MOUSE

The mouse is a pointing device with one or more buttons that allows users to interact with computers.

Despite the emergence of alternative input devices like touchscreens and light pens, the mouse remains the primary input device for desktop computers.

This is due to its versatility, ease of use, and affordability.

Windows 98 supports mice with one, two, or three buttons. It can also use joysticks or light pens to mimic mouse input. In the early days of Windows, applications were designed to work with one-button mice, as many users didn't have two-button mice.

However, two-button mice have become the standard, and most applications now utilize the second button for various functions, such as invoking context menus or performing special drag operations.

Determining Mouse Presence and Button Count

To determine if a mouse is present, you can use the GetSystemMetrics function with the SM_MOUSEPRESENT parameter.

However, this function always returns TRUE in Windows 98, regardless of whether a mouse is attached. To get accurate information, use this function in Microsoft Windows NT.

To determine the number of buttons on the installed mouse, use the GetSystemMetrics function with the SM_CMOUSEBUTTONS parameter.

This function should also return 0 if a mouse is not installed. However, in Windows 98, the function returns 2 if a mouse is not installed.

Left-Handed Mouse Users

Left-handed users can switch the mouse buttons using the Windows Control Panel. While an application can determine this by calling GetSystemMetrics with the **SM_SWAPBUTTON** parameter, this is usually unnecessary. The button triggered by the index finger is considered the left button, even if it's physically on the right side of the mouse.

Setting Mouse Parameters

You can set other mouse parameters, such as the double-click speed, using the SystemParametersInfo function. This function allows you to set or obtain various mouse-related settings from within your Windows application.

Fun facts:

- The mouse cursor is a small bitmapped picture that moves on the display as the user moves the mouse.
- The hot spot is the single-pixel point on the cursor that indicates the precise location on the display.
- Windows supports several predefined mouse cursors, such as IDC_ARROW, IDC_CROSS, and IDC_WAIT.
- Programmers can also design their own custom cursors.
- The default cursor for a particular window is specified when defining the window class structure.
- Common mouse actions include clicking, double-clicking, and dragging.
- On a three-button mouse, the buttons are called the left button, the middle button, and the right button.
- On a two-button mouse, there is only a left button and a right button.
- The single button on a one-button mouse is a left button.
- The plural of "mouse" is a matter of debate, with both "mice" and "mouses" being considered acceptable.
- The Microsoft Manual of Style for Technical Publications recommends avoiding the plural "mice" and using "mouse devices" instead.

Overview

Client-area mouse messages are notifications sent by Windows to a window's procedure when mouse events occur within the window's client area. These messages provide information about the mouse's position, button state, and modifier keys.

Mouse Messages vs. Keyboard Messages

Unlike keyboard messages, which are only sent to the window that has the input focus, mouse messages are sent to any window that the mouse cursor passes over or clicks within, regardless of whether the window is active or has the input focus. This allows windows to respond to mouse interactions even when they are not in the foreground.

Types of Mouse Messages

Windows defines 21 mouse messages, but only 10 of them relate to the client area. These messages can be categorized into three types:

- Mouse movement: The WM_MOUSEMOVE message is sent when the mouse cursor moves within the client area.
- Button press/release: When a mouse button is pressed or released within the client area, the window procedure receives one of the messages shown in the table in the text.
- Double-click: Double-click messages are sent only if the window class has been defined to receive them.

Extracting Mouse Position

The value of lParam in the client-area mouse messages contains the position of the mouse cursor. The low word is the x-coordinate, and the high word is the y-coordinate, both relative to the upper-left corner of the client area. These values can be extracted using the LOWORD and HIWORD macros.

Extracting Mouse Button State and Modifier Keys

The value of wParam in the client-area mouse messages indicates the state of the mouse buttons and the Shift and Ctrl keys. These states can be tested using the bit masks defined in the WINUSER.H header file, which have the prefix "MK" for "mouse key".

WM_LBUTTONDOWN Message and Active Window

Clicking the left mouse button in the client area of an inactive window causes Windows to make the clicked window active and then send the WM_LBUTTONDOWN message to the window procedure. This allows the window to respond to the click even if it was not previously active.

WM_LBUTTONDOWN and WM_LBUTTONUP Messages

A window procedure may receive a WM_LBUTTONDOWN message without a corresponding WM_LBUTTONUP message, or vice versa. This can happen if the mouse button is pressed or released outside the window's client area.

Button	Pressed	Released	Pressed (Second Click)
Left	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
Middle	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK
Right	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCLK

Mouse Capture

A window procedure can capture the mouse and continue to receive mouse messages even when the mouse is outside the window's client area. This is useful for operations that require continuous mouse tracking, such as drawing or dragging.

System Modal Message Boxes and Dialog Boxes

When a system modal message box or dialog box is on the display, no other program can receive mouse messages. These modal boxes prevent switching to another window while they are active.

Here's an example of how to handle the WM_LBUTTONDOWN message in C code using WinAPI:

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)

{
    switch (msg) {
        case WM_LBUTTONDOWN:
        {
            int x = LOWORD(lParam);
            int y = HIWORD(lParam);

            // Handle the left mouse button click at position (x, y)
            ...
        }
        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
}
```

This code snippet defines a window procedure function called WindowProc that handles the WM_LBUTTONDOWN message. When the left mouse button is pressed within the window's client area, the function extracts the mouse coordinates (x, y) from the lParam parameter and performs the corresponding action.

Connect folder in Chapter 7 has the code. Here's the video for it's working…

Connect Mouse chapter 7 Part 1.mp4

The CONNECT program is a simple mouse-driven demo program that allows users to connect dots on the screen. The program processes three mouse messages:

- WM_LBUTTONDOWN: Clears the client area.
- WM_MOUSEMOVE: If the left mouse button is down, draws a black dot on the client area at the mouse position and saves the coordinates.
- WM_LBUTTONUP: Connects every dot shown in the client area to every other dot.

Notes:

- The program uses three GDI function calls: SetPixel, MoveToEx, and LineTo.
- The program stores a maximum of 1000 points.
- The program switches to an hourglass cursor while processing the WM_PAINT message.
- The program calls ShowCursor twice to change the cursor visibility.
- The term "tracking" refers to the way programs handle mouse movement.

Additional Points

- The CONNECT program works best for a curved pattern of a few dots.
- If you move the mouse cursor out of the client area before releasing the button, CONNECT does not connect the dots.
- You can continue a design after releasing the button outside the client area by pressing the left button again while the mouse is outside the client area and then moving the mouse back inside.
- The CONNECT program might take some time to draw the lines, depending on your hardware.
- Because CONNECT is a preemptive multitasking environment, you can switch to other programs while the program is busy.

Processing Shift Keys with wParam

The CONNECT program utilizes the wParam value to determine the state of the Shift keys when handling the WM_MOUSEMOVE message. This value is obtained from the mouse message and provides information about the mouse button presses and the Shift and Ctrl keys.

To check if the Shift key is pressed, you can perform a bitwise AND operation between wParam and MK_SHIFT. The MK_SHIFT constant represents the state of the Shift key. If the result of the operation is non-zero (TRUE), then the Shift key is down.

Here's an example of how to check if the Shift key is pressed:

```
if (wParam & MK_SHIFT) {
  // Shift key is down
} else {
  // Shift key is not down
}
```

Advanced Shift Key Handling

You can also use wParam to check for specific combinations of keys, such as Shift and Ctrl together. For instance, if you need to differentiate between Shift, Ctrl, and both Shift and Ctrl being pressed, you can use nested if-else statements to handle each case separately.

Here's an example of how to check for Shift and Ctrl key combinations:

```
if (wParam & MK_SHIFT) {
  if (wParam & MK_CONTROL) {
    // Shift and Ctrl keys are down
  } else {
    // Shift key is down, Ctrl key is not
  }
} else {
  if (wParam & MK_CONTROL) {
    // Ctrl key is down, Shift key is not
  } else {
    // Neither Shift nor Ctrl key is down
  }
}
```

Emulating Right Button Click with Shift and Left Button

If you want to support both left and right mouse buttons in your program and accommodate users with a one-button mouse, you can make the Shift key in combination with the left button act like the right button.

This can be done by checking for the Shift key state in the WM_LBUTTONDOWN message handler and then handling it accordingly.

Here's an example of how to emulate the right button click:

```
case WM_LBUTTONDOWN:
  if (!(wParam & MK_SHIFT)) {
    // Left button logic
  } else {
    // Shift key is down, treat it as right button click
    // Handle right button logic
  }
  return 0;
```

Using GetKeyState for Mouse and Key States

The GetKeyState function can also be used to retrieve the state of the mouse buttons or shift keys using the virtual key codes VK_LBUTTON, VK_RBUTTON, VK_MBUTTON, VK_SHIFT, and VK_CONTROL. If the value returned from GetKeyState is negative, the corresponding button or key is down.

Unlike wParam, GetKeyState provides the current state of the mouse buttons or keys, even if they were pressed in a previous message. This allows you to check the state of a button or key at any point during message processing.

However, it's important to note that GetKeyState should not be used to wait for a button or key press. Instead, you should rely on message-based processing and handle button or key presses within the respective message handlers.

Here's an example of how to check the state of the left button using GetKeyState:

```
if (GetKeyState(VK_LBUTTON) < 0) {
  // Left button is down
} else {
  // Left button is not down
}
```

In summary, processing Shift keys and mouse button states in Windows applications can be achieved using both wParam and GetKeyState. wParam provides information about the state of the buttons and keys within the current message, while GetKeyState provides the current state of the buttons and keys, regardless of the current message.

Understanding Mouse Double-Clicks

A mouse double-click is a common interaction technique in Windows applications. It involves quickly clicking the mouse button twice in close proximity, typically within a specified time interval called the "double-click speed." The default double-click speed is set by the system, but users can modify it through the Control Panel.

Handling Double-Click Messages

To enable your window procedure to receive double-click messages, you must include the CS_DBLCLKS flag in the window class style when registering the window class.

This flag instructs the system to send WM_LBUTTONDBLCLK messages to your window procedure instead of generating separate WM_LBUTTONDOWN messages for each click.

Default Double-Click Behavior

If you include CS_DBLCLKS in the window class style, the window procedure receives the following messages for a double-click:

WM_LBUTTONDOWN: This message indicates the first click of the double-click.

WM_LBUTTONUP: This message indicates the release of the mouse button after the first click.

WM_LBUTTONDBLCLK: This message replaces the second WM_LBUTTONDOWN message and signals that a double-click has occurred.

WM_LBUTTONUP: This message indicates the release of the mouse button after the double-click.

Processing Double-Clicks

When implementing double-click logic, it's often advantageous for the first click to perform the same action as a single click.

This allows users to perform the single-click action without worrying about accidentally triggering a double-click. The second click (the WM_LBUTTONDBLCLK message) can then perform an additional action.

For instance, consider how double-clicks are handled in Windows Explorer. A single click selects a file, highlighting it with a reverse-video bar.

A double-click performs two actions: it selects the file like a single click, and it also directs Explorer to open the file. This design is straightforward and user-friendly.

Complex Double-Click Logic

Handling double-clicks becomes more complex if the first click does not perform the same action as a single click.

In such cases, the window procedure needs to track click events and distinguish between single clicks and double-clicks based on the time interval between clicks.

This can involve using the GetMessageTime function to obtain the relative times of WM_LBUTTONDOWN messages.

Understanding Nonclient-Area Mouse Messages

Windows applications receive mouse messages when the user interacts with the mouse within the window's client area.

However, if the mouse interaction occurs within the window's nonclient area, which includes the title bar, menu, and window scroll bars, Windows sends a different set of messages called nonclient-area mouse messages.

Purpose of Nonclient-Area Mouse Messages

Nonclient-area mouse messages are primarily used for system functions, such as resizing windows, minimizing or maximizing windows, and dragging windows around the screen.

Typically, you don't need to directly process these messages in your application's window procedure. Instead, you can pass them to DefWindowProc to allow Windows to handle the default system behavior.

Similarity to System Keyboard Messages

Nonclient-area mouse messages share similarities with the system keyboard messages WM_SYSKEYDOWN, WM_SYSKEYUP, and WM_SYSCHAR. These messages are also handled by DefWindowProc to perform system-level actions in response to keyboard events.

Nonclient-Area Mouse Messages vs. Client-Area Mouse Messages

The nonclient-area mouse messages closely parallel the client-area mouse messages. However, they are distinguished by the prefix "NC" in their message identifiers.

For instance, when the mouse moves within a nonclient area, the window procedure receives WM_NCMOUSEMOVE, which corresponds to WM_MOUSEMOVE for client-area mouse movements.

Nonclient-Area Mouse Messages for Mouse Buttons

The nonclient-area mouse messages for mouse button presses and releases follow a similar pattern:

Button	Pressed	Released	Double-Clicked
Left	WM_NCLBUTTONDOWN	WM_NCLBUTTONUP	WM_NCLBUTTONDBLCLK
Middle	WM_NCMBUTTONDOWN	WM_NCMBUTTONUP	WM_NCMBUTTONDBLCLK
Right	WM_NCRBUTTONDOWN	WM_NCRBUTTONUP	WM_NCRBUTTONDBLCLK

wParam and lParam Parameters for Nonclient-Area Mouse Messages

The wParam and lParam parameters for nonclient-area mouse messages differ slightly from those for client-area mouse messages.

The wParam parameter indicates the specific nonclient area where the mouse interaction occurred. It is set to one of the identifiers defined in the WINUSER.H header file, starting with the prefix "HT" (for "hit-test").

The lParam parameter contains the screen coordinates of the mouse position, with the:

x-coordinate in the low word and the y-coordinate in the high word.

These coordinates are based on the entire screen, not just the window's client area.

Converting Screen Coordinates to Client-Area Coordinates

To convert screen coordinates to client-area coordinates and vice versa, you can use the Windows functions ScreenToClient and ClientToScreen. These functions take a POINT structure as input and modify its coordinates accordingly.

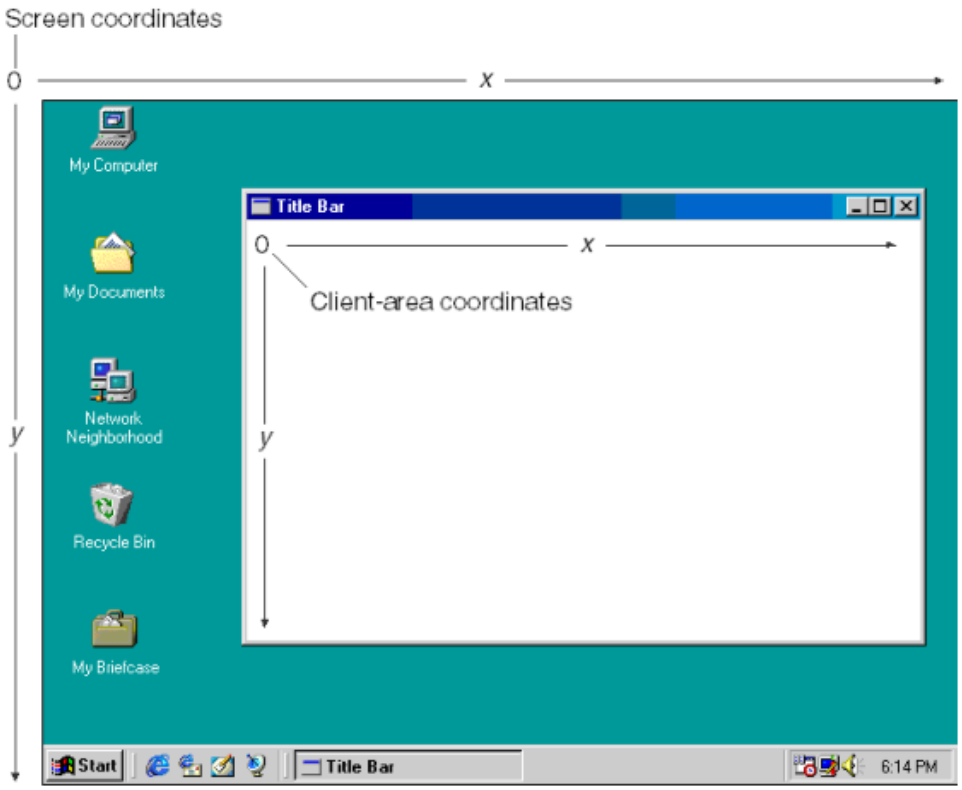


Figure 7–3. *Screen coordinates and client–area coordinates.*

Summary

Nonclient-area mouse messages are used for system-level interactions with a window's nonclient area. They are typically handled by DefWindowProc to maintain the default system behavior. Understanding these messages and their parameters is essential for creating applications that interact seamlessly with the Windows environment.

UNDERSTANDING THE WM_NCHITTEST MESSAGE

The WM_NCHITTEST message, also known as the "nonclient hit test" message, is a crucial part of the mouse input handling mechanism in Windows applications.

It is the final mouse message out of the 21 defined messages and plays a pivotal role in determining how mouse interactions are interpreted and handled.

Purpose of WM_NCHITTEST

The primary purpose of WM_NCHITTEST is to determine the specific part of the window that the mouse cursor is currently over. This information is crucial for generating appropriate mouse messages and enabling system-level behaviors associated with different window regions.

When WM_NCHITTEST is Sent

The WM_NCHITTEST message is sent to a window's window procedure before any other client-area or nonclient-area mouse messages are generated. This ensures that the window procedure has the opportunity to interpret the mouse position and provide an accurate hit test result.

Parameters of WM_NCHITTEST

The WM_NCHITTEST message has two parameters:

wParam: This parameter is not used and is ignored by the window procedure.

lParam: This parameter contains the screen coordinates of the mouse cursor position. The x-coordinate is stored in the low word, and the y-coordinate is stored in the high word.

Handling WM_NCHITTEST

Windows applications typically pass the WM_NCHITTEST message to DefWindowProc, the default window procedure provided by the system. DefWindowProc uses this message to determine the appropriate hit test value based on the mouse position and the window's layout.

Hit Test Values

DefWindowProc can return one of several hit test values, each indicating a specific part of the window or its surrounding area:

HTCLIENT: This value indicates that the mouse cursor is over the client area of the window.

HTNOWHERE: This value indicates that the mouse cursor is not over any window.

HTTRANSPARENT: This value indicates that the mouse cursor is over a transparent area of the window, meaning that another window is visible beneath it.

HTERROR: This value is used to generate an error sound.

Other HT values: There are additional HT values corresponding to specific nonclient areas of the window, such as the title bar, menu bar, and sizing borders.

Disabling Mouse Interaction with HTNOWHERE

Similar to how you can disable system keyboard interactions by trapping WM_SYSKEYDOWN, you can disable all mouse interactions by trapping WM_NCHITTEST and returning HTNOWHERE. This effectively prevents the generation of any client-area or nonclient-area mouse messages, making the mouse unresponsive within the window.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        case WM_NCHITTEST:
            return HTNOWHERE;

        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
}
```

This code will prevent any mouse clicks or movements from being registered within the window. If you want to allow mouse interactions in certain parts of the window, such as the client area, you can modify the code to return different hit test values depending on the mouse position.

MESSAGE CHAINING: A SERIES OF EVENTS

Windows applications often involve a chain of messages triggered by a single user action. This interconnectedness allows the system to handle complex interactions and respond appropriately to various user inputs.

The WM_NCHITTEST message serves as a common starting point for this chain of events, leading to the generation of subsequent messages based on the hit test result.

Example: Double-Clicking the System Menu Icon

Consider the scenario of double-clicking the system menu icon in a Windows program. This action initiates a sequence of messages:

WM_NCHITTEST Messages: The double-click generates multiple WM_NCHITTEST messages, indicating the mouse position over the system menu icon.

WM_NCLBUTTONDBLCLK Message: DefWindowProc, upon receiving the WM_NCHITTEST messages, determines the mouse position over the system menu icon and returns HTSYSMENU. This results in a WM_NCLBUTTONDBLCLK message being placed in the message queue with wParam equal to HTSYSMENU.

WM_SYSCOMMAND Message: The window procedure typically passes the WM_NCLBUTTONDBLCLK message to DefWindowProc. In turn, DefWindowProc interprets the message and generates a WM_SYSCOMMAND message with wParam equal to SC_CLOSE, indicating a request to close the window.

WM_CLOSE Message: The window procedure usually passes the WM_SYSCOMMAND message to DefWindowProc. DefWindowProc handles the message by sending a WM_CLOSE message to the window procedure.

WM_DESTROY Message: If the program does not require user confirmation before closing, the window procedure processes the WM_CLOSE message by calling DestroyWindow. Among other actions, DestroyWindow sends a WM_DESTROY message to the window procedure.

WM_QUIT Message: Typically, the window procedure handles WM_DESTROY by sending a PostQuitMessage(0) instruction. This causes Windows to place a WM_QUIT message in the message queue. The WM_QUIT message signals the end of the message loop and leads to the program's termination.

Code Example: Trapping WM_CLOSE for Confirmation

If the program requires confirmation before closing, the window procedure can trap WM_CLOSE and handle it accordingly. For instance, you could display a dialog box to confirm the user's intention to close the program.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        case WM_CLOSE:
            if (MessageBox(hwnd, L"Are you sure you want to close?", L"Confirmation", MB_YESNO | MB_ICONQUESTION) == IDYES)
            {
                DestroyWindow(hwnd);
            }
            else
            {
                return 0;
            }

        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
}
```

In this example, the window procedure captures the WM_CLOSE message and displays a confirmation dialog box. If the user confirms the closure, the program proceeds to close by destroying the window. Otherwise, the WM_CLOSE message is not processed, and the window remains open.

UNDERSTANDING HIT-TESTING

Hit-testing is a fundamental aspect of user interaction in Windows applications. It involves determining the specific item or element that the user is interacting with, typically based on the mouse cursor position.

This information is crucial for handling user actions appropriately and providing a responsive user interface.

Hit-Testing in List View Controls

List view controls are commonly used in Windows applications to display a list of items, often with multiple columns. They handle hit-testing internally, making it easier for developers to manage their list data and respond to user interactions.

Hit-Testing Without List View Controls

Sometimes, developers may need to implement their own hit-testing logic, especially when dealing with custom controls or user interfaces. In these cases, the process involves performing calculations based on the mouse cursor coordinates and the layout of the displayed elements.

Hypothetical Example: Hit-Testing Files in Columns

Consider a scenario where you need to display a list of files in multiple columns without using a list view control. You would need to perform your own hit-testing to determine which file the user is interacting with.

Assumptions:

- File names are stored in an array of pointers to character strings named szFileNames.
- The file list starts at the top of the client area, which is cxClient pixels wide and cyClient pixels high.
- Columns are cxColWidth pixels wide; characters are cyChar pixels high.
- The number of files that can fit in each column is iNumInCol = cyClient / cyChar.

Calculating File Index from Mouse Coordinates

- Obtain the mouse coordinates cxMouse and cyMouse from the lParam parameter of the mouse message.
- Calculate the column index (iColumn) based on mouse position and column width: iColumn = cxMouse / cxColWidth.
- Calculate the position of the filename relative to the top of the column: iFromTop = cyMouse / cyChar.
- Calculate the index (iIndex) into the szFileNames array: iIndex = iColumn * iNumInCol + iFromTop.
- Check if iIndex exceeds the number of files in the array. If it does, the user is clicking on an empty area.

Handling Non-List View Scenarios

Hit-testing can become more complex when dealing with graphical images or variable font sizes. It requires understanding the layout and mapping mouse coordinates to the underlying data or objects.

Code Example:

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        case WM_LBUTTONDOWN:
        {
            // Get mouse coordinates
            int cxMouse = LOWORD(lParam);
            int cyMouse = HIWORD(lParam);

            // Calculate column index
            int iColumn = cxMouse / cxColWidth;

            // Calculate position relative to top of column
            int iFromTop = cyMouse / cyChar;

            // Calculate index into file array
            int iIndex = iColumn * iNumInCol + iFromTop;

            // Validate index
            if (iIndex < 0 || iIndex >= numFiles)
            {
                // Clicked on empty area
                return 0;
            }

            // Handle file selection or action based on iIndex
            // ...

            break;
        }

        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
}
```

The first hittest program…

Hittesting program
1-I was the one click

Sure, here are the main points of the WM_KEYDOWN logic in CHECKER2:

Determines cursor position: The logic uses GetCursorPos to get the current cursor position in screen coordinates and then converts it to client-area coordinates using ScreenToClient.

Constrains cursor position: The x and y coordinates are passed through the min and max macros to ensure they are within the range of 0 to 4. This is necessary because the cursor might not be within the client area when a key is pressed.

Handles arrow keys: For arrow keys, the logic increments or decrements x and y appropriately, allowing the user to navigate between rectangles using the keyboard.

Simulates mouse clicks with Enter or Spacebar: If the Enter key or the Spacebar is pressed, the logic uses SendMessage to send a WM_LBUTTONDOWN message to itself, simulating a mouse click on the currently focused rectangle.

Centers cursor on rectangle: The logic calculates the client-area coordinates that point to the center of the rectangle, converts them to screen coordinates using ClientToScreen, and sets the cursor position using SetCursorPos. This ensures that the cursor is always positioned at the center of the selected rectangle.

CHECKER 2 and 3 PROGRAMS

Leveraging Child Windows for Hit-Testing

In certain applications, such as the Windows Paint program, the client area is divided into distinct logical regions. Paint, for instance, has a dedicated area for its icon-based tool menu on the left and another for the color menu at the bottom.

When handling mouse clicks in these areas, Paint must consider their placement within the overall client area before identifying the specific item selected by the user.

Child Windows: Simplifying Drawing and Hit-Testing

Paint simplifies both the drawing and hit-testing of these smaller areas by utilizing child windows. These child windows effectively partition the entire client area into smaller rectangular regions.

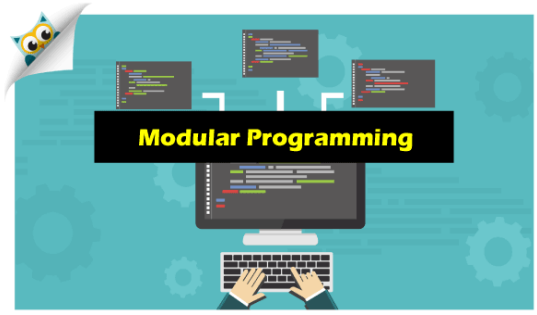
Each child window possesses its own window handle, window procedure, and client area. Mouse messages directed at each child window are only processed by the corresponding child window procedure.

The lParam parameter within the mouse message contains coordinates relative to the upper-left corner of the child window's client area, not the client area of the parent window (which is Paint's main application window).

Advantages of Child Windows

Child windows offer several advantages for structuring and modularizing applications:

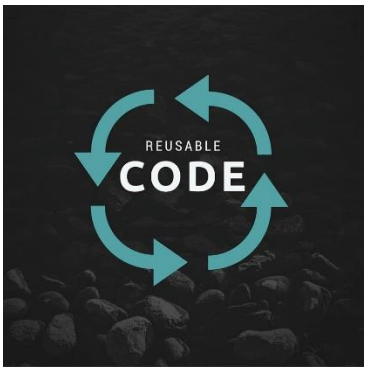
Modularization: Each child window can have its own window procedure if different window classes are used. This promotes modularity and code organization.



Customizable Appearance: Different window classes can define distinct background colors and default cursors, allowing for customization of each child window's appearance.

Simplified Hit-Testing: Child windows simplify hit-testing by isolating mouse interactions to their respective client areas, eliminating the need for complex calculations based on the entire client area.

Reusability: Child windows can be reused across multiple applications, promoting code reuse and efficiency.



Child Windows in CHECKER3

CHECKER3, shown in Figure 7-7, It utilizes 25 child windows to manage mouse clicks within specific areas of the client area.

It lacks a keyboard interface, which can be added in CHECKER4 as discussed later in the chapter.

Key Points

- Child windows provide a structured and modular approach to handling mouse interactions within specific regions of the client area.
- Child windows simplify hit-testing by confining mouse events to their respective client areas.
- Child windows promote code reuse and customization of individual areas within the client area.

CHECKER3 is an enhanced version of the CHECKER2 program, introducing the concept of child windows for handling user interactions.

Window Procedures: WndProc and ChildWndProc

CHECKER3 employs two window procedures: WndProc and ChildWndProc. WndProc remains the window procedure for the main (parent) window, responsible for overall window management. ChildWndProc, on the other hand, serves as the window procedure for the 25 child windows, handling mouse interactions within each individual child window.

Defining Window Classes

Since window procedures are associated with specific window class structures, CHECKER3 requires two window classes:

- Main Window Class ("Checker3"): This class defines the main window's behavior and appearance.
- Child Window Class ("Checker3_Child"): This class defines the child windows' behavior and appearance.

Window Class Registration

The WinMain function handles the registration of both window classes. After registering the main window class, it reuses most of the fields for the child window class. However, it modifies four specific fields:

- lpfnWndProc: Set to ChildWndProc, the child window procedure.
- cbWndExtra: Set to sizeof(long), indicating 4 bytes of extra storage per child window.
- hIcon: Set to NULL, as child windows don't require icons.
- lpszClassName: Set to "Checker3_Child", the child window class name.

Message Handling in ChildWndProc

ChildWndProc handles messages specific to the child windows:

- WM_CREATE: Initializes the child window's internal state flag to 0.
- WM_LBUTTONDOWN: Toggles the child window's internal state flag and invalidates the window to trigger a repaint.
- WM_PAINT: Handles painting the child window's background and the diagonal line if the state flag is set.

Argument	Main Window	Child Window
Window class	"Checker3"	"Checker3_Child"
Window caption	"Checker3"	NULL
Window style	WS_OVERLAPPEDWINDOW	WS_CHILDWINDOW
Horizontal position	CW_USEDEFAULT	0
Vertical position	CW_USEDEFAULT	0
Width	CW_USEDEFAULT	0
Height	CW_USEDEFAULT	0
Parent window handle	NULL	hwnd
Menu handle/child ID	NULL	(HMENU) (y << 8
Instance handle	hInstance	(HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE)
Extra parameters	NULL	NULL

Child Window Positioning and Sizing

The position and size parameters are not specified immediately when creating the child windows. Instead, they are set later in WndProc when the WM_SIZE message is received. This allows for dynamic positioning and sizing of the child windows based on the available client area.

Child Window Identification

The child windows are identified using a unique child ID, which is a composite of the x and y positions of the child window within the 5-by-5 array. This child ID is used to differentiate between the child windows when handling messages.

Child Window State Tracking

ChildWndProc maintains the current state (X or no X) of each child window using the extra space reserved in the window structure. This state is toggled when the child window is clicked and used to draw the diagonal line during WM_PAINT processing.

CHECKER4: IMPLEMENTING KEYBOARD NAVIGATION

CHECKER4 builds upon CHECKER3 by introducing keyboard navigation for toggling check marks using the Spacebar or Enter key and moving the input focus among child windows using the cursor keys.

Checker 4
Program-Navigating

Global Variable: idFocus

CHECKER4 introduces a global variable, idFocus, to track the child window ID of the window that currently has the input focus. This variable is used to identify the target window for keyboard interactions.

WM_SETFOCUS and WM_KILLFOCUS Handling

In CHECKER4, the parent window handles the WM_SETFOCUS message by setting the input focus to the child window identified by idFocus. Conversely, it invalidates the focused child window when receiving the WM_KILLFOCUS message.

Keyboard Navigation in ChildWndProc

ChildWndProc processes keyboard events:

WM_KEYDOWN: If the pressed key is not the Spacebar or Enter key, it forwards the message to the parent window for further processing.

Spacebar or Enter Press: It toggles the check mark of the focused child window and invalidates the window to trigger a repaint.

WM_LBUTTONDOWN Handling

ChildWndProc handles mouse clicks:

WM_LBUTTONDOWN: It toggles the check mark of the clicked child window, sets the input focus to that window, and invalidates it for a repaint.

Additional Notes

- ✓ CHECKER4 utilizes the GetDlgItem and GetDlgCtrlID functions to access child windows based on their ID.
- ✓ The WM_KEYDOWN handling in ChildWndProc differentiates between keys used for toggling check marks and those for moving the input focus.
- ✓ The WM_LBUTTONDOWN handling in ChildWndProc updates the check mark and sets the input focus to the clicked child window.
- ✓ CHECKER4 introduces keyboard navigation for toggling check marks and moving the input focus. The global variable idFocus tracks the child window ID of the focused window. Parent and child windows collaborate to handle keyboard and mouse interactions.

ChildWndProc Processing

WM_SETFOCUS:

- ➢ Saves the child window ID of the window receiving the input focus in the global variable idFocus.
- ➢ Invalidates the window to trigger a repaint.

WM_KILLFOCUS:

- ➢ Invalidates the window to trigger a repaint.

WM_PAINT:

- ➢ If the window has the input focus, draws a rectangle with a PS_DASH pen style to indicate the focus.

WM_KEYDOWN:

- ➢ For Spacebar or Enter key presses, toggles the check mark and invalidates the window.
- ➢ For other keys, sends the message to the parent window.

Parent Window Processing

Cursor Movement Key Handling:

- ➢ Obtains the x and y coordinates of the child window with the input focus.
- ➢ Changes the coordinates based on the pressed cursor key.
- ➢ Sets the input focus to the new child window using SetFocus.

BLOKOUT1: CAPTURING THE MOUSE

BLOKOUT1 demonstrates the concept of capturing the mouse to draw a rectangle. It utilizes the WM_LBUTTONDOWN, WM_MOUSEMOVE, and WM_LBUTTONUP messages to track the mouse movements and draw the rectangle accordingly.

BlockOut 1.mp4

WM_LBUTTONDOWN:

- Records the initial position of the rectangle (ptBeg).
- Sets the cursor to IDC_CROSS.
- Sets the fBlocking flag to indicate that the mouse is captured.

WM_MOUSEMOVE:

- If the mouse is captured, updates the end position of the rectangle (ptEnd).
- Draws the outline of the rectangle using DrawBoxOutline.

WM_LBUTTONUP:

- Draws the outline of the rectangle using DrawBoxOutline.
- Records the final position of the rectangle (ptBoxEnd).
- Sets the cursor to IDC_ARROW.
- Sets the fBlocking flag to false.
- Invalidates the window to trigger a repaint.

WM_PAINT:

- Fills the rectangle using SelectObject and Rectangle.
- If the mouse is captured, draws the outline of the rectangle using SetROP2 and DrawBoxOutline.

WM_CHAR (Escape key):

- Cancels the rectangle drawing.
- This program demonstrates the use of mouse capture to interactively draw shapes on a window.

Mouse Capturing: Maintaining Control

BLOKOUT1 demonstrates a common issue with mouse interaction: when the cursor moves outside the window, the program stops receiving mouse events. Mouse capturing addresses this issue by allowing a window to maintain control over mouse events even when the cursor is outside the window.

Initiating Mouse Capture

To capture the mouse, a program calls the SetCapture(hwnd) function, where hwnd is the handle of the window that should receive mouse events. This function ensures that all subsequent mouse messages are sent to the specified window, regardless of the cursor's position.

Mouse Capture Behavior

With mouse capture enabled, mouse messages are always treated as client-area messages, even if the cursor is in a nonclient area. The coordinates provided in the lParam parameter represent the position of the mouse relative to the client area, and negative values indicate that the cursor is outside the client area.

Releasing Mouse Capture

To release mouse capture and restore normal mouse behavior, a program calls the ReleaseCapture() function. This function allows other windows to receive mouse events once again.

Mouse Capture in 32-bit Windows

In 32-bit versions of Windows, mouse capturing is more restrictive. If the mouse is captured, a button is not pressed, and the cursor moves over another window, the window beneath the cursor will receive mouse events instead of the capturing window. This safeguard prevents a single program from disrupting the system by indefinitely capturing the mouse.

Recommended Practice

To avoid issues, a program should only capture the mouse when a button is pressed within the client area and release the capture when the button is released. This ensures that mouse events are handled appropriately and prevents conflicts with other windows.

Main Differences

Mouse Capture: BLOKOUT2 introduces mouse capture using the SetCapture() function in the WM_LBUTTONDOWN message. This ensures that the program receives mouse events even when the cursor moves outside the window.

Mouse Capture Release: BLOKOUT2 adds calls to ReleaseCapture() in the WM_LBUTTONDOWN and WM_CHAR messages to properly release mouse capture when the button is released or the Escape key is pressed.

BLOKOUT2 vs. BLOKOUT1: BLOKOUT2 is identical to BLOKOUT1 except for the addition of mouse capture handling.

Mouse Capture Benefits: Mouse capture allows the program to track mouse movements even when the cursor is outside the window, providing a smoother and more consistent user experience.

Appropriate Mouse Capture Usage: Mouse capture should be used when tracking WM_MOUSEMOVE messages after a mouse button press within the client area until the button is released.

Simplified Program Logic: Using mouse capture simplifies the program logic and ensures that the user's expectations are met.

Additional Notes

- Enlarging the window after capturing the mouse reveals the entire rectangle drawn.
- Mouse capture is not limited to oddball applications; it should be used whenever consistent mouse tracking is required after a button press.
- Proper mouse capture handling ensures a simpler program and met user expectations.

BlockOut 2.mp4

MOUSE WHEEL

The SYSMETS program implements mouse wheel support to allow users to scroll through the system metrics list using the mouse wheel.

This is achieved by handling the WM_MOUSEWHEEL message.

WM_MOUSEWHEEL Message Handling

When the WM_MOUSEWHEEL message is received, the program retrieves the scroll delta from the HIWORD parameter of the wParam parameter. The scroll delta indicates the amount of scrolling in units of WHEEL_DELTA, which is typically 120.

Scrolling Logic

The program accumulates scroll deltas using the iAccumDelta variable. When the accumulated delta reaches or exceeds the iDeltaPerLine value, the program sends a WM_VSCROLL message to the window to scroll the system metrics list vertically.

iDeltaPerLine Calculation

The iDeltaPerLine variable determines how much scrolling occurs for each mouse wheel rotation. It is calculated using the ulScrollLines value obtained from the SystemParametersInfo function. If ulScrollLines is zero, no scrolling occurs.

Smooth Scrolling

The program accumulates scroll deltas to provide smoother scrolling. This prevents the system metrics list from jumping excessively when the mouse wheel is quickly rotated.

Mouse Wheel Usability

Mouse wheel support enhances the usability of the SYSMETS program by allowing users to quickly scroll through the system metrics list without using the scroll bars. This is particularly useful when working with a large number of system metrics.

Mouse Wheel Events

The SYSMETS program responds to mouse wheel events by handling the WM_MOUSEWHEEL message. This message contains information about the scroll delta, which indicates the amount of scrolling that should occur.

Scroll Delta Interpretation

The scroll delta is typically either 120 or -120, representing three lines of scrolling up or down, respectively. Future mouse wheels may generate finer scroll deltas, such as 40 or -40, indicating scrolling one line up or down.

Generalized Scrolling

To accommodate varying scroll delta values, SYSMETS retrieves the SystemParametersInfo value for SPI_GETWHEELSCROLLLINES. This value indicates the number of lines to scroll for a delta value of WHEEL_DELTA, which is typically 120.

Delta Accumulation

SYSMETS accumulates scroll deltas in the iAccumDelta variable. When this variable reaches or exceeds the iDeltaPerLine value (calculated using SPI_GETWHEELSCROLLLINES), SYSMETS generates WM_VSCROLL messages to scroll the window.

Smooth Scrolling Implementation

The accumulation of scroll deltas ensures smooth scrolling behavior, preventing abrupt jumps when the mouse wheel is quickly rotated.

Future Mouse Wheel Enhancements

The SYSMETS code is designed to handle future mouse wheels with finer scroll delta values. The program will adjust its scrolling behavior based on the delta values generated by the specific mouse wheel in use.

Custom Mouse Cursors

The creation of customized mouse cursors is covered in Chapter 10 along with other Windows resources. The video…

Sysmets with mousewheel.mp4