CHAPTER 17: DIVING DEEPER INTO TEXT AND FONTS WITH WINAPI

Welcome to Chapter 17, where we'll delve into the fascinating world of text and fonts in the context of WinAPI! As you mentioned, displaying text was our initial foray into graphics programming, and now it's time to refine our skills by exploring:

Font Varieties in Windows: We'll delve into the diverse world of fonts available in Microsoft Windows, including the revolutionary TrueType technology that brought WYSIWYG to life.

Font Manipulation Magic: TrueType's power goes beyond simple display. We'll explore exciting techniques like font scaling, rotation, pattern fills, and even using fonts as clipping regions!

Justifying Text: Learn how to make your text visually appealing by aligning it to the left, right, or center of the window.

Remember, you're currently learning WinAPI, so we'll focus on using its functionalities to achieve these effects. Buckle up and get ready to unleash your inner typography guru!

TrueType: The Game Changer for Text in Windows

The introduction of TrueType in Windows 3.1 marked a significant milestone in text rendering. Unlike older bitmap fonts, TrueType uses mathematical outlines to define character shapes. This offers several advantages:

Scalability: TrueType fonts can be smoothly scaled to any size without losing quality, unlike pixelated bitmap fonts. This is crucial for WYSIWYG, ensuring what you see on screen is what gets printed.



Platform Independence: TrueType fonts work across different platforms like Windows and macOS, promoting compatibility and flexibility.



Advanced Features: TrueType's outline-based approach opens doors for exciting font manipulation techniques like:

* Rotation: You can rotate characters to create unique effects.
* Pattern Filling: Fill character interiors with custom patterns for visual flair.
* Clipping Regions: Use font outlines to define clipping regions for other graphical elements.

We'll explore these functionalities later in the chapter, so stay tuned!

Beyond Basic Display: Mastering Text Manipulation

TrueType's capabilities extend far beyond simply displaying text on the screen. With WinAPI, you can unleash your creativity and manipulate fonts in various ways:

Scaling: Change the size of text without compromising quality using functions like CreateScalableFontResource and SetTextScale.



Rotation: Rotate individual characters or entire strings for a dynamic and eye-catching look. Utilize functions like GetGlyphOutline and PolyDraw to achieve this.



Pattern Filling: Fill the interiors of characters with custom patterns using WinAPI functions like FillPath. Imagine text shimmering with stripes or polka dots!



Clipping Regions: Define clipping regions based on font outlines using functions like SelectClipRgn. This allows you to mask other graphical elements behind the text, creating interesting effects.



Remember, these are just a few examples. As you delve deeper into WinAPI text manipulation, you'll discover a vast playground for experimentation and creative expression.

Justifying Text for Visual Balance

Justified text, where both margins are aligned, adds a touch of refinement to your applications. WinAPI provides functions like SetTextAlign and GetTextExtentPoint32 to achieve this effect. You can choose left, right, or center alignment based on your desired layout.



By mastering text justification, you can elevate the visual appeal of your WinAPI applications and create a more polished and professional user experience.

WINDOWS TEXT OUTPUT FUNCTIONS

In Windows programming, text output is facilitated by various functions, and one commonly used function is:

TextOut:



Core function for displaying text.

Arguments:

* hdc: Handle to the device context.
* xStart: Horizontal starting position (logical coordinates).
* yStart: Vertical starting position (logical coordinates).
* pString: Pointer to the character string.
* iCount: Length of the string (not NULL-terminated).

Coordinates and Positioning

The xStart and yStart parameters determine the starting position of the text in logical coordinates.

Typically, Windows starts drawing at the upper left corner of the first character.

The function requires a pointer to the character string (pString) and the length of the string (iCount). Notably, it does not recognize NULL-terminated character strings.

The positioning of the text can be influenced by the SetTextAlign function. Flags like TA\_LEFT, TA\_RIGHT, TA\_CENTER, TA\_TOP, TA\_BOTTOM, and TA\_BASELINE affect how xStart and yStart are used for horizontal and vertical positioning.

SetTextAlign Function

By calling SetTextAlign with the TA\_UPDATECP flag, the xStart and yStart arguments in TextOut are ignored.

Instead, Windows uses the current position set by functions like MoveToEx or LineTo.

The TA\_UPDATECP flag also updates the current position after a TextOut call, which is useful for displaying multiline text.

Controls horizontal and vertical positioning of text.

Flags:

* Horizontal: TA\_LEFT, TA\_RIGHT, TA\_CENTER
* Vertical: TA\_TOP, TA\_BOTTOM, TA\_BASELINE

TA\_UPDATECP flag:

* Ignores xStart/yStart in TextOut, uses current position.
* Updates current position after TextOut (except for TA\_CENTER).

TabbedTextOut Function

An alternative to multiple TextOut calls for columnar text is the TabbedTextOut function:



If the text string contains tab characters (\t or 0x09), TabbedTextOut expands the tabs based on an array of tab stops.

Handles text with embedded tab characters.

Expands tabs based on specified tab stops.

Arguments:

* iNumTabs: Number of tab stops.
* piTabStops: Array of tab stop positions (pixels).
* xTabOrigin: Starting position for measuring tab stops.

Tab Stops and Customization

The sixth argument (iNumTabs) is the number of tab stops, and the seventh argument (piTabStops) is an array of tab stops in pixels.

You can customize tab stops by providing specific pixel values. If these arguments are set to 0 or NULL, tab stops are set at every eight average character widths.

ADVANCED TEXT OUTPUT FUNCTIONS IN WINDOWS

In Windows programming, the ExtTextOut function provides extended capabilities for text rendering:



Arguments:

* hdc: Handle to the device context.
* xStart, yStart: Starting position (logical coordinates).
* iOptions: Flags for clipping and background:
* ETO\_CLIPPED: Clips text to the specified rectangle.
* ETO\_OPAQUE: Fills the rectangle with the background color before drawing text.
* &rect: Pointer to a rectangle structure for clipping or background.
* pString: Pointer to the character string.
* iCount: Length of the string.
* pxDistance: Optional array of integers for intercharacter spacing (NULL for default).

Clipping and Background Rectangles

The fifth argument (rect) in ExtTextOut is a pointer to a rectangle structure. If iOptions is set to ETO\_CLIPPED, it serves as a clipping rectangle; if set to ETO\_OPAQUE, it becomes a background rectangle filled with the current background color. Both options can be specified or omitted as needed.

Character Spacing

The last argument (pxDistance) is an array of integers specifying the spacing between consecutive characters. This feature allows fine-tuning of intercharacter spacing, which proves valuable for justifying text in narrow columns. Setting it to NULL defaults to the standard character spacing.

DrawText Function

A higher-level text rendering function in Windows is DrawText:



Arguments:

* hdc: Handle to the device context.
* pString: Pointer to the character string.
* iCount: Length of the string (-1 for NULL-terminated strings).
* &rect: Pointer to a rectangle structure defining the text area.
* iFormat: Flags controlling text formatting.

*The iFormat above has flags that control various aspects of text formatting. Here's a breakdown of each flag and its purpose:*

Alignment:

* DT\_LEFT (default): Specifies left justification of the text.
* DT\_RIGHT: Specifies right justification of the text.
* DT\_CENTER: Specifies center alignment of the text.

Line Breaking:

* DT\_SINGLELINE: Treats carriage returns and linefeeds as displayable characters, meaning they will be shown as they are.
* DT\_TOP (default): Places the text at the top of the rectangle.
* DT\_BOTTOM: Places the text at the bottom of the rectangle.
* DT\_VCENTER: Vertically centers the text within the rectangle.
* DT\_WORDBREAK: Breaks lines at the end of words if they don't fit within the rectangle. This ensures that words are not split in the middle.

Clipping:

* DT\_NOCLIP: Disables clipping, allowing text to extend beyond the boundaries of the rectangle. This means the text may overflow outside the specified area.

Spacing:

* DT\_EXTERNALLEADING: Includes external leading in line spacing. External leading refers to the extra space between lines of text.

Tabs:

* DT\_EXPANDTABS: Expands tab characters (\t) to spaces, aligning the text based on the specified tab stops.
* DT\_TABSTOP (use cautiously): Sets custom tab stops. The upper byte of iFormat specifies the positions of the tab stops. This flag should be used carefully, as incorrect tab stops can lead to inconsistent or unexpected text alignment.

These flags provide control over the alignment, line breaking, clipping, spacing, and tab behavior of the text. By combining different flags, you can achieve the desired formatting for displaying text within a given rectangle.

Key Points:

* Use ExtTextOut for granular control over clipping, background, and intercharacter spacing.
* Use DrawText for simplified text output within a rectangle, with various formatting options.
* Understand the available flags to tailor text output to your specific needs.
* Consider using DrawText for common text output tasks due to its convenience.
* Use ExtTextOut when you need precise control over text rendering.

Specifying Text within a Rectangle

Instead of specifying a coordinate starting position, DrawText uses a RECT structure defining a rectangle where the text should appear.

The function requires a pointer to the character string (pString) and its length (iCount). For NULL-terminated strings, setting iCount to -1 prompts Windows to calculate the length automatically.

Text Formatting Options

The iFormat argument in DrawText allows customization of the text's appearance within the specified rectangle.

Flags such as DT\_LEFT (default for left-justified), DT\_RIGHT (right-justified), and DT\_CENTER (centered) control the horizontal alignment.

Including DT\_SINGLELINE prevents newline characters, and DT\_TOP, DT\_BOTTOM, and DT\_VCENTER dictate vertical alignment.

Handling Line Breaks and Word Wrapping

Windows interprets carriage return and linefeed characters as newline characters by default. The DT\_SINGLELINE flag changes this behavior.

For multi-line displays, using DT\_WORDBREAK breaks lines at the end of words, ensuring more readable text.

Additionally, the DT\_NOCLIP flag prevents text truncation outside the specified rectangle.

Tab Handling

For text containing tab characters (\t or 0x09), including the DT\_EXPANDTABS flag in DrawText ensures proper rendering.

By default, tab stops are set every eighth character position.

While the DT\_TABSTOP flag allows custom tab settings, caution is advised due to potential conflicts with other flags in the iFormat argument.

In programming and character encoding, the term "tab character" refers to a control character that is commonly represented as \t in escape sequences or as the hexadecimal value 0x09. It is a non-printable character used to advance the cursor to the next tab stop.

When you encounter a tab character (\t or 0x09) in a string, it serves as an instruction to move the cursor to the next predefined position, which is typically at regular intervals.

The default convention is to set tab stops every eight character positions, but this can be customized.

DrawTextEx: Enhanced Text Handling with Tab Stops

Purpose:

* Offers more control over text formatting compared to DrawText, particularly for handling tab stops.
* Introduced to address limitations of DT\_TABSTOP flag in DrawText.

Syntax:



Arguments:

* hdc: Handle to the device context.
* pString: Pointer to the character string.
* iCount: Length of the string.
* &rect: Pointer to a rectangle structure defining the text area.
* iFormat: Flags controlling text formatting (same as in DrawText).
* &drawtextparams: Pointer to a DRAWTEXTPARAMS structure for additional settings.

DRAWTEXTPARAMS Structure:

* cbSize: Size of the structure (set to sizeof(DRAWTEXTPARAMS)).
* iTabLength: Size of each tab stop, in units of average character width.
* iLeftMargin: Left margin, in units of average character width.
* iRightMargin: Right margin, in units of average character width.
* uiLengthDrawn: Receives the number of characters processed.

Key Points:

* Use DrawTextEx when you need precise control over tab stops.
* Set iTabLength to define the spacing between tab stops.
* Margins are optional for further text positioning.
* uiLengthDrawn provides feedback on the amount of text drawn.
* Avoids conflicts with other flags in iFormat.
* DrawTextEx might not be available on older Windows systems.
* For simple tab handling without margins, DT\_TABSTOP in DrawText might still suffice.
* Choose the appropriate function based on your specific tab stop requirements and compatibility needs.

Utilizing Enhanced Settings

With DrawTextEx, developers gain greater flexibility in text layout and formatting. The DRAWTEXTPARAMS structure allows for fine-tuning tab stops, adjusting margins, and obtaining information about the processed text.

This enhanced functionality is particularly valuable in scenarios where precise text alignment, tabulation, and margin control are essential.

By leveraging the DrawTextEx function with the DRAWTEXTPARAMS structure, developers can create visually appealing and precisely formatted text displays in Windows applications. This improved feature set contributes to a more robust and customizable text rendering experience.

DEVICE CONTEXT ATTRIBUTES FOR TEXT RENDERING IN WINDOWS

In Windows programming, several device context attributes play a crucial role in determining how text is displayed. These attributes allow developers to customize aspects such as text color, background color, background mode, and intercharacter spacing.

Customizing Text Color

The default text color in the device context is black, but developers can alter it using the SetTextColor function: Text Color is controlled by:



The rgbColor parameter represents the desired color, and Windows converts this value into a pure color. To retrieve the current text color, developers can use the GetTextColor function.

Background Mode and Color

Windows displays text within a rectangular background area, which can be colored based on the background mode setting. Developers can change the background mode using SetBkMode:



The iMode parameter can be either OPAQUE or TRANSPARENT. The default is OPAQUE, where Windows fills the background with the specified color. The background color can be set using:



The rgbColor parameter is converted to a pure color. In TRANSPARENT mode, Windows ignores the background color, enhancing text visibility.

Handling Intercharacter Spacing

Intercharacter spacing can be adjusted using the SetTextCharacterExtra function:



The iExtra parameter, in logical units, determines the spacing between characters. A value of 0 means no additional space.

Negative values are converted to their absolute values, preventing spacing less than 0. Developers can retrieve the current intercharacter spacing with GetTextCharacterExtra.

Adapting to System Colors

For consistency with system color settings, developers can set text and background colors using system colors:



In case of system color changes, developers should handle the WM\_SYSCOLORCHANGE message and update the display accordingly:



Understanding and manipulating these device context attributes provide developers with the tools needed to create visually appealing and adaptable text displays in Windows applications. Fine-tuning text color, background settings, and intercharacter spacing contribute to a more polished and user-friendly graphical interface.

Key Points:

* These attributes influence text appearance and background rendering.
* Choose appropriate settings for text readability and visual appeal.
* Consider system-consistent colors for seamless user experience.
* Adjust intercharacter spacing for fine-tuning text layout.