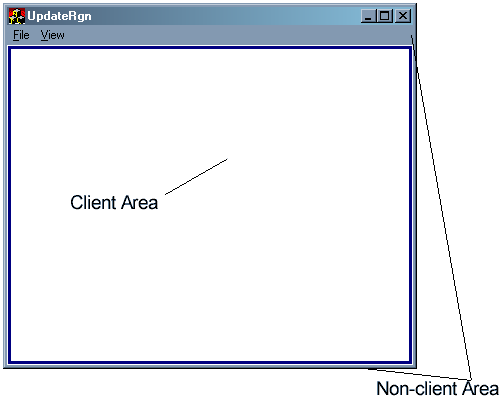
**CHAPTER 4**

In Windows programming, the client area is the part of the application window that is not taken up by the title bar, window-sizing border, and other elements.

Windows programs must be able to handle client areas of varying sizes, from very small to very large.

The process of displaying text or graphics in a Windows program's client area is called "painting."



Windows provides a **Graphics Device Interface (GDI)** for painting, but in this chapter we will focus on displaying simple lines of text.

Windows programs should use the system font as the default font, as this ensures consistent appearance across different systems.

Device-independent programming is the practice of writing software that can run on a variety of hardware and software configurations.



Windows programs can obtain information about their environment using Windows facilities.

The size of the client area can change at any time, so Windows programs need to be able to react to these changes.

Windows programs can use a variety of techniques to make their output look good on different screen sizes.



Device-independent programming is an important skill for any Windows programmer.

In Windows, programs can only draw text and graphics in the client area of their window.

Windows may inform a window procedure that part of the window's client area needs painting by posting a WM\_PAINT message.

A window procedure can receive a WM\_PAINT message for a variety of reasons, such as when a hidden area of the window is brought into view or when the user resizes the window or a tooltip is displayed.

In some cases, Windows may post a WM\_PAINT message when it is not strictly necessary to repaint the client area.

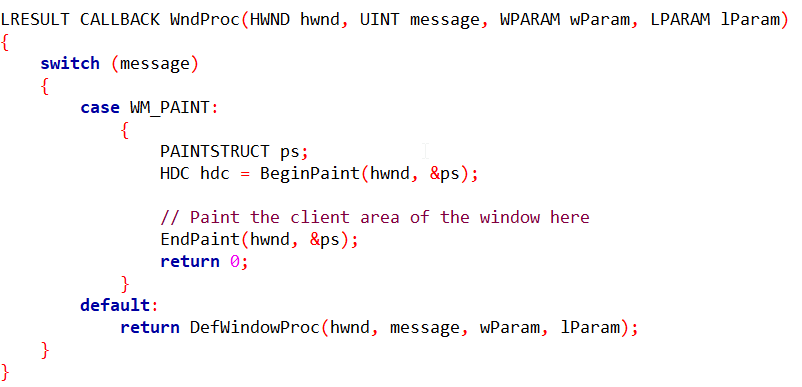
Windows always saves the area of the display it overwrites and then restores it in a few cases, such as when the mouse cursor is moved across the client area or when an icon is dragged across the client area.

Windows programs should be prepared to receive WM\_PAINT messages at any time.

Windows programs should be able to repaint their entire client area if necessary.

Windows programs can use the InvalidateRect or InvalidateRgn function to explicitly generate a WM\_PAINT message.

Here is an example of how to handle a WM\_PAINT message in a Windows program:



This code is the window procedure for a Windows program. The window procedure is responsible for handling all messages sent to the window, including WM\_PAINT messages.

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam): This line defines the window procedure function. The function takes four parameters:

* hwnd: The handle of the window
* message: The message that was sent to the window
* wParam: Additional message-specific information
* lParam: Additional message-specific information

switch (message): This switch statement checks the type of message that was sent to the window.

case WM\_PAINT:: This case block handles the WM\_PAINT message. This message is sent to the window when the client area needs to be painted.

PAINTSTRUCT ps;: This line declares a variable of type PAINTSTRUCT. This structure contains information about the area of the client area that needs to be painted.

HDC hdc = BeginPaint(hwnd, &ps);: This line calls the BeginPaint function to obtain a device context (HDC) for the window. The HDC is used to draw on the client area of the window.

// Paint the client area of the window here`: This comment indicates where the painting code should go. The painting code should use the HDC to draw on the client area of the window.

EndPaint(hwnd, &ps);: This line calls the EndPaint function to release the HDC.

return 0;: This line returns 0 to indicate that the message was handled successfully.

default:: This block handles all messages that are not specifically handled by the switch statement.

return DefWindowProc(hwnd, message, wParam, lParam);: This line calls the DefWindowProc function to handle the message. The DefWindowProc function will handle the message in the default way for the window class.

**CLIENT AREA PAINTING**

In traditional character-based environments, programs have direct control over the entire video display.



They can write text and graphics anywhere on the screen, and the modifications will remain visible until explicitly overwritten.

This simplicity allows programs to manage the screen contents without worrying about external factors.

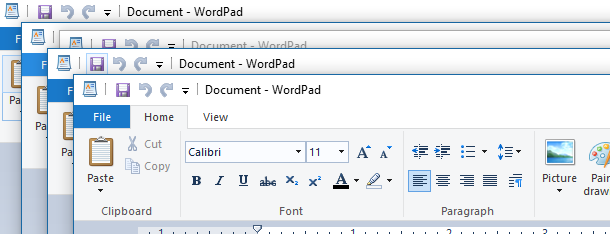
However, in Windows, the situation is more complex. Programs can only draw on the client area of their own window, a designated rectangular region within the overall window frame.

This restriction is primarily due to the multitasking nature of Windows, where multiple programs share the screen space.

Additionally, the content of the client area is not guaranteed to persist indefinitely.

**There are several scenarios where the client area may need to be repainted:**

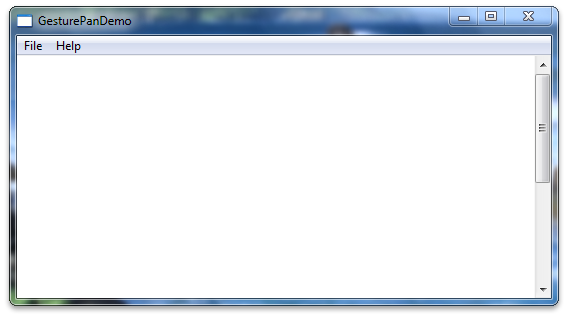
Revealing Previously Hidden Areas: When a hidden portion of the window is brought into view, either by moving the window or uncovering it from behind another window, Windows will send a WM\_PAINT message to the window procedure. This message signals the need to redraw the exposed area.



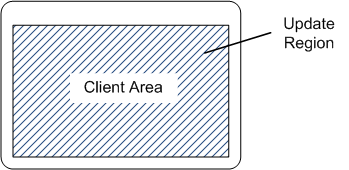
Window Resizing: If the user resizes the window, and the window class style has the CS\_HREDRAW and CW\_VREDRAW bits set, Windows will again send a WM\_PAINT message. This ensures that the client area adapts to the new window size.



Scrolling: When a program uses the ScrollWindow or ScrollDC functions to scroll part of the client area, Windows will generate a WM\_PAINT message to update the visible portion of the client area.



Explicit Invalidation: Programs can explicitly request a repaint of specific areas of the client area using the InvalidateRect or InvalidateRgn functions. This is useful when the program makes changes to its internal data that affect the displayed content.



Temporary Overwriting: In some cases, Windows may attempt to save and restore an area of the display when it is temporarily overwritten, such as when a dialog box or menu is displayed over the client area. However, this process is not always reliable, and Windows may sometimes send a WM\_PAINT message even when the client area was not actually altered.

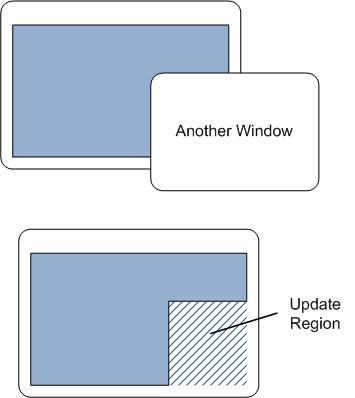


Mouse Cursor and Icon Dragging: In a few specific situations, Windows always saves and restores the overwritten area, triggering a WM\_PAINT message only when necessary. These cases include moving the mouse cursor across the client area or dragging an icon within the client area.



**Dealing with WM\_PAINT Messages**

Handling WM\_PAINT messages requires a shift in how you perceive drawing on the screen. Instead of directly updating the display whenever your program needs to, you should structure your program to accumulate all the necessary drawing information and only perform the actual rendering when Windows sends a WM\_PAINT message. This may seem like an indirect approach, but it promotes a more structured and manageable programming style.



**On-demand Painting**

The WM\_PAINT message acts as a trigger, informing your program that the client area needs to be updated. By deferring the actual painting until this message is received, you maintain a clear separation between the data model and the visual representation, making your program more organized and easier to maintain.



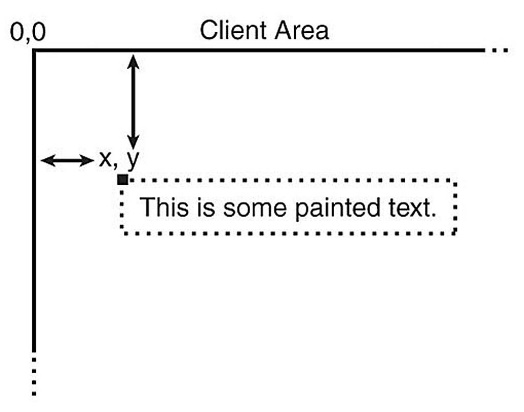
**Forcing Repaints**

In situations where your program needs to update the client area outside of a WM\_PAINT message, you can explicitly force Windows to generate this message by calling the InvalidateRect function. This will invalidate the specified rectangular region, prompting Windows to add a WM\_PAINT message to the message queue.



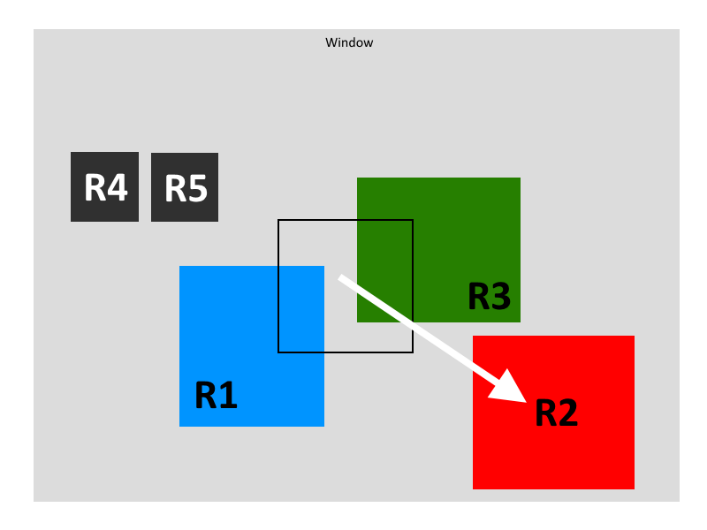
**Valid and Invalid Rectangles**

While a window procedure should be prepared to repaint the entire client area in response to a WM\_PAINT message, it's often more efficient to update only a smaller portion, typically a rectangular region within the client area. This is particularly relevant when a dialog box or other element temporarily overlaps the client area.



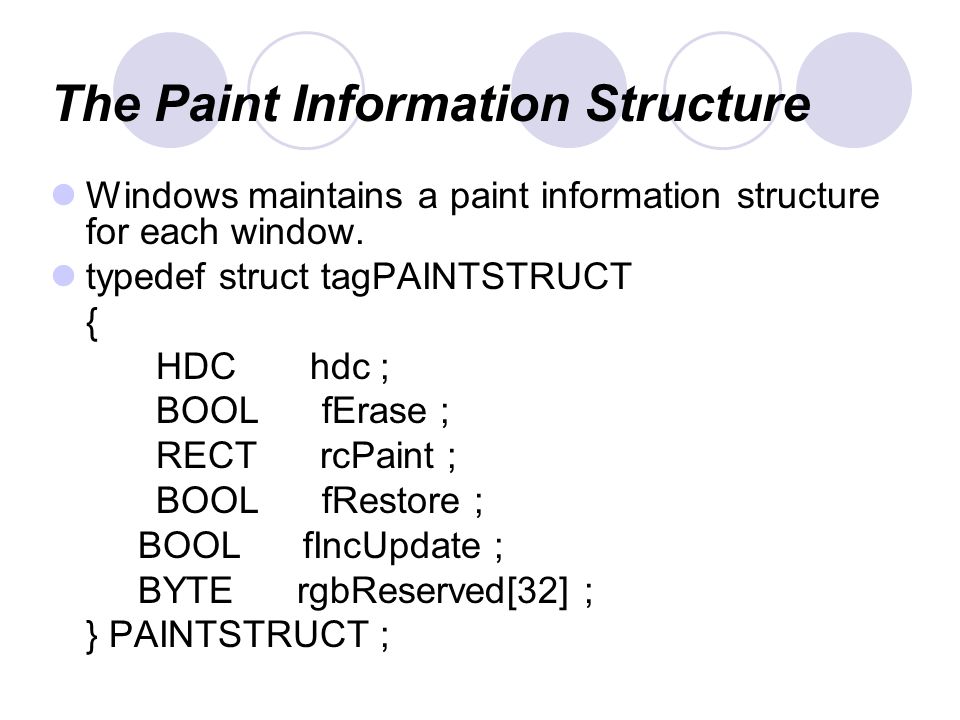
**Invalid Region**

The area of the client area that needs to be repainted is known as an "invalid region" or "update region." The presence of an invalid region is what triggers Windows to send a WM\_PAINT message. Your window procedure will only receive a WM\_PAINT message if there's an invalid region in your client area.



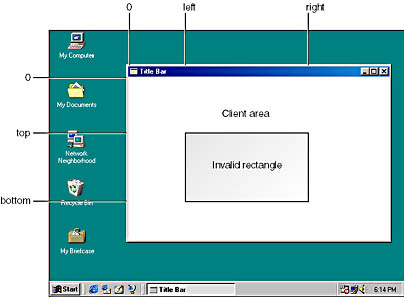
**Paint Information Structure**

Windows maintains an internal "paint information structure" for each window. This structure holds various information, including the coordinates of the smallest rectangle that encompasses the invalid region. This rectangle is called the "invalid rectangle."



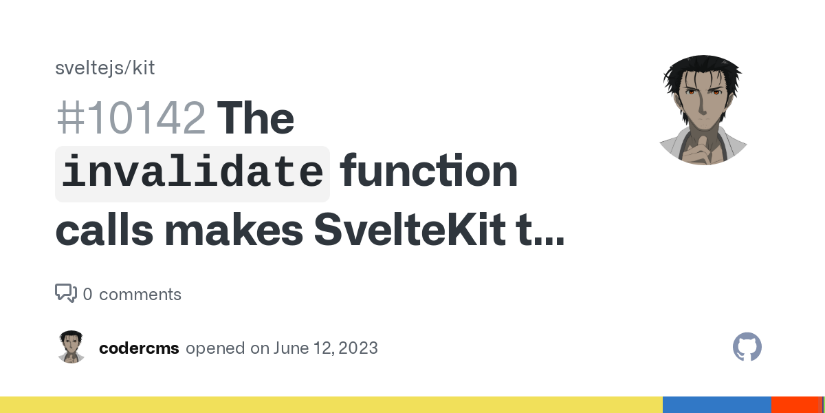
**Invalid Rectangle Updates**

If another part of the client area becomes invalid before the window procedure processes the pending WM\_PAINT message, Windows will recalculate the invalid region and invalid rectangle to encompass both areas. It will then update the paint information structure with this new information. Windows avoids placing multiple WM\_PAINT messages in the queue for the same window.



**Invalidating Rectangles**

A window procedure can explicitly invalidate a rectangle in its own client area by calling the InvalidateRect function. If there's already a WM\_PAINT message in the message queue, Windows will recalculate the invalid rectangle accordingly. Otherwise, it will add a WM\_PAINT message to the queue.



**Retrieving Invalid Rectangle Coordinates**

The window procedure can retrieve the coordinates of the invalid rectangle when it receives a WM\_PAINT message. It can also obtain these coordinates at any other time by calling the GetUpdateRect function.



**Validating Rectangles**

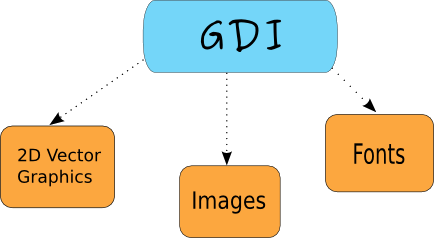
Once the window procedure calls BeginPaint during the WM\_PAINT message, the entire client area is considered validated. A program can also explicitly validate any rectangular area within the client area by calling the ValidateRect function. If this call effectively validates the entire invalid area, any WM\_PAINT message currently in the queue is removed.



**INTRODUCTION TO GDI**

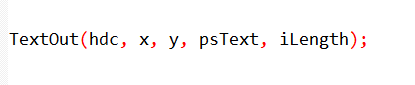
The Graphics Device Interface (GDI) is a set of functions provided by Windows for drawing text, graphics, and other visual elements on the screen.

To paint the client area of your window, you'll utilize these GDI functions.

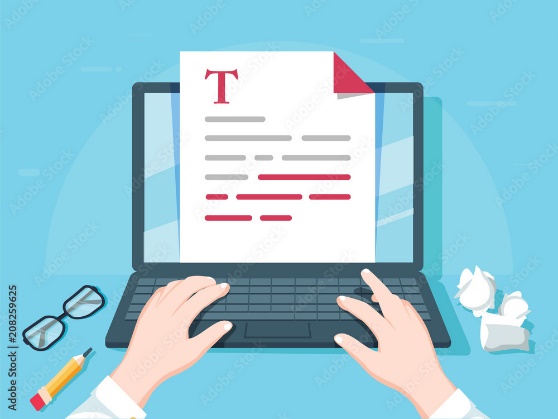
B

TextOut: A Versatile Text Output Function

Windows offers several GDI functions for writing text to the client area, but the most commonly used is undoubtedly TextOut. It takes the following format:



This function displays a character string in the client area. The psText argument is a pointer to the character string, and iLength specifies its length in characters. The x and y coordinates define the starting position of the text within the client area.



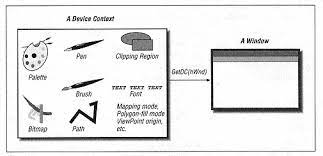
The Device Context: A Crucial GDI Element

The hdc argument in the TextOut function is a "handle to a device context" (DC).

A handle is simply a numerical identifier that Windows uses internally to reference objects.

You obtain the DC handle from Windows and use it in various GDI functions.

The DC handle serves as your window's authorization to interact with GDI functions, enabling you to draw on the client area.



The device context (DC) is a data structure maintained internally by GDI.

It's associated with a specific display device, such as a monitor or a printer.

For a video display, the DC is typically linked to a particular window on the screen.

Graphics Attributes: Defining the Look and Feel

The DC contains various values known as graphics attributes, which determine how GDI drawing functions operate.

For instance, in the case of TextOut, these attributes specify the text color, background color, font to use, and how the x and y coordinates from the function are mapped to the client area.



Acquiring and Releasing the Device Context Handle

Before painting, a program must obtain a handle to the device context.

When you do this, Windows initializes the internal DC structure with default attribute values.

These defaults can be modified using specific GDI functions.

You can also retrieve the current values of these attributes and utilize other GDI functions to draw on the client area.



Proper Handling of the Device Context Handle

Once a program has finished painting, it's essential to release the device context handle.



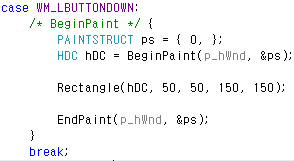
Releasing the handle invalidates it and prevents its further use. The program should acquire and release the handle within the processing of a single message.

With the exception of a DC created using the CreateDC function, which is beyond the scope of this chapter, you should not maintain a DC handle between messages.

Common Methods for Obtaining a Device Context Handle

Windows applications generally employ two methods to obtain a DC handle for screen painting:

Using BeginPaint: The BeginPaint function retrieves the DC handle for the window and prepares it for painting. This function should be called at the beginning of the WM\_PAINT message processing.



Using GetDC: The GetDC function directly retrieves the DC handle for the window. This function can be used outside of the WM\_PAINT message processing.



These methods provide the necessary access to the device context, enabling you to paint on the client area using GDI functions.

**Method One: Acquiring a Device Context Handle with BeginPaint and EndPaint**

This method is specifically used when processing WM\_PAINT messages, which signal the need to repaint the client area of a window. Two crucial functions are involved in this process: BeginPaint and EndPaint.

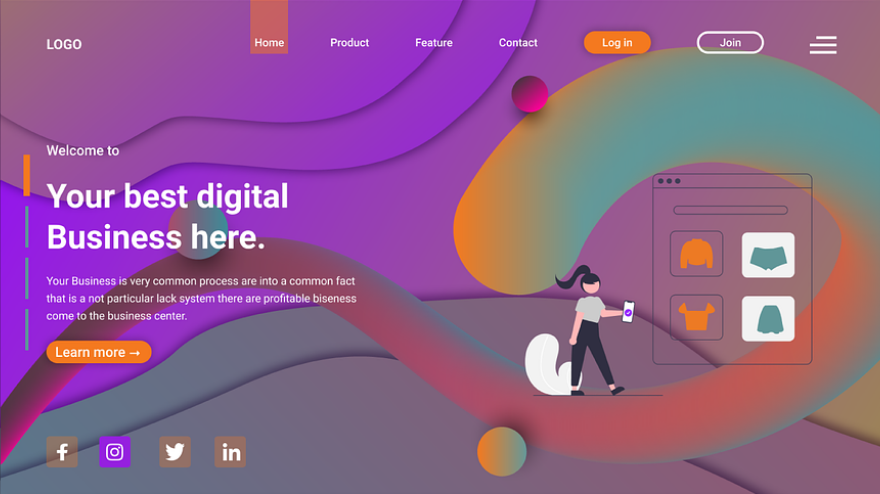
****

Figure 1: BeginPaint and EndPaint

BeginPaint: Preparing for Painting

The BeginPaint function marks the beginning of the painting process. It performs several essential tasks:

* Background Erasure: It erases the background of the invalid region, ensuring a clean slate for new rendering.
* PAINTSTRUCT Structure: It fills in the fields of the PAINTSTRUCT structure, providing information about the painting operation.
* Device Context Handle: It returns the device context handle (HDC), which is a unique identifier for the window's drawing context.

EndPaint: Releasing the Device Context

The EndPaint function serves as the counterpart to BeginPaint, marking the end of the painting process. It performs the following actions:

* Device Context Release: It releases the device context handle, making it available for other applications to use.
* Painting Validation: It validates the previously invalid region, indicating to Windows that the repainting is complete.

Typical WM\_PAINT Message Handling

A typical implementation of WM\_PAINT message handling using BeginPaint and EndPaint involves the following steps:

* BeginPaint Call: The window procedure calls BeginPaint, obtaining the device context handle and preparing for painting.
* GDI Function Calls: The program utilizes various GDI functions, such as TextOut, to draw on the client area using the acquired device context handle.
* EndPaint Call: The window procedure calls EndPaint, releasing the device context handle and validating the painting operation.

Error in Skipping BeginPaint and EndPaint

Attempting to handle a WM\_PAINT message without calling BeginPaint and EndPaint is a serious error.



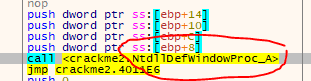
Windows places the WM\_PAINT message in the message queue because part of the client area is invalid and requires repainting.

Failing to call BeginPaint and EndPaint will prevent Windows from validating the invalid region, leading to a continuous stream of WM\_PAINT messages without any actual painting.

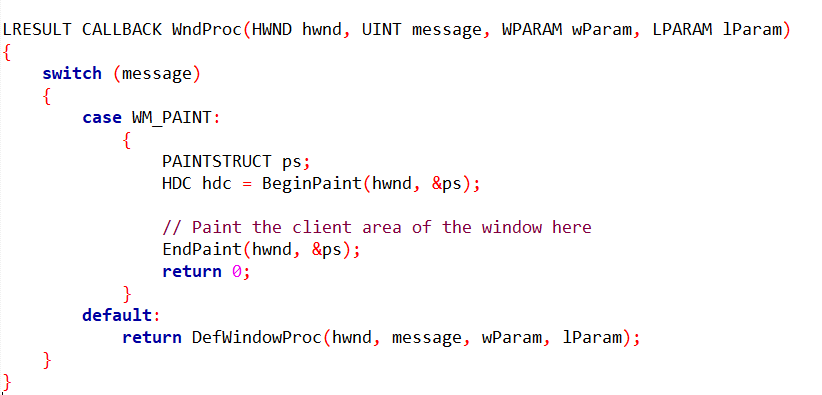
Default Window Procedure Handling

If a window procedure does not handle WM\_PAINT messages, Windows will pass the message to the default window procedure, DefWindowProc.

DefWindowProc will automatically call BeginPaint and EndPaint, effectively validating the invalid region and handling the painting process.



Here is an example of how to use the BeginPaint and EndPaint functions to process a WM\_PAINT message:



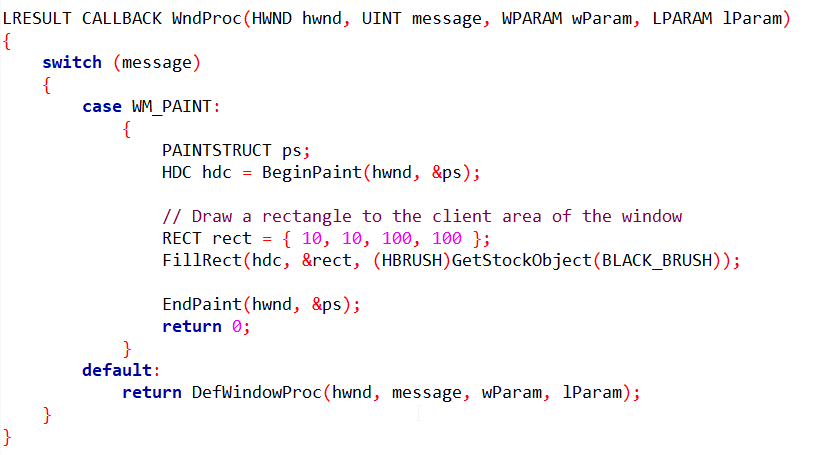
This code will first call BeginPaint to obtain a device context handle (HDC) for the window. The BeginPaint function will also erase the background of the invalid region of the client area.

Next, the code will paint the client area of the window using the HDC. The code can use any GDI functions to paint the client area.

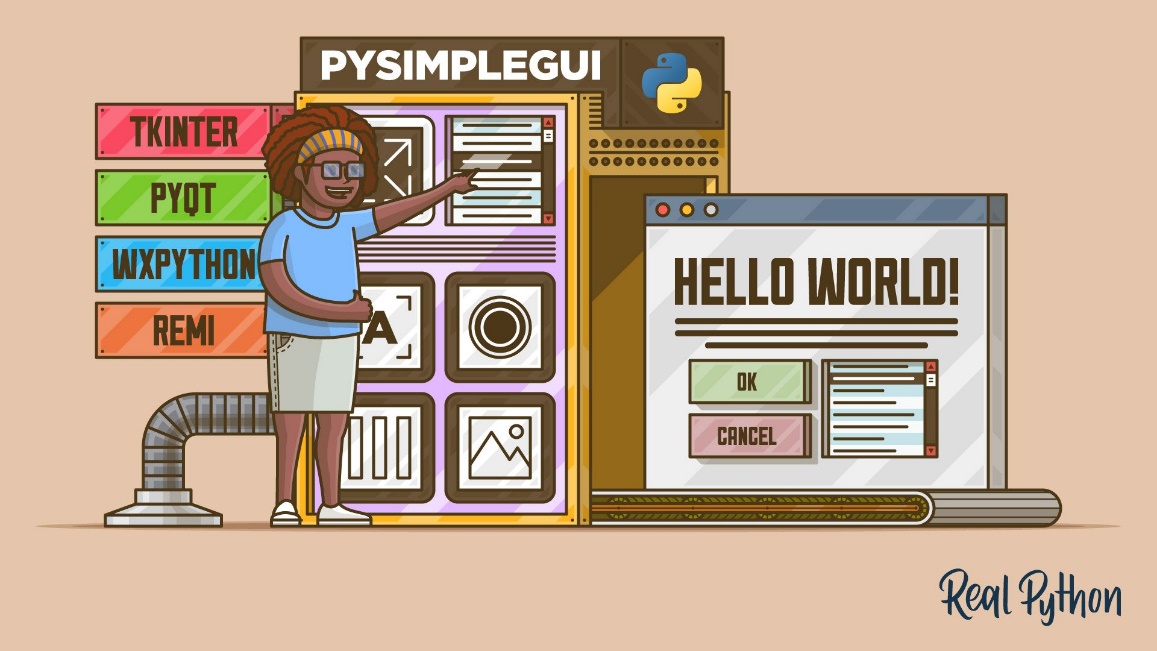
Finally, the code will call EndPaint to release the HDC and validate the invalid region of the client area.

If the window procedure does not process WM\_PAINT messages, it must pass the message to DefWindowProc. DefWindowProc will automatically call BeginPaint and EndPaint, effectively validating the invalid region and handling the painting process.

Here is an example of how to use the BeginPaint and EndPaint functions to draw a rectangle to the client area of a window:



This code will draw a black rectangle to the client area of the window. The rectangle will be 100 pixels wide and 100 pixels tall, and it will be positioned 10 pixels from the left edge of the window and 10 pixels from the top edge of the window.

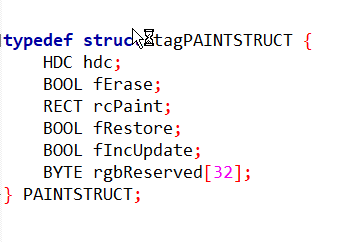


**PAINTSTRUCT STRUCTURE IN DEPTH**

The PAINTSTRUCT structure, referred to as "paint information structure" earlier, plays a crucial role in Windows programming. It holds essential information about the painting process and is populated by Windows when the BeginPaint function is called.

PAINTSTRUCT Structure Definition

The PAINTSTRUCT structure is defined as follows:



Key Fields of PAINTSTRUCT

* hdc: This field holds the handle to the device context (DC), which is a unique identifier for the window's drawing context.
* fErase: This Boolean flag indicates whether Windows has already erased the background of the invalid rectangle. If TRUE (nonzero), the background has been erased.
* rcPaint: This field is a RECT structure that defines the boundaries of the invalid rectangle. The values represent pixel coordinates relative to the client area's top-left corner.

Windows Initialization of PAINTSTRUCT Fields

When the BeginPaint function is called, Windows fills in the relevant fields of the PAINTSTRUCT structure:

* hdc: Windows assigns the device context handle to this field.
* fErase: In most cases, fErase will be FALSE (0), indicating that Windows has already erased the background.
* rcPaint: Windows sets this field to the coordinates of the invalid rectangle.

Programmatic Access to PAINTSTRUCT Fields

Your program can access and utilize only the first three fields of the PAINTSTRUCT structure: hdc, fErase, and rcPaint. The remaining fields are reserved for internal Windows operations.

Understanding the fErase Flag

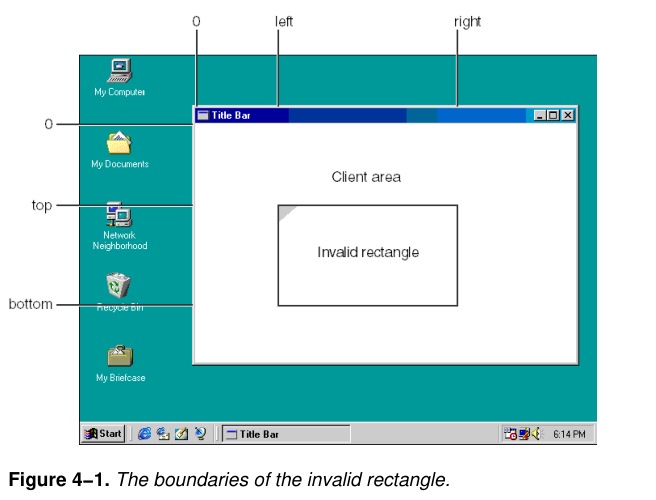
The fErase flag plays a crucial role in determining whether background erasing is necessary.

By default, Windows erases the background of the invalid rectangle before calling BeginPaint.

However, if your program invalidates a rectangle using InvalidateRect, specifying FALSE (0) as the last argument, Windows will not erase the background, and the fErase flag will be TRUE (nonzero) after calling BeginPaint.

Significance of rcPaint

The rcPaint field provides the coordinates of the invalid rectangle, defining the area that your program should repaint. This rectangle is expressed in pixel units relative to the client area's top-left corner.



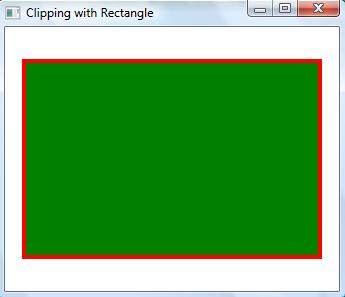
In summary, the PAINTSTRUCT structure serves as a vital information carrier during the painting process. It provides access to the device context handle, indicates whether background erasing is necessary, and defines the boundaries of the invalid rectangle, ensuring that your program paints efficiently and accurately.

The Clipping Rectangle: Beyond the Invalid Rectangle

The rcPaint rectangle in the PAINTSTRUCT structure serves not only as a representation of the invalid rectangle but also as a clipping rectangle.

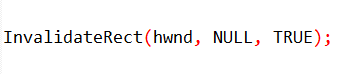
This means that Windows restricts painting operations to occur solely within the boundaries of the clipping rectangle.

In other words, even if the actual invalid region is not rectangular, Windows will still confine painting to the rectangular area defined by rcPaint.



Invalidating the Entire Client Area

To enable painting outside the invalid rectangle while handling WM\_PAINT messages, you can invalidate the entire client area using the InvalidateRect function before calling BeginPaint. This involves making the following call:

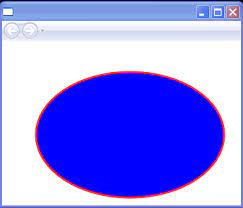


This code invalidates the entire client area, prompting Windows to erase the background when BeginPaint is called. Specifying TRUE as the last argument ensures background erasing. Alternatively, passing FALSE will preserve the existing background.

Convenience of Repainting the Entire Client Area

For most Windows programs, it's generally more efficient to simply repaint the entire client area whenever a WM\_PAINT message is received, regardless of the rcPaint structure's contents.

This approach is particularly advantageous when the client area contains graphical elements that extend beyond the invalid rectangle.

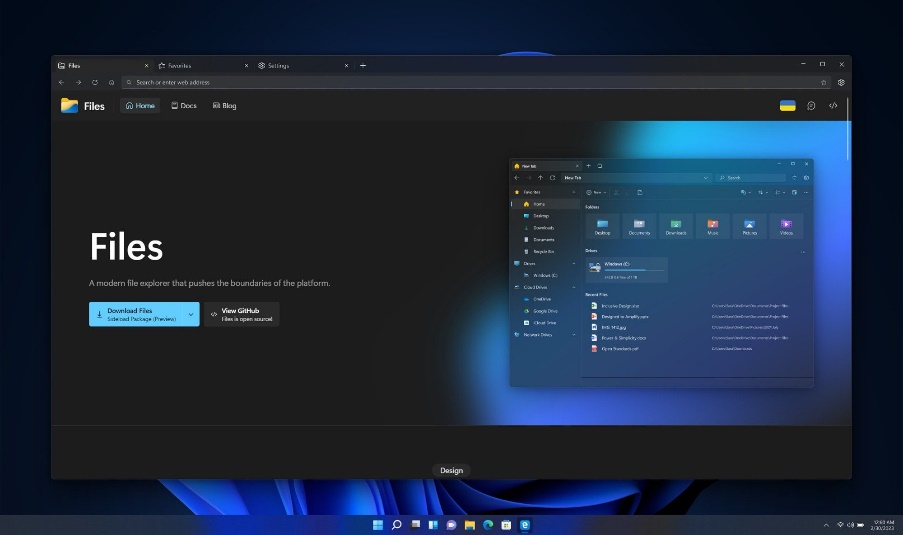


For instance, if a circle is partially within the invalid rectangle, it's more efficient to redraw the entire circle than just the invalid portion.

Windows' Limitation of Painting Outside rcPaint

Even if you draw outside the rcPaint rectangle, Windows will still clip the drawing operation, ensuring that nothing appears beyond the defined boundaries.

This behavior highlights the importance of considering the rcPaint rectangle when processing WM\_PAINT messages.



Performance Considerations

Programmers who prioritize performance and efficiency should utilize the invalid rectangle information during WM\_PAINT message processing to minimize unnecessary GDI calls.

This is especially crucial when painting operations involve accessing disk files, such as bitmaps. By adhering to the clipping rectangle, you can optimize rendering performance and avoid unnecessary file access.



Conclusion

The rcPaint rectangle plays a dual role in Windows programming, both as a representation of the invalid region and as a clipping rectangle that restricts painting operations. Understanding this concept and utilizing it effectively can lead to more efficient and performant code.

Method Two: Acquiring a Device Context Handle with GetDC and ReleaseDC

While Method One, using BeginPaint and EndPaint, is preferred for handling WM\_PAINT messages, Method Two offers flexibility for painting outside the invalid rectangle or obtaining device context handles for other purposes.

Obtaining a Device Context Handle with GetDC

The GetDC function retrieves a device context handle (HDC) for the client area of the specified window. The HDC is a unique identifier for the window's drawing context. To use GetDC, follow these steps:

* Call GetDC(hwnd), passing the window handle (hwnd) as an argument.
* Use GDI functions to render graphics on the client area using the obtained HDC.
* Call ReleaseDC(hwnd, hdc) to release the HDC and make it available for other applications.

Comparison with BeginPaint and EndPaint

Unlike BeginPaint and EndPaint, the GetDC and ReleaseDC functions should be called in pairs within the same message processing cycle.

**NB:** Avoid calling GetDC in one message and ReleaseDC in another.

Clipping Rectangle Considerations

The device context handle returned by GetDC has a clipping rectangle equal to the entire client area. This means you can paint on any part of the client area, unlike the restricted area defined by the rcPaint rectangle in BeginPaint.

Validating the Client Area

GetDC does not automatically validate any invalid regions. If you need to validate the entire client area, explicitly call ValidateRect(hwnd, NULL).

Typical Use Cases of GetDC and ReleaseDC

Primarily, GetDC and ReleaseDC are used to respond to keyboard or mouse messages, allowing real-time drawing in word processors or drawing programs without invalidating the client area.

Handling WM\_NCPAINT Messages for GetWindowDC

The GetWindowDC function returns a device context handle for drawing on the entire window, including the title bar. However, your program must also process WM\_NCPAINT ("nonclient paint") messages to handle repainting of non-client areas.

Conclusion

Method Two provides a flexible approach to acquiring device context handles for painting outside the invalid rectangle or for other purposes.

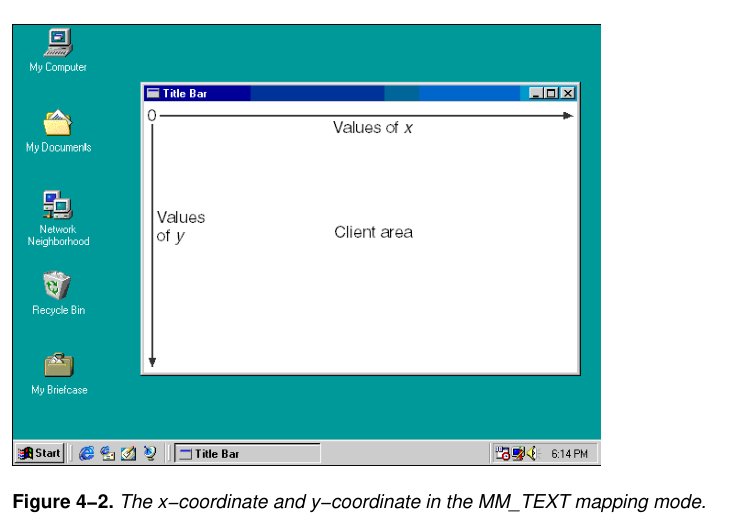
However, it's important to remember to properly pair GetDC and ReleaseDC calls and handle WM\_NCPAINT messages when using GetWindowDC.

THE TEXTOUT FUNCTION

The TextOut function is a fundamental tool in the GDI (Graphics Device Interface) library, enabling programmers to display text on the screen. Its syntax is concise and straightforward:



Breaking Down the TextOut Function Parameters:



**hdc:** This parameter represents the device context handle (HDC), which acts as a unique identifier for the window's drawing context. It can be obtained either from the GetDC function or from the BeginPaint function during WM\_PAINT message processing.

**x:** This parameter specifies the horizontal position of the text string's starting point within the client area. Values of x increase as you move to the right in the client area.

**y:** This parameter specifies the vertical position of the text string's starting point within the client area. Values of y increase as you move down in the client area.

**psText:** This parameter is a pointer to the character string that will be displayed. The string should not contain any ASCII control characters, as Windows will interpret them as non-displayable symbols.

**iLength:** This parameter indicates the number of characters in the psText string. For Unicode character strings, the number of bytes in the string is double the iLength value.

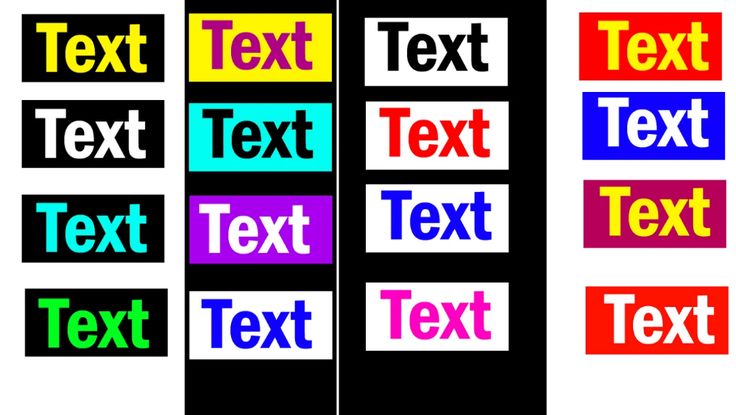
Understanding Device Context Attributes:

The attributes of the device context play a crucial role in determining the appearance of the displayed text. These attributes include:

Text Color: The default text color is black, but you can modify it using the SetTextColor function.



Text Background Color: The default text background color is white, and it fills in the rectangular space surrounding each character, known as the "character box."



Font: The font used to render the text is specified using the SelectObject function in combination with a created font object.

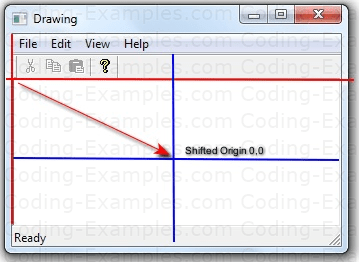


Coordinate System and Mapping Modes:

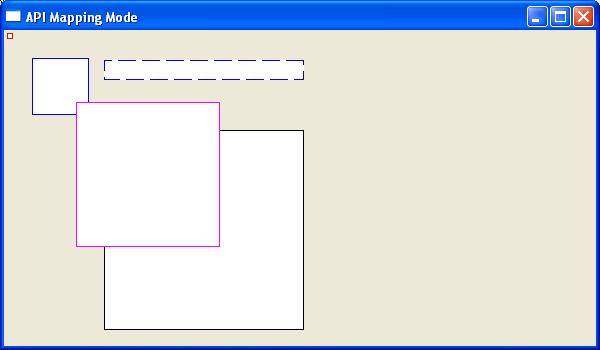
The x and y coordinates passed to TextOut represent logical coordinates, which are not directly translated to physical pixels on the display.

Windows employs various mapping modes to bridge the gap between logical and physical coordinates.

MM\_TEXT Mapping Mode: The default mapping mode, MM\_TEXT, directly equates logical units to physical pixels, making it convenient for text rendering.



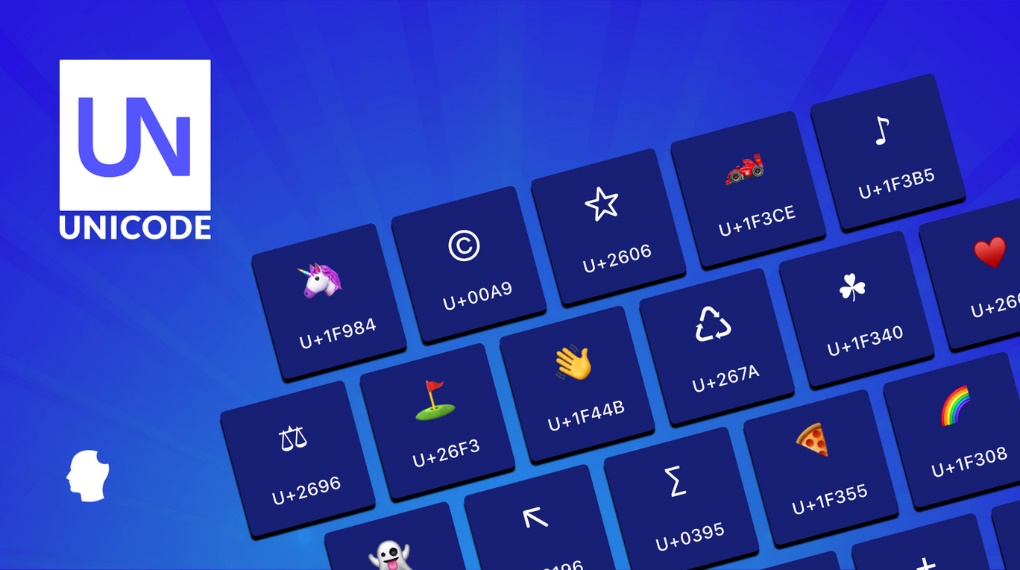
Other Mapping Modes: Other mapping modes, such as MM\_HIENGLISH and MM\_LOENGLISH, introduce scaling and offsetting to logical coordinates, influencing the text placement.



Unicode Support:

TextOut supports Unicode character strings, allowing for the display of multilingual text. The number of bytes in a Unicode string is double the iLength value, and the string should not contain any ASCII control characters.

We looked at unicode in-depth in chapter 1.



Clipping Region and Its Impact on Text Rendering

The device context, which serves as the drawing context for a window, also defines a clipping region.

This region determines the area within which graphical elements, including text, will be rendered.

By default, the clipping region is set to the entire client area for a device context handle obtained using GetDC and to the invalid region for a device context handle obtained using BeginPaint.

When the TextOut function is called to display text, Windows will only render the portions of the character string that fall within the clipping region.

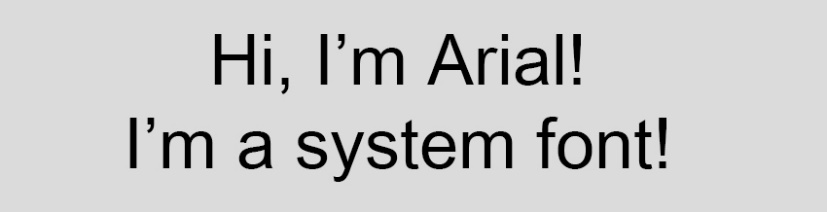
Any part of a character that lies outside the clipping region will not be displayed. Similarly, if a character partially overlaps the clipping boundary, only the portion of the character inside the region will be rendered.

This clipping mechanism ensures that text is displayed only within the visible boundaries of the window, preventing it from extending beyond the client area or overlapping with other graphical elements.

SYSTEM FONT

The device context also defines the default font that Windows uses when displaying text using the TextOut function.

This default font is known as the "system font" or, using the identifier defined in the WINGDI.H header file, SYSTEM\_FONT.



The system font is the font that Windows employs by default for text strings in title bars, menus, and dialog boxes.

It serves as a baseline font for text rendering and is commonly used throughout the Windows user interface.

Evolution of the System Font: From Fixed-Pitch to Variable-Pitch

In the early versions of Windows, the system font was a fixed-pitch font, meaning that all characters had the same width.

This was similar to the font used in typewriters. However, with the release of Windows 3.0, the system font transitioned to a variable-pitch font.



Variable-pitch fonts allow different characters to have different widths, reflecting their natural shapes and sizes.

This change from fixed-pitch to variable-pitch fonts was driven by research indicating that text displayed in variable-pitch fonts is more readable and visually appealing.

Impact of Variable-Pitch Fonts on Programming

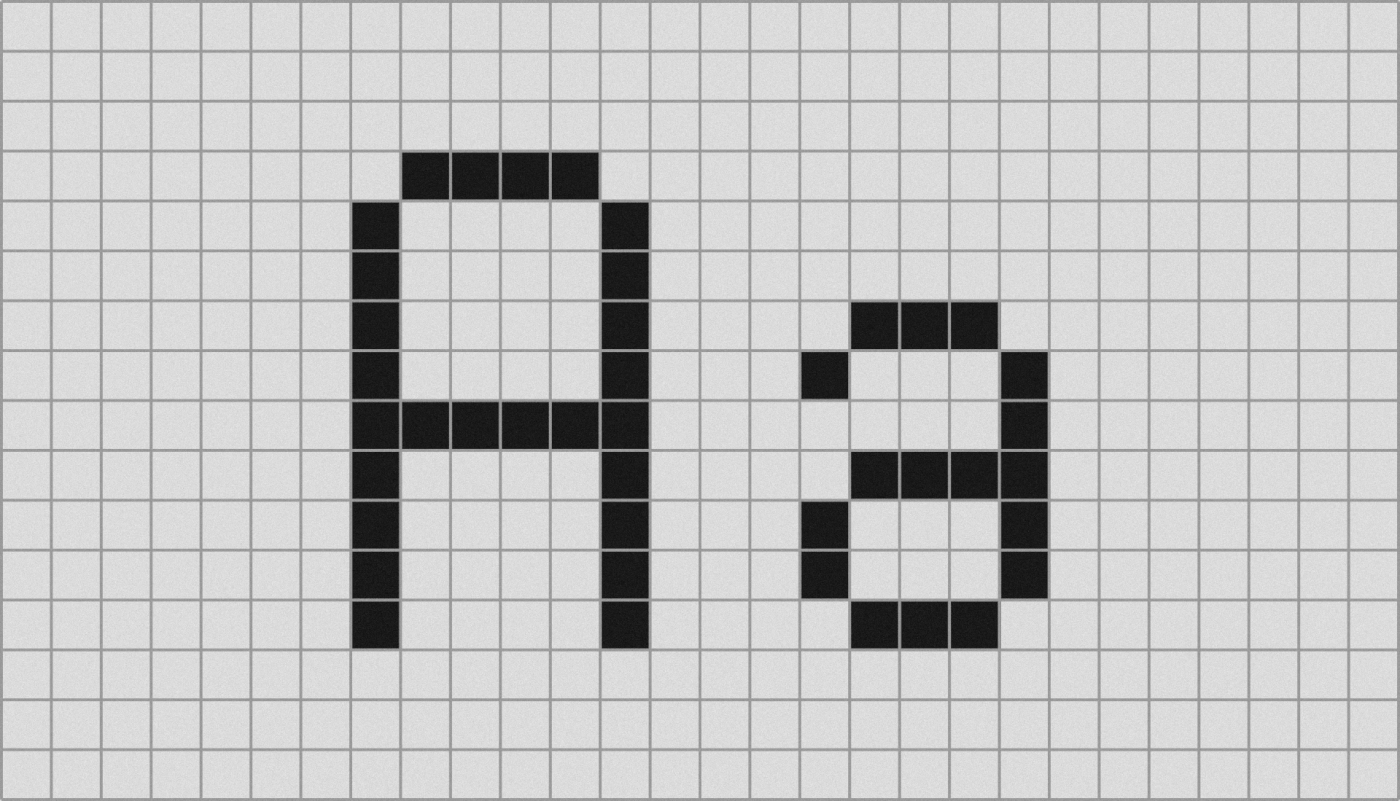
The adoption of variable-pitch fonts necessitated adjustments in programming practices.

Developers had to adapt their code to account for the varying widths of characters and ensure proper text alignment and formatting.

Raster Fonts vs. TrueType Fonts: A Distinction

The system font is a raster font, meaning that the characters are defined as blocks of pixels. This definition is tied to a specific resolution and may not scale well to different display sizes.

Play this video: 👇👇



In contrast, TrueType fonts, which we'll explore in the last Chapters, are defined by scalable outlines.

TrueType fonts are vector-based fonts, meaning that they are defined by mathematical formulas rather than fixed bitmaps.

They can be rendered at various resolutions without losing quality, making them more versatile for various display sizes and text rendering scenarios.



Conclusion

The device context's clipping region and the system font play crucial roles in text rendering within Windows applications.

The clipping region ensures that text is displayed only within the visible boundaries of the window, while the system font provides a default text rendering style. Understanding these concepts is essential for creating visually appealing and user-friendly Windows applications.

The TextOut function is an essential tool for displaying text in Windows applications. Its straightforward syntax and integration with device context attributes enable programmers to create visually appealing and informative user interfaces.

Understanding the concept of mapping modes and Unicode support further enhances the versatility of this function.

Also, read the GDI documentation.

CHARACTER DIMENSIONS AND TEXT RENDERING

When displaying multiple lines of text using the TextOut function, it is crucial to determine the dimensions of the characters in the font.

This information allows for proper spacing between lines and columns of text, ensuring a visually appealing and readable layout.

Dynamic Character Dimensions: Impact of Display Size and Font Selection

The dimensions of characters are not static and can vary depending on the pixel size of the video display.

Windows allows a range of display resolutions, such as 640x480, 800x600, and 1024x768.



Additionally, users can choose different font sizes for the system font. These factors contribute to the dynamic nature of character dimensions.

Determining Character Dimensions Using GetTextMetrics

To determine the character dimensions for a specific font, a program can utilize the GetTextMetrics function.

This function requires a handle to the device context, as it retrieves information about the font currently selected in that context.

The retrieved information is stored in a TEXTMETRIC structure, which contains various fields related to font metrics.

Key Fields of the TEXTMETRIC Structure. Among the 20 fields in the TEXTMETRIC structure, the following seven are particularly relevant for character dimensions:

* tmHeight: Represents the total height of a character, including both the ascent and descent.
* tmAscent: Indicates the portion of the character that extends above the baseline.
* tmDescent: Represents the portion of the character that extends below the baseline.
* tmInternalLeading: Refers to the additional spacing between lines of text within the tmHeight boundary.
* tmExternalLeading: Represents the additional spacing between lines of text outside the tmHeight boundary.
* tmAveCharWidth: Indicates the average width of characters in the font.
* tmMaxCharWidth: Represents the width of the widest character in the font.

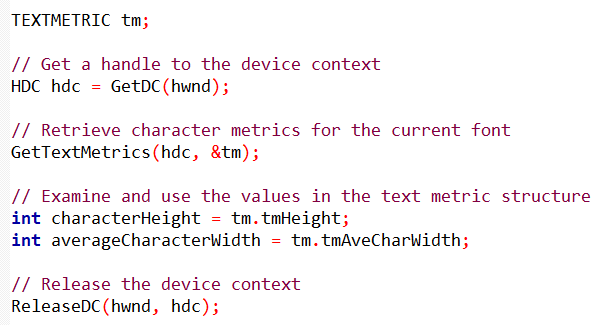
Units of Measurement for Character Dimensions

The values in the TEXTMETRIC structure are measured in units based on the mapping mode currently selected for the device context.

In the default device context, the mapping mode is MM\_TEXT, meaning the dimensions are in pixels.

Code Example for Retrieving Character Dimensions

The following code snippet demonstrates how to retrieve character dimensions using GetTextMetrics:





This code uses the Windows API to obtain the default font on the system, creates a font from that information, selects it into a device context, and then retrieves the text metrics using GetTextMetrics. Finally, it prints out the various fields of the TEXTMETRIC structure.

Please note that this code assumes you are working in a Windows environment, as it relies on Windows API functions. If you are using a different platform, you might need to use a different approach.

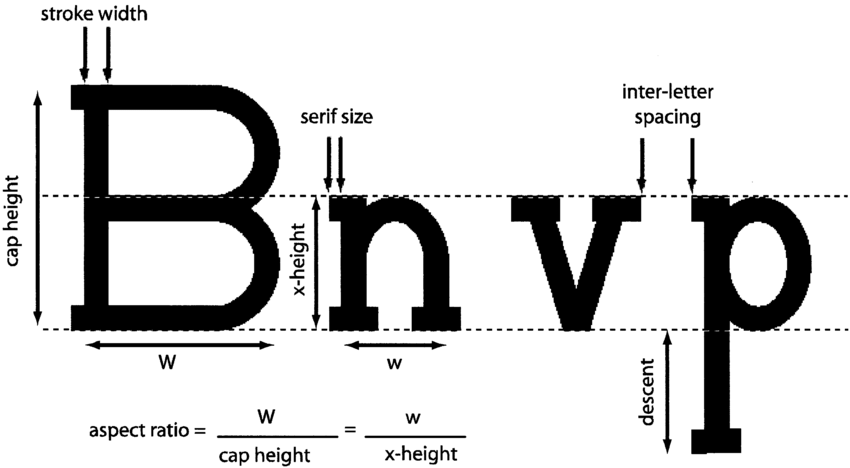
Delving into the TEXTMETRIC Structure for Character Dimensions

The TEXTMETRIC structure provides comprehensive information about the font currently selected in the device context.

Among its various fields, four are particularly relevant for understanding character dimensions: tmHeight, tmAscent, tmDescent, and tmInternalLeading.

tmHeight: Defining the Overall Character Height

The tmHeight field represents the total height of a character, encompassing both the ascent and descent. It serves as a crucial parameter for determining the vertical spacing between lines of text.



tmAscent and tmDescent: Character Extents Above and Below the Baseline

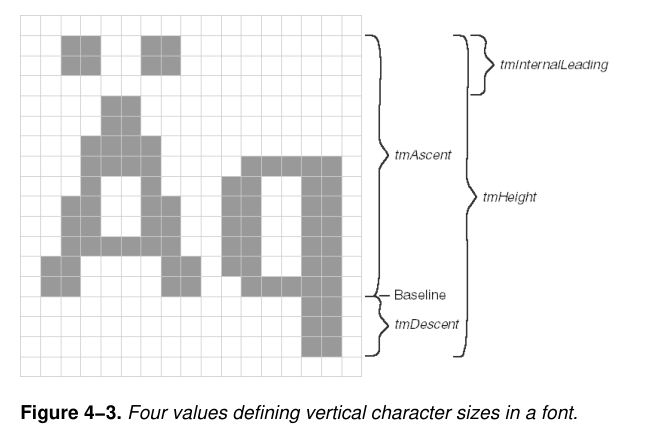
The tmAscent field indicates the portion of the character that extends above the baseline, while tmDescent represents the portion that extends below the baseline. These values are essential for proper positioning of text relative to the baseline.

tmInternalLeading: Accounting for Accent Marks

tmInternalLeading refers to the additional spacing between lines of text within the tmHeight boundary. This spacing typically accommodates accent marks, which are small characters placed above or below other characters.

External Leading: An Optional Spacing Suggestion

The TEXTMETRIC structure also includes a field named tmExternalLeading, which represents an additional spacing suggestion from the font designer. This spacing is intended to be added between successive rows of displayed text. Programmers have the flexibility to accept or reject this suggestion based on their desired text layout.



Character Widths: Average and Maximum

Character widths are represented by two fields in the TEXTMETRIC structure: tmAveCharWidth and tmMaxCharWidth. tmAveCharWidth indicates the average width of lowercase characters, while tmMaxCharWidth represents the width of the widest character in the font.

Calculating Uppercase Character Width

For the sample programs in this chapter, the average width of uppercase letters is required. This value can be approximated by calculating 150% of tmAveCharWidth.

Dynamic Character Dimensions: Adapting to Display Size and Font Selection

It is crucial to recognize that the dimensions of a system font are not static and can vary depending on the pixel size of the video display on which Windows runs.

Additionally, some users may have customized the system font size, further influencing character dimensions.

Importance of Accurate Character Dimensions

Relying on guesswork or hard-coded values for character dimensions can lead to inconsistent and visually unappealing text layouts across different display setups and user preferences.

Role of GetTextMetrics in Obtaining Accurate Character Dimensions

The GetTextMetrics function provides a reliable and device-independent mechanism for retrieving accurate character dimensions. By utilizing this function, programmers can ensure that their text rendering adapts to various display configurations and font selections.

In summary, the TEXTMETRIC structure offers valuable insights into font metrics, particularly character dimensions. Understanding the significance of each field and employing the GetTextMetrics function empowers programmers to create visually consistent and user-friendly text layouts.

Formatting text and optimizing GetTextMetrics usage:

Efficient Utilization of GetTextMetrics for Text Formatting

Since the dimensions of the system font remain constant throughout a Windows session, it is unnecessary to repeatedly call GetTextMetrics.

A single call during program initialization is sufficient.

An ideal location for this call is within the window procedure's response to the WM\_CREATE message, the first message received by the window procedure.

Defining Variables for Character Dimensions

Consider a Windows program that displays multiple lines of text within the client area.

To calculate the appropriate spacing between lines and characters, the average character width (cxChar) and total character height (cyChar) need to be determined.

These values can be stored in static variables defined within the window procedure: