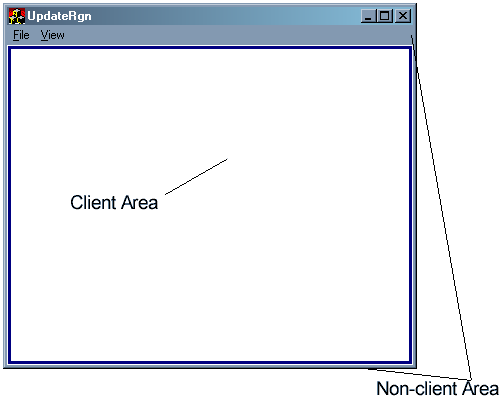
**CHAPTER 4**

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.

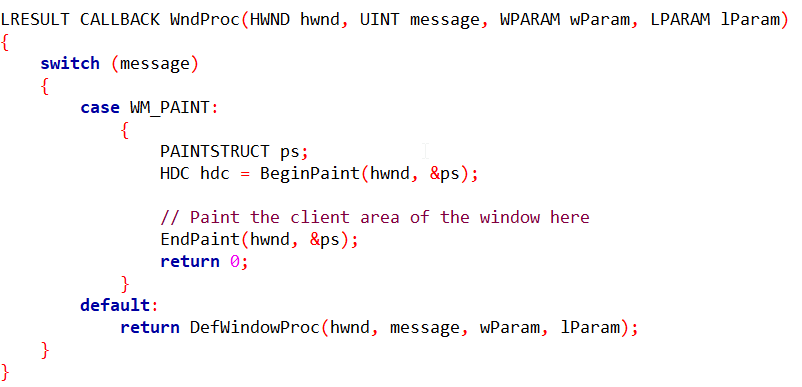
Windows always saves the area of the display it overwrites and then restores it in a few cases, such as when the mouse cursor is moved across the client area or when an icon is dragged across the client area.

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.

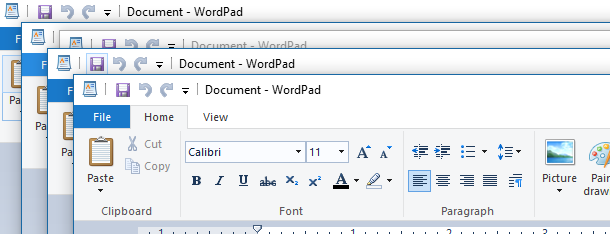
However, in Windows, the situation is more complex. Programs can only draw on the client area of their own window, a designated rectangular region within the overall window frame.

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:**

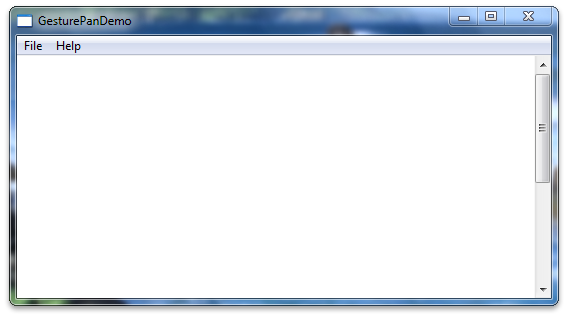
Revealing Previously Hidden Areas: When a hidden portion of the window is brought into view, either by moving the window or uncovering it from behind another window, Windows will send a WM\_PAINT message to the window procedure. This message signals the need to redraw the exposed area.



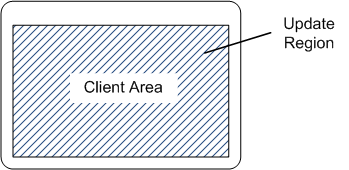
Window Resizing: If the user resizes the window, and the window class style has the CS\_HREDRAW and CW\_VREDRAW bits set, Windows will again send a WM\_PAINT message. This ensures that the client area adapts to the new window size.



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.

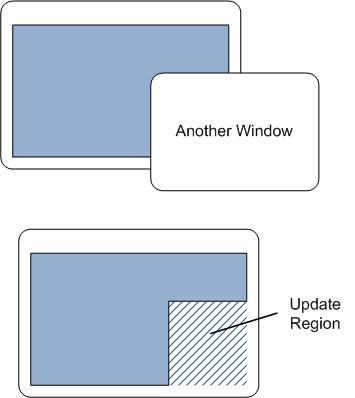


Mouse Cursor and Icon Dragging: In a few specific situations, Windows always saves and restores the overwritten area, triggering a WM\_PAINT message only when necessary. These cases include moving the mouse cursor across the client area or dragging an icon within the client area.



**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.



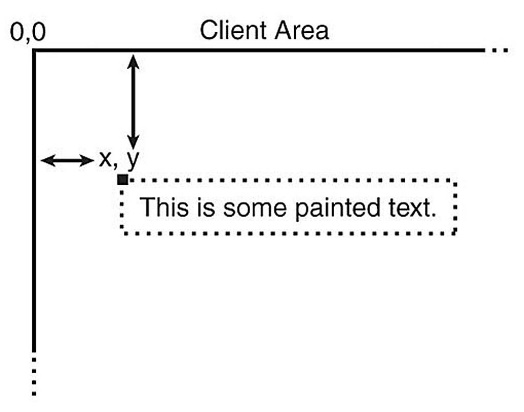
**Forcing Repaints**

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.



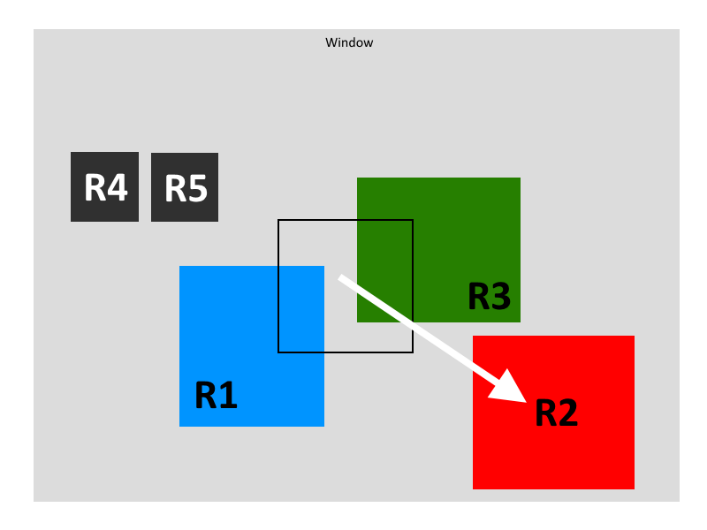
**Valid and Invalid Rectangles**

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.



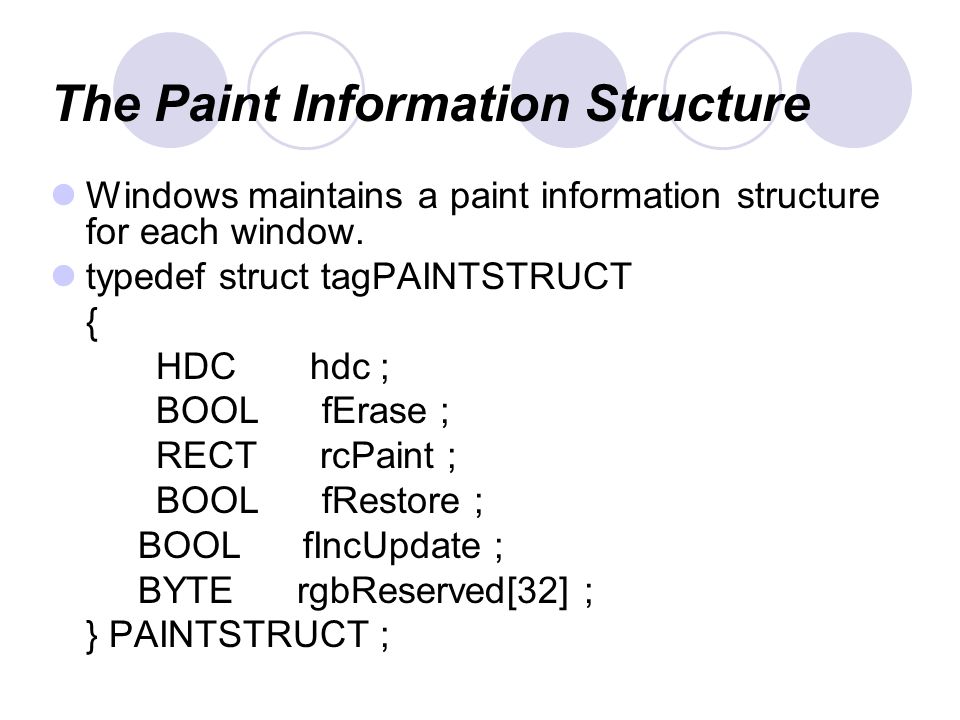
**Invalid Region**

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.



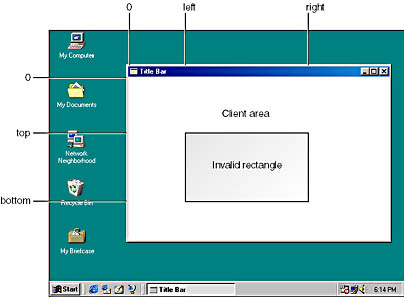
**Paint Information Structure**

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."



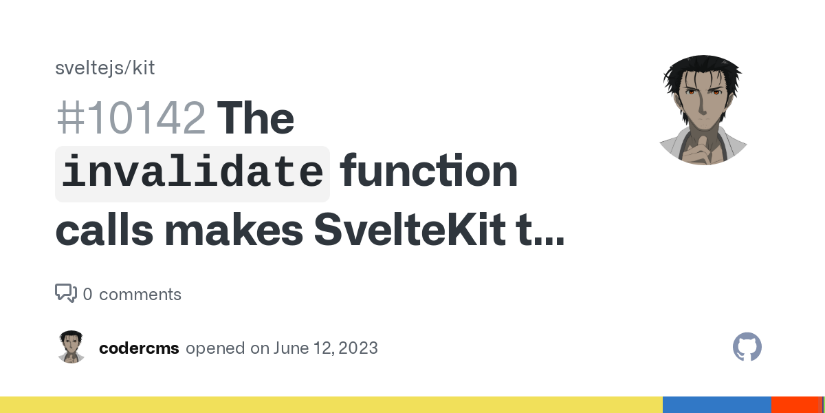
**Invalid Rectangle Updates**

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.



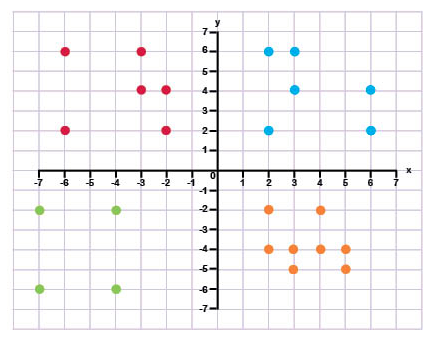
**Invalidating Rectangles**

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**

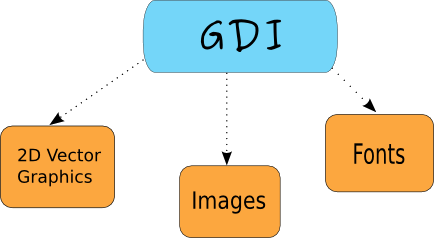
Once the window procedure calls BeginPaint during the WM\_PAINT message, the entire client area is considered validated. A program can also explicitly validate any rectangular area within the client area by calling the ValidateRect function. If this call effectively validates the entire invalid area,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



**INTRODUCTION TO GDI**

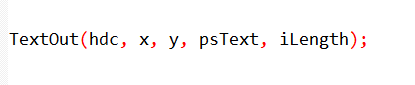
The Graphics Device Interface (GDI) is a set of functions provided by Windows for drawing text, graphics, and other visual elements on the screen.

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

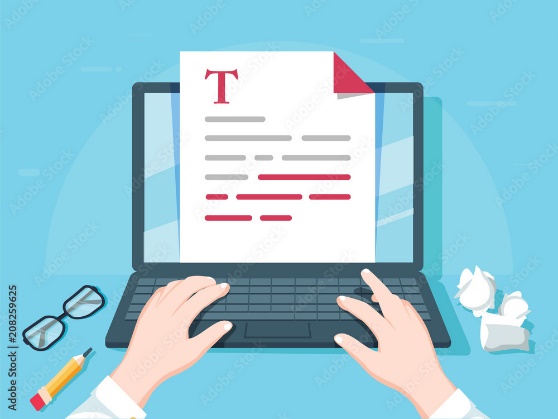
B

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.



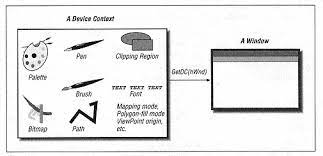
The Device Context: A Crucial GDI Element

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.



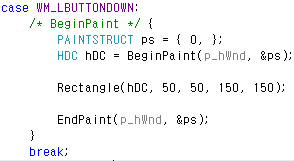
Releasing the handle invalidates it and prevents its further use. The program should acquire and release the handle within the processing of a single message.

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.

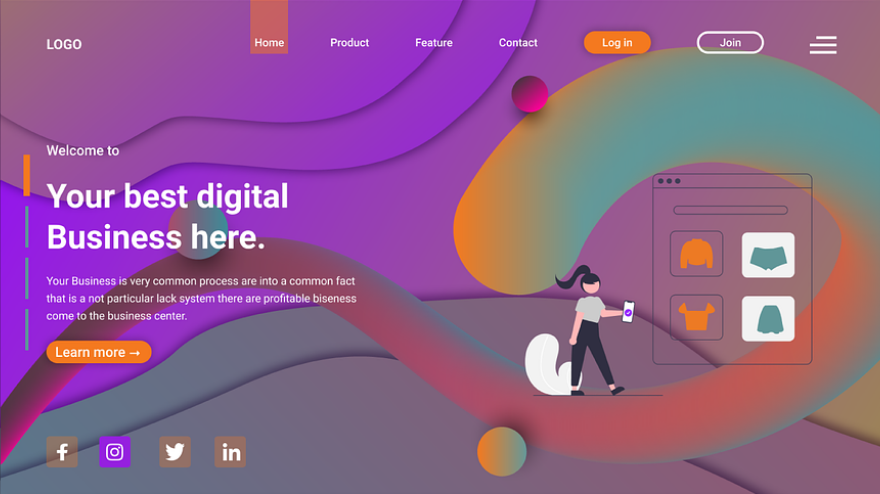
****

Figure : 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.

