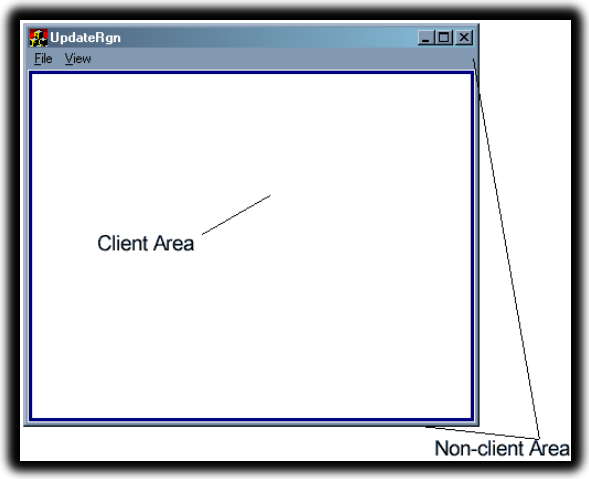
CHAPTER 4: DEALING WITH TEXT OUTPUT

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.

What is the Client Area?

The **client area** is the part of the window where your app does the actual drawing — think of it as your painting canvas.

It's everything **except** the title bar, borders, and scroll bars.



What is Painting?

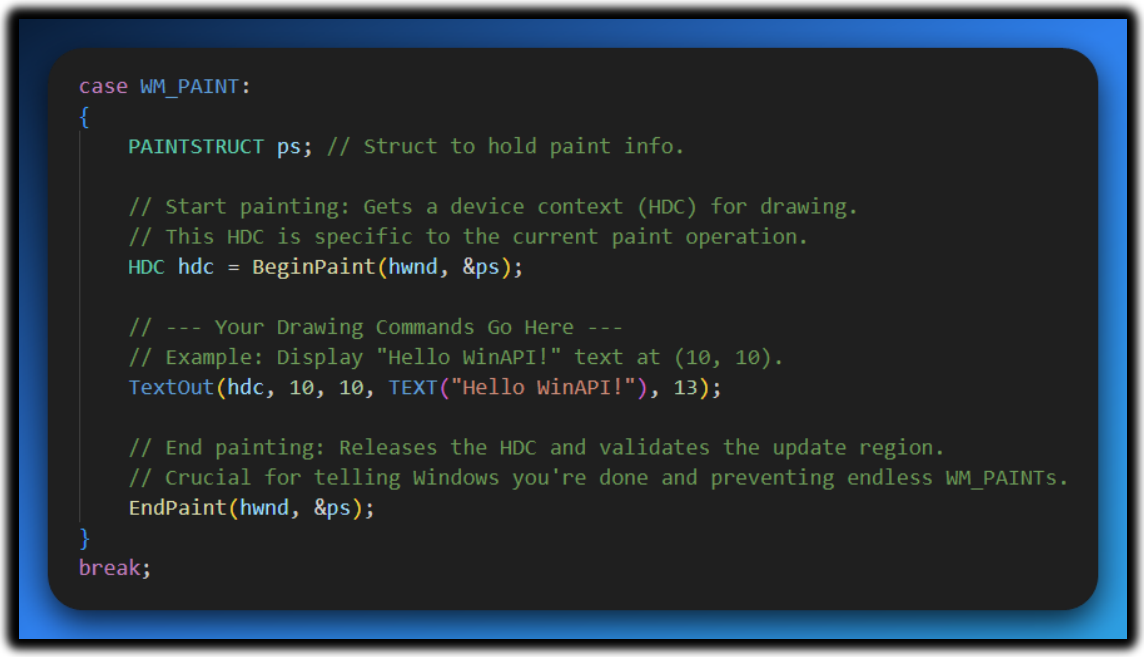
The process of displaying text or graphics in a Windows program's client area is called "painting."

We don’t draw whenever we want — Windows controls **when** we’re allowed to paint, by sending us a message: WM_PAINT.

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

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.



```
case WM_PAINT:
{
    PAINTSTRUCT ps; // Struct to hold paint info.

    // Start painting: Gets a device context (HDC) for drawing.
    // This HDC is specific to the current paint operation.
    HDC hdc = BeginPaint(hwnd, &ps);

    // --- Your Drawing Commands Go Here ---
    // Example: Display "Hello WinAPI!" text at (10, 10).
    TextOut(hdc, 10, 10, TEXT("Hello WinAPI!"), 13);

    // End painting: Releases the HDC and validates the update region.
    // Crucial for telling Windows you're done and preventing endless WM_PAINTs.
    EndPaint(hwnd, &ps);
}
break;
```

HDC (Device Context): Think of this as your canvas. All GDI (Graphics Device Interface) drawing functions need an HDC to know where to draw.

✅ Always use BeginPaint() / EndPaint() inside WM_PAINT.

🧠 Device-Independent Programming

Device-independent programming is the practice of writing software that can run on a variety of hardware and software configurations. Different screens, resolutions, DPI settings... your code needs to adapt.

Tips:

- *Use system font (GetStockObject(SYSTEM_FONT))*
- *Use window metrics (GetClientRect, GetDeviceCaps) to stay responsive*
- *Avoid hardcoded positions and sizes*

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.

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

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_PAINT:
            {
                PAINTSTRUCT ps;
                HDC hdc = BeginPaint(hwnd, &ps);

                // Paint the client area of the window here
                EndPaint(hwnd, &ps);
                return 0;
            }
        default:
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
}
```

🔷 HWND hwnd

Handle to the window that got the message. Think of this as the *unique ID* for your window.

🔷 UINT message

Tells you *what happened*. Could be:

- WM_PAINT → Time to redraw
- WM_KEYDOWN → Key pressed
- WM_MOUSEMOVE → Mouse moved

🔷 WPARAM wParam & LPARAM lParam

Extra info, depending on the message:

- If it's a keypress, wParam might be the key code.
- If it's mouse movement, lParam might pack the (x, y) coords.

🧠 Message Handling Logic

You handle these messages using a switch (message) block:

```
switch (message)
 {
 // You can comment out this part because PlaySound requires the winmm.lib library,
 // which is not linked by default in most Windows projects.
 // Unlike kernel32.lib, user32.lib, and gdi32.lib (which are included automatically),
 // winmm.lib is an external multimedia library that handles sounds and music.
 // If you don't link winmm.lib, calling PlaySound will cause linker errors.
 // So, unless you explicitly link winmm.lib in your project settings,
 // it's safer to comment out PlaySound to avoid build problems.
 case WM_CREATE:
     //PlaySound(TEXT("hellowin.wav"), NULL, SND_FILENAME | SND_ASYNC);
     return 0;

 case WM_PAINT:
 {
     PAINTSTRUCT ps; // Struct to hold paint info.

     // Start painting: Gets a device context (HDC) for drawing.
     // This HDC is specific to the current paint operation.
     HDC hdc = BeginPaint(hwnd, &ps);

     // --- Your Drawing Commands Go Here ---
     // Example: Display "Hello WinAPI!" text at (10, 10).
     TextOut(hdc, 10, 10, TEXT("Hello WinAPI!"), 13);

     // End painting: Releases the HDC and validates the update region.
     // Crucial for telling Windows you're done and preventing endless WM_PAINTs.
     EndPaint(hwnd, &ps);
     return 0;
 }

 case WM_DESTROY:
     PostQuitMessage(0);
     return 0;
 }
```

🎨 WM_PAINT – Drawing Time

When your window needs to update its client area (the drawable part), Windows sends WM_PAINT.

🛠️ What Triggers WM_PAINT?

- Window is resized.
- A covered section becomes visible.
- A tooltip disappears.
- You manually request repainting (InvalidateRect).

🧱 The Painting Steps (explained line-by-line)

📦 PAINTSTRUCT ps;

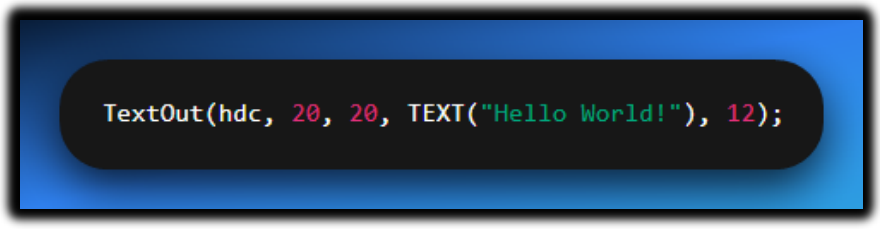
This holds data about the area that needs to be redrawn.
Windows fills this when you call BeginPaint().

🖌️ BeginPaint(hwnd, &ps);

Starts the painting.
Returns an HDC (Handle to Device Context) — this is your *drawing tool*.
You'll use it to draw text, shapes, etc.

🖼️ Draw Your Content

Between BeginPaint() and EndPaint(), do things like:



🧼 EndPaint(hwnd, &ps);

Always call this when you're done.
It releases the HDC and tells Windows: "I'm done painting."

🚪 DefWindowProc – The Default Handler

Not every message is one **you** need to handle yourself.

If your window gets a message you **don't explicitly process**, you need to *pass it along* to DefWindowProc(). This function is built into Windows — it knows how to handle the boring but critical stuff like:

- Moving the window around.
- Resizing.
- Clicking the close ❌ button.
- Minimizing/maximizing.
- System commands (like Alt+F4).

If you don't call it for unhandled messages, weird stuff happens — your window might stop behaving like a normal window. It won't close right, it won't respond to default key combos, and users will think it's broken 💀.

🧠 What You're Passing In:

```
return DefWindowProc(hwnd, message, wParam, lParam);
```

This ensures default behavior (like dragging, closing, etc.) still works.

- **hwnd:** Which window this is about
- **message:** The type of message (e.g. WM_CLOSE, WM_MOVE, etc.)
- **wParam / lParam:** Any extra info tied to the message — depends on the message type.

Basically, you're saying:

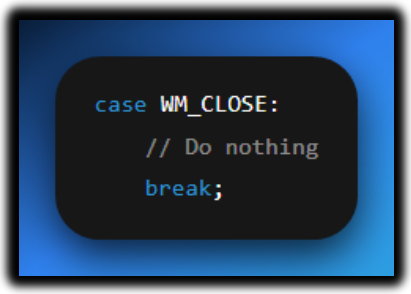
"Hey Windows, I don't know what to do with this message, so *you* deal with it."

🧪 Real Example

1. **User clicks the ❌ (close) button** on your app window.
2. Windows **sends a WM_CLOSE message** to your window procedure (WndProc).
3. Now you have two choices in your WndProc function:
   - ✓ **Handle WM_CLOSE yourself** (maybe ask "Are you sure?" with a message box, or save work).
   - ✓ **Or** if you **don't care to do anything special**, **pass it to DefWindowProc**.

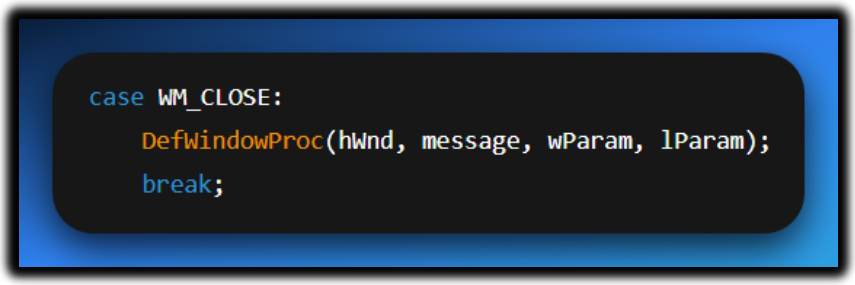
🔍 What happens if you do *not* pass WM_CLOSE to DefWindowProc?

If your WndProc looks like this:



Then Windows will NOT close your window. Your app stays open, but you can't interact with it. It's now a zombie window — looks alive, but it doesn't respond or close.

✅ Correct Way (if you don't want to do anything special):



Or, more commonly, let it fall through to the default handler:

```
default:
    return DefWindowProc(hWnd, message, wParam, lParam);
```

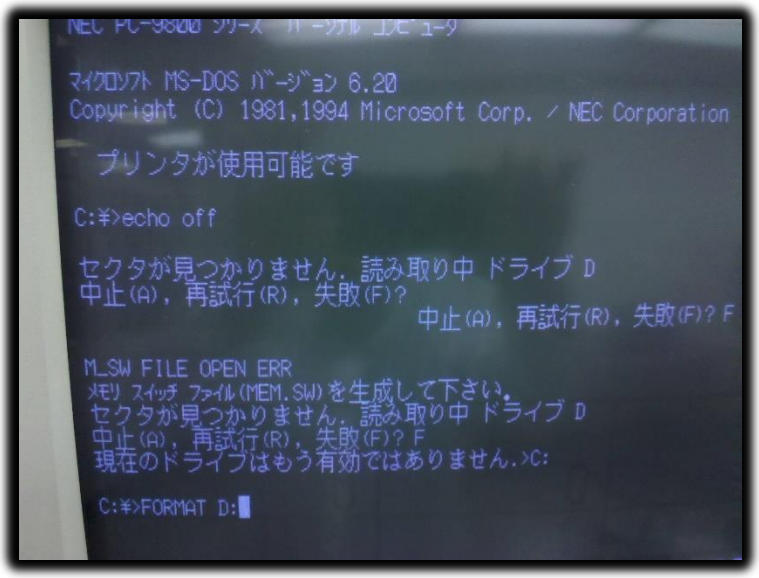
This ensures Windows does its job: destroying the window, sending WM_DESTROY, and eventually posting a WM_QUIT to end the message loop.

🧹 Bottom Line

Yes — if you don't handle WM_CLOSE, and you don't pass it to DefWindowProc, your window won't close. Always call DefWindowProc for messages you don't explicitly handle.

CLIENT AREA PAINTING

Back in the day — in good old DOS or other character-mode environments — you could draw *anywhere* on the screen.



The entire display was yours: a giant grid of characters or pixels. You could scribble on it, and whatever you wrote stayed put until you changed it.

🎨 Total control. No rules.

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

They could 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.

Windows is much complex, so 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.

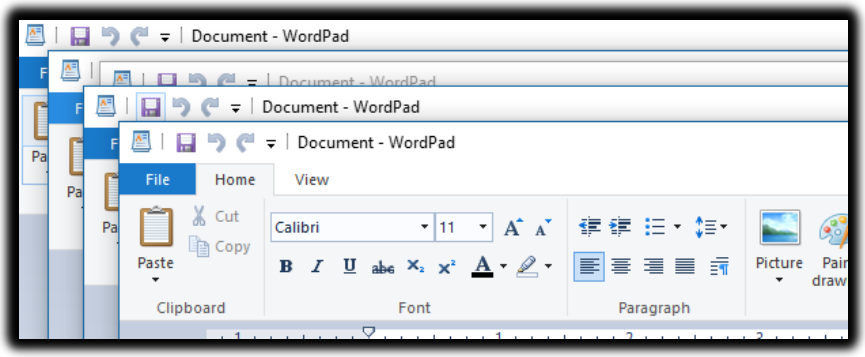
Additionally, the content of the client area is not guaranteed to persist indefinitely.

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.

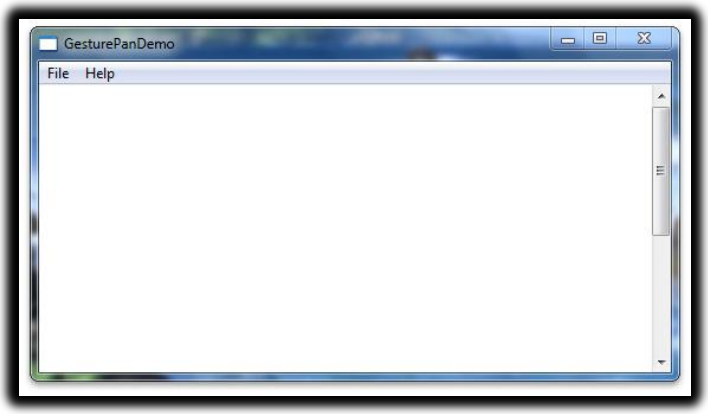
Windows tells your program:

"Hey, that section was in the shadows, now it's in the light — redraw it."



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.

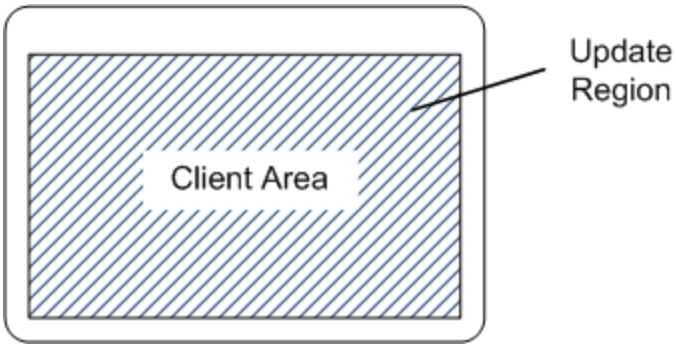
Scrolling: When a program uses the ScrollWindow or ScrollDC functions to scroll part of the client area, Windows will generate a WM_PAINT message to update the visible portion of the client area.



Explicit Invalidation: Programs can explicitly request a repaint of specific areas of the client area using the InvalidateRect or InvalidateRgn functions.

✅ Use InvalidateRect(hwnd, NULL, TRUE) — to redraw everything
✅ Or use InvalidateRgn() for custom shapes

This is great when your internal data changes and the screen needs to match.



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 a popup 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.

In short, paint repaint the area after the overlay is gone, restore the visuals beneath, just to make sure.

Mouse Cursor and Icon Dragging:

This is one of the rare **"clean" overwrite cases**.
When the mouse or an icon drags over your client area:

🖱️ Windows **always saves and restores** the bits underneath — *no repaint needed*.
🧠 Unless something actually changed, no WM_PAINT is fired. Efficient and smart.

⚠️ Final Pro Tip

Always assume a WM_PAINT could hit **any time**.
Be ready to redraw *everything needed*, even if you just painted it a second ago.
That's the **Windows way**.

DEALING WITH WM_PAINT MESSAGES

🎯 Understanding How to Handle WM_PAINT Properly

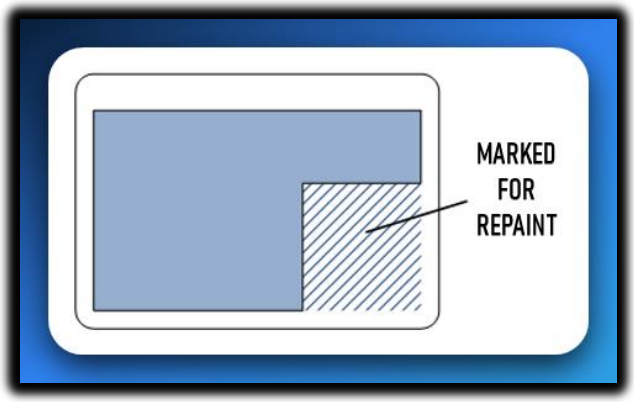
🧠 It's Not About Drawing Whenever You Want

In older systems (think DOS or bare metal), if you wanted something on screen — you drew it *right then and there*.
✏️ You directly touched the screen memory and said:
"Put this text here. Draw that shape there."
Done.

But in **Windows**, you **don't control the screen** like that anymore.
You're playing in a **multitasking, window-sharing, constantly redrawing OS**.

📼 In Windows: You Wait For the WM_PAINT Signal

When your app wants to update the visuals:
❌ **Don't draw right away.**
✅ **Instead, mark the area as needing a repaint.**



🚦 Then, when Windows sends a WM_PAINT message, *that's your* ***green light.***
That's when you actually do the drawing.

Real-World:

Think of it like a traffic system. When cars approach an intersection (like your application windows), they wait for the green light (the WM_PAINT message). The green light signals that it's their turn to move.

You don’t let each car move whenever it feels like it—you wait for a coordinated signal (like the WM_PAINT message), and only then do you allow all the cars (representing drawing operations) to proceed in an organized, synchronized way.

Just like traffic lights control the flow of cars, WM_PAINT controls the flow of painting operations.

Windows doesn’t start rendering (or painting) until it’s explicitly told to do so, ensuring everything is done at the right time and in the right order.

This keeps the process smooth and avoids chaos, making sure that multiple requests don’t interfere with each other.

💡 Why This Weird Flow?

Because Windows is juggling tons of things:

- Other apps
- Minimizing and restoring windows
- Moving stuff around
- Tooltips, dialogs, drag-drops — all fighting for screen space

So, it’s **Windows** that controls ***when*** something is drawn.

You just tell it:

“Hey, I changed something—next time you're redrawing, include this region I’ve marked for repaint,” says the program.

And how fast does Windows handle these repaints?

“Fast enough that we hardly even notice them. It’s so seamless that the updates often feel instant, and we’re rarely aware of the precise moment the screen refreshes.”

Do all Windows apps, even those written in other languages and frameworks like Qt, C#, Tkinter, and Electron, rely on WinAPI concepts, just in a higher-level way?

Yes! All Windows apps, whether using low-level WinAPI (like in Charles Petzold's examples) or high-level frameworks like C#, Qt, Tkinter, or Electron, are ultimately built on WinAPI concepts.

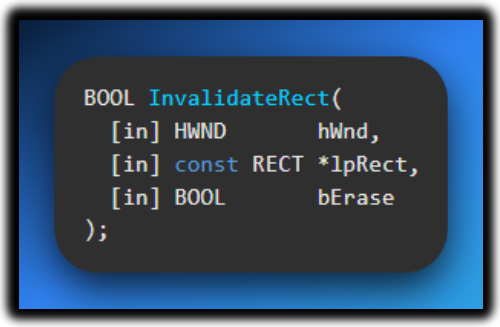
These frameworks abstract the complexity but still depend on Windows' core system calls for window creation, message handling, painting, and user input.

- **WinAPI (C, Petzold):** Direct interaction with the Windows message loop and handling events like WM_PAINT yourself.
- **C# (WinForms, WPF):** Handles high-level UI details but still relies on Windows' message system for painting and event dispatching.
- **Qt:** Cross-platform but on Windows, it still uses WinAPI behind the scenes.
- **Tkinter:** Python's Tk toolkit wraps around WinAPI for window management and events.
- **Electron:** Runs on Chromium and Node.js, but for windowing and events, it depends on WinAPI concepts on Windows.

In short, even though these frameworks make things easier, they all tap into the same underlying Windows mechanisms.

InvalidateRect

A function in Windows programming that adds a rectangle to a window's update region, signaling that the **specified area needs to be redrawn.** It doesn't immediately trigger a repaint, but rather **accumulates changes** to be processed later when a **WM_PAINT message** is sent or when **ValidateRect** is called. The **ValidateRect function** removes a rectangle from the update region, essentially marking it as already redrawn.



- **Key parameters:**
  - ✓ **hWnd:** A handle to the window to invalidate.
  - ✓ **lpRect:** A pointer to a RECT structure defining the rectangle to invalidate, or NULL to invalidate the entire client area.
  - ✓ **bErase:** A boolean indicating whether to erase the background of the invalidated area.
- **Return value:** Nonzero indicates success; zero indicates failure.

```
// Example usage:
RECT rectToInvalidate = {10, 10, 50, 50};
InvalidateRect(hwnd, &rectToInvalidate, TRUE); // Invalidate the rectangle and erase the background
```

This doesn't paint instantly — it **adds a WM_PAINT to the message queue**. Windows will call you back soon to handle it. This may feel like you're giving up control — but trust, it's for the better.

You're now coding in a way that's clean, predictable and works with the OS, not against it.

Structured drawing = stable apps.

Let the OS decide ***when*** to paint. You (your program) just decides ***what*** to paint.

🧩 Valid vs Invalid Regions — What's Dirty?

🟥 Invalid Region

An **invalid region** is just a part of your window that needs to be redrawn because something changed there.

Windows keeps track of it silently in the background.

The moment you:

- *Resize your window*
- *Uncover a hidden section e.g. previously covered by dialog boxes and other windows.*
- *Invalidate a rectangle yourself, Boom — that region becomes "invalid."*

As soon as one of these things happens, Windows immediately marks that region as "invalid" — meaning it needs to be redrawn.

🧠 You don’t need to memorize this. Just remember:

No WM_PAINT = No invalid region = No redrawing.

That means, If Windows doesn’t send a WM_PAINT message, it means nothing needs to be redrawn, because no invalid regions have been marked.

✅ Valid Region

After you **handle the WM_PAINT message** (which means you’ve drawn the content that needed updating), you **call EndPaint()** to tell Windows that you're done with the drawing.

When you do that, **the invalid region is cleared** — meaning the part of the window that needed a redraw is now considered **valid** again.

- **Valid region** = No more painting needed for this part until something changes again (like resizing or uncovering it).
- When you finish painting, your window has a **clean slate**: everything that needed updating is now updated, so there’s nothing left for the system to redraw until something changes again.

In simpler terms:

- **Before you paint**: The area of the window that needs updating is marked as **invalid** (it needs to be redrawn).
- **After you paint**: Once you finish drawing and call EndPaint(), the system **clears the invalid region**, and that part of the window is now considered **valid** again.
  Windows won’t ask you to repaint that area until the next time it needs to change.

So after handling WM_PAINT and calling EndPaint(), **there's no more redrawing needed until something else happens** to make the window "invalid" again.

🚨 Forcing Repaints (When You Gotta Break the Flow)

Sometimes, you can't wait for Windows to decide when to repaint — you need an immediate update to reflect some change in your UI, like when a button changes state or something dynamic happens that needs to be shown right away.

You can:

1. Invalidate the entire client area (the whole window):

```
InvalidateRect(hwnd, NULL, TRUE); // Mark the entire client area as dirty
```

OR

2. Invalidate just a specific region that needs to change:

```
// Example usage:
RECT rectToInvalidate = {10, 10, 50, 50};
InvalidateRect(hwnd, &rectToInvalidate, TRUE); // Invalidate the rectangle and erase the background
```

🎨 The Paint Information Structure: Windows' Dirty Canvas Tracker

Whenever Windows tells you **"Hey! Redraw this area!"** via WM_PAINT, it's not just shouting into the void —

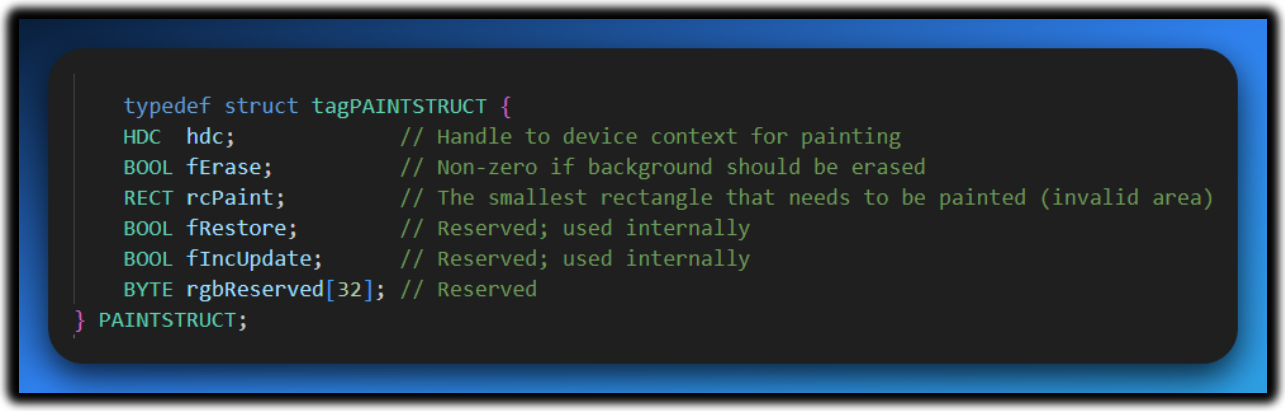
It quietly hands you a **little bundle of info** behind the scenes: the PAINTSTRUCT.

This structure is what Windows uses to manage repainting —
it's like a "painting invoice" saying:

"Here's what needs redrawing. Here's the rectangle. Go crazy, but only inside this box."

📦 PAINTSTRUCT — The Real Deal Struct

Here's what it looks like under the hood (from <windows.h>):



```
typedef struct tagPAINTSTRUCT {
    HDC  hdc;              // Handle to device context for painting
    BOOL fErase;           // Non-zero if background should be erased
    RECT rcPaint;          // The smallest rectangle that needs to be painted (invalid area)
    BOOL fRestore;         // Reserved; used internally
    BOOL fIncUpdate;       // Reserved; used internally
    BYTE rgbReserved[32]; // Reserved
} PAINTSTRUCT;
```

🖌️ HDC hdc

Handle to Device Context — this is your **drawing tool**.
Comes from BeginPaint() and lets you draw directly onto the window.

🧽 BOOL fErase

Tells you whether Windows wants the **background cleared** before drawing.

If TRUE, you might want to call FillRect() to repaint the background.
Windows often sets this to TRUE if it thinks leftover graphics are around.

📦 RECT rcPaint

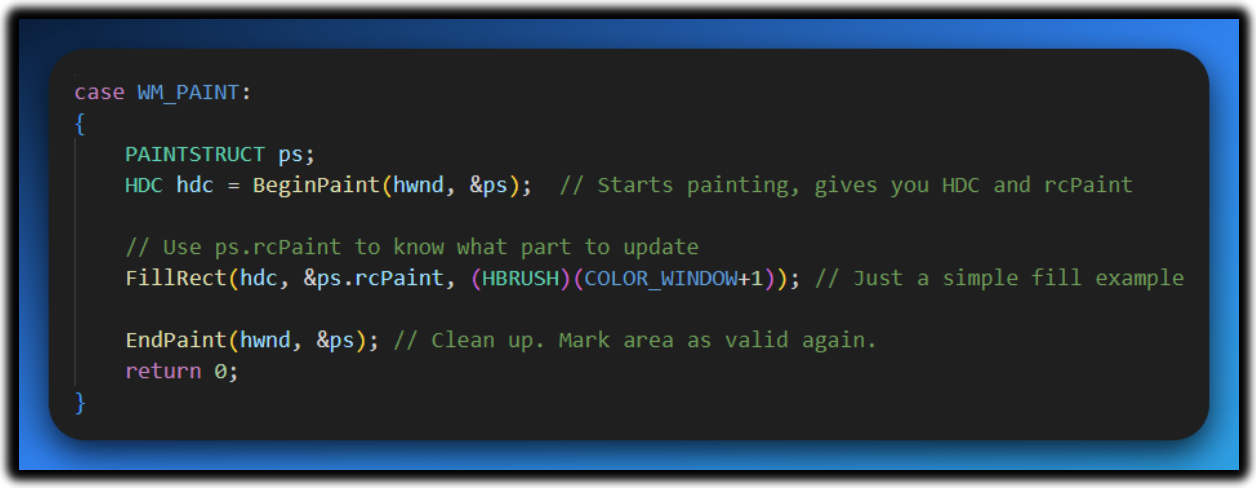
The **exact rectangle** that Windows has marked as invalid. This is the region your app *should* repaint. Ignore the rest. This is your **scope**, your **mission zone**, your **red zone**.
Don't go repainting the entire window unless this says so.

🛠️ BOOL fRestore, fIncUpdate, BYTE rgbReserved[32]

Ignore these. They're internal stuff for the OS.
Legacy or reserved — don't touch unless you're writing your own OS (and if you are, I bow to you 😶).

🧪 Real-World Use

In practice, this is how you use PAINTSTRUCT:



```
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hwnd, &ps);  // Starts painting, gives you HDC and rcPaint

    // Use ps.rcPaint to know what part to update
    FillRect(hdc, &ps.rcPaint, (HBRUSH)(COLOR_WINDOW+1)); // Just a simple fill example

    EndPaint(hwnd, &ps); // Clean up. Mark area as valid again.
    return 0;
}
```

This is **how you stay efficient**.
No wasted redraws. Just precision.

🔁 Recap Time

- PAINTSTRUCT holds everything you need during a repaint.
- You get it from BeginPaint(), and pass it to EndPaint().
- Its rcPaint member tells you *exactly* what needs redrawing.
- Only repaint that region — keeps your app responsive and slick.

This paint pipeline is foundational in **any GUI Windows app** — get this down, and the rest builds on it.

Hit me with the next part. We're moving like pros now. 🔥💻

🧩 🖼 Invalid Rectangle Updates — No Duplicate Paint Chaos

🧠 What Happens If New Damage Occurs Before You Paint?

Imagine this:

1. A part of your window becomes "damaged" — say, something was moved, resized, or revealed.
2. Windows sends you a WM_PAINT, saying: "Yo, redraw this bit here."
3. **But you haven't handled that paint message yet.**

Then boom — **another part** of the client area gets damaged **before you even start painting**.

❗So... What Does Windows Do?

Instead of flooding your message queue with **multiple WM_PAINTs**, Windows gets slick:

- It **combines** all the damaged regions into a single **invalid region**.
- It recalculates the **invalid rectangle (rcPaint)** to cover **both areas**.
- The original paint info (PAINTSTRUCT) is updated to reflect the **new, larger zone**.
- Still **just one WM_PAINT** message in the queue for your window.

🔁 Efficient Like a Pro

This is done to **optimize performance** and reduce flickering or redundant painting.

Even if more parts get messed up while you're procrastinating the WM_PAINT, Windows patiently waits, **bundles all the junk**, and **sends one big cleanup mission** when you're finally ready to paint.

📦 Real World Analogy

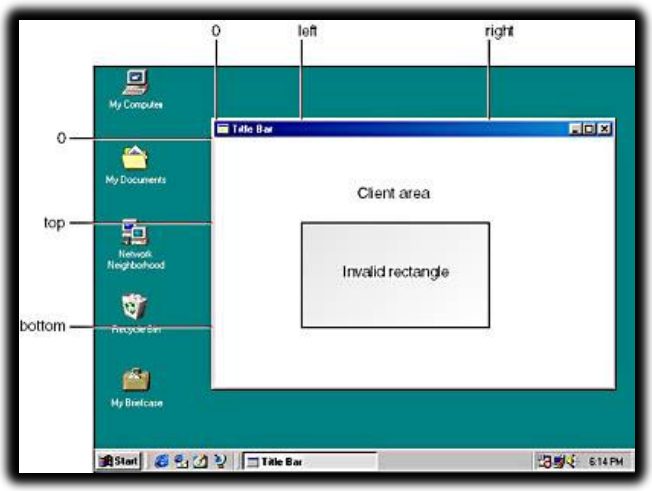
Think of it like a housekeeper:
If they see one cup on the floor, they're about to clean it... but before they do, someone knocks over a plate too.

Rather than running back and forth for each mess, they wait, see it all, and **clean everything in one go**. That's Windows.

🧪 Gotchas and Tips

- **Never assume** rcPaint only covers the original damage. Always check it fresh when handling WM_PAINT.
- **Don't try to queue multiple WM_PAINTs** — Windows *won't* let you. Just use InvalidateRect() again, and it'll expand the region.
- **Efficiency is the name of the game** — only paint the invalid region, not the whole window, unless needed.

This is **paint region management 101** — it's all about letting Windows juggle those rectangles so you don't have to.



🧨 Invalidating Rectangles — Manually Saying "Repaint This!"

📜 What's the Deal?

Sometimes your app changes something — maybe some data updates, or a UI element needs refreshing — but **Windows doesn't know that**. So, **you** have to tell Windows:

"Hey, this rectangle right here? It's dirty. Needs repainting."

You do that with:

```
InvalidateRect(hwnd, &rect, TRUE);
```

📦 What It Does

- Marks the specified rect in the **client area** as invalid (or the **entire client area**, if rect is NULL).
- Doesn't immediately redraw.
- Instead, it **queues a WM_PAINT message** — but **only if** one isn't already waiting.

If there's already a WM_PAINT in the queue?

Windows just updates the invalid region — no duplicate message. Efficient AF.

🧠 Example

Say your app has a button that changes text on the screen. You'd do something like:

```
InvalidateRect(hwnd, NULL, TRUE);  // Repaint the whole client area
```

Or target a specific zone:

```
RECT updateArea = { 20, 20, 200, 50 };
InvalidateRect(hwnd, &updateArea, TRUE);
```

Boom 💥 — the system goes:

"Okay, I'll send WM_PAINT soon — this region needs some fresh perfume."

📍 Retrieving the Invalid Rectangle Coordinates

So now you get WM_PAINT — how do you know **which part** to repaint?

🔍 1. Inside the WM_PAINT handler

When you do:

```
PAINTSTRUCT ps;
HDC hdc = BeginPaint(hwnd, &ps);
```

You get ps.rcPaint — that's your **invalid rectangle**. That's what you repaint.

🔍 2. Outside WM_PAINT? Use GetUpdateRect()

Maybe you're not inside WM_PAINT, but you're curious what needs repainting.

You can call:

```
RECT rect;
BOOL hasInvalid = GetUpdateRect(hwnd, &rect, FALSE);
```

- If hasInvalid is TRUE, there's a region that needs repainting.
- rect will hold its coordinates.

Don't use this during actual painting though — just a heads-up check.

🧠 Final Thoughts

In summary:

InvalidateRect: This function manually marks part or all of the window as "dirty" (i.e., needing to be redrawn).

Windows queues a WM_PAINT message: If there isn't already a WM_PAINT message queued, it adds one. If there's already a queued WM_PAINT, it simply updates the invalid region.

BeginPaint(): When handling the WM_PAINT, BeginPaint() provides you with rcPaint, which tells you the exact region that needs to be repainted.

GetUpdateRect(): This function gives you the invalidated region outside of the painting process — helpful if you need to know what needs repainting without triggering the paint cycle.

💡 Real World Analogy

Think of it like telling a janitor:

"This aisle is dirty — add it to your list." (InvalidateRect)

If he's already planning to clean it, just update his list.

When he shows up to clean (WM_PAINT), he brings a clipboard (rcPaint) to show exactly what areas need to be cleaned.

✅ Validating Rectangles — Marking Areas as "All Good"

Windows sends a WM_PAINT message when it thinks a part of your window needs to be redrawn.

But what if your app's like:

"Yo, I'm good. No need to repaint this."

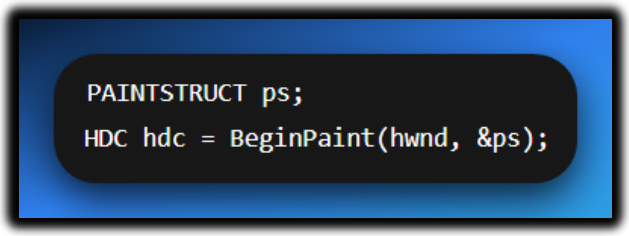
That's where ValidateRect() comes in.

This tells Windows:
"This rectangle? It's already fresh. No need to send a WM_PAINT for it."

- If you pass **NULL** as the rect, it means you're telling Windows that **the entire client area** is valid and doesn't need any redrawing.
- If a **WM_PAINT** message is already in the queue and the validated area fully covers the invalid region, Windows will **cancel** that WM_PAINT because it thinks everything is good now. It's like telling Windows, "Forget that repaint request; everything's fine."

🔁 When Does This Happen?

1. **Automatically after BeginPaint()**:
   When you call BeginPaint(), Windows automatically marks the invalid region as valid.



What happens: You start painting, and Windows automatically assumes the invalid region is taken care of. It's like saying:

- **You**: "I'm painting now."
- **Windows**: "Okay, I'll consider that area cleaned up, no need to worry about it anymore."

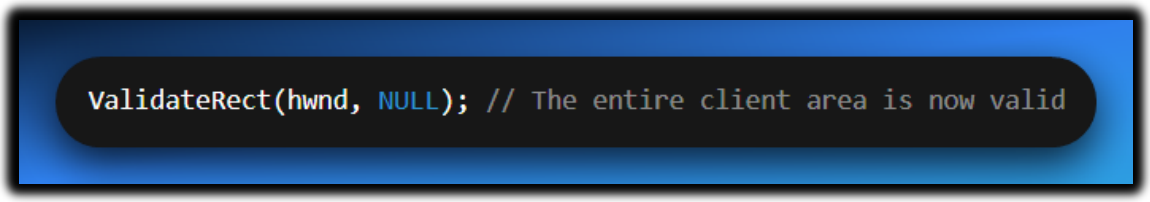
You usually **don't need to manually validate** inside the WM_PAINT message because BeginPaint() already does this for you.

Manually with ValidateRect():

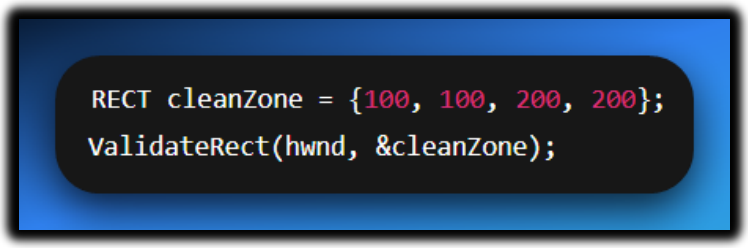
If you want to skip repainting or let Windows know that **nothing needs to be redrawn**, you can use ValidateRect() manually.

For example:

- To **validate the whole window** (mark it all as good):



- Or to **validate a specific region** (like part of the window):



Why would you do this?

You might want to do this if there's a **false alarm** (like an event that looks like it needs redrawing but doesn't) or if the **visuals didn't actually change**. This can **stop a pending WM_PAINT** or **prevent a new one from being queued**.

Summary:

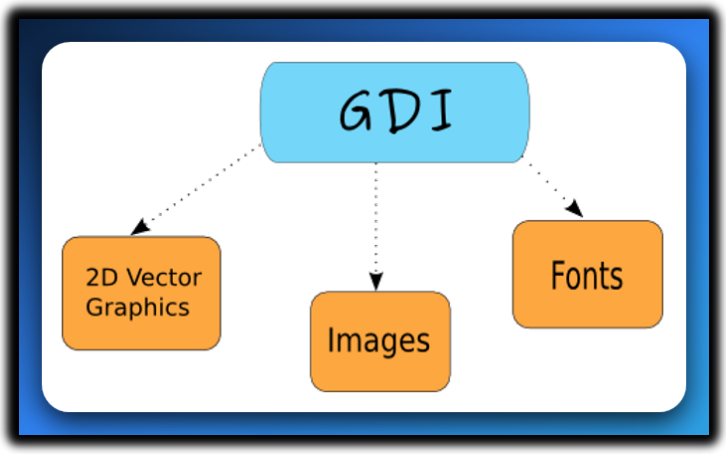
- *ValidateRect() tells Windows, "This area is fine, no need for repainting".*
- *After BeginPaint(), Windows automatically validates the region you're painting, so no need to call ValidateRect() in most cases.*
- *You can manually call ValidateRect() to stop or clear a WM_PAINT if you don't actually need to redraw anything.*

This section is about **telling Windows to stop worrying about certain parts of the window**, so it doesn't waste time or resources sending unnecessary WM_PAINT messages.

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 (text, lines, shapes, even bitmaps).



🚧 (Read this carefully, you'll meet these everywhere!)

🪟 HWND: Handle to a Window (Your Window's ID Tag)

In the world of WinAPI, HWND is your **window's unique identity**.

Think of it like:

- *A username for your window.*
- *Or better — a pointer to your window's profile in memory.*

Whenever you create a window, Windows gives you an HWND —

This is a ***handle*** — or a **number-like ID** that points to an internal structure inside the OS containing:

- 📼 Size
- 📼 Position
- 📼 Visibility
- 📼 Styles
- 📼 Input
- 📼 Owner relationships
- 📼...and more

That **structure tracks *everything*** about your window: size, position, visibility, input, styles, ownership… the whole deal.

🖐️ Easy Analogy: The "Human Hand" Metaphor

HWND = "hand to the window"

- Want to **move** the window? Use MoveWindow(hwnd, ...)
- Want to **resize**? SetWindowPos(hwnd, ...)
- Want to **show or hide** it? ShowWindow(hwnd, SW_SHOW)
- Want to **send a message**? SendMessage(hwnd, WM_CLOSE, ...)

If you don't have the HWND, you've got **no way to talk to the window.**

🧠 How Windows Thinks Behind the Scenes

Every window lives somewhere in memory.
The HWND is like a **map pin** that tells the OS,

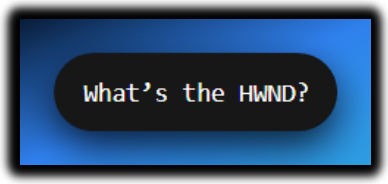
"Yo, that window you created earlier? Yeah, it lives here."

📌 When you drag, resize, or close a program like Word:

- You're interacting with a GUI layer.
- Under the hood, the OS uses the HWND to locate that window's internal data.
- That data includes instructions (from your program's code) on how it should react.

🛠️ So Why Do We Care?

Every serious WinAPI function starts by asking:



It's like showing your ID card at the door. **No HWND? No access.**

- *Draw inside the window?*
- *Resize it?*
- *Send it a message?*

You better have that HWND — it's your ID badge.

Even message processing (WndProc) is tied to your HWND — Windows routes messages to the correct window **using its HWND**.

✅ TLDR:

- *HWND is the handle (ID) that uniquely represents a window. It's the address of the house.*
- *The OS uses it to find and manage that window's memory and behavior.*
- *Without it, you can't interact with the window using WinAPI.*

🧑🎨 HDC: Handle to Device Context (Your Drawing Toolkit)

Meet HWND's younger brother: HDC - He's the artist, the technician, the graffiti guy painting walls all day, while HWND handles the business side of windows things.

If HWND is the **address of a house**, then HDC is your **access pass** to paint the walls inside it, to the drawing tools and space inside that window.

HDC stands for Handle to Device Context.

It's the **identifier** you use when you want to draw on the window.

It gives you access to a **drawing environment**:

- Text
- Shapes
- Lines
- Bitmaps
- Anything GUI-related

Think of it as a **"key" that unlocks the drawing surface**, allowing your program to interact with the graphical environment of that window.

🧠 How Windows Thinks About HDC:

🔷 “Okay, Nick made a window — here’s the HWND.”
🔷 “Oh he wants to draw now? He called GetDC()? Cool.”
🔷 “Here’s an HDC for that window’s drawable surface.”
🔷 “But he better return it later… GDI is *not* unlimited.”

📐 HDC = Surface + Settings

It’s not just a canvas — it’s the full drawing setup:

🎯 Target: screen, printer, bitmap, etc.
🎨 State: pen, brush, font, color
📏 Transform: where/how drawings appear

Windows tracks this under the hood — and HDCs are not cheap.

That’s why functions like BeginPaint() or GetDC() give you one, and **you must release it** when done:

```
HDC hdc = GetDC(hwnd);  // Step 1: Get access to the window's drawing surface
// ... perform drawing operations ...
ReleaseDC(hwnd, hdc);   // Step 2: Release the access to the drawing surface
```

- *Give me access to the drawing surface for this window.*
- *The OS provides you with an HDC, which represents the drawing environment for that window.*
- *It’s like receiving an access pass to interact with the window’s graphical content.*
- *With this HDC, you can start using various drawing functions (like TextOut(), MoveToEx(), LineTo(), etc.) to draw on the window.*
- *Once you’re done with the drawing operations, you call ReleaseDC(hwnd, hdc) to release the HDC back to the operating system.*
- *You’re telling the operating system, “I’m done using the drawing surface for this window.”*

🎬 What Actually Happens When a Window is Created?

When you launch your app and create a window, the OS does **not** draw your content for you. Here's what really goes down:

🧠 Step 1: The Shell is Born

When you call something like CreateWindowEx():

- Windows allocates memory and resources.
- It builds the *structure* of the window (size, borders, style, etc.).
- It gives you back the HWND.

But that's it.
Your window now **exists**…
…but it's empty inside. Blank. A white void.

Think of it as: An apartment with walls, doors, and lighting, but no furniture, no paint, no posters — just a shell.

🎨 Step 2: You (the Program) Must Do the Painting

Now Windows says:

"Hey, this window needs something to show — I'll send a WM_PAINT message."

At that point:

- You handle WM_PAINT in your WndProc.
- You call BeginPaint() to get an HDC.
- Now you have access to **drawing tools**: pens, brushes, fonts, etc.

Your program is now responsible for:

- Painting text with TextOut()
- Drawing shapes
- Rendering bitmaps
- Laying out UI elements

Nothing shows up unless **you** draw it with GDI.

🔧 The OS Just Gives You Tools — You Do the Work

Let's reframe the big picture:

🧠 The OS is like a huge construction company.
It builds the **outer frame** of your app window (via HWND).
But when it comes to **furnishing and decorating** (what the user sees inside),
That's your job — and the only way in is through HDC.

You don't draw stuff **anytime** you want — you draw **when Windows tells you,** during the WM_PAINT cycle. And you use the HDC it hands you.

✅ TLDR Recap:

- *When a window is created (via HWND), it starts off completely blank.*
- *The OS gives you access to the drawable surface (via HDC).*
- *Your program draws the actual GUI using GDI functions like TextOut(), Rectangle(), etc.*
- *Nothing visible happens unless you handle painting during WM_PAINT.*

✅ This is critical background that makes the **entire paint cycle** make sense. Leave this in your notes exactly where it is — just tighten the formatting, and maybe even call it:

⚠️ Common Misunderstanding:

"I created a window… why isn't anything showing up?"
→ Because you haven't painted anything yet, chief. Windows gave you the space — now pick up the brush.

Whenever you're ready, we can dive into:

- WM_PAINT lifecycle
- PAINTSTRUCT
- BeginPaint vs GetDC
- and how GDI fits into all of it like clockwork.

Finally, lets close out this discussion on HWND and HDC.

🧠 HWND vs HDC — Brothers From the WinAPI Womb

- **HWND = the handle to your window.**
  It's like a **home address**. It tells the OS, *"Yo, this is where the window lives."*
  It identifies *what* you want to draw on, manage, move, minimize, etc.
- **HDC = the handle to device context.**
  It's like a **toolbox + canvas**. Once you have the window (HWND), HDC is how you say,
  "Okay, give me the brush and let's start painting." 🎨

🤖 Under the Hood (How Windows Thinks)

Windows separates the *what* (the window itself) from the *how* (the tools used to draw into it):

- The **OS gives you the HWND** so it can manage windows, events, messages, and layout.
- The **OS gives you HDCs** so you can draw to surfaces — whether that's a screen, a printer, or an in-memory bitmap.

This separation is deliberate and powerful. Imagine HWND as the **body** 🧍 and HDC as the **painted clothes, tattoos, or makeup** you apply with care.

📜 True Statement Recap

✅ "**While HWND is the handle that identifies your window, HDC is the handle that provides the drawing tools and the canvas inside that window.**"
That's **absolutely correct** and beautifully put. Keep that line — it slaps both technically and poetically.

🎯 Final Reminder: Don't Be this Guy

Once you're done with drawing, release stuff:

```
EndPaint(hWnd, &ps); // Or ReleaseDC(hWnd, hdc);
```

Because…

💥 **Forget to release? You leak GDI objects. Windows *will* retaliate.**

- **HWND** → The **address** of the window (the "building").
- **HDC** → The **drawing tools and surface** inside the window (your "blueprint & brushes").
- You get an HDC via GetDC() or BeginPaint().
- You release it via ReleaseDC() or EndPaint().

We're done with HDC and HWND, now we head back to GDI topic.

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.

```
BOOL TextOut(
  HDC hdc,          // device context (where to draw)
  int x, int y,     // position on screen (in pixels)
  LPCSTR psText,    // pointer to the text string
  int iLength       // number of characters to draw
);
```

or

```
BOOL TextOut(HDC hdc, int x, int y, LPCSTR psText, int iLength);
```

The function prints a string of text at position (x, y) in the client area.

Simple enough — just give it:

- A valid DC (handle to a drawing surface) - that just means asking Windows for the tools (pens, brushes, etc.) and the surface (where to draw) so your app can actually put pixels on the screen."
- The coordinates (where the text begins),
- A string to draw, and
- How many characters to draw.

🧠 Pro Tip:

If you pass **strlen(yourString)** as **iLength**, you won't need to count chars manually. I know that's sorcery, you didn't hear nothing. 😆 Welcome to C.

That line's a **mic-drop tip** for using text functions like TextOut() or DrawText() in WinAPI where you have to **tell Windows how many characters you're printing**.

Say you've got this:

```
char* myMessage = "Hello, Windows!";
TextOutA(hdc, 10, 10, myMessage, ???);
```

That last parameter is asking:

"How many characters should I draw?"

You *could* count manually like a caveman:

```
TextOutA(hdc, 10, 10, myMessage, 15); // hardcoded 💀
```

But then you change the message... and forget to change the length... and now your app is drawing random garbage or cuts off early.

💡 Pro tip to the rescue:

```
TextOutA(hdc, 10, 10, myMessage, strlen(myMessage));
```

Now C counts the characters **automatically**, right before passing it to TextOutA() — no typos, no mental math, no pain.

⚠️ Gotchas:

- Make sure the string is **null-terminated** (\0), or strlen() might run off into memory madness.
- Don’t use strlen() if you're drawing only **part of the string** (like just the first 5 letters). Use 5 directly in that case.

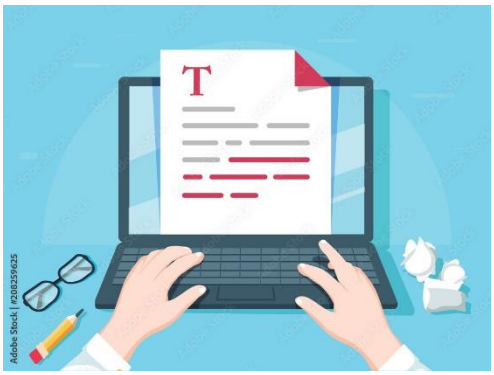
🧠 TLDR:

strlen(yourString) = C doing the counting for you
🧼 Clean, automatic, no guesswork

🔒 Bonus bar for your notes:

```
// Pass strlen(msg) to TextOut() if you want C to count the characters for you.
// No need to hardcode lengths manually.
// Just make sure msg is null-terminated.
```

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.



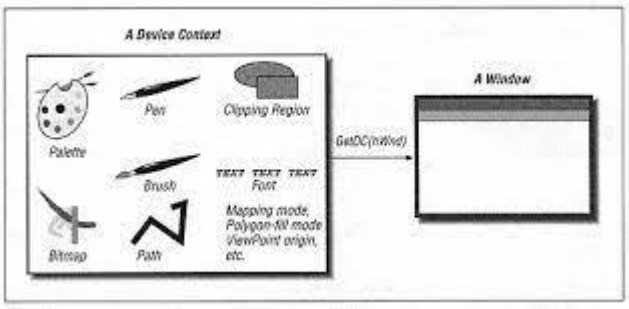
The Device Context: A Crucial GDI Element

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.

```
case WM_LBUTTONDOWN:
    /* BeginPaint */ {
        PAINTSTRUCT ps = { 0, };
        HDC hDC = BeginPaint(p_hWnd, &ps);

        Rectangle(hDC, 50, 50, 150, 150);

        EndPaint(p_hWnd, &ps);
    }
    break;
```

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.

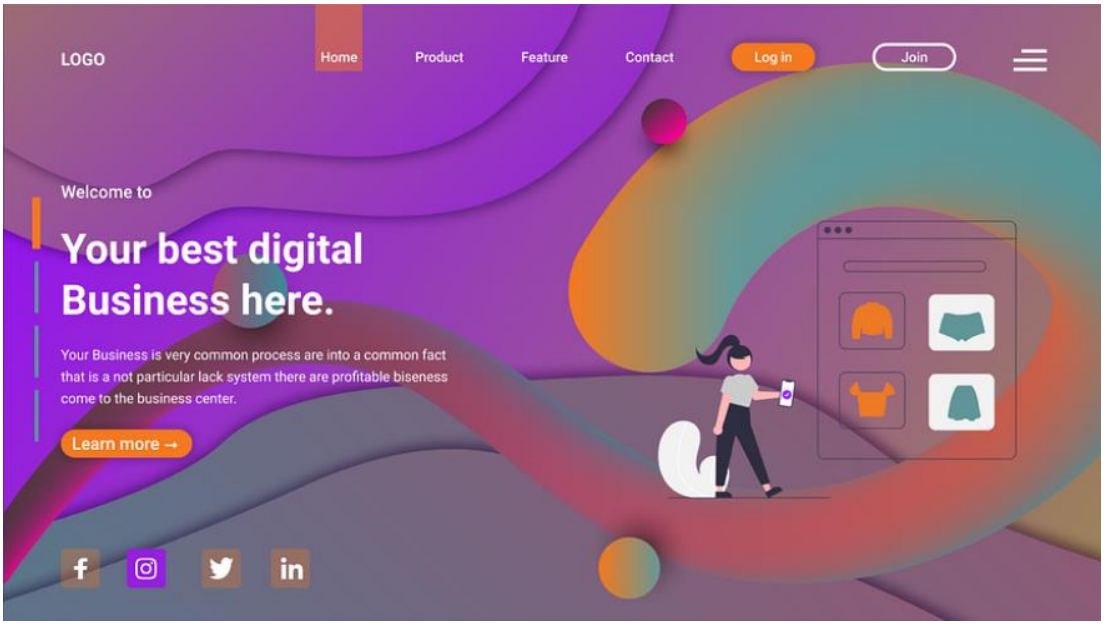


Figure 1: BeginPaint and EndPaint

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.

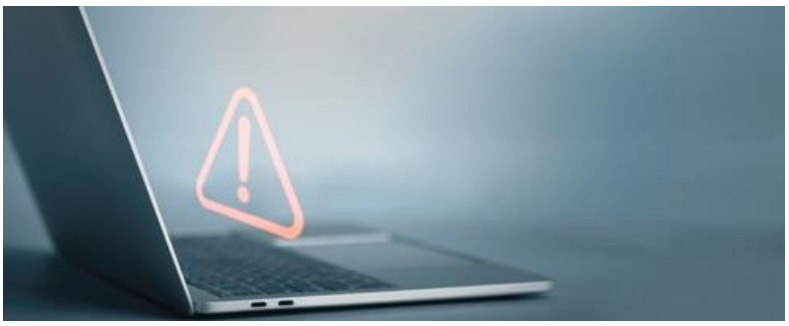
Typical WM_PAINT Message Handling

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.



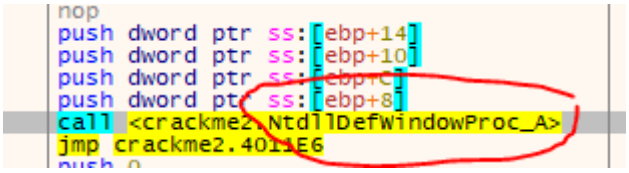
Windows places the WM_PAINT message in the message queue because part of the client area is invalid and requires repainting.

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:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_PAINT:
            {
                PAINTSTRUCT ps;
                HDC hdc = BeginPaint(hwnd, &ps);

                // Paint the client area of the window here
                EndPaint(hwnd, &ps);
                return 0;
            }
        default:
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
}
```

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.

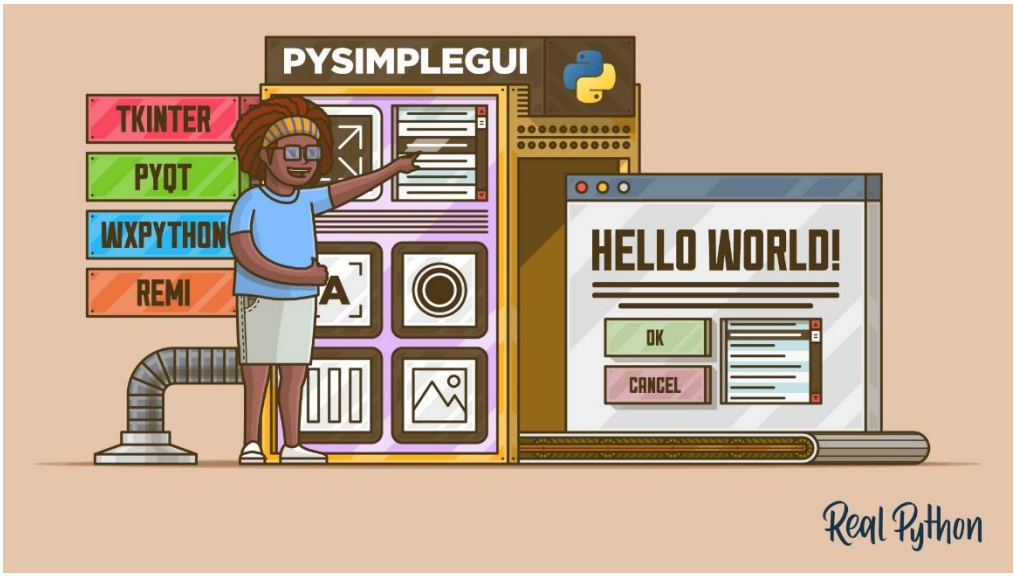
Here is an example of how to use the BeginPaint and EndPaint functions to draw a rectangle to the client area of a window:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_PAINT:
            {
                PAINTSTRUCT ps;
                HDC hdc = BeginPaint(hwnd, &ps);

                // Draw a rectangle to the client area of the window
                RECT rect = { 10, 10, 100, 100 };
                FillRect(hdc, &rect, (HBRUSH)GetStockObject(BLACK_BRUSH));

                EndPaint(hwnd, &ps);
                return 0;
            }
        default:
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
}
```

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:

```
typedef struct tagPAINTSTRUCT {
    HDC hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
} PAINTSTRUCT;
```

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.

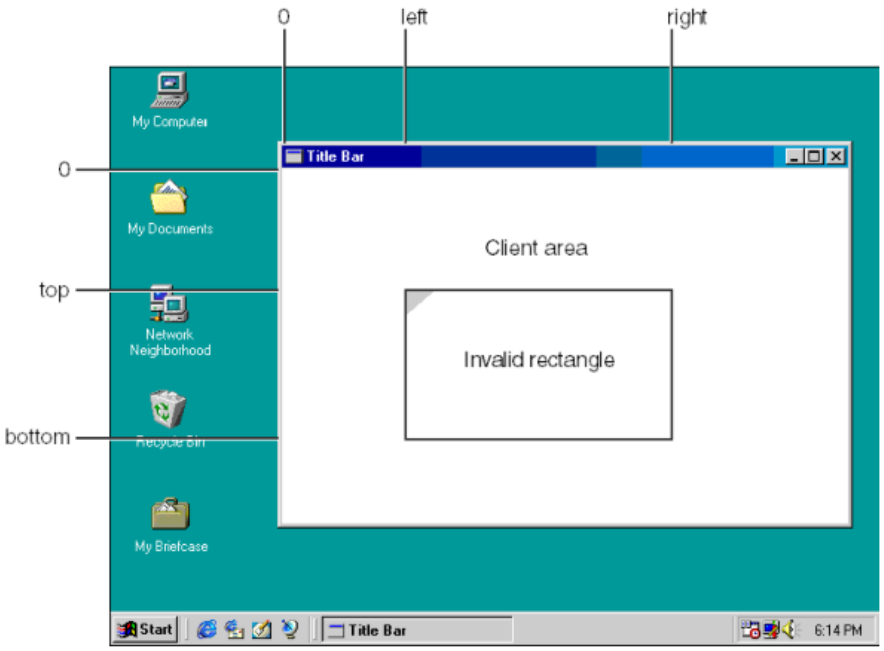


Figure 4–1. *The boundaries of the invalid rectangle.*

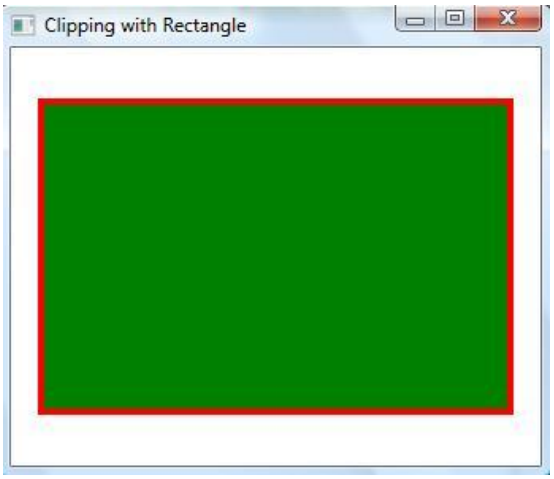
In summary, the PAINTSTRUCT structure serves as a vital information carrier during the painting process. It provides access to the device context handle, indicates whether background erasing is necessary, and defines the boundaries of the invalid rectangle, ensuring that your program paints efficiently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

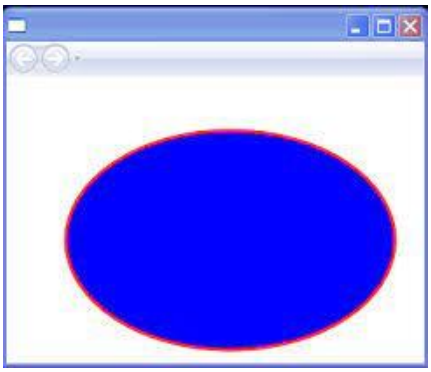
```
InvalidateRect(hwnd, NULL, TRUE);
```

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.

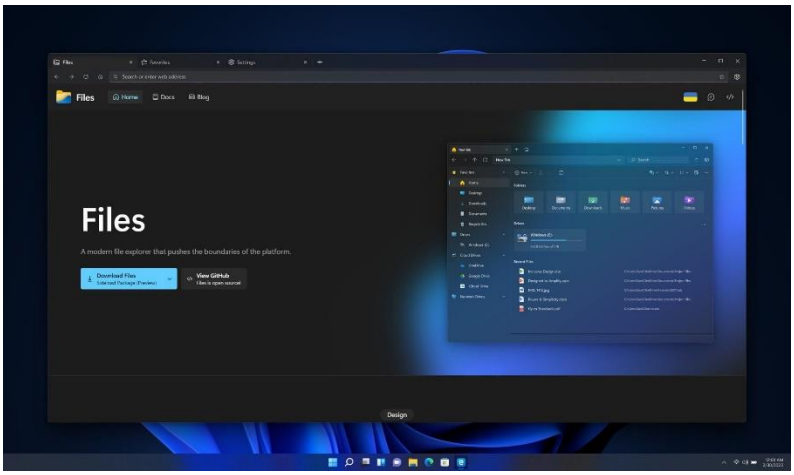


For instance, if a circle is partially within the invalid rectangle, it's more efficient to redraw the entire circle than just the invalid portion.

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:

```
TextOut(hdc, x, y, psText, iLength);
```

Breaking Down the TextOut Function Parameters:

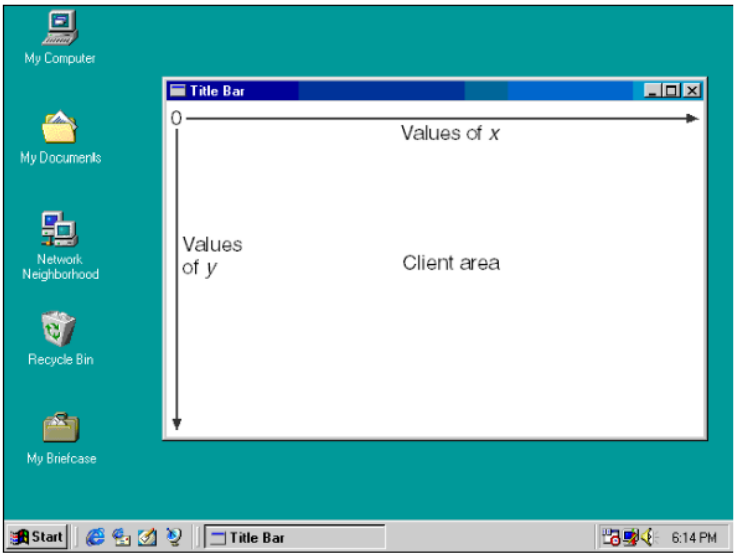


Figure 4–2. *The x–coordinate and y–coordinate in the MM_TEXT mapping mode.*

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.

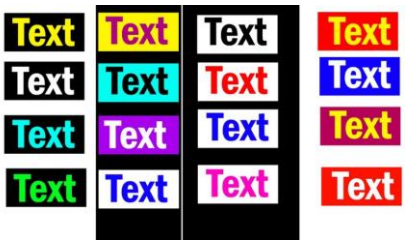
iLength: This parameter indicates the number of characters in the psText string. For Unicode character strings, the number of bytes in the string is double the iLength value.

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."



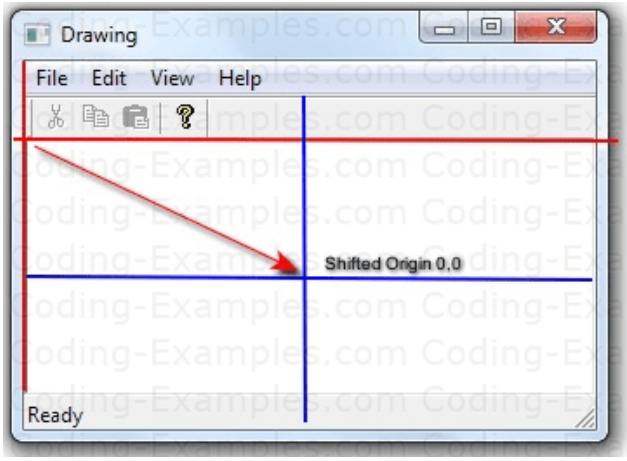
Font: The font used to render the text is specified using the SelectObject function in combination with a created font object.

Coordinate System and Mapping Modes:

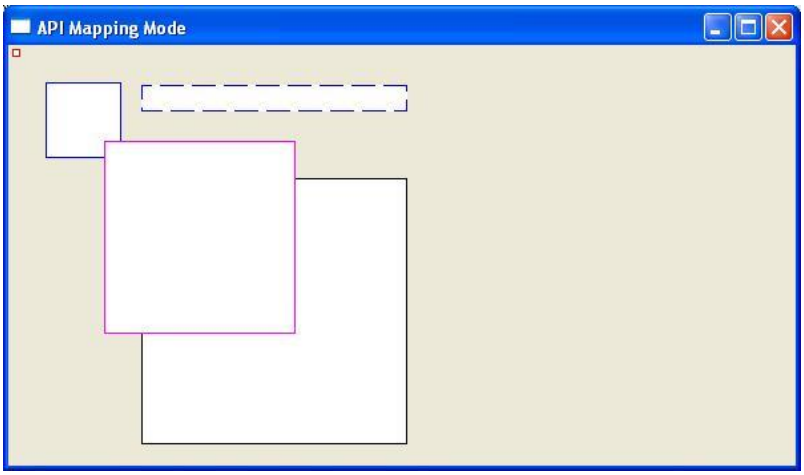
The x and y coordinates passed to TextOut represent logical coordinates, which are not directly translated to physical pixels on the display.

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.



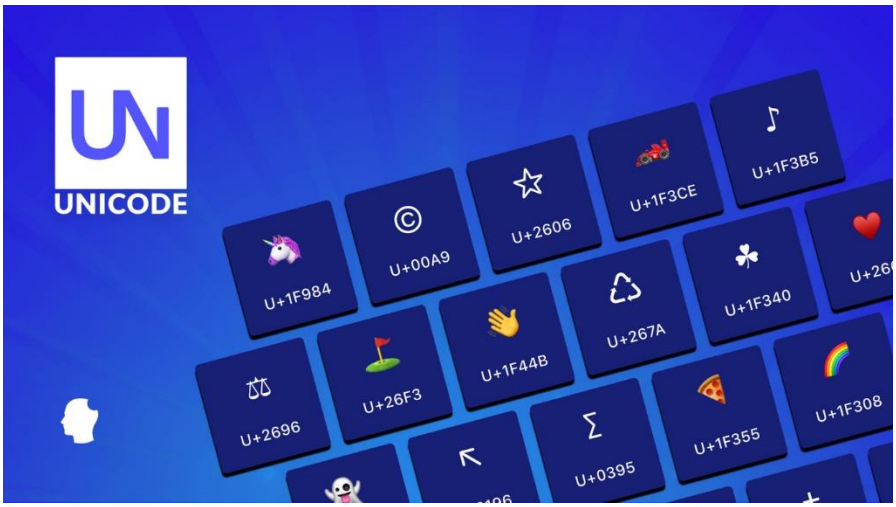
Other Mapping Modes: Other mapping modes, such as MM_HIENGLISH and MM_LOENGLISH, introduce scaling and offsetting to logical coordinates, influencing the text placement.



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.

Hi, I'm Arial!
I'm a 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

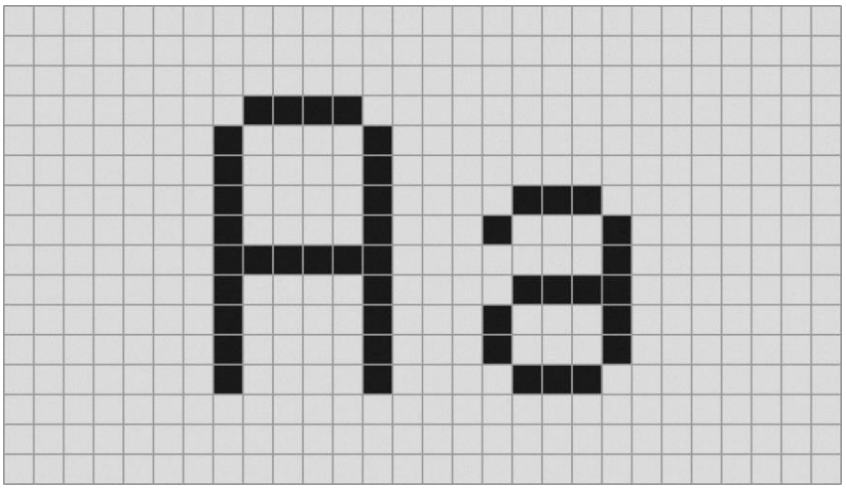
The adoption of variable-pitch fonts necessitated adjustments in programming practices.

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.

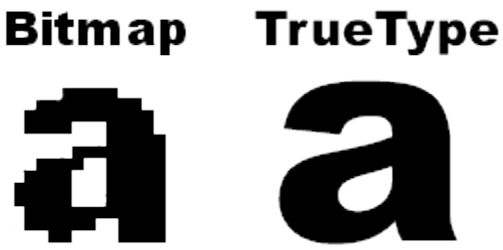
Play this video: 👇👇



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:

```
TEXTMETRIC tm;

// Get a handle to the device context
HDC hdc = GetDC(hwnd);

// Retrieve character metrics for the current font
GetTextMetrics(hdc, &tm);

// Examine and use the values in the text metric structure
int characterHeight = tm.tmHeight;
int averageCharacterWidth = tm.tmAveCharWidth;

// Release the device context
ReleaseDC(hwnd, hdc);
```

```c
#include <windows.h>
typedef struct tagTEXTMETRIC {
    LONG tmHeight;
    LONG tmAscent;
    LONG tmDescent;
    LONG tmInternalLeading;
    LONG tmExternalLeading;
    LONG tmAveCharWidth;
    LONG tmMaxCharWidth;
    // Add other structure fields as needed
} TEXTMETRIC, *PTEXTMETRIC;

int main() {
    HDC hdc = GetDC(0); // Get the device context of the screen
    // Get the font information
    LOGFONT lf;
    GetObject(GetStockObject(DEFAULT_GUI_FONT), sizeof(LOGFONT), &lf);
    // Create a font using the information obtained
    HFONT hFont = CreateFontIndirect(&lf);
    // Select the font into the device context
    SelectObject(hdc, hFont);
    // Get the text metrics
    TEXTMETRIC tm;
    GetTextMetrics(hdc, &tm);
    // Display the text metrics
    printf("Text Metrics:\n");
    printf("  Height: %ld\n", tm.tmHeight);
    printf("  Ascent: %ld\n", tm.tmAscent);
    printf("  Descent: %ld\n", tm.tmDescent);
    printf("  Internal Leading: %ld\n", tm.tmInternalLeading);
    printf("  External Leading: %ld\n", tm.tmExternalLeading);
    printf("  Average Character Width: %ld\n", tm.tmAveCharWidth);
    printf("  Maximum Character Width: %ld\n", tm.tmMaxCharWidth);
    // Clean up
    DeleteObject(hFont);
    ReleaseDC(0, hdc);
    return 0;
}
```

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.

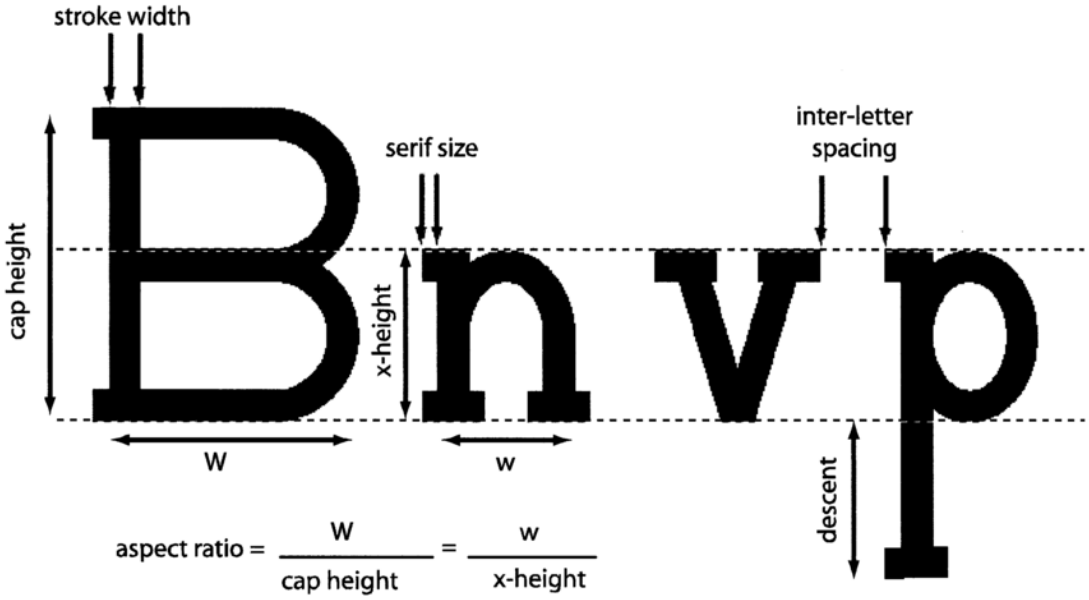
Delving into the TEXTMETRIC Structure for Character Dimensions

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.

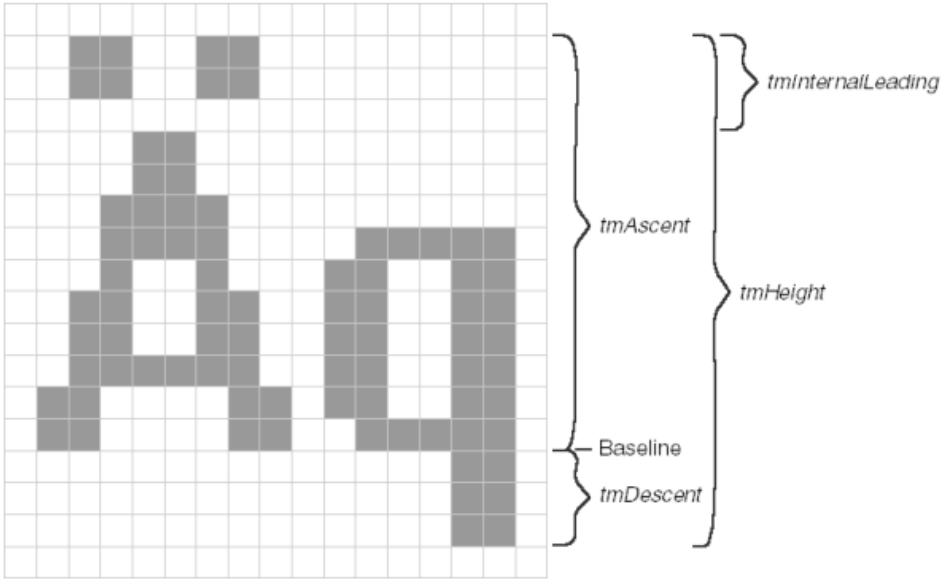
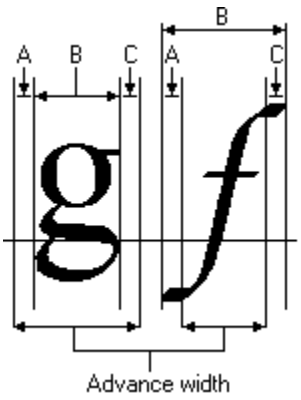


Figure 4–3. *Four values defining vertical character sizes in a font.*

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

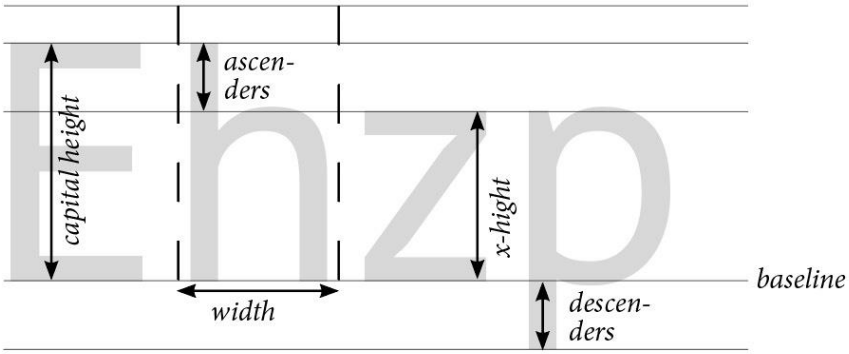
The GetTextMetrics function provides a reliable and device-independent mechanism for retrieving accurate character dimensions. By utilizing this function, programmers can ensure that their text rendering adapts to various display configurations and font selections.

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.



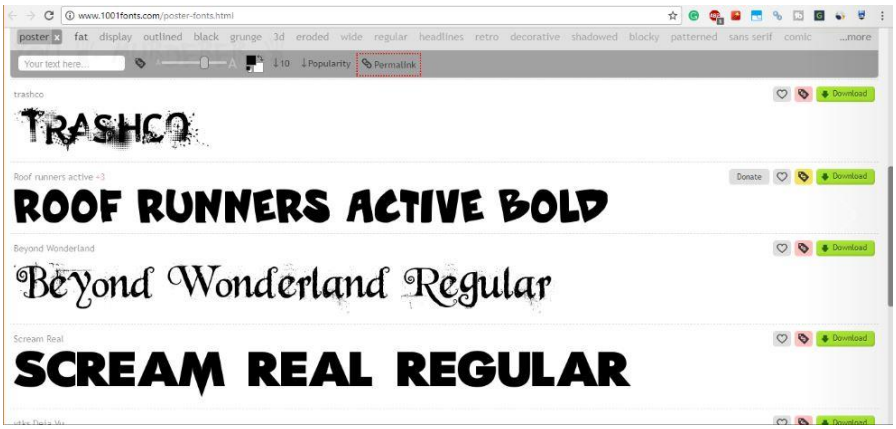
A single call during program initialization is sufficient.

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:

The static variables cxChar and cyChar are defined within the WM_CREATE message handler of the window procedure.

This message handler is executed when the window is created, and it's an appropriate place to initialize these variables since they need to be valid for subsequent message processing, such as WM_PAINT.

Here's an example of how to define and initialize these variables in the WM_CREATE message handler:

```
case WM_CREATE:
  hdc = GetDC(hwnd);
  GetTextMetrics(hdc, &tm);
  cxChar = tm.tmAveCharWidth;
  cyChar = tm.tmHeight + tm.tmExternalLeading;
  ReleaseDC(hwnd, hdc);
  return 0;
```

FORMATTING TEXT IN WINDOWS

In Windows programming, formatting text involves using functions like GetTextMetrics, TextOut, wsprintf, and sprintf to control the appearance of text displayed on the screen.

These functions allow you to modify font characteristics, line spacing, and text alignment.

Obtaining Character Metrics

The GetTextMetrics function retrieves information about the system font, including character width and height.

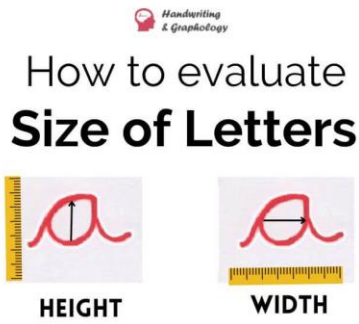
This information is crucial for positioning text accurately and ensuring proper line spacing.

The function takes a device context handle (HDC) as input and returns a TEXTMETRIC structure containing relevant font metrics.

Storing Character Metrics

The cxChar and cyChar variables are used to store the average character width and total character height, respectively.

These variables are defined as static to maintain their values across multiple message processing calls.



WM_CREATE Message Handling

The WM_CREATE message is the first message received by a window procedure after window creation.

It's an ideal place to initialize the text formatting variables. The WM_CREATE message handler typically performs the following steps:

Retrieve the device context handle (HDC) using GetDC.

Call GetTextMetrics to obtain character metrics and store them in cxChar and cyChar.

Release the device context using ReleaseDC.

Return 0.

Displaying Formatted Text

To display formatted text, you can use functions like wsprintf and TextOut.

wsprintf formats a string according to specified format specifiers and stores the result in a buffer.

TextOut then renders the formatted string to the specified coordinates on the device context.

Example Code for Formatted Text Output

The provided code snippet demonstrates the usage of wsprintf and TextOut to display the sum of two numbers:

```
int iLength;
TCHAR szBuffer[40];

// Other program lines

iLength = wsprintf(szBuffer, TEXT("The sum of %i and %i is %i"), iA, iB, iA + iB);
TextOut(hdc, x, y, szBuffer, iLength);
```

Concise Code Combining wsprintf and TextOut

While not as elegant, you can combine wsprintf and TextOut into a single statement:

```
TextOut(hdc, x, y, szBuffer, wsprintf(szBuffer, TEXT("The sum of %i and %i is %i"), iA, iB, iA + iB));
```

This approach directly passes the formatted string to TextOut without the need for an intermediate buffer.

Summary

Formatting text in Windows involves using specific functions to control font characteristics, line spacing, and text alignment. The GetTextMetrics function provides character width and height information, while TextOut and wsprintf enable formatted text output. These functions empower developers to create visually appealing and informative text displays in their Windows applications.

Creating a Header File for GetSystemMetrics Information

The provided code snippet below will introduce the concept of creating a header file named SYSMETS.H to manage the information retrieved from the GetSystemMetrics function.

This header file defines an array of structures containing both the GetSystemMetrics index identifier and the corresponding text for each value returned by the function.

Structure Definition for GetSystemMetrics Information

The SMETRICS structure is defined to hold the GetSystemMetrics index identifier and the associated text:

```
typedef struct {
  int smIndex;
  LPCTSTR szLabel;
} SMETRICS;
```

The smIndex member stores the index identifier for GetSystemMetrics, while the szLabel member holds the corresponding text label.

Array of SMETRICS Structures

An array of SMETRICS structures is defined to store multiple GetSystemMetrics index-text pairs:

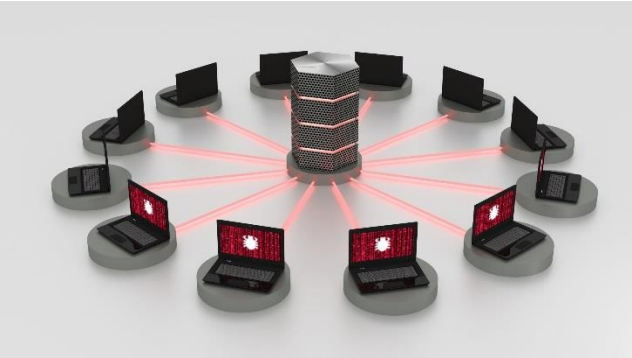
```
SMETRICS aSMETRICS[] = {
  { SM_CXSIZE, TEXT("Width in pixels") },
  { SM_CYSIZE, TEXT("Height in pixels") },
  { SM_CXICON, TEXT("Icon width in pixels") },
  // ... Additional SMETRICS entries ...
};
```

Each structure in the array associates a GetSystemMetrics index with a descriptive text label.

Benefits of Using a Header File

Creating a header file like SYSMETS.H offers several advantages:

Centralized Information Management: The header file centralizes the information about GetSystemMetrics indices and their corresponding text labels, making it easier to maintain and update.



Improved Code Readability: By encapsulating the index-text mapping in a separate file, the main program code becomes more readable and less cluttered with GetSystemMetrics index constants.

```
/**
* Code Readability
*/
if (readable()) {
    be_happy();
} else {
    refactor();
}
```

Error Reduction: The header file serves as a single source of truth for the GetSystemMetrics index-text associations, reducing the risk of errors in the main program code.

Overall, using a header file like SYSMETS.H promotes code organization, maintainability, and readability, especially when dealing with large sets of data or complex information management.

```c
/* SYSMETS.H - System metrics display structure */

#ifndef SYSMETS_H
#define SYSMETS_H

#include <tchar.h>

typedef struct {
    int iIndex;
    TCHAR* szLabel;
    TCHAR* szDesc;
} SMETRIC_ENTRY;

#define NUMLINES ((int)(sizeof(sysmetrics) / sizeof(sysmetrics[0])))

SMETRIC_ENTRY sysmetrics[] = {
    { SM_CXSCREEN, TEXT("SM_CXSCREEN"), TEXT("Screen width in pixels") },
    { SM_CYSCREEN, TEXT("SM_CYSCREEN"), TEXT("Screen height in pixels") },
    { SM_CXVSCROLL, TEXT("SM_CXVSCROLL"), TEXT("Vertical scroll width") },
    { SM_CYHSCROLL, TEXT("SM_CYHSCROLL"), TEXT("Horizontal scroll height") },
    { SM_CYCAPTION, TEXT("SM_CYCAPTION"), TEXT("Caption bar height") },
    { SM_CXBORDER, TEXT("SM_CXBORDER"), TEXT("Window border width") },
    { SM_CYBORDER, TEXT("SM_CYBORDER"), TEXT("Window border height") },
    { SM_CXFIXEDFRAME, TEXT("SM_CXFIXEDFRAME"), TEXT("Dialog window frame width") },
    { SM_CYFIXEDFRAME, TEXT("SM_CYFIXEDFRAME"), TEXT("Dialog window frame height") },
    // ... Other metric entries ...
};

#endif // SYSMETS_H
```

```c
#include <windows.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
     static TCHAR szAppName[] = TEXT("SysMets1");
     HWND hwnd;
     MSG msg;
     WNDCLASS wndclass;

     wndclass.style = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc = WndProc;
     wndclass.cbClsExtra = 0;
     wndclass.cbWndExtra = 0;
     wndclass.hInstance = hInstance;
     wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName = NULL;
     wndclass.lpszClassName = szAppName;

     if (!RegisterClass(&wndclass)) {
          MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
          return 0;
     }

     hwnd = CreateWindow(szAppName, TEXT("Get System Metrics No. 1"), WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
          NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

     while (GetMessage(&msg, NULL, 0, 0)) {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }

     return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
     static int cxChar, cxCaps, cyChar;
     HDC hdc;
     int i;
     PAINTSTRUCT ps;
     TCHAR szBuffer[10];
     TEXTMETRIC tm;

     switch (message) {
     case WM_CREATE:
          hdc = GetDC(hwnd);
          GetTextMetrics(hdc, &tm);
          cxChar = tm.tmAveCharWidth;
          cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2;
          cyChar = tm.tmHeight + tm.tmExternalLeading;
          ReleaseDC(hwnd, hdc);
          return 0;

     case WM_PAINT:
          hdc = BeginPaint(hwnd, &ps);
          for (i = 0; i < NUMLINES; i++) {
               TextOut(hdc, 0, cyChar * i, sysmetrics[i].szLabel, lstrlen(sysmetrics[i].szLabel));
               TextOut(hdc, 22 * cxCaps, cyChar * i, sysmetrics[i].szDesc, lstrlen(sysmetrics[i].szDesc));
               SetTextAlign(hdc, TA_RIGHT | TA_TOP);
               TextOut(hdc, 22 * cxCaps + 40 * cxChar, cyChar * i, szBuffer,
                    wsprintf(szBuffer, TEXT("%5d"), GetSystemMetrics(sysmetrics[i].iIndex)));
               SetTextAlign(hdc, TA_LEFT | TA_TOP);
          }
          EndPaint(hwnd, &ps);
          return 0;

     case WM_DESTROY:
          PostQuitMessage(0);
          return 0;
     }

     return DefWindowProc(hwnd, message, wParam, lParam);
}
```

The SYSMETS1.C source code file contains the implementation of the SYSMETS program, which displays various system metrics on the screen.

The program utilizes the GetSystemMetrics function to retrieve information about various graphical elements, such as icon sizes, scroll bar dimensions, and window borders.

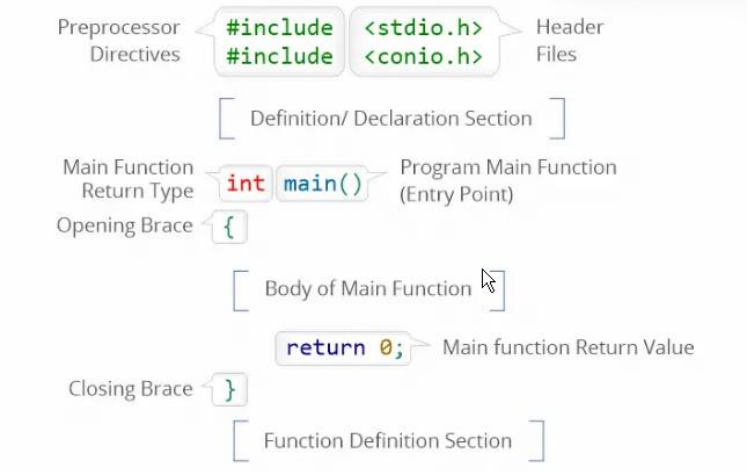
The WinMain function serves as the entry point for the program. It performs the following tasks:

Register the window class: This step defines the characteristics of the program's window, including its style, icon, and cursor.

Create the window: Using the registered window class, WinMain creates the program's window and assigns a handle to it.

Display the window: The window is made visible using the ShowWindow function.

Enter the message loop: The program enters a message loop, where it continuously processes messages received from the operating system.

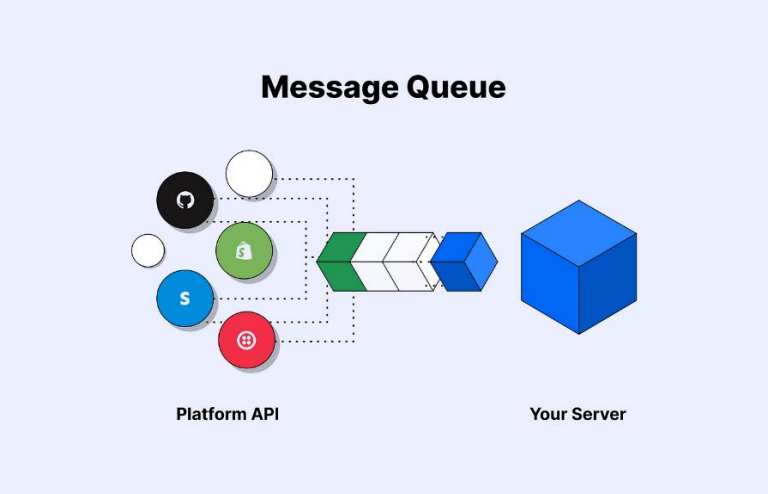


The WndProc function is responsible for handling messages sent to the program's window. It processes various messages, including:

WM_CREATE: This message is received when the window is created. The SYSMETS program uses this message to initialize the text formatting variables.

WM_PAINT: This message is received when the window needs to be repainted. The SYSMETS program uses this message to retrieve system metrics using GetSystemMetrics and display the information on the screen using TextOut.

WM_DESTROY: This message is received when the window is destroyed. The SYSMETS program uses this message to perform any necessary cleanup tasks.

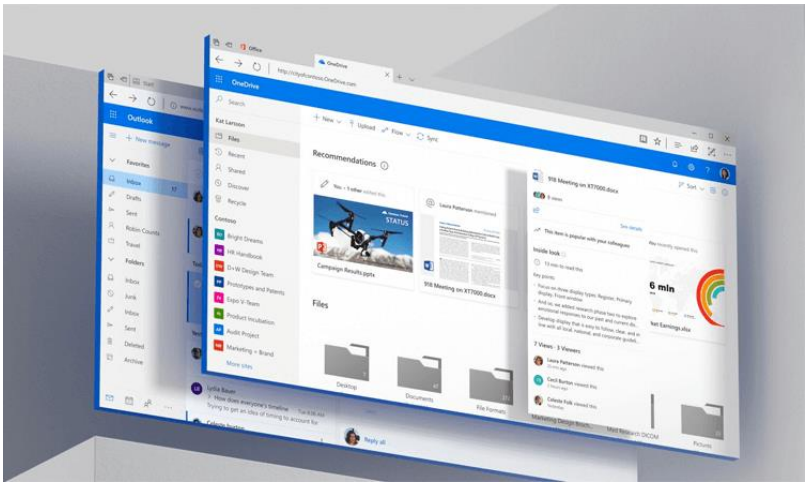


Relationship to Previous Code

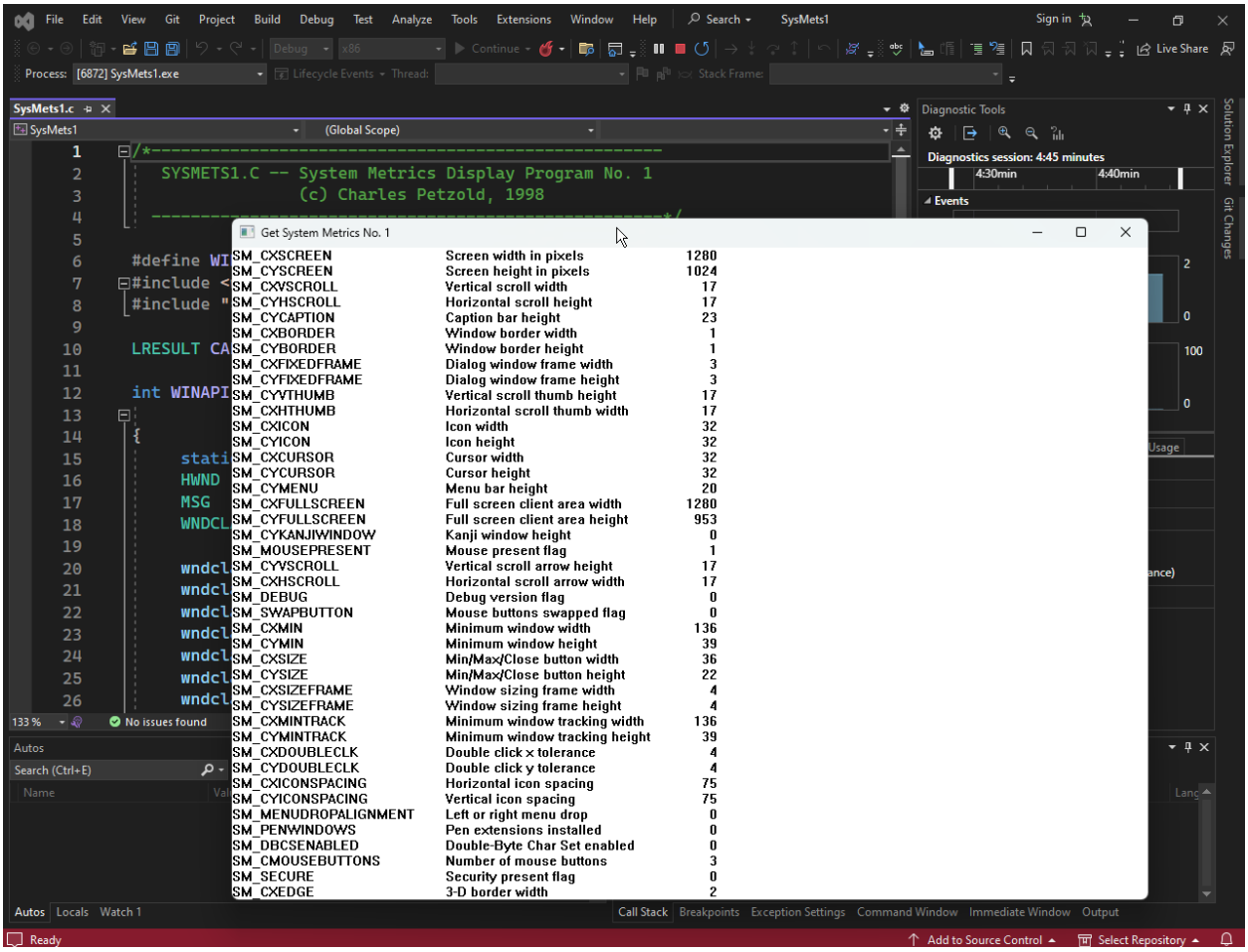
The WinMain function in SYSMETS1 is similar to the one in HELLOWIN, as both involve registering the window class, creating the window, and displaying it.

The WndProc function in SYSMETS1 incorporates elements discussed in previous chapters, such as obtaining character metrics and using TextOut for formatted text output.

The *SYSMETS1 program* demonstrates the practical application of concepts covered in earlier chapters, including window creation, message handling, and text formatting. It showcases how to utilize the GetSystemMetrics function to retrieve system-specific information and display it in a structured manner.



The program runs well in Visual studio community 2022.



The SYSMETS1 Window Procedure

The WndProc window procedure in the SYSMETS1.C program is responsible for handling messages sent to the window. It processes three main types of messages:

WM_CREATE: This message is sent when the window is created. In response, the WndProc procedure obtains a device context for the window using the GetDC function and retrieves the text metrics for the default system font using the GetTextMetrics function. The text metrics include the average character width (stored in cxChar) and the total height of the characters (including external leading) (stored in cyChar).

WM_PAINT: This message is sent when the window needs to be repainted. In response, the WndProc procedure draws the window's contents using the device context obtained in the WM_CREATE message. The procedure uses the TextOut function to display three columns of text: the system metric identifier, the system metric description, and the system metric value. The position of each column is calculated based on the text metrics and the width of the uppercase letters (stored in cxCaps).

WM_DESTROY: This message is sent when the window is destroyed. In response, the WndProc procedure cleans up any resources that were allocated in the WM_CREATE message.

Detailed Explanation of WM_PAINT Processing

The WM_PAINT message processing involves the following steps:

BeginPaint: The WndProc procedure obtains a handle to the device context using the BeginPaint function. The device context is used for drawing the window's contents.

TextOut for Identifiers: The WndProc procedure uses the TextOut function to display the uppercase identifiers in the first column. The second argument to TextOut is set to 0 to start the text at the left edge of the client area. The text is obtained from the szLabel field of the sysmetrics structure. The length of the string is calculated using the lstrlen function and passed as the last argument to TextOut.

TextOut for Descriptions: The WndProc procedure uses the TextOut function to display the descriptions of the system metric values in the second column. The descriptions are stored in the szDesc field of the sysmetrics structure. The second argument to TextOut is set to 22 * cxCaps to position the start of the text to the right of the identifiers.

TextOut for Values: The WndProc procedure uses the TextOut function to display the numeric values obtained from the GetSystemMetrics function in the third column. To right-justify the numbers, the WndProc procedure sets the text alignment using the SetTextAlign function before calling TextOut. The position of the numbers is calculated based on the width of the digits and the width of the uppercase letters.

EndPaint: The WndProc procedure releases the device context obtained in the BeginPaint function using the EndPaint function.

The cxCaps variable is calculated to be 150% of cxChar for variable-width fonts and equal to cxChar for fixed-pitch fonts.

The SetTextAlign function is used to set the text alignment to right-justified for the column of numbers and then set back to the default alignment before the next iteration of the loop.

This detailed explanation provides a deeper understanding of the SYSMETS1 window procedure and its role in displaying system metric information.

The "Not Enough Room" Problem

The SYSMETS1 program displays a list of system metrics along with their corresponding values.

However, due to limitations in how the program determines the available space for displaying text, the entire list may not be visible on screens with smaller resolutions or window sizes.

The primary issue lies in the program's reliance on the default behavior of Windows to clip text that extends beyond the client area of the window.

This approach assumes that the program has sufficient space to display the entire list, which may not be the case on all systems or window configurations.

Proposed Solution

To address this issue, the program needs to explicitly determine the available space within the client area before attempting to display the text.

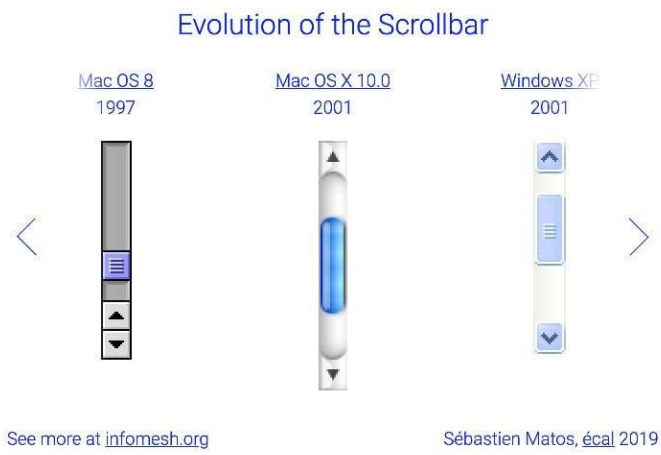
This can be achieved by retrieving the dimensions of the client area using the GetClientRect function.

Implementation Details

Determining Available Space: Before displaying the list of system metrics, the program should call the GetClientRect function to obtain the width and height of the client area.

Adjusting Text Positioning: Based on the available client area dimensions, the program should adjust the positioning of the text to ensure it fits within the visible area. This may involve modifying the vertical position of the text or truncating the list if necessary.

Handling Overflow: If the entire list cannot be displayed within the client area, the program should consider displaying a notification or implementing a scrolling mechanism to allow users to view the remaining entries.



By incorporating these changes, the SYSMETS1 program can ensure that the system metrics list is displayed appropriately, regardless of the screen resolution or window size.

Determining the Client Area Size

The client area is the portion of a window that is available for displaying application content. It excludes the title bar, menu bar, and any other non-client elements.

Determining the size of the client area is essential for ensuring that application content is displayed correctly and efficiently.

Traditional Method: GetClientRect Function

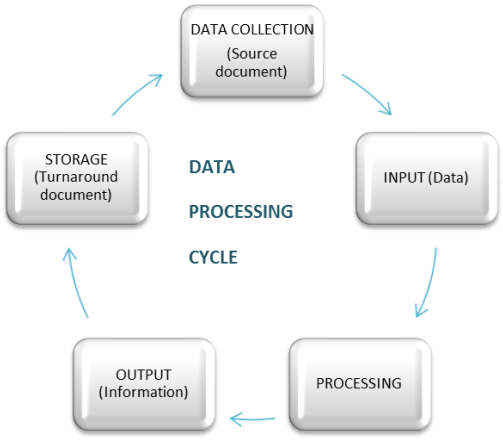
The GetClientRect function is a traditional method for retrieving the dimensions of the client area. It takes a window handle as input and returns a RECT structure containing the client area's coordinates. However, calling this function repeatedly can be inefficient, especially if the client area size changes frequently.

Efficient Method: Processing WM_SIZE Message

A more efficient approach is to process the WM_SIZE message in the window procedure. This message is sent whenever the window's size changes, providing the updated client area dimensions in the lParam parameter.

Storing Client Area Dimensions

To store the client area dimensions, define static variables cxClient and cyClient in your window procedure. These variables are declared as static to retain their values between message processing cycles.



Handling WM_SIZE Message

The WM_SIZE message is handled within the window procedure using the following code:

```
case WM_SIZE:
  cxClient = LOWORD(lParam);
  cyClient = HIWORD(lParam);
  return 0;
```

The LOWORD and HIWORD macros extract the low-order and high-order words from the lParam parameter, respectively. These values represent the client area's width and height.

Calculating Displayable Text Lines and Characters

Once the client area dimensions are known, you can calculate the number of full lines of text and lowercase characters that can be displayed:

- Number of text lines: cyClient / cyChar.
- Number of lowercase characters: cxClient / cxChar.

These calculations rely on the values of cxChar and cyChar, which represent the average character width and total character height, respectively. These values are typically determined during the WM_CREATE message when the window is created.

Scroll Bars for Content Overflow

If the client area is not large enough to hold all the content, scroll bars can be implemented to provide a way for the user to view the remaining information. Scroll bars are essential for applications that handle large amounts of text or data.

Conclusion

Determining the size of the client area is a crucial aspect of Windows programming. By utilizing the WM_SIZE message and storing the client area dimensions efficiently, developers can ensure that application content is displayed correctly and efficiently, even when window sizes change dynamically.

SCROLL BARS: ENHANCING USER INTERFACE INTERACTION

Scroll bars are a ubiquitous and essential component of graphical user interfaces (GUIs), enabling users to navigate through extensive content that exceeds the visible area of a window.

They provide a user-friendly and intuitive way to scroll through documents, spreadsheets, images, web pages, and other types of content.

Types and Placement of Scroll Bars

Scroll bars come in two main types: vertical and horizontal.

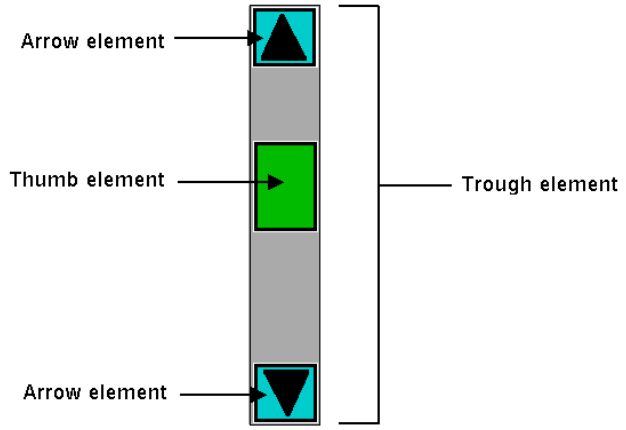
Vertical scroll bars are used for scrolling up and down, typically found on the right side of a window.

Horizontal scroll bars, used for scrolling left and right, are typically positioned at the bottom of a window.

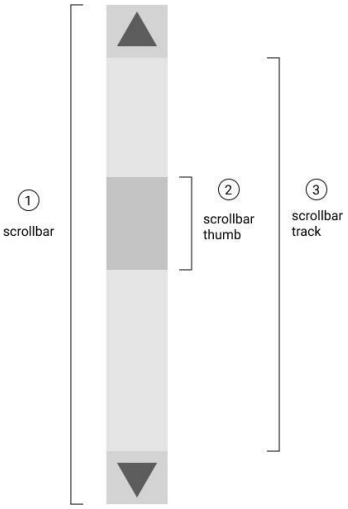
Scroll Bar Components and Functionality

Each scroll bar comprises several key components:

Scroll Bar Arrows: Clicking on the arrows at either end of the scroll bar initiates scrolling in the corresponding direction (up or down for vertical scroll bars, left or right for horizontal scroll bars).



Scroll Box (Thumb): The scroll box, also known as the thumb, is a movable indicator that represents the approximate position of the visible content within the entire document. Dragging the scroll box allows for direct navigation to a specific location in the content.



Scroll Bar Track: The scroll bar track provides the area along which the scroll box can move. Clicking within the scroll bar track, either above or below the scroll box, initiates scrolling in the corresponding direction.

Scroll Bar Integration into Windows Applications

Integrating scroll bars into Windows applications is a straightforward process.

During window creation using the CreateWindow function, the desired scroll bar type can be specified by including the corresponding window style identifiers:

- WS_VSCROLL: Vertical scroll bar.
- WS_HSCROLL: Horizontal scroll bar.
- WS_VSCROLL | WS_HSCROLL: Both vertical and horizontal scroll bars.

Once included, scroll bars are automatically positioned and sized according to the window's client area.

Windows handles all mouse interactions with the scroll bars, providing the necessary scrolling behavior.

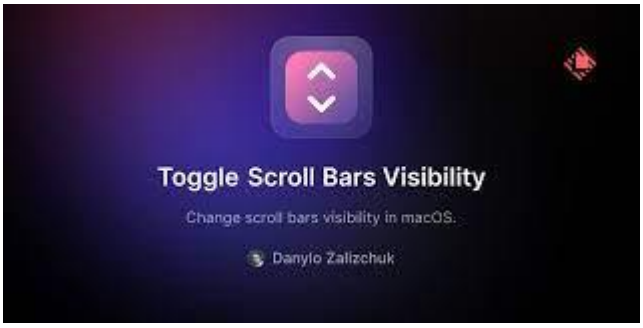
Keyboard Interface for Scroll Bars

While Windows manages mouse interactions with scroll bars, it does not provide an automatic keyboard interface. If you want to enable keyboard navigation using cursor keys, you'll need to implement the necessary logic within your application.

Enhancing Scroll Bar Usability

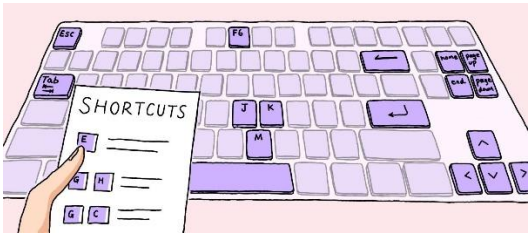
Scroll bars can be further enhanced to improve user experience:

Scroll Bar Visibility: Implement logic to hide scroll bars when the content fits within the visible area and show them only when scrolling is necessary.



Scroll Bar Feedback: Provide visual feedback during scrolling, such as highlighting the scroll box or changing its color, to indicate the current position within the content.

Keyboard Navigation: Implement keyboard shortcuts using cursor keys to provide an alternative scrolling method.



Scroll Bar Customization: Allow users to customize scroll bar appearance, such as color, size, and placement.

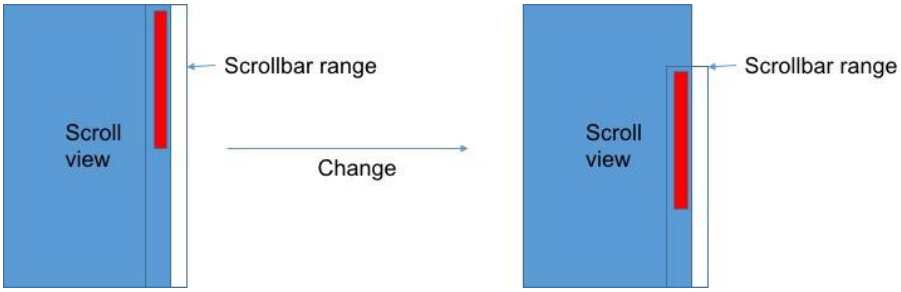
Scroll Bar Range and Position: Controlling Scrolling Behavior

Scroll bars play a crucial role in graphical user interfaces (GUIs) by enabling users to navigate through extensive content that exceeds the visible area of a window.

They provide a visual representation of the overall content and allow users to scroll smoothly through it.

Scroll Bar Range

The scroll bar range defines the minimum and maximum values associated with the scroll bar. These values represent the extent of the content that can be scrolled through. For instance, a scroll bar with a range of 0 to 100 indicates that the content can be scrolled from its initial position (0) to its full length (100).



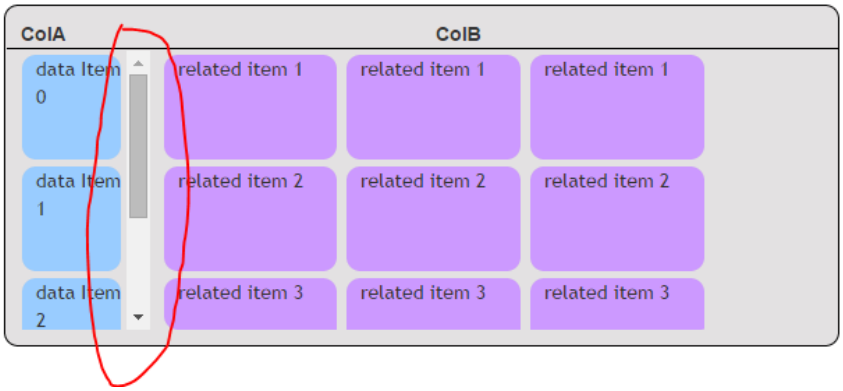
Setting Scroll Bar Range

The SetScrollRange function is used to modify the range of a scroll bar. It takes five arguments:

- hwnd: The window handle of the window containing the scroll bar.
- iBar: Specifies the scroll bar type (SB_VERT for vertical, SB_HORZ for horizontal).
- iMin: The new minimum value of the scroll bar range.
- iMax: The new maximum value of the scroll bar range.
- bRedraw: A Boolean value indicating whether to redraw the scroll bar based on the new range (TRUE for redrawing, FALSE to avoid unnecessary redraws).

Scroll Bar Position

The scroll bar position indicates the current location of the visible content within the overall content. It corresponds to the placement of the scroll box (thumb) along the scroll bar track. When the scroll box is at the top or left, the position is the minimum value of the range. At the bottom or right, the position is the maximum value of the range.



Setting Scroll Bar Position

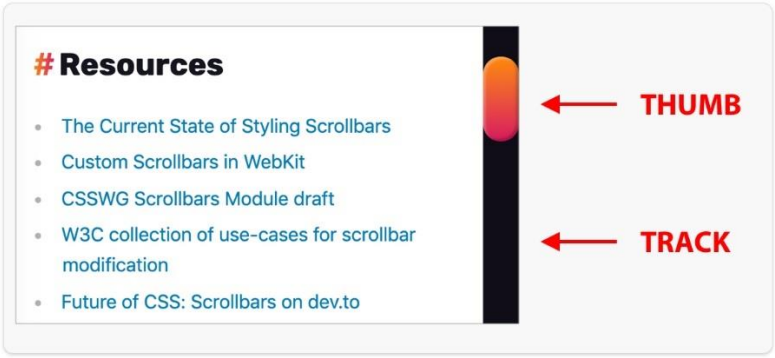
The SetScrollPos function is used to set the position of the scroll box within the scroll bar range. It takes four arguments:

- hwnd: The window handle of the window containing the scroll bar
- iBar: Specifies the scroll bar type (SB_VERT for vertical, SB_HORZ for horizontal).
- iPos: The new position of the scroll box within the range (must be between the minimum and maximum values).
- bRedraw: A Boolean value indicating whether to redraw the scroll bar based on the new position (TRUE for redrawing, FALSE to avoid unnecessary redraws).

Querying Scroll Bar Range and Position

The GetScrollRange and GetScrollPos functions can be used to retrieve the current range and position of a scroll bar, respectively.

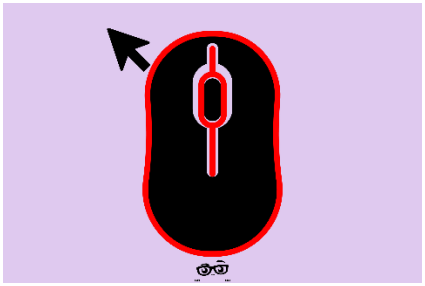
These functions provide access to the scroll bar's state and allow applications to adjust scrolling behavior accordingly.



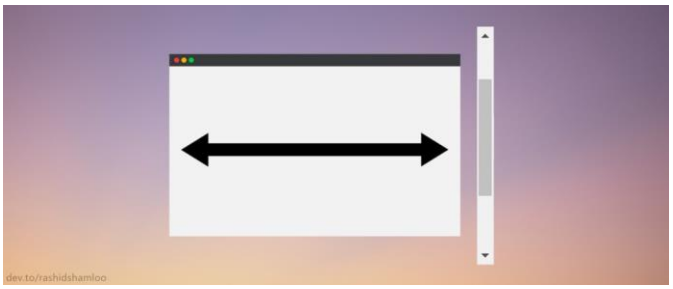
Windows' Responsibilities for Scroll Bars

Windows handles the low-level interactions with scroll bars, providing the necessary functionality and visual feedback to the user. Its responsibilities include:

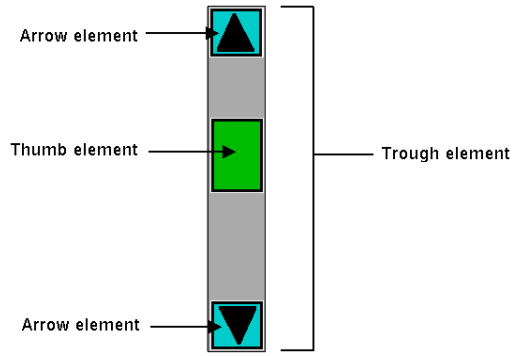
Mouse Message Processing: Windows handles all mouse events directed towards scroll bars, interpreting clicks and drags to initiate and control scrolling.



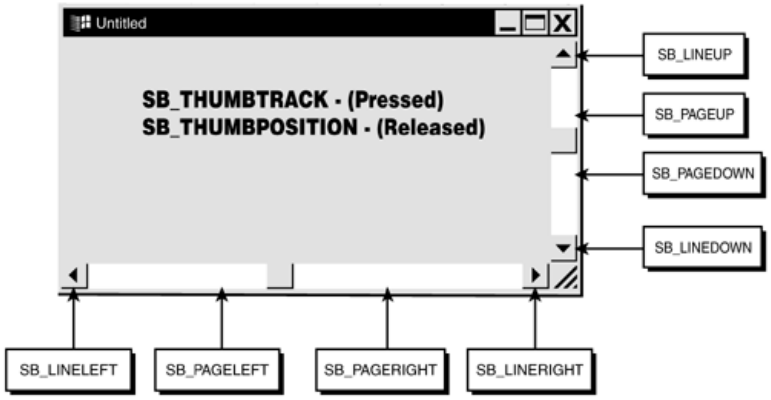
Reverse-Video Flash: When the user clicks within the scroll bar track, Windows provides a visual cue by flashing the area in reverse video, indicating that scrolling action is taking place.



Thumb Movement: As the user drags the scroll box (thumb) within the scroll bar, Windows updates its position accordingly, reflecting the user's intent to scroll through the content.



Scroll Bar Messages: Windows sends scroll bar messages to the window procedure of the window containing the scroll bar. These messages inform the application about scroll bar interactions, such as clicks, drags, and changes in position.



Your Program's Responsibilities for Scroll Bars

While Windows handles the low-level mechanics of scroll bars, your program plays a crucial role in managing the overall scrolling behavior and content updates:

Scroll Bar Range Initialization: You are responsible for setting the initial range of the scroll bar using the SetScrollRange function. The range defines the minimum and maximum values associated with the scroll bar, representing the extent of the content that can be scrolled through.

Scroll Bar Message Processing: Your program's window procedure needs to process the scroll bar messages sent by Windows. These messages provide information about scroll bar interactions and allow your program to react accordingly.

Scroll Bar Thumb Update: Based on the scroll bar messages and the current position of the scroll box (thumb), your program should update the position of the thumb to reflect the user's scrolling actions.

Content Area Updates: When the scroll bar position changes, your program should update the contents of the client area to display the corresponding portion of the content. This ensures that the visible content matches the user's scrolling actions.

Scroll Bar Messages: Communicating Scroll Bar Interactions

Scroll bars provide a crucial user interface element for navigating through extensive content that exceeds the visible area of a window.

When users interact with scroll bars, Windows sends messages to the window procedure of the window containing the scroll bar to inform the application about these interactions and enable appropriate responses.

WM_VSCROLL and WM_HSCROLL Messages

Two primary messages are used for scroll bar interactions:

- WM_VSCROLL: This message is sent for vertical scroll bars.
- WM_HSCROLL: This message is sent for horizontal scroll bars.

Each mouse action on the scroll bar triggers at least two of these messages: one when the mouse button is pressed and another when it is released. This allows the application to track the user's scrolling actions and update the content accordingly.

wParam and lParam Message Parameters

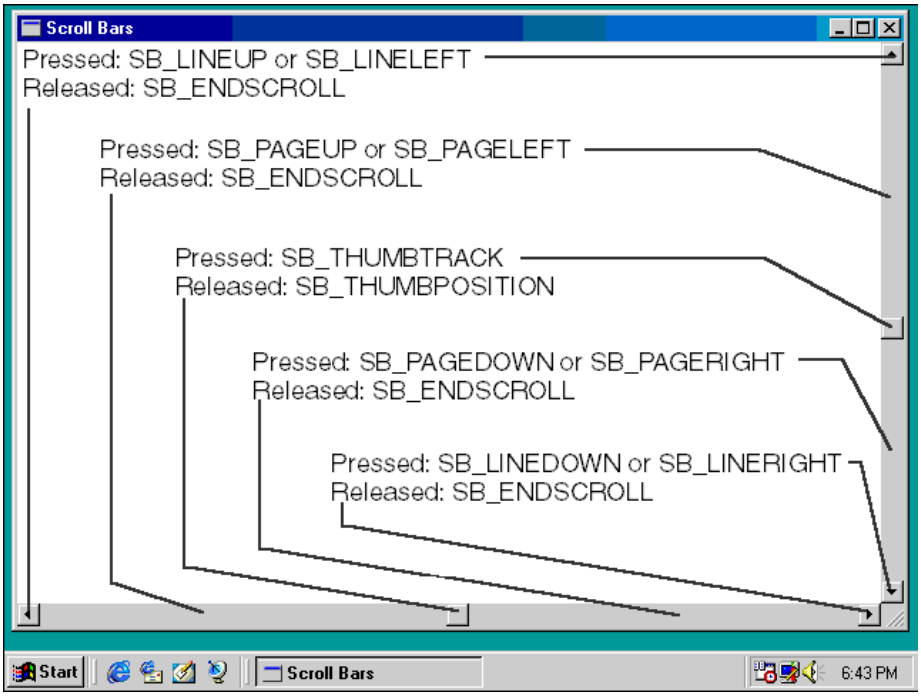
Like all Windows messages, WM_VSCROLL and WM_HSCROLL messages are accompanied by two additional parameters:

- **wParam:** This parameter contains the notification code, which indicates the specific action the user is performing with the scroll bar.
- **lParam:** For scroll bars created as part of the main window, lParam can be ignored. However, for scroll bars created as child windows, lParam provides additional information, typically related to the child window's position.

Notification Codes

Notification codes are represented by identifiers that begin with SB, indicating "scroll bar." These codes specify the type of action the user is performing with the scroll bar. Some of the commonly used notification codes include:

- **SB_LINEUP:** Scrolls one line up (vertical scroll bar).
- **SB_LINEDOWN:** Scrolls one line down (vertical scroll bar).
- **SB_PAGEUP:** Scrolls one page up (vertical scroll bar).
- **SB_PAGEDOWN:** Scrolls one page down (vertical scroll bar).
- **SB_THUMBPOSITION:** Indicates the current position of the scroll box (thumb).
- **SB_THUMBTRACK:** Indicates that the scroll box (thumb) is being dragged.
- **SB_TOP:** Scrolls to the top (vertical scroll bar).
- **SB_BOTTOM:** Scrolls to the bottom (vertical scroll bar).
- **SB_LEFT:** Scrolls to the left (horizontal scroll bar).
- **SB_RIGHT:** Scrolls to the right (horizontal scroll bar).
- **SB_ENDSCROLL:** Indicates that the scroll bar operation has ended.



The scrollbar image you sent shows a vertical and horizontal scrollbar with a scroll box (thumb) positioned at the top of the scrollbar track. This indicates that the visible content is at the top of the overall content.

The scrollbar track is divided into a number of equal segments, each representing a specific portion of the overall content.

The scroll box (thumb) is sized to indicate the relative amount of visible content. For instance, a large scroll box indicates that a significant portion of the content is visible, while a small scroll box indicates that only a small portion of the content is visible.

The scrollbar arrows at either end of the scrollbar track allow users to scroll up or down through the content.

Clicking on one of the arrows scrolls the content in the corresponding direction by a small amount.

Alternatively, users can drag the scroll box (thumb) along the scrollbar track to scroll through the content more quickly.

The scrollbar image also includes a number of other elements:

- Scroll bar track: The vertical bar that the scroll box (thumb) moves along.
- Scroll box (thumb): The movable indicator that represents the approximate position of the visible content within the overall content.
- Scroll bar arrows: The arrows at either end of the scrollbar track that allow users to scroll up or down through the content.
- Scroll bar range: The minimum and maximum values associated with the scroll bar, representing the extent of the content that can be scrolled through.
- Scroll bar position: The current location of the visible content within the overall content, indicated by the placement of the scroll box (thumb) along the scrollbar track.

By interpreting these notification codes, the application can determine the user's scrolling intent and make the necessary changes to the visible content. This ensures that the user's scrolling actions are reflected in the displayed information.

SB_THUMBTRACK and SB_THUMBPOSITION Notification Codes

When the user drags the scroll box (thumb), scroll bar messages with notification codes of SB_THUMBTRACK and SB_THUMBPOSITION are generated. These codes provide information about the current and final positions of the scroll box.

SB_THUMBTRACK: The high word of wParam represents the current position of the scroll box as the user is dragging it. This position is within the minimum and maximum values of the scroll bar range.

SB_THUMBPOSITION: The high word of wParam represents the final position of the scroll box when the user released the mouse button.

Processing SB_THUMBTRACK or SB_THUMBPOSITION Messages

Your program can choose to process either SB_THUMBTRACK or SB_THUMBPOSITION messages, but not both. The choice depends on how you want to handle the scrolling behavior.

Processing SB_THUMBTRACK Messages: If you process SB_THUMBTRACK messages, you can update the contents of the client area as the user is dragging the thumb, providing real-time feedback. However, this approach can be more demanding on the program's resources.

Processing SB_THUMBPOSITION Messages: If you process SB_THUMBPOSITION messages, you will only update the contents of the client area when the user stops dragging the thumb. This approach is less demanding but may not provide as smooth of a scrolling experience.

SB_TOP, SB_BOTTOM, SB_LEFT, and SB_RIGHT Notification Codes

The notification codes SB_TOP, SB_BOTTOM, SB_LEFT, and SB_RIGHT indicate that the scroll bar has been moved to its minimum or maximum position. However, these codes are never sent for scroll bars created as part of your application window.

32-bit Scroll Bar Range

While it is possible to use 32-bit values for the scroll bar range, the high word of wParam, which is only a 16-bit value, cannot properly indicate the position for SB_THUMBTRACK and SB_THUMBPOSITION actions. In this case, you need to use the GetScrollInfo function to retrieve the necessary information.

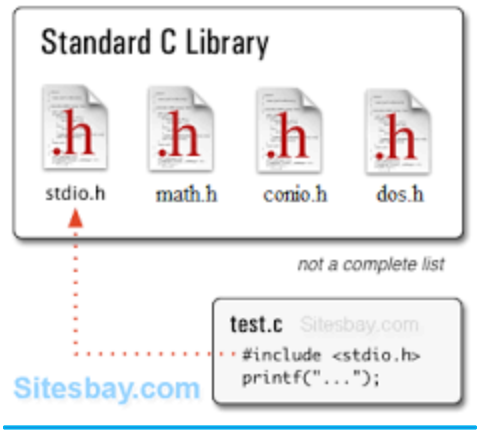
SYSMETS PROGRAM

You can find this code in the chapter 4 sysmets2 after running it, you will see the same program, but with a scroll bar that moves when clicked.In-depth explanation of the SYSMETS2.C code, with specific details and insights:

Header Files and Preprocessor Directives

windows.h: This header file provides essential Windows programming functionalities, including function declarations, data types, and macros. It's crucial for interacting with the Windows operating system and managing window elements.

sysmets.h: This custom header file presumably contains definitions and structures related to the system metrics being displayed. It serves as a local information repository for the application's specific data.



WndProc Function: Message Handling

WM_CREATE Message: This message indicates the creation of the application window. In response, the code performs essential initialization tasks:

- Text Metrics Retrieval: It retrieves the text metrics for the current font using GetTextMetrics. This provides information about character width and height, which is crucial for positioning text elements within the window.
- Scroll Bar Range Setup: It sets the scroll bar range using SetScrollRange. This defines the minimum and maximum values of the scroll bar, ensuring that the user can scroll through the entire list of system metrics.
- Initial Scroll Position: It initializes the current scroll position to 0 using SetScrollPos. This sets the starting point for displaying the system metrics within the visible area of the window.

WM_SIZE Message: This message is triggered when the window size changes. The code simply updates the cyClient variable, which stores the client area height, to reflect the new dimensions. This information is used later for adjusting the layout of the displayed text.

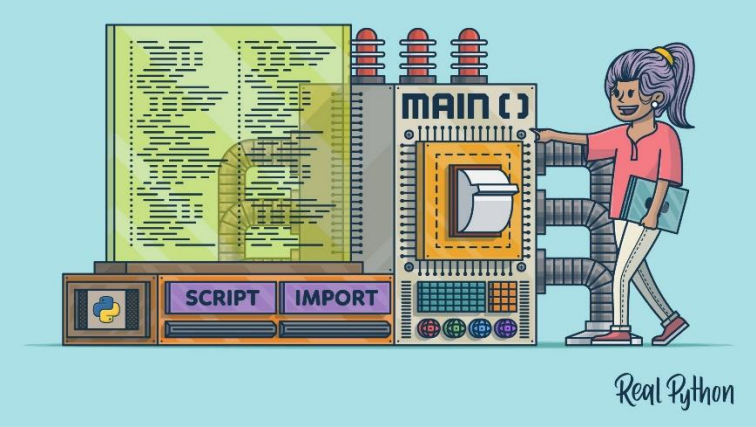
WM_VSCROLL Message: This message is generated when the user interacts with the scroll bar. The code handles various scroll bar actions:

- Line Up/Down: For line-by-line scrolling, it adjusts the iVscrollPos variable accordingly, either decreasing or increasing it by 1.
- Page Up/Down: For page-by-page scrolling, it adjusts the iVscrollPos variable by the height of the client area in character units. This ensures that a full page of system metrics is scrolled at a time.
- Thumb Position: When the user drags the scroll box (thumb), the code updates the iVscrollPos variable to the high word of wParam, which represents the new position of the thumb.
- Invalidation and Update: After adjusting the scroll position, the code ensures it stays within the valid range using max and min functions. If the scroll position changes, it updates the scroll bar position using SetScrollPos and invalidates the window's client area using InvalidateRect. This triggers a repaint to reflect the updated scroll position and display the corresponding system metrics.

WM_PAINT Message: This message prompts the code to repaint the window's client area. The code iterates through the system metrics array and draws each metric's label, description, and corresponding value using the TextOut function. This ensures that the system metrics are presented in a clear and organized manner.

WM_DESTROY Message: This message indicates the destruction of the application window. The code responds by sending a PostQuitMessage to the message queue, signaling the end of the application's execution.

Main Function: Application Entry Point



Window Class Registration: The WinMain function starts by registering the window class using RegisterClass. This defines the characteristics of the window, such as its style, window procedure, instance handle, icon, cursor, background brush, and class name.

Window Creation: It creates the window using the CreateWindow function. This allocates memory for the window structure, establishes its position and size, and associates it with the previously registered window class.

Window Visibility and Update: It displays the window using the ShowWindow function, making it visible to the user. It then updates the window's layout and contents using the UpdateWindow function.

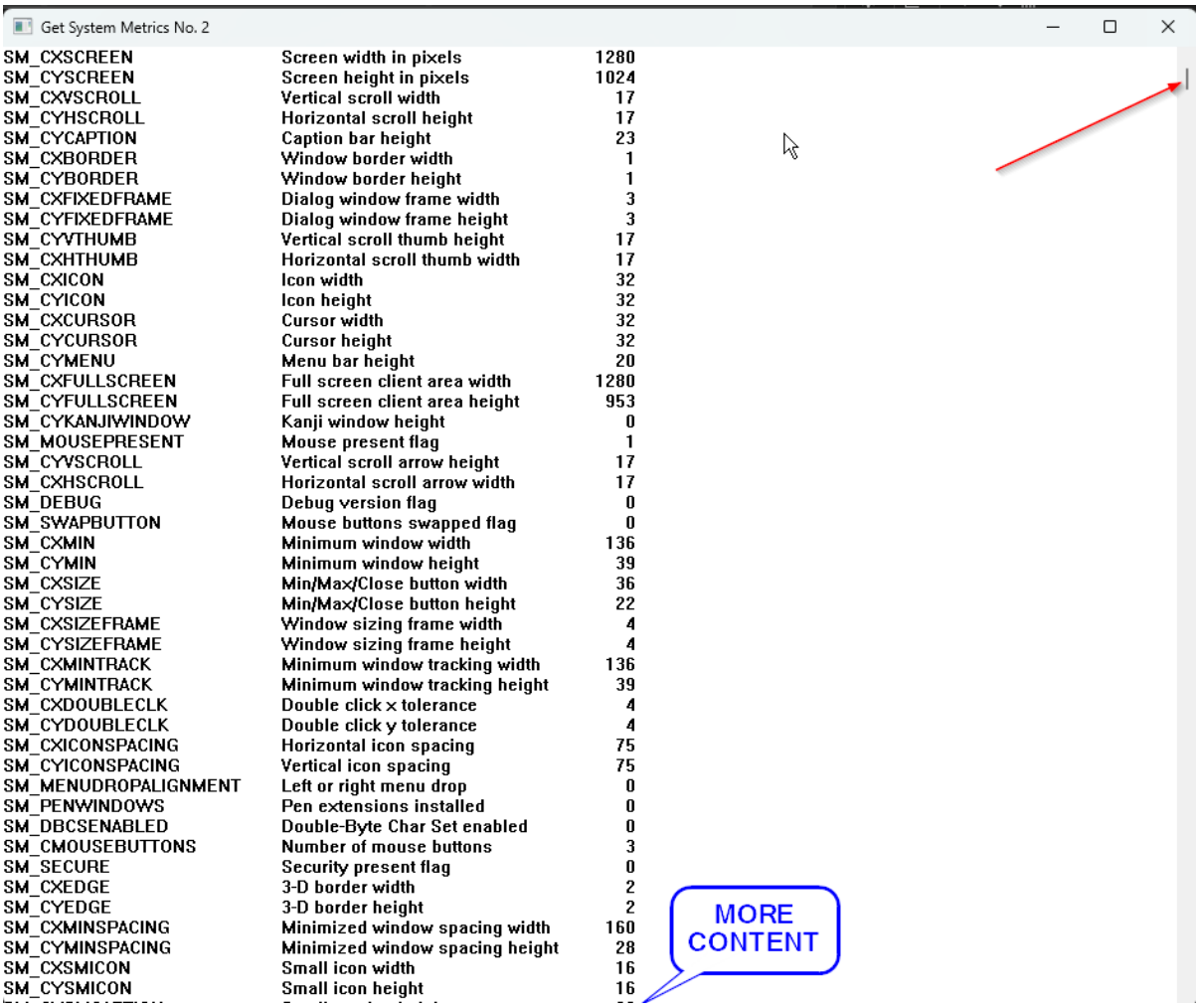
Message Loop: It enters a message loop using a while loop. This continuously retrieves messages from the message queue using GetMessage, translates them using TranslateMessage, and dispatches them to the WndProc callback function using DispatchMessage.

Application Termination: The message loop continues until a WM_QUIT message is received. In this case, the GetMessage function returns 0, signaling the end of the message queue. The WinMain function exits the loop and returns the message parameter from GetMessage.

Vertical Scroll Bar Integration

The CreateWindow call now includes the WS_VSCROLL window style in the third argument, adding a vertical scroll bar to the window.

This allows users to scroll through the list of system metrics if the entire list cannot be displayed within the visible area of the window.



Setting Scroll Bar Range and Initial Position

In response to the WM_CREATE message, the WndProc window procedure sets the scroll bar range and initial position using the SetScrollRange and SetScrollPos functions, respectively.

SetScrollRange: This function establishes the minimum and maximum values of the scroll bar, ensuring that it can cover the entire range of system metrics (0 to NUMLINES - 1).

SetScrollPos: This function initializes the current scroll position to 0, placing the first line of system metrics at the top of the client area.

Scroll Position Variable

A static variable named iVscrollPos is defined within the WndProc window procedure to track the current position of the scroll bar thumb. This variable plays a crucial role in managing the scroll bar behavior and updating the displayed system metrics.

Handling WM_VSCROLL Messages

The WndProc window procedure handles various WM_VSCROLL messages to respond to user interactions with the scroll bar:

SB_LINEUP and SB_LINEDOWN: For line-by-line scrolling, the scroll position is adjusted by 1, either decreasing or increasing it accordingly.

SB_PAGEUP and SB_PAGEDOWN: For page-by-page scrolling, the scroll position is adjusted by the height of the client area in character units (cyClient divided by cyChar). This ensures that a full page of system metrics is scrolled at a time.

SB_THUMBPOSITION: When the user drags the scroll box (thumb), the new thumb position is stored in the high word of wParam. This value is then assigned to iVscrollPos to reflect the current scroll position.

SB_ENDSCROLL and SB_THUMBTRACK: These messages are ignored in this version of the program.

Validating Scroll Position and Updating Window

After adjusting the scroll position based on the WM_VSCROLL message type, the code ensures that the value of iVscrollPos remains within the valid range of the scroll bar using the min and max macros.

If the scroll position has changed, it is updated using SetScrollPos, and the entire window is invalidated using InvalidateRect to trigger a repaint.

Updating Client Area

When repainting the client area in response to a WM_PAINT message, the y-coordinate of each line is calculated using the formula cyChar * (i - iVscrollPos).

This formula takes into account the current scroll position, ensuring that the appropriate lines of system metrics are displayed within the visible area of the window.

Painting in Response to WM_PAINT

The preferred approach in Windows programming is to handle all client area painting in response to a WM_PAINT message.

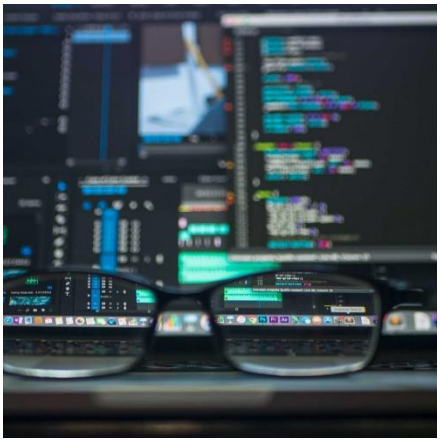
This ensures that the entire **client area is repainted consistently** and efficiently.

Duplicating painting logic in response to other messages can lead to redundancy and potential inconsistencies.

Discipline of WM_PAINT-based Painting

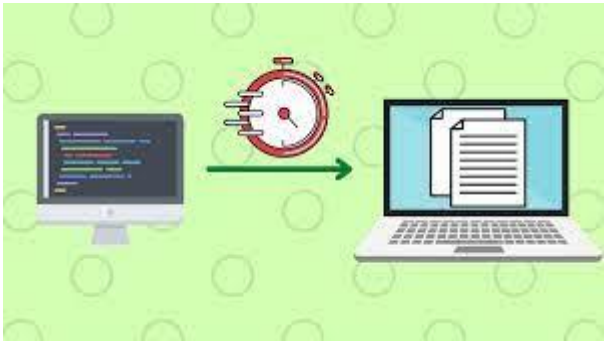
While it may seem indirect, this approach offers several benefits:

Code Clarity and Maintainability: By centralizing painting logic in the WM_PAINT handler, the code becomes more organized and easier to maintain.



Consistent Repainting: The WM_PAINT handler should be capable of repainting the entire client area, ensuring that the window's contents are always displayed correctly.

Efficiency: Windows optimizes painting operations, so relying on WM_PAINT messages for repainting can lead to better performance.



InvalidateRect for Invalidating Client Area

When a program needs to trigger a repaint of a specific area or the entire client area, it can use the InvalidateRect function.

This marks the specified rectangles as invalid, prompting Windows to generate a WM_PAINT message.

Considerations with InvalidateRect

While InvalidateRect is useful for invalidating specific areas, it's important to note that WM_PAINT messages are handled with **lower priority** than other messages.

This means that **if the system is busy with other tasks**, it may take some time for the WM_PAINT message to be processed, resulting in visible "holes" in the window.

UpdateWindow for Immediate Repainting

To address this delay, you can use the UpdateWindow function after calling InvalidateRect.

This causes the window procedure to be immediately called with a WM_PAINT message, bypassing the message queue and ensuring immediate repainting of the invalidated area.

UpdateWindow's Role in Window Creation

UpdateWindow is also used during window creation to trigger the initial painting of the client area.

When a window is first created, the entire client area is marked as invalid, and UpdateWindow directs the window procedure to paint it accordingly.

Conclusion

SYSMETS2.C introduces vertical scrolling functionality, allowing users to navigate through the list of system metrics efficiently. The code effectively manages the scroll bar position and updates the displayed content accordingly.

However, this implementation is still wasteful and inefficient, as it renders lines of text outside the client area. Improvements will be made to address this issue in subsequent versions.

BUILDING A BETTER SCROLLBAR

Obsolete Scroll Bar Functions

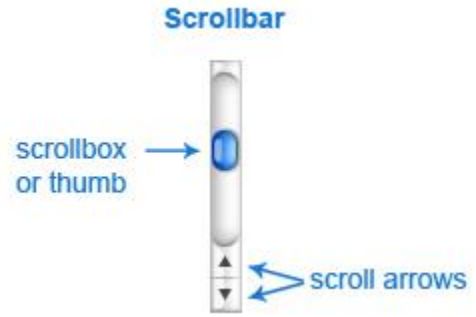
The scroll bar functions introduced in earlier versions of Windows, including SetScrollRange, SetScrollPos, GetScrollRange, and GetScrollPos, are considered obsolete in the Win32 API.

While they are still functional and can handle 32-bit arguments, they have been superseded by newer functions that offer additional capabilities.

New Scroll Bar Functions: SetScrollInfo and GetScrollInfo

The Win32 API introduced two new scroll bar functions: SetScrollInfo and GetScrollInfo. These functions provide all the functionality of the older functions and add two significant enhancements:

Thumb Size Control: The size of the scroll bar thumb can now be dynamically adjusted based on the page size, which represents the portion of the document displayed in the window. This allows for a more **visually intuitive representation** of the document's size and scroll position.



32-bit Thumb Position Retrieval: The GetScrollInfo function provides access to the actual 32-bit value of the thumb position, even if the range exceeds 65,536. This addresses a limitation of the older functions, which only provided 16 bits of thumb position information.

Benefits of SetScrollInfo and GetScrollInfo. These new functions offer several advantages over the obsolete ones:

Enhanced Functionality: They provide greater control over the appearance and behavior of scroll bars, including thumb size adjustment and accurate thumb position retrieval.

Future-Proofing: They are designed to handle larger ranges and provide more flexibility for future scroll bar implementations.

Improved User Experience: By enabling dynamic thumb sizing and accurate position tracking, they contribute to a more intuitive and responsive user experience.

Here is an in-depth explanation of the syntax and usage of the SetScrollInfo and GetScrollInfo functions:

Function Syntax

```
SetScrollInfo(hwnd, iBar, &si, bRedraw);
GetScrollInfo(hwnd, iBar, &si);
```

Parameters

hwnd: Handle of the window that contains the scroll bar.

iBar: Specifies which scroll bar to set or get information for. It can be SB_VERT for a vertical scroll bar, SB_HORZ for a horizontal scroll bar, or SB_CTL for a scroll bar control.

&si: A pointer to a SCROLLINFO structure that contains the scroll bar information to set or get.

bRedraw: (For SetScrollInfo only) A Boolean value indicating whether to redraw the scroll bar after applying the changes. TRUE to redraw, FALSE to skip redrawing.

SCROLLINFO Structure

The SCROLLINFO structure, defined as follows, holds the scroll bar information:

```
typedef struct tagSCROLLINFO {
    UINT cbSize; // Size of the SCROLLINFO structure
    UINT fMask; // Flags indicating which fields to set or get
    int nMin; // Minimum value of the scroll bar range
    int nMax; // Maximum value of the scroll bar range
    UINT nPage; // Page size, used for proportional thumb sizing
    int nPos; // Current position of the scroll bar thumb
    int nTrackPos;
    // Current tracking position of the scroll bar thumb
    //(only valid during SB_THUMBTRACK or SB_THUMBPOSITION messages)
} SCROLLINFO, * PSCROLLINFO;
```

Setting the cbSize Field

Before calling either SetScrollInfo or GetScrollInfo, you must initialize the cbSize field of the SCROLLINFO structure to the size of the structure:

```
si.cbSize = sizeof(si);
```

This allows for future compatibility with potential enhancements to the structure without breaking existing programs.

Example Usage

To set the scroll bar range for a vertical scroll bar:

```
SCROLLINFO si;
si.cbSize = sizeof(si);
si.fMask = SIF_RANGE;
si.nMin = 0;
si.nMax = 100;
SetScrollInfo(hwnd, SB_VERT, &si, TRUE);
```

To retrieve the current position of a horizontal scroll bar:

```
SCROLLINFO si;
si.cbSize = sizeof(si);
si.fMask = SIF_POS;
GetScrollInfo(hwnd, SB_HORZ, &si);
int currentPos = si.nPos;
```

Flag Syntax and Usage

The fMask field of the SCROLLINFO structure is used to specify which fields of the structure to set or get. The flags begin with the SIF prefix and can be combined using the bitwise OR (|) operator.

SIF_RANGE Flag

This flag is used to set or get the scroll bar range.

When used with SetScrollInfo, the nMin and nMax fields of the SCROLLINFO structure must be set to the desired minimum and maximum values, respectively.

When used with GetScrollInfo, the nMin and nMax fields will be updated to reflect the current scroll bar range.

SIF_POS Flag

This flag is used to set or get the current position of the scroll bar.

When used with SetScrollInfo, the nPos field of the SCROLLINFO structure must be set to the desired scroll bar position.

When used with GetScrollInfo, the nPos field will be updated to reflect the current scroll bar position.

SIF_PAGE Flag

This flag is used to set or get the page size, which determines the proportional size of the scroll bar thumb.

When used with SetScrollInfo, the nPage field of the SCROLLINFO structure must be set to the desired page size.

When used with GetScrollInfo, the nPage field will be updated to reflect the current page size.

SIF_TRACKPOS Flag

This flag is used to get the current 32-bit thumb position when processing a WM_VSCROLL or WM_HSCROLL message with a notification code of SB_THUMBTRACK or SB_THUMBPOSITION.

The nTrackPos field of the SCROLLINFO structure will be updated to reflect the current thumb position.

SIF_DISABLENOSCROLL Flag

This flag is used with SetScrollInfo to disable the scroll bar instead of hiding it when the scroll bar arguments would normally make it invisible.

SIF_ALL Flag

This flag is a combination of SIF_RANGE, SIF_POS, SIF_PAGE, and SIF_TRACKPOS. It is useful for setting the scroll bar arguments during a WM_SIZE message or processing a scroll bar message.

SIF stands for Scroll Information Flag. It is a bitmask used in the SetScrollInfo() and GetScrollInfo() functions to specify which fields of the SCROLLINFO structure are to be set or retrieved.

Here's a table summarizing the flag usage:

Flag	Usage
`SIF_RANGE`	Set or get the scroll bar range
`SIF_POS`	Set or get the current position of the scroll bar
`SIF_PAGE`	Set or get the page size
`SIF_TRACKPOS`	Get the current 32-bit thumb position
`SIF_DISABLENOSCROLL`	Disable the scroll bar instead of hiding it
`SIF_ALL`	Combination of `SIF_RANGE`, `SIF_POS`, `SIF_PAGE`, and `SIF_TRACKPOS`

The SetScrollInfo and GetScrollInfo functions represent a significant advancement in scroll bar control and provide a more powerful and flexible approach to managing scroll bars in Windows applications.

Determining the Maximum Scroll Position

In SYSMETS2, the scroll bar range is unnecessarily large, extending beyond the visible area of the client area.

To address this, we can dynamically adjust the scroll bar range based on the window size and the height of the text elements.

Calculating Scroll Bar Range Using WM_SIZE

Instead of setting the scroll bar range during WM_CREATE, we can defer it until the WM_SIZE message is received, ensuring that the range is adjusted based on the actual window dimensions.

```
iVscrollMax = max(0, NUMLINES - cyClient / cyChar);
SetScrollRange(hwnd, SB_VERT, 0, iVscrollMax, TRUE);
```

In this code, NUMLINES represents the total number of lines of information, cyClient is the client area height, and cyChar is the character height. The max function ensures that the maximum scroll position (iVscrollMax) is never less than 0.

Using Scroll Bar Page Size for Automatic Range Adjustment

The new scroll bar functions introduce the concept of scroll bar page size, which represents the portion of the document displayed in the window.

This feature simplifies the calculation of the maximum scroll position and ensures that the scroll bar thumb is sized proportionally to the visible content.

```
si.cbSize = sizeof(SCROLLINFO);
si.fMask = SIF_RANGE | SIF_PAGE;
si.nMin = 0;
si.nMax = NUMLINES - 1;
si.nPage = cyClient / cyChar;
SetScrollInfo(hwnd, SB_VERT, &si, TRUE);
```

In this code, si is a SCROLLINFO structure, NUMLINES is the total number of lines, and cyClient and cyChar are as defined earlier.

The SetScrollInfo function automatically adjusts the maximum scroll position based on the page size (si.nPage), ensuring that the scroll bar range is appropriate for the visible content.

Handling Large Page Sizes

If the page size is equal to or larger than the scroll bar range, Windows automatically hides the scroll bar. This is a convenient behavior that eliminates unnecessary scroll bar controls when the entire content is visible.

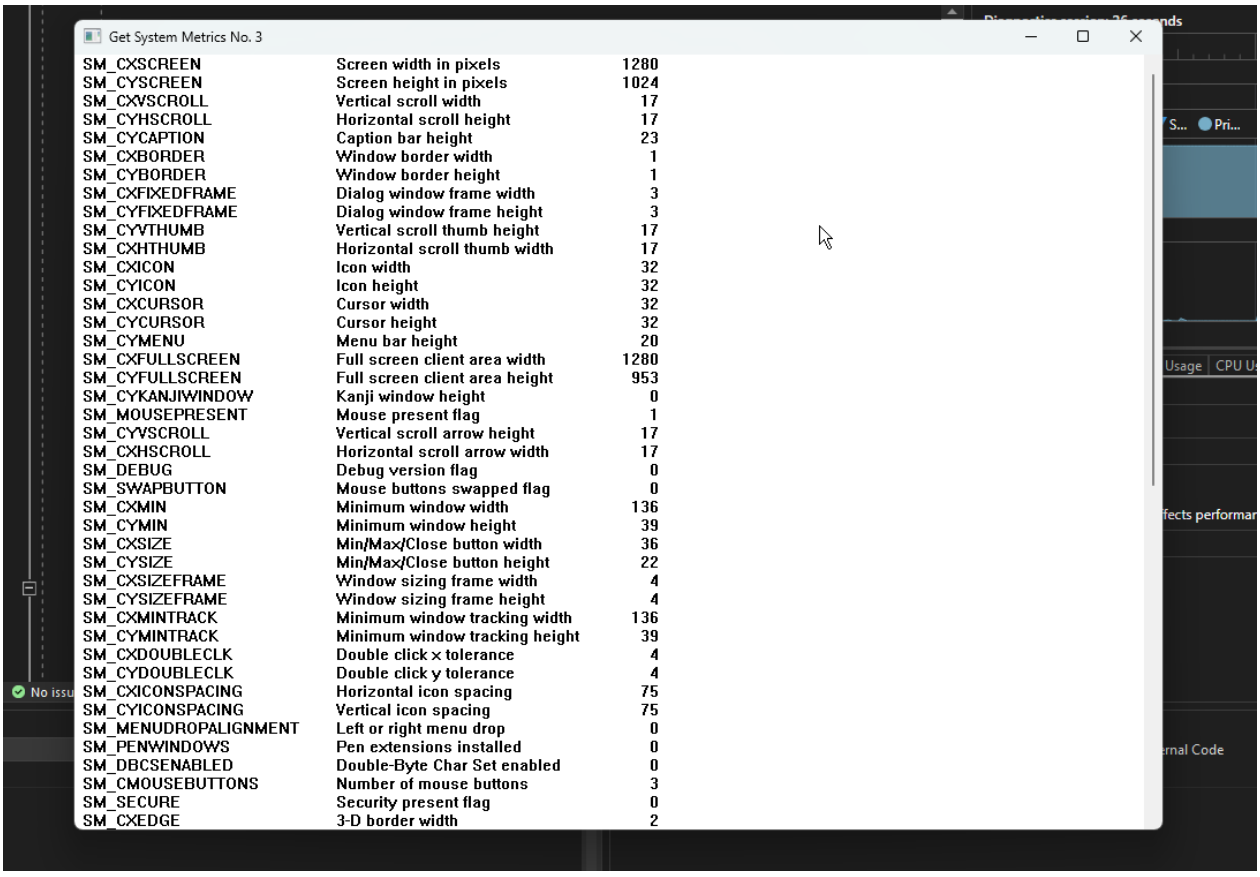
Disabling Scroll Bar Instead of Hiding

If you prefer to keep the scroll bar visible even when the page size is large, you can use the SIF_DISABLENOSCROLL flag when calling SetScrollInfo.

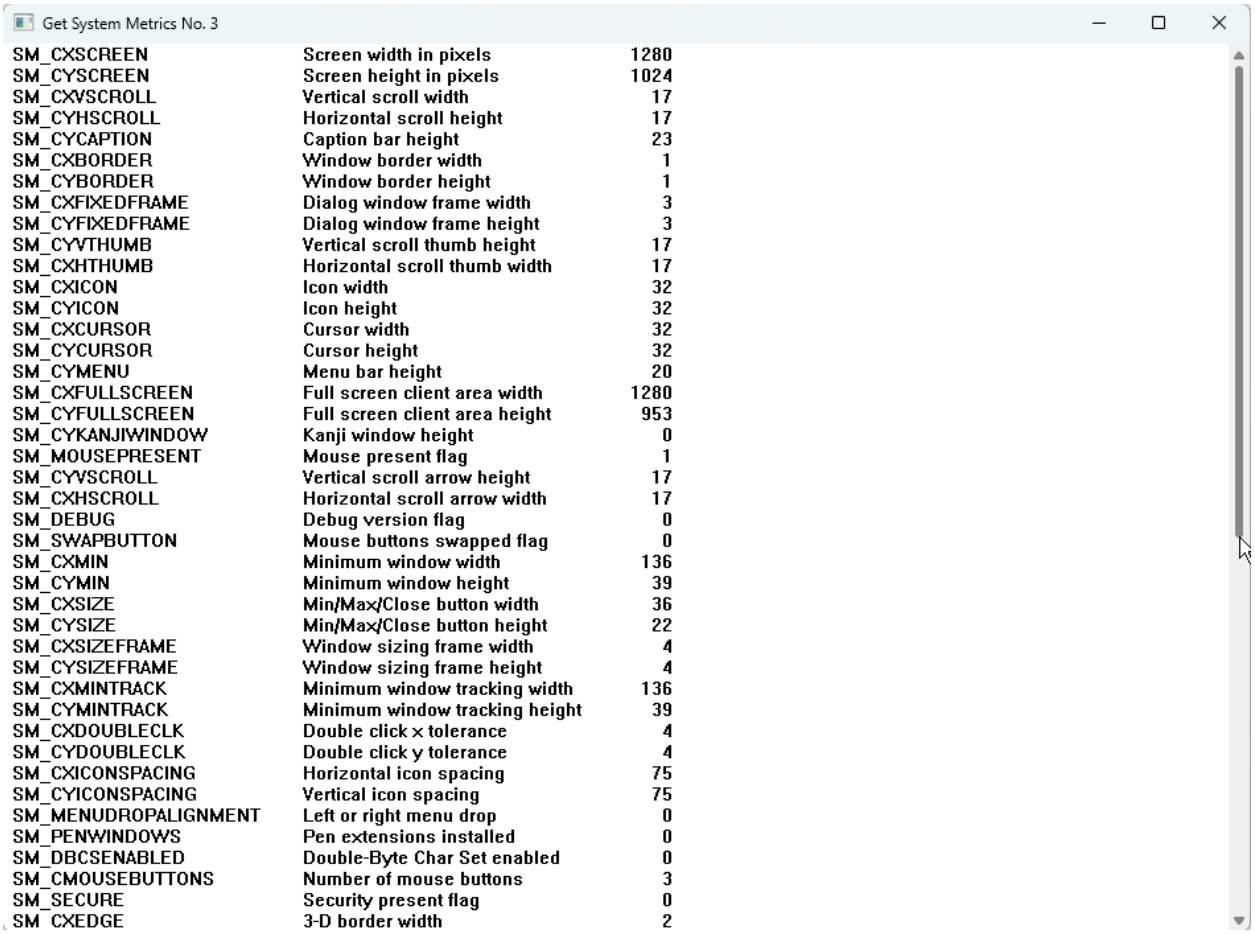
This will disable the scroll bar instead of hiding it, allowing users to manually scroll through the content if needed.

By employing these techniques, you can prevent excessive scrolling and ensure that the scroll bar behavior is consistent with the actual amount of visible content.

I was able to run sysmets3 code. It's actually a better scrollbar with a hide feature:



And an unhide feature, and it flows better than the former one:



Reliance on Windows for Scroll Bar Management

SYSMETS3 simplifies scroll bar management by relying on Windows to handle most of the scroll bar information and boundary checking. Instead of manually tracking scroll bar positions and enforcing limits, the program retrieves the current scroll bar information at the beginning of WM_VSCROLL and WM_HSCROLL processing.

Handling Scroll Bar Notifications

The program adjusts the scroll bar position based on the notification code received in these messages. For instance, if the notification code indicates that the scroll bar thumb has been moved, the program adjusts the scroll bar position accordingly.

Setting and Retrieving Scroll Bar Positions

After adjusting the scroll bar position, SYSMETS3 calls SetScrollInfo to inform Windows of the new position. This allows Windows to maintain consistent scroll bar information across the system. To verify the updated position, the program calls GetScrollInfo and checks if the position was adjusted by Windows due to any boundary constraints.

Utilizing ScrollWindow for Efficient Scrolling

Instead of repainting the entire client area whenever scrolling occurs, SYSMETS3 employs the ScrollWindow function to perform the scrolling operation more efficiently. This function scrolls the client area horizontally and vertically based on the specified amounts.

Invalidation and WM_PAINT Message Handling

By setting the last two arguments of ScrollWindow to NULL, the program indicates that the entire client area should be scrolled. Windows automatically invalidates the exposed rectangle in the client area, triggering a WM_PAINT message. This eliminates the need to explicitly call InvalidateRect.

Ignoring SB_THUMBTRACK for Horizontal Scrolling

In WM_HSCROLL processing, SYSMETS3 ignores the SB_THUMBTRACK notification code and only responds to SB_THUMBPOSITION. This means that the program does not scroll the window horizontally while the user is dragging the thumb on the horizontal scroll bar. Instead, it updates the scroll bar position only when the user releases the mouse button.

Responsive Vertical Scrolling

In contrast to horizontal scrolling, SYSMETS3 scrolls the window vertically in real-time as the user drags the thumb on the vertical scroll bar. This is achieved by capturing SB_THUMBTRACK messages and updating the scroll bar position accordingly.

Potential Performance Issues with Frequent Scrolling

While responsive scrolling provides a more intuitive user experience, it can lead to performance issues if users rapidly drag the scroll bar thumb back and forth. This is because the program continuously updates the scroll position and repaints the client area in response to these rapid movements.

Considering Alternative Scrolling Strategies for Slow Machines

To optimize performance on slower machines, SYSMETS3 could incorporate the SB_SLOWMACHINE flag when calling GetSystemMetrics. This flag provides information about the system's processing speed and could be used to trigger less frequent scrolling updates or utilize alternative scrolling mechanisms.

Efficient WM_PAINT Handling

To optimize WM_PAINT processing, SYSMETS3 identifies the lines within the invalid rectangle and rewrites only those lines instead of repainting the entire client area. This approach reduces the amount of repainting required and improves overall performance.

USERS AND SCROLLBAR

User Preferences and Mouse Usage

While mice have become the standard input device for personal computers, some users still prefer to operate their systems primarily using the keyboard. This preference may be due to familiarity, comfort, or accessibility considerations.

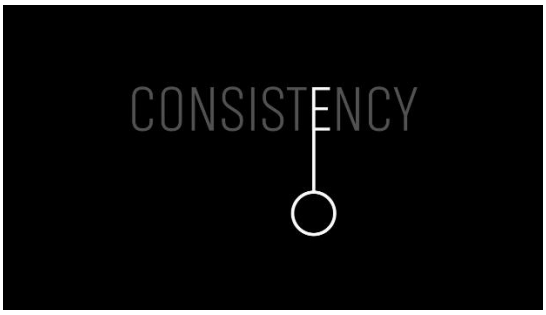
Early Windows and Mouseless Operation

In the early versions of Windows, a significant portion of users relied solely on the keyboard for navigation and interaction. This was reflected in the design of the operating system and many applications, which were designed to be fully functional without mouse input.

Providing keyboard alternatives for scroll bar operations offers several advantages:

Accessibility: Users with limited hand mobility or dexterity may find it easier to use keyboard shortcuts rather than relying on a mouse.

Consistency: Keyboard shortcuts provide a consistent and predictable way to interact with scroll bars, regardless of the user's preference or ability to use a mouse.



Efficiency: Experienced users can often navigate through scroll bars more quickly using keyboard shortcuts than with a mouse.

Maintaining Compatibility with Older Windows Behavior

The inclusion of code that handles WM_VSCROLL messages with notification codes SB_TOP and SB_BOTTOM in SYSMETS3 is likely intended to maintain compatibility with older versions of Windows, where these messages were sent for scroll bar operations.

Future Enhancements with Keyboard Interface

In the next chapter, you'll explore techniques for implementing a keyboard interface for scroll bar operations, allowing users to navigate through the content using keyboard shortcuts. This will enhance the program's accessibility and provide alternative input methods for those who prefer or require them.