 75) and (475, 75) specify the starting and ending points of the arc within that rectangle.

A chord is drawn using the Chord function, with coordinates (400, 150) and (500, 200) for the bounding rectangle and (425, 175) and (475, 175) for the starting and ending points.

Finally, a pie slice is drawn using the Pie function. The coordinates (400, 250) and (500, 300) define the bounding rectangle, and (425, 275) and (475, 275) set the starting and ending points for the pie slice.

The EndPaint function marks the end of the painting process, and the WM_DESTROY message ensures that the program exits when the window is closed. The default case forwards any unhandled messages to the default window procedure for processing.

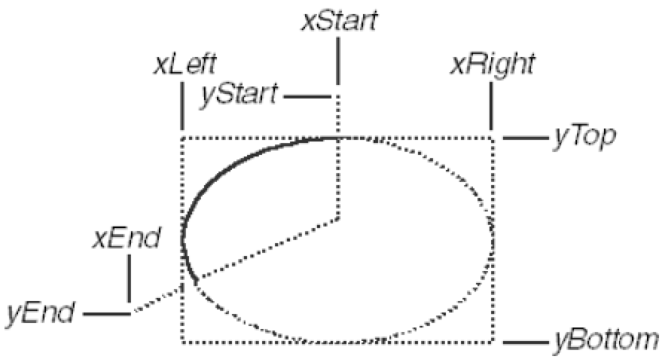


Figure 5–11. *A line drawn using the* Arc *function.*

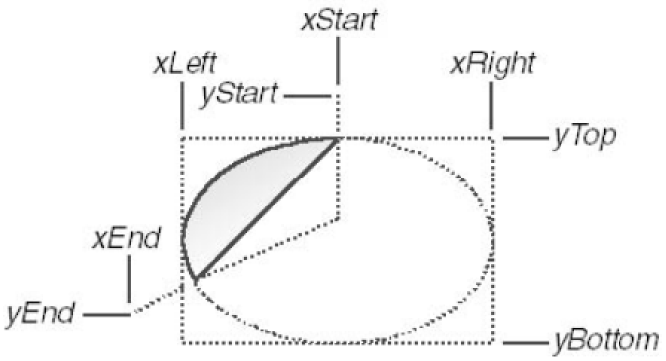


Figure 5–12. *A figure drawn using the* Chord *function.*

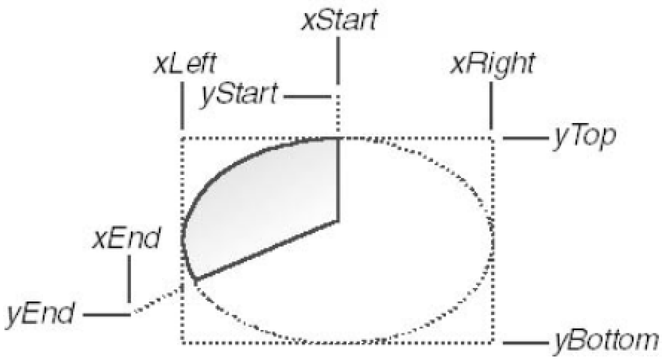


Figure 5–13. *A figure drawn using the* Pie *function.*

Relationship between Starting and Ending Points

The Arc, Chord, and Pie functions utilize an imaginary line connecting the starting point to the ellipse's center and another imaginary line connecting the ending point to the ellipse's center.

These imaginary lines are crucial in determining the arc's extent and the enclosed area to be filled.

Precision vs. Convenience

While it's possible to directly specify starting and ending points on the circumference of the ellipse, this approach requires more precise calculations of arc angles and positions.

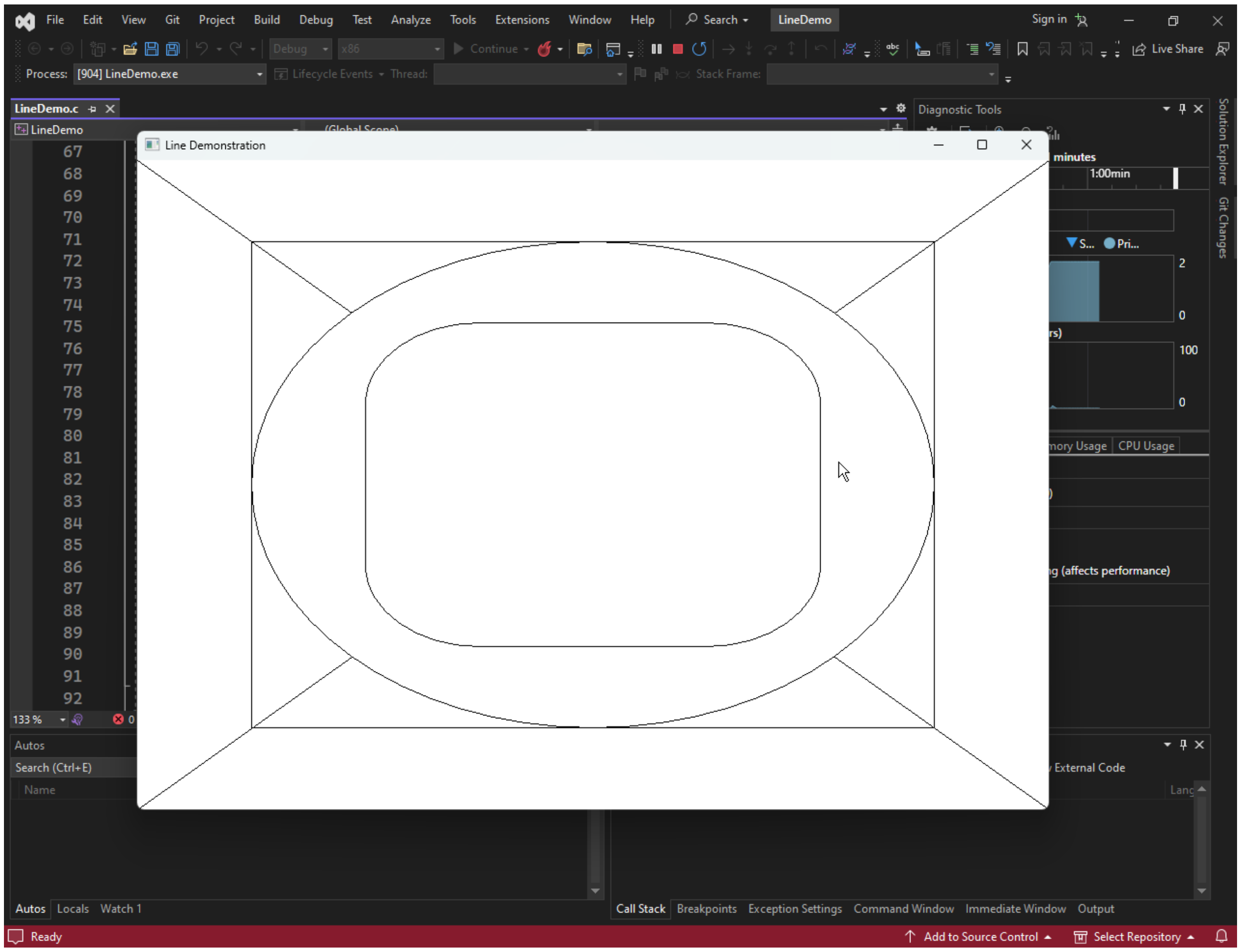
Windows' method, using starting and ending points relative to the ellipse's center, offers a simpler and more intuitive way to define the arc, chord, and pie slice without sacrificing functionality.

LINEDEMO Demonstration

The LINEDEMO program showcases the ability of these functions to fill closed areas.

The lines drawn in the program are hidden behind the ellipse, demonstrating that the ellipse's filled area extends beyond its boundaries, effectively covering the underlying lines.

The code is in Chapter 5 LineDemo folder. Run the .sln file in visual studio community.



In summary, the code orchestrates the drawing of various shapes on a window, showcasing the capabilities of the GDI library in Windows programming.

Each function call corresponds to a specific shape, and the coordinates provided determine their position and appearance on the window.

💥💥💥💥💥💥💥💥💥

BEZIER SPLINES

A Bezier spline is a parametric curve used in computer graphics and related fields.

It is defined by a set of control points that determine the shape of the curve. The more control points you use, the more complex the curve can be.

Bezier splines are often used to draw smooth, continuous curves, such as those found in fonts, logos, and graphical user interfaces.

How do Bezier Splines Work?

Bezier splines are based on the concept of Bernstein polynomials.

A Bernstein polynomial is a weighted sum of basis functions.

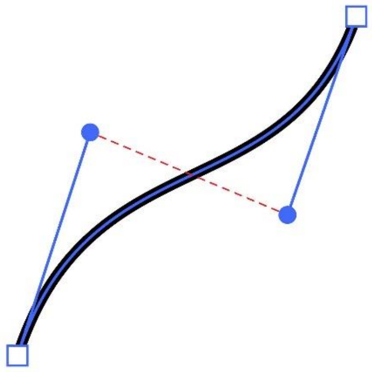
The weights of the basis functions are determined by the control points.

The higher the degree of the Bernstein polynomial, the more control points you need.

Advantages of Bezier Splines

Bezier splines have several advantages over other types of curves, such as B-splines and Hermite splines. They are:

Smooth and continuous: Bezier splines are always smooth and continuous, even at the joints between different segments of the curve.



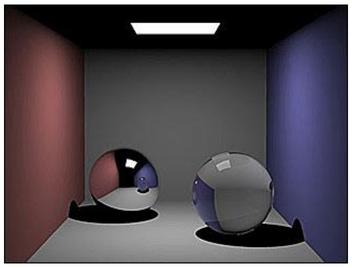
Easy to control: Bezier splines are easy to control. You can change the shape of the curve by simply moving the control points.

Computational efficient: Bezier splines are computationally efficient. They can be evaluated quickly and easily, even for complex curves.

Applications of Bezier Splines

Bezier splines are used in a wide variety of applications, including:

Computer graphics: Bezier splines are used to draw smooth, continuous curves in computer graphics applications, such as Adobe Illustrator and Inkscape.



Font design: Bezier splines are used to define the outlines of PostScript fonts.

Animation: Bezier splines are used to animate objects in animation software, such as Adobe After Effects and Maya.

Robotics: Bezier splines are used to control the motion of robots. For example, they can be used to make a robot arm move smoothly from one point to another.

Example of a Bezier Spline

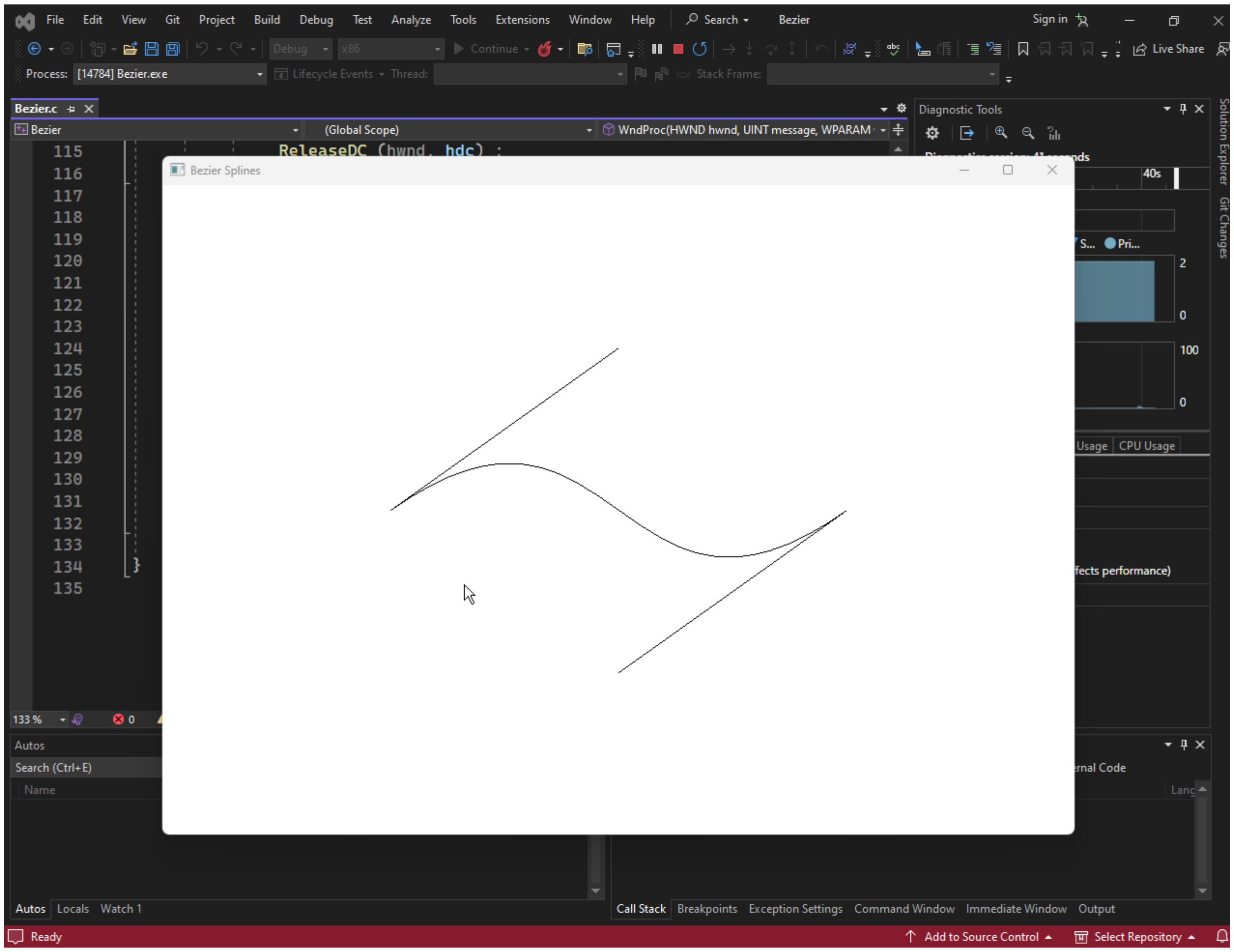
A simple Bezier spline can be defined by four points: P0, P1, P2, and P3.

The curve starts at P0 and ends at P3.

The control points P1 and P2 determine the shape of the curve.

The curve is always tangent to the line segment from P0 to P1 at P0, and it is always tangent to the line segment from P2 to P3 at P3.

Program in Chapter 5 Bezier folder.



```
    ReleaseDC (hwnd, hdc) ;
}
```

The BEZIER.C code demonstrates the creation of Bezier curves using the PolyBezier function from the Windows API. Bezier curves are a type of parametric curve that is commonly used in computer graphics to create smooth, rounded shapes.

The code starts by including the necessary header files, including windows.h. The LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) function is the window procedure, which handles all messages sent to the window.

The WinMain function is the entry point for the program. It creates a window, initializes the control points for the Bezier curve, and enters the message loop.

The DrawBezier function draws the Bezier curve to the specified device context (HDC). It first calls the PolyBezier function to draw the curve itself. Then, it draws the control points and their connecting lines to show how they affect the shape of the curve.

The WndProc function handles the following messages:

- WM_SIZE: This message is sent when the window is resized. The function updates the control points for the Bezier curve based on the new size of the window.

- WM_LBUTTONDOWN, WM_RBUTTONDOWN, and WM_MOUSEMOVE: These messages are sent when the mouse button is pressed or moved. If the left mouse button is pressed, the function updates the position of the first control point. If the right mouse button is pressed, the function updates the position of the second control point. In all cases, the function redraws the Bezier curve.
- WM_PAINT: This message is sent when the window needs to be repainted. The function calls the DrawBezier function to redraw the curve.
- WM_DESTROY: This message is sent when the window is destroyed. The function posts a quit message to the message queue, which causes the program to exit.

The text describes the BEZIER.C program from the book Graphics Programming in C by Charles Petzold.

The program allows users to manipulate Bezier splines by dragging the two control points with the mouse.

The control points are initially set to be halfway down the client area, and ¼ and ¾ of the way across the client area.

The text also discusses several characteristics of Bezier splines that make them useful for computer-assisted design (CAD) work:

- Easy to manipulate: With a little practice, users can usually manipulate Bezier splines into the desired shape.
- Well controlled: Bezier splines are always anchored at the two end points and bounded by a convex hull formed by connecting the end points and control points. This means that the curve will never veer off into infinity, which is important for CAD work.
- Aesthetically pleasing: Bezier splines often produce smooth, rounded curves that are pleasing to the eye.

The BEZIER.C program uses the following code to draw the Bezier curve:

```
DrawBezier (hdc, apt);
```

The DrawBezier function calls the PolyBezier function from the Windows API to draw the curve. The PolyBezier function takes three parameters:

- **hdc:** The device context to draw the curve to.
- **apt:** An array of points that define the Bezier curve.
- **iCount:** The number of points in the array.

In the BEZIER.C program, the apt array contains four points: the two end points and the two control points. The iCount parameter is set to 4.

The **DrawBezier function also draws straight lines** from the first control point to the first end point and from the second control point to the second end point. This is done to show how the control points affect the shape of the curve.

The mathematical equations for the Bezier curve are as follows:

```
x(t) = (1 – t)3 x0 + 3t (1 – t)2 x1 + 3t2 (1 – t) x2 + t3 x3
y(t) = (1 – t)3 y0 + 3t (1 – t)2 y1 + 3t2 (1 – t) y2 + t3 y3
```

These equations define the x and y coordinates of a point on the Bezier curve for a given value of t. The t parameter ranges from 0 to 1.

When t is 0, the curve is at the first end point. When t is 1, the curve is at the second end point. For values of t in between, the curve is at some point along the curve.

The equations are based on the following assumptions:

- The curve is anchored at the two end points.
- The curve is tangent to and in the same direction as a straight line drawn from the begin point to the first control point.
- The curve is tangent to and in the same direction as a straight line drawn from the second control point to the end point.

In both the PolyBezier and PolyBezierTo functions, the apt parameter is an array of POINT structures. Each POINT structure contains an x-coordinate and a y-coordinate, which specify a point on the screen.

💥💥💥💥💥💥💥💥💥💥💥💥💥💥💥💥💥💥

Using PolyBezier

The PolyBezier function takes four arguments:

- **hdc:** The device context to draw the curve to.

- **apt:** An array of POINT structures that define the curve.
- **iCount:** The number of points in the array.
- **nCurves:** The number of connected curves to draw.

In the PolyBezier function, the first four points in the apt array define the first Bezier curve. The next three points define the second Bezier curve, and so on. Each Bezier curve requires three points because the end point of one curve is the same as the begin point of the next curve.

The iCount parameter is always one plus three times the number of nCurves. For example, if you are drawing two connected Bezier curves, then iCount would be 7. The nCurves parameter is typically 1, but it can be greater than 1 if you are drawing multiple connected curves.

Using PolyBezierTo

The PolyBezierTo function takes three arguments:

- **hdc:** The device context to draw the curve to.
- **apt:** An array of POINT structures that define the curve.
- **nCurves:** The number of connected curves to draw.

The PolyBezierTo function uses the current position for the first begin point. The first and each subsequent Bezier spline requires only three points. When the function returns, the current position is set to the last end point.

Note on Smooth Connections

When you draw a series of connected Bezier splines, the point of connection will be smooth only if the second control point of the first Bezier, the end point of the first Bezier (which is also the begin point of the second Bezier), and the first control point of the second Bezier are colinear; that is, they lie on the same straight line.

This is because the Bezier spline is always tangent to the line formed by its two control points. If the control points are not colinear, then the tangents of the two splines will not be the same, and the connection will not be smooth.

The PolyBezier and PolyBezierTo functions are powerful tools for drawing Bezier splines in Windows. By using these functions, you can create smooth, rounded curves that are ideal for a variety of applications.