CHAPTER 16: THE PALETTE MANAGER - UNVEILING THE WORLD OF 256 COLORS

This chapter delves into the realm of the Windows Palette Manager, a tool born out of necessity due to certain hardware limitations.

A paletter lookup table on video boards works like:



While modern video adapters often support higher color depths like 24-bit or 16-bit, certain setups, particularly on laptops or in high-resolution modes, are constrained to 8 bits per pixel. This limitation translates to a palette of only 256 simultaneous colors.

The question arises: What can be accomplished with a palette of 256 colors? While 16 colors are insufficient for displaying realistic images, and thousands or millions of colors are more than ample for such tasks, the middle ground of 256 colors presents unique challenges.

To effectively showcase real-world images with this limited palette, colors must be carefully selected for each specific image. A one-size-fits-all "standard" set of 256 colors isn't feasible, as it won't cater to the diverse needs of every application.

Enter the Windows Palette Manager. This tool is designed for precisely specifying the colors required by a program when operating in an 8-bit video mode.

If your programs exclusively run in higher bit depths, you may not encounter the need for the Palette Manager. Nevertheless, this chapter holds valuable insights, particularly in tying up loose ends related to bitmap handling.

Key Points:

Hardware Limitations: Certain video adapters restrict color depth to 8 bits per pixel, allowing only 256 colors simultaneously.

Palette Manager's Purpose: Tailored for programs operating in 8-bit video modes, the Palette Manager enables the specification of essential colors.

Color Selection Challenge: Unlike higher color depths, where a standard set suffices, 256 colors require careful curation for each application's unique needs.

Understanding the Palette Manager is crucial for developers navigating the constraints of 8-bit video modes.

While it may not be applicable in all scenarios, its insights into color management are invaluable, especially when working with real-world images in resource-limited environments.

The 20 reserved colors in 256-color video modes:



The 20 reserved colors in 256-color video modes are part of the Windows Palette Manager, and they serve as standard colors that are predefined for system use. These colors are reserved to maintain consistency across applications running in an 8-bit video mode. Here is a description of each of the 20 reserved colors:

1. Black (Pixel Bits: 00000000, RGB Value: 00 00 00)
2. Dark Red (Pixel Bits: 00000001, RGB Value: 80 00 00)
3. Dark Green (Pixel Bits: 00000010, RGB Value: 00 80 00)
4. Dark Yellow (Pixel Bits: 00000011, RGB Value: 80 80 00)
5. Dark Blue (Pixel Bits: 00000100, RGB Value: 00 00 80)
6. Dark Magenta (Pixel Bits: 00000101, RGB Value: 80 00 80)
7. Dark Cyan (Pixel Bits: 00000110, RGB Value: 00 80 80)
8. Light Gray (Pixel Bits: 00000111, RGB Value: C0 C0 C0)
9. White (Pixel Bits: 11111111, RGB Value: FF FF FF)
10. Cyan (Pixel Bits: 11111110, RGB Value: 00 FF FF)
11. Magenta (Pixel Bits: 11111101, RGB Value: FF 00 FF)
12. Blue (Pixel Bits: 11111100, RGB Value: 00 00 FF)
13. Dark Gray (Pixel Bits: 11111000, RGB Value: 80 80 80)
14. Medium Gray (Pixel Bits: 11110111, RGB Value: A0 A0 A4)
15. Cream (Pixel Bits: 11110110, RGB Value: FF FB F0)
16. Sky Blue (Pixel Bits: 11110101, RGB Value: A6 CA F0)
17. Money Green (Pixel Bits: 11110100, RGB Value: C0 DC C0)
18. Reserved
19. Reserved
20. Reserved

In 256-color display modes, Windows manages a system palette that mirrors the video card's hardware color lookup table. This system palette controls the available colors for display.

By default, Windows provides a specific set of colors as the system palette, which is depicted in Figure above.

It holds 256 colors, 20 of which are fixed for system elements, while applications can customize the remaining 236.

Applications can adjust these colors using logical palettes. If multiple applications use logical palettes, Windows prioritizes the active window (the one in the foreground with a highlighted title bar), ensuring its color choices take precedence.

In scenarios where multiple applications are using logical palettes simultaneously, Windows gives the highest priority to the active window.

The active window refers to the window that currently has the highlighted title bar and appears in the foreground of all other windows.

This ensures that the active application's color choices take precedence over other applications, providing a consistent and coherent visual experience.

To explore this concept practically, we'll examine a sample program later in this chapter. To align with these examples, consider switching your display to 256-color mode.

Access display settings by right-clicking on your desktop, selecting "Properties," and choosing the "Settings" tab.

Key points:

* System palette: Master color table for Windows in 256-color modes.
* Logical palettes: Application-specific color customizations.
* Active window priority: Windows prioritizes the active application's colors.
* Switching to 256-color mode: Recommended for compatibility with examples.

Additional notes:

* While modern systems often use higher color depths, understanding system palettes is still relevant for legacy applications and specific development scenarios.
* The specific process for adjusting display settings may vary slightly depending on your Windows version.

GRAYS1 PROGRAM



The GRAYS1 program is designed to display 65 shades of gray as a "fountain" of color, ranging from black to white. It does not use the Windows Palette Manager but instead directly creates and fills rectangles with varying shades of gray.

The program starts by defining the necessary headers and function prototypes. It then defines the WinMain function, which is the entry point of the program. Inside WinMain, the program registers a window class, creates a window, and enters the message loop.

The WndProc function is the window procedure for handling messages related to the program's window. It handles messages such as WM\_SIZE, WM\_PAINT, and WM\_DESTROY.

In the WM\_SIZE message case, the cxClient and cyClient variables are updated with the width and height of the client area of the window, respectively.

In the WM\_PAINT message case, the program prepares to paint the window. It begins by obtaining a device context (hdc) and a paint structure (ps) using the BeginPaint function. Then, a loop is executed 65 times to create and fill rectangles with varying shades of gray.

Inside the loop, the rect structure is defined to represent the dimensions of each rectangle. The left and right coordinates of the rectangle are calculated based on the current iteration and the total number of shades. The top and bottom coordinates are set to cover the entire height of the client area.

A brush (hBrush) is created using the CreateSolidBrush function, specifying the RGB values for the gray color. The RGB values are derived from the current iteration to create a gradient effect.

The FillRect function is then used to fill the current rectangle with the gray color represented by the brush. After filling the rectangle, the brush is deleted to release the associated resources.

Finally, the program calls EndPaint to signal the end of the painting process and returns 0 to indicate that the message has been handled.

In the WM\_DESTROY message case, the program posts a quit message to exit the message loop and terminate the program.

If any other messages are received or not handled in the WndProc function, the program calls DefWindowProc to perform the default window procedure for those messages.



*Let's dive deeper into the GRAYS1 program and explore the section where color palettes are mentioned.*

In the given code, the GRAYS1 program does not use the Windows Palette Manager. Instead, it directly creates and fills rectangles with varying shades of gray using the RGB color model.

The concept of color palettes in computer graphics refers to a limited set of colors that are available for use in a particular system or application. In the Windows operating system, a palette is a data structure that holds a fixed number of colors, typically 256 colors. The Windows Palette Manager is responsible for managing and mapping colors from the system palette to the colors used by an application.

However, in the GRAYS1 program, the focus is on displaying shades of gray rather than utilizing a predefined color palette. The program achieves this by dynamically calculating and creating shades of gray using the CreateSolidBrush function and RGB values.

Within the WM\_PAINT message case, a loop is executed 65 times to create 65 rectangles, each representing a different shade of gray. The RGB values for each shade are calculated as min (255, 4 \* i), where i is the current iteration of the loop.

By multiplying i by 4 and clamping the result to a maximum of 255, the program ensures that the RGB values stay within the valid range for a grayscale color. This calculation creates a gradient effect, where the shades of gray become progressively lighter as i increases.

The CreateSolidBrush function is then used to create a brush with the calculated RGB values, representing the current shade of gray. The FillRect function fills the current rectangle with the gray color represented by the brush.

It's important to note that by creating and using brushes directly, the GRAYS1 program bypasses the Windows Palette Manager and the limitations of a fixed color palette. Instead, it dynamically generates and displays the shades of gray as a "fountain" of color in the client area of the window.

This approach allows for greater flexibility in displaying a wider range of shades and gradients, as it leverages the full RGB color space rather than being constrained by a predefined palette.

Here's a clearer and more concise explanation of the program's behavior in 256 color mode:

During the WM\_PAINT message, the program paints 65 rectangles using different gray shades, ranging from black to white.

Dithering: To achieve these shades, Windows employs a technique called "dithering." It blends combinations of the four pure colors available in the system palette (black, dark gray, light gray, and white) to simulate additional shades, resulting in a grainy pattern.

Lines and text: In contrast, lines and text in Windows are typically drawn using only the pure colors, without dithering.

Bitmaps: When displaying bitmaps in 256-color mode, Windows approximates them using the 20 standard system colors, often leading to color inaccuracies. Dithering is not typically applied to bitmaps.

Palette Manager functions and messages: The GRAYS2 program, featured in Figure 16-3, demonstrates key concepts of color management in Windows through the use of Palette Manager functions and messages.

Key takeaways:

Limited color palette: In 256-color mode, Windows has a restricted set of colors.

Dithering for filled areas: Windows uses dithering to simulate additional colors within filled areas.

Pure colors for lines and text: Lines and text are drawn using only the pure colors available in the system palette.

Color approximations for bitmaps: Bitmaps are approximated using the standard system colors, often resulting in inaccuracies.

Palette Manager for color control: The Palette Manager functions and messages provide a way for applications to manage color palettes in Windows.

The GRAYS2 program is a Windows application that displays a gradient of gray shades using the Palette Manager. Let's break down its functionality into paragraphs:

GRAYS2 PROGRAM





Application Structure:

The program is structured as a Windows application with a standard WinMain function and a window procedure (WndProc).

It defines a window class (WNDCLASS) with basic attributes such as the window procedure, background brush, icon, and cursor.

Palette Initialization (WM\_CREATE):

Upon window creation, the program sets up a LOGPALETTE structure to define a custom palette containing 65 shades of gray.

Each palette entry is assigned RGB values representing varying intensities of gray.

The CreatePalette function is then called to create a logical palette based on the provided information.

Handling Window Size (WM\_SIZE):

The program handles the WM\_SIZE message to update the client area dimensions whenever the window is resized. This information is crucial for drawing the gradient.

Painting the Gradient (WM\_PAINT):

In response to the WM\_PAINT message, the program begins painting by obtaining a device context (HDC) using BeginPaint.

It selects the custom palette into the device context and realizes the palette to make it effective for the current device.

The gradient is drawn by creating a series of rectangles, each filled with a solid brush of a specific gray shade. The shades vary from black to white.

Palette Interaction (WM\_QUERYNEWPALETTE, WM\_PALETTECHANGED):

The program handles palette-related messages to ensure proper interaction with the system and other applications.

WM\_QUERYNEWPALETTE is used to respond to a request for palette selection, indicating whether the application has a palette to select.

WM\_PALETTECHANGED is used to handle changes in the system palette, updating the application's palette if necessary.

Cleanup on Window Destruction (WM\_DESTROY):

Upon window destruction, the program deletes the logical palette using DeleteObject and posts a quit message to terminate the application.

The Palette Manager plays a crucial role in managing colors in Windows applications, particularly when dealing with limited color environments such as 8-bit video modes.

The GRAYS2 program exemplifies the use of the Palette Manager to create a logical palette and display a gradient of gray shades. Let's delve into the detailed explanation:



Creating a Logical Palette:

The initial step involves creating a logical palette using the CreatePalette function.

In the WM\_CREATE message handler, the program initializes a LOGPALETTE structure to define its custom palette. The structure includes a version, the number of entries, and an array of PALETTEENTRY structures.

Each PALETTEENTRY structure represents an RGB color value. The program sets up 65 shades of gray, calculating the appropriate intensity values and storing them in the palette.

Palette Initialization Details:

The LOGPALETTE structure is defined with compatibility information and an array of PALETTEENTRY structures.

Each PALETTEENTRY structure consists of red, green, and blue color components, each represented by a byte. The peFlags field is set to 0.

Memory is allocated for the LOGPALETTE structure and additional PALETTEENTRY structures to accommodate the desired number of shades.

Palette Selection and Realization:

During the WM\_PAINT message processing, the program selects the logical palette into the device context using SelectPalette. The logical palette is then "realized" in the device context using RealizePalette.

Selecting a palette into the device context is crucial for the system to map the colors to the actual physical palette of the video board.

The SelectPalette function returns the handle of the previous logical palette, allowing for restoration if needed.

Color Specification and Use of PALETTERGB:

The program continues to use the familiar RGB macro to specify colors in the logical palette. However, it introduces the concept of "Palette RGB" values using the PALETTERGB macro.

A "Palette RGB" color is similar to an RGB color but with the high byte of the COLORREF value set to 2.

Rules are provided for using colors in the logical palette, emphasizing the need to specify colors using Palette RGB values or Palette Index values when working with a selected palette.

Handling Palette Management Support:

Notably, the program does not explicitly check whether the video display driver supports palette management.

In scenarios where palette management is not supported (non-256 color video modes), the program functions similarly to a version (GRAYS1) that does not utilize the Palette Manager.

In essence, GRAYS2 demonstrates a meticulous process of creating, selecting, and realizing a logical palette to effectively manage and display a gradient of gray shades, showcasing the nuanced interaction with the Palette Manager in a Windows environment.

LOGICAL PALETTE DEFINITION:



In the provided code snippet, we're dealing with the implementation of the Windows Palette Manager in the context of the GRAYS2 program. Let's break down the code and discuss its functionality in the context of logical palettes, color representation, and palette management.

The LOGPALETTE structure is a fundamental part of Windows Palette Manager. It holds information about a logical palette, including the palette version, the number of entries, and an array of PALETTEENTRY structures. Each PALETTEENTRY structure represents an RGB color value and flags.

Palette Entry Structure:

The PALETTEENTRY structure defines individual entries within the logical palette. It consists of fields representing the red, green, and blue color components, along with additional flags.

Logical Palette Creation:

The GRAYS2 program dynamically allocates memory for the LOGPALETTE structure, including space for an array of PALETTEENTRY structures. It then initializes the logical palette by setting the version, the number of entries, and RGB color values for each entry in a loop. The logical palette is created using the CreatePalette function, and memory is freed after its creation.

WM\_PAINT Handling:

When handling the WM\_PAINT message, the program begins painting by obtaining the device context (hdc) using BeginPaint. It then selects and realizes the logical palette in the device context using SelectPalette and RealizePalette. This step is crucial for mapping the colors in the logical palette to the system palette, aligning them with the physical palette of the video board.

The program proceeds to draw a "fountain of grays" by iterating over the 65 shades defined in the logical palette. For each shade, it creates a solid brush using the CreateSolidBrush function and the PALETTERGB macro. This macro is used to specify colors from the logical palette, ensuring that the additional colors are utilized. The drawn rectangle is then filled with the brush, and the brush is appropriately deleted.

Finally, the painting process is concluded using EndPaint.

Explanation Summary:

In summary, the provided code demonstrates the creation and utilization of a logical palette in the GRAYS2 program. This logical palette is essential for managing and displaying a range of gray shades, and the program ensures proper integration with the Windows Palette Manager during the painting process.

PALETTE MANAGER PRIORITY AND COLOR MAPPING:

In a multi-program environment using the Palette Manager, the active window takes precedence over the palette. The system prioritizes the most recently active window and subsequent windows accordingly.

When a new program becomes active, the Windows Palette Manager may need to reorganize the system palette table.

If a program defines a color in its logical palette identical to one of the 20 reserved colors, Windows will map that logical palette entry to the corresponding reserved color.

Moreover, if multiple applications specify the same color in their logical palettes, they will share the system palette entry.

The program can influence this behavior by using the PC\_NOCOLLAPSE flag in the peFlags field of the PALETTEENTRY structure.

Palette Manager Messages:

Two crucial messages, WM\_QUERYNEWPALETTE and WM\_PALETTECHANGED, are integral to organizing the system palette.

WM\_QUERYNEWPALETTE:

Sent to a main window when it is about to become active, this message requires processing by programs utilizing the Palette Manager.

In the context of GRAYS2, the program handles this message by obtaining a device context handle, selecting the palette into it, calling RealizePalette, and then invalidating the window to trigger a WM\_PAINT message.

The window procedure returns TRUE if it realizes its logical palette and FALSE otherwise.

WM\_PALETTECHANGED:

When the system palette changes due to a WM\_QUERYNEWPALETTE message, Windows sends the WM\_PALETTECHANGED message to all main windows, starting with the most active window.

The wParam value passed to the window procedure is the handle of the active window.

A program should process this message only if wParam is not equal to the program's window handle.

Generally, programs using a customized palette call SelectPalette and RealizePalette while processing this message.

System Palette Organization:

Upon subsequent calls to RealizePalette during the WM\_PALETTECHANGED message, Windows checks for matches of RGB colors in the logical palette with those already loaded in the system palette.

If two programs require the same color, the same system palette entry is shared. If no unused system palette entries exist, the color in the logical palette is mapped to the closest color from the 20 reserved entries.

Handling WM\_PALETTECHANGED:

For programs concerned about the appearance of the client area when inactive, processing the WM\_PALETTECHANGED message is essential.

GRAYS2 demonstrates one approach by obtaining a device context, selecting the palette into it, and calling RealizePalette.

Instead of calling InvalidateRect, GRAYS2 opts for UpdateColors, a function typically more efficient than repainting the window.

It changes the values of pixels in the window to help preserve the previous colors.

GRAYS3 PROGRAM



In the GRAYS3 program, several changes and enhancements have been introduced compared to GRAYS2, primarily focusing on how colors are managed and displayed using the Palette Manager. Let's delve into the key modifications:



Palette Management Approach:

Instead of using the PALETTERGB macro as in GRAYS2, GRAYS3 employs the PALETTEINDEX macro during the WM\_PAINT message processing. This macro is used to specify colors based on the index within the logical palette.

Palette Entry Specification:

The program still creates a logical palette using the LOGPALETTE structure, similar to GRAYS2. The palette consists of 65 entries, each representing a different shade of gray. The entries are specified with varying levels of red, green, and blue components, providing a gradient effect.

Palette Realization in WM\_PAINT:

During the WM\_PAINT message processing, the program selects the logical palette into the device context and calls RealizePalette to map the colors to the system palette. However, in GRAYS3, the colors are drawn using the PALETTEINDEX macro, indicating an index within the logical palette, which simplifies color specification.

Drawing with PALETTEINDEX:

The fountain of grays is drawn in the client area of the window using a loop. For each iteration, a rectangle representing a segment of the window is filled with a solid brush created using the PALETTEINDEX macro. This approach directly references the index in the logical palette to determine the color.

Handling Palette Change Messages:

The program responds to the WM\_QUERYNEWPALETTE message, which is sent when the window is about to become active. In this case, it obtains the device context, selects and realizes the palette, and invalidates the window to trigger a repaint.

Additionally, the WM\_PALETTECHANGED message is processed, ensuring that the program updates its colors if another window's palette has changed. This involves selecting and realizing the palette in response to the system palette change.

Efficient UpdateColors Function:

GRAYS3 introduces the use of the UpdateColors function instead of directly repainting the window in response to a palette change.

This function efficiently updates the pixel values in the window to preserve the previous colors without the need for a full repaint.

Palette indices and Palette RGB colors represent different ways of specifying colors in Windows applications.

A Palette Index color consists of a high byte set to 1, and the low byte represents an index into the logical palette currently selected in the device context.

In the case of GRAYS3, the logical palette used has a total of 65 entries, with indices ranging from 0 to 64.

Each index corresponds to a specific color in the palette, allowing for a wide range of shades.

For example, PALETTEINDEX(0) represents black, PALETTEINDEX(32) represents medium gray, and PALETTEINDEX(64) represents white.

Using palette indices offers efficiency advantages over using RGB values because Windows does not need to perform a nearest-color search.

Instead of specifying the RGB values directly, the program can simply reference the desired color by its corresponding palette index.

This approach eliminates the need for Windows to search for the closest matching color in the palette, resulting in faster color retrieval and rendering.

By utilizing palette indices, Windows applications can leverage the predefined logical palette and directly access the desired colors without additional calculations.

This approach is particularly beneficial in scenarios where palette manipulation is supported and a limited color depth, such as 256 colors, is used.

In summary, palette indices provide a more efficient means of specifying colors in Windows applications compared to RGB values.

By referencing the desired color through the index, the need for a nearest-color search is eliminated, resulting in improved performance and streamlined color retrieval.

GRAYS3 refines the color management process by utilizing the PALETTEINDEX macro, simplifying color specification, and maintains responsiveness to palette changes with efficient updates, offering an enhanced visual experience compared to GRAYS2.

QUERY PALETTE SUPPORT

In Windows applications, the Palette Manager provides efficient color mapping and manipulation capabilities, particularly in video display modes with limited color depths like 256 colors.

To determine whether the device driver supports the Palette Manager, the program can use the GetDeviceCaps function with the RASTERCAPS parameter.

By performing a bitwise AND operation between the return value and the constant RC\_PALETTE, the program can check if the value is nonzero, indicating support for palette manipulation.

The GetDeviceCaps function also provides other important information related to the palette.

The call GetDeviceCaps (hdc, SIZEPALETTE) returns the total size of the palette table on the video board, typically 256 for 8 bits per pixel video display modes.

The function call GetDeviceCaps (hdc, NUMRESERVED) returns the number of colors in the palette table that the device driver reserves for system purposes, usually 20.

These reserved colors are the only pure colors that a Windows application can use without invoking the Palette Manager in a 256-color video mode.

The GetDeviceCaps (hdc, COLORRES) function call provides the resolution (in bits) of the RGB color values loaded into the hardware palette table, indicating the precision of the digital-to-analog converters (ADCs) used by the video display adapter. For example, a value of 18 indicates the usage of 6-bit ADCs, while a value of 24 indicates the usage of 8-bit ADCs.

By examining the color resolution value, a Windows program can make informed decisions about color usage.

For instance, if the color resolution is 18, requesting 128 shades of gray would be unnecessary because only 64 discrete shades of gray are possible.

Requesting more shades would lead to redundant entries in the hardware palette table.

In summary, by querying the Palette Manager support and using the GetDeviceCaps function, Windows applications can determine palette capabilities, such as support for palette manipulation, the size of the palette table, the reserved colors, and the color resolution of the hardware palette. This information assists in making optimal color choices and utilizing the available colors effectively in various video display modes.

*How does the Palette Manager handle situations where the device driver does not support palette manipulation? Can you provide an example of how the Palette Manager functions are used to work with the palette indices in GRAYS3? What are some other advantages of using palette indices instead of RGB values in Windows applications?*

When the device driver does not support palette manipulation, the Palette Manager functions gracefully adapt, and Windows continues to function without palette enhancements.

In such cases, the application essentially operates as if the Palette Manager were not utilized.

The advantages provided by the Palette Manager, such as accessing additional colors beyond the standard system palette, become relevant only in video modes that support palette management, typically those with 256 colors.

Let's delve into an example of how the Palette Manager functions are employed to work with palette indices in GRAYS3.

In this program, the logical palette is created with 65 entries during the WM\_CREATE message.

Later, during the WM\_PAINT message, the selected and realized palette is used to draw a series of rectangles representing different shades of gray.

Notably, the PALETTEINDEX macro is employed to specify the color of the solid brush when creating these rectangles.

Here's an excerpt of the relevant code from GRAYS3:



In this code, the PALETTEINDEX macro is employed to specify the color when creating a solid brush (CreateSolidBrush). The index 'i' is used to access different shades of gray from the logical palette. This method is more efficient than using RGB values, as Windows does not need to perform a nearest-color search, and it directly references the index in the logical palette.

The advantages of using palette indices over RGB values in Windows applications extend beyond efficiency. Some notable advantages include:

Reduced Memory Usage: Palette indices typically require less memory than RGB values, as they refer to predefined colors in a palette rather than storing full RGB color information.



Improved Performance: Since palette indices directly map to entries in the logical palette, color selection and rendering are faster compared to the computation involved in matching RGB values.



Compatibility: Palette indices facilitate compatibility with legacy systems and older display technologies that rely on palette-based color representation.



Consistent Color Mapping: Using palette indices ensures consistent color mapping, especially when dealing with limited color environments, leading to predictable and reliable color rendering.



In summary, the Palette Manager, when used with palette indices, offers efficiency, reduced memory usage, improved performance, and compatibility advantages in Windows applications, particularly in scenarios where palette manipulation is supported by the display hardware.

*How can a Windows program utilize the Palette Manager functions to access the remaining 236 colors in a 256-color video mode? What are some examples of decisions a Windows program can make based on the color resolution value? Can you provide more information on how the Palette Manager functions work in Windows applications?*

A Windows program can utilize the Palette Manager functions to access the remaining 236 colors in a 256-color video mode by creating and managing a logical palette. Here are the general steps involved:

Create a Logical Palette:

Allocate memory for a LOGPALETTE structure that includes information about the palette.

Set the palVersion field to 0x0300 for Windows 3.0 compatibility.

Set the palNumEntries field to the number of entries in the palette table, typically 256 for a 256-color video mode.

Populate the palPalEntry array with RGB values for each color in the palette.

Create the Palette:

Use the CreatePalette function to create a logical palette based on the information in the LOGPALETTE structure.

The function returns a handle to the logical palette.

Select and Realize the Palette:

During the WM\_PAINT message, use SelectPalette to select the logical palette into the device context.

Call RealizePalette to map the colors to the system palette, corresponding to the physical palette of the video board.

Use Palette Indices:

To access colors from the logical palette, use palette indices or the PALETTEINDEX macro instead of normal RGB values.

Palette indices range from 0 to 255 (or the total size of the palette table), allowing access to the 236 additional colors beyond the 20 reserved colors.

Examples of decisions a Windows program can make based on the color resolution value (COLORRES) obtained from GetDeviceCaps:

Optimal Color Choices:

If the color resolution is lower (e.g., 18 bits), the program might prioritize using fewer distinct colors to avoid filling the limited color space with redundant entries.

Color Depth Adjustments:

The program can dynamically adjust its use of colors based on the available color depth. For instance, it might use more colors in higher color depth scenarios.

Palette Efficiency:

In situations with lower color resolution, the program might limit the number of requested shades, optimizing the use of the available color range.

*Regarding how the Palette Manager functions work in Windows applications:*

SelectPalette Function:

Selects a logical palette into a device context.

Only one logical palette can be selected into a device context at a time.

RealizePalette Function:

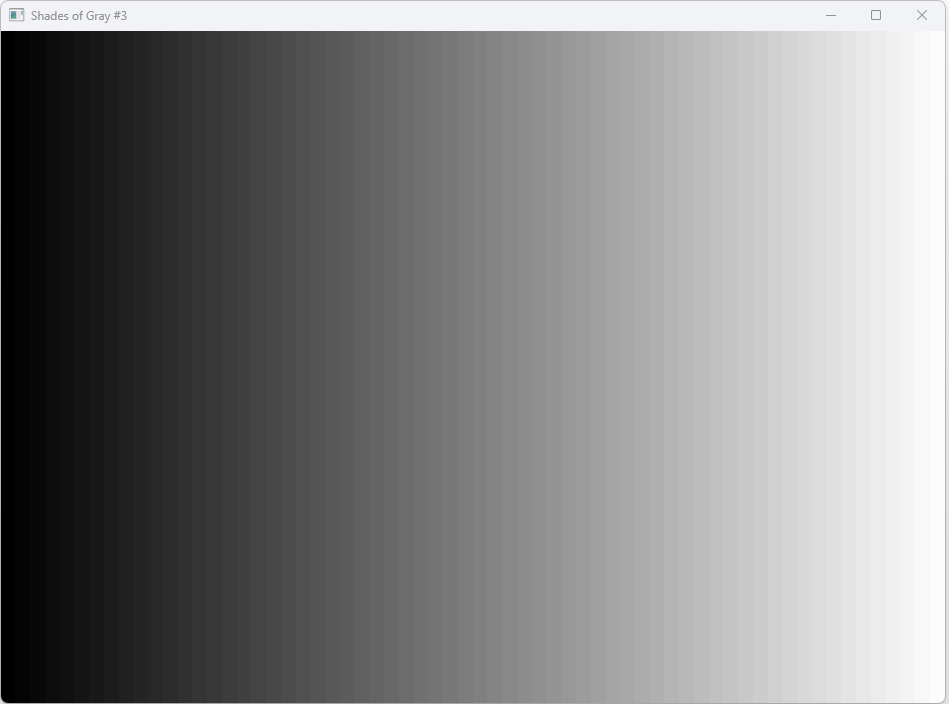
Causes Windows to realize the logical palette in the device context by mapping colors to the system palette.

The actual work, including determining window activity and notifying other windows about palette changes, occurs during this function call.

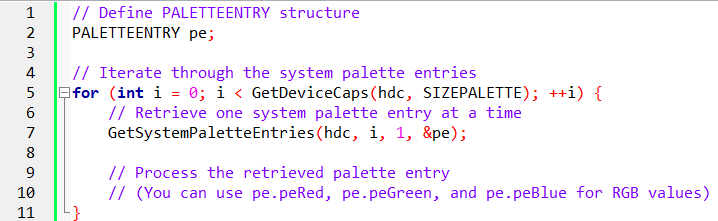
Palette Messages (WM\_QUERYNEWPALETTE and WM\_PALETTECHANGED):

Windows sends these messages to main windows to manage palette changes.

Programs using the Palette Manager should process these messages to ensure proper palette realization and updating.



The system palette in Windows is directly linked to the hardware palette lookup table on the video adapter board, although the color resolution of the hardware palette may be lower than that of the system palette.



The function GetSystemPaletteEntries allows a program to retrieve individual or multiple RGB entries from the system palette, provided that the video adapter mode supports palette manipulation.

The function's parameters include the device context handle (hdc), an unsigned integer indicating the index of the first palette entry (uStart), an unsigned integer specifying the number of palette entries to retrieve (uNum), and a pointer to a PALETTEENTRY structure (&pe).

There are various ways to use the GetSystemPaletteEntries function. A program can define a single PALETTEENTRY structure and make multiple calls to the function, incrementing the index (i) from 0 to the value returned by GetDeviceCaps with the SIZEPALETTE index, typically 255.

Alternatively, a program can retrieve all system palette entries by defining a pointer to a PALETTEENTRY structure, allocating memory to accommodate the required number of PALETTEENTRY structures based on the palette size.

The function essentially allows detailed examination of the hardware palette table. The order of entries in the system palette corresponds to the increasing values of pixel bits used to represent colors in the video display buffer.

This information can be valuable for understanding the organization of colors in the palette and may be utilized for various purposes within a program.

For a more practical demonstration, let's consider an example where a program iterates through the system palette entries, gaining insights into the arrangement of colors in the hardware palette table.

In summary, the Palette Manager functions enable Windows programs to efficiently manage and utilize color palettes in scenarios where there are limitations on available colors, such as in 256-color video modes.

Programs can make informed decisions based on the color resolution value to optimize color usage and provide a better visual experience.

THE RASTER OPERATIONS PROBLEM

Understanding Raster Operations (ROPs):

Purpose: Raster operations, or ROPs, are techniques in GDI (Graphics Device Interface) for combining pixels of different objects during drawing operations.

Types:

* Binary ROPs involve two objects (e.g., line and background).
* Tertiary ROPs involve three objects (e.g., source bitmap, brush, and destination bitmap).

SetROP2 Function: Sets the drawing mode for binary ROPs.

Bitwise Operations: ROPs function by performing bitwise operations (AND, OR, XOR, etc.) on the pixel bits of the involved objects.

The Raster-Op Problem:

Root Cause: ROPs manipulate pixel bits directly, often without regard to their corresponding colors in the system palette.

Color Distortions: When the palette is changed, the relationship between pixel bits and colors can be altered, leading to unexpected color results in ROP-based drawing operations.

Example: Dragging a sizing border in GRAYS2 or GRAYS3 might produce random colors due to inverted pixel bits mapping to incorrect palette entries.

Reserved Colors and Limitations:

Reserved Colors: The system palette reserves 20 standard colors to minimize ROP issues.

Black and White Guarantee: Only black (pixel bits 0) and white (pixel bits 1) are guaranteed to behave predictably in ROPs.

Predicting Results: To anticipate ROP behavior with colors, examine the system palette table to understand the RGB values associated with various pixel bit combinations.

Key Considerations for Handling Raster-Op Issues:

Respect Reserved Colors: Avoid modifying the 20 reserved colors to maintain some ROP consistency.

Exercise Caution with Palette Changes: Be mindful of potential color distortions when altering the palette.

Prioritize Black and White: Use black and white whenever possible for reliable ROP results.

Analyze Palette for Color ROPs: Inspect the system palette table to anticipate color outcomes when using ROPs with non-black/white colors.

Consider Alternative Approaches: Explore techniques that don't rely heavily on ROPs for color-sensitive drawing operations.

Additional Insights:

Figure 16-1: (1st image) May visually illustrate the arrangement of reserved colors in the system palette.

GRAYS2 and GRAYS3 Programs: (Covered) Could be used for hands-on experimentation with ROP behavior.

Remember: Understanding the interplay between ROPs, palettes, and pixel bits is crucial for achieving predictable and visually accurate drawing results in GDI.

The raster operations in GDI, governed by functions like SetROP2 and used in drawing lines and filling areas, involve binary or tertiary operations on pixel bits.

The numeral "2" in SetROP2 signifies a binary raster operation, while tertiary raster operations are employed with functions like BitBlt.

These operations determine how the pixels of the drawn object interact with the pixels of the surface, allowing for diverse effects such as bitwise exclusive-OR combinations.

In the context of changing the palette, the raster operations become particularly nuanced.

These operations manipulate pixel bits, which may not necessarily align with actual colors.

An illustrative example can be observed by running the GRAYS2 or GRAYS3 program and dragging the top or bottom sizing border.

Windows typically uses a raster operation to invert the background pixel bits, ensuring the dragged sizing border remains visible.

However, with these programs, the result may manifest as various random colors, corresponding to unused entries in the palette table resulting from the inversion of pixel bits.

It's crucial to note that only the pixel bits are inverted, not the visible color.

Figure 16-1 demonstrates that the 20 standard reserved colors in the system palette, positioned at the top and bottom, maintain normalcy in the results of raster operations.

However, as palette changes occur, especially if reserved colors are altered, raster operations on colored objects can lose their intended meaning.

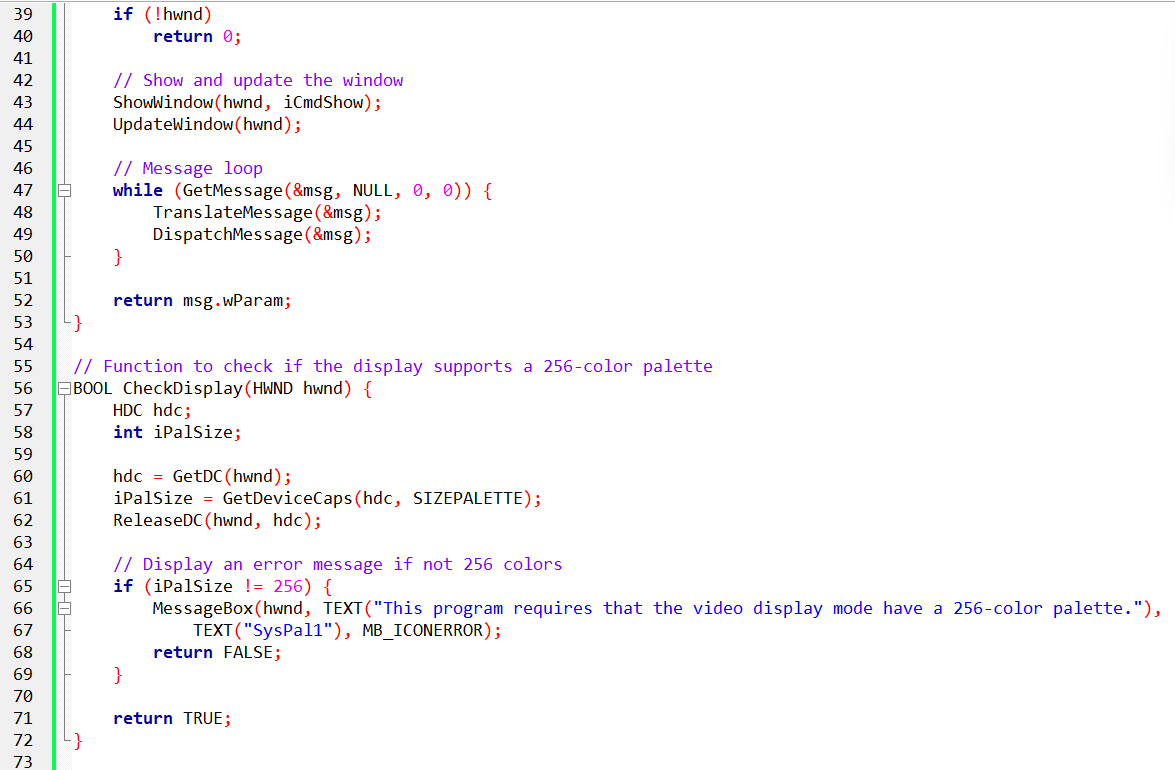
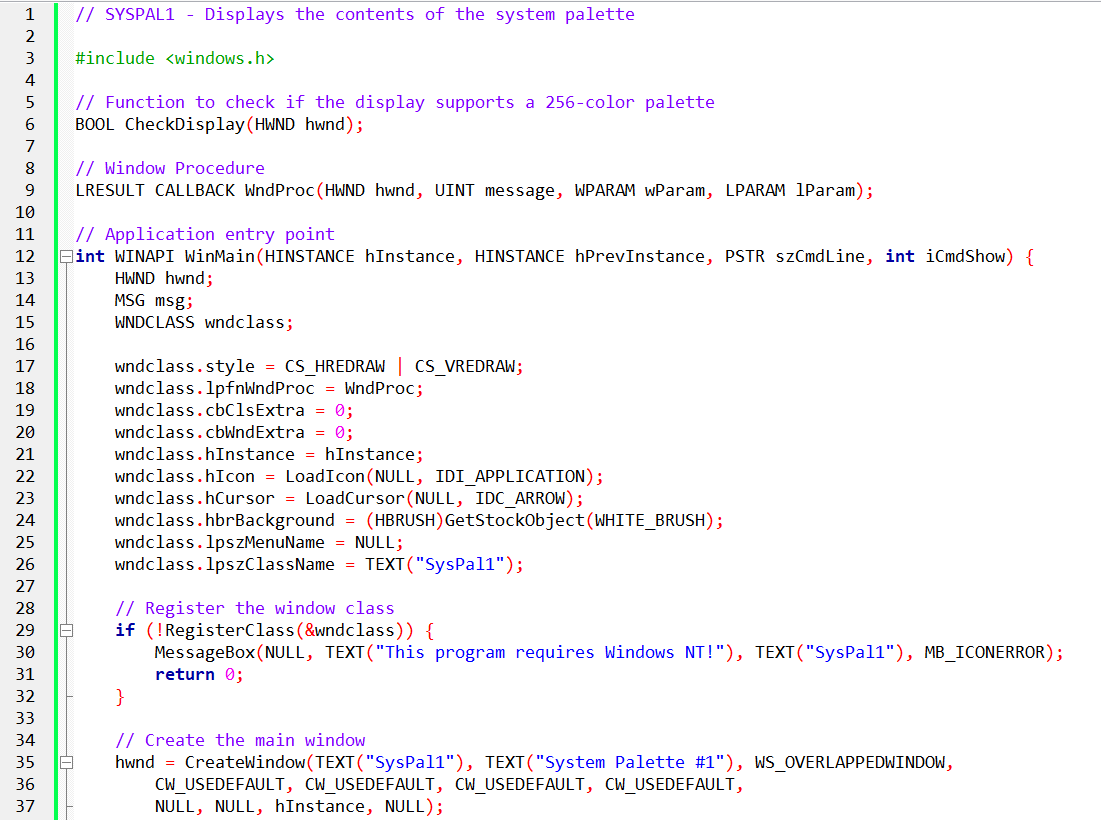
A fundamental guarantee is that raster operations will consistently work with black and white.

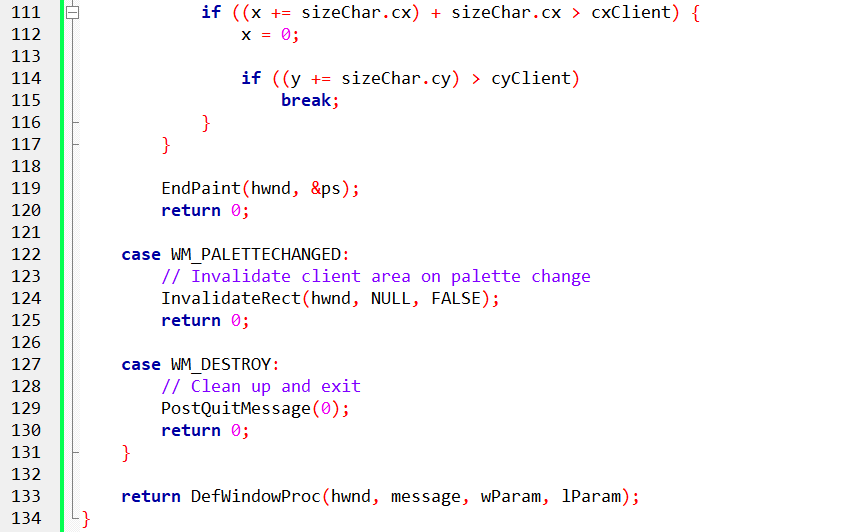
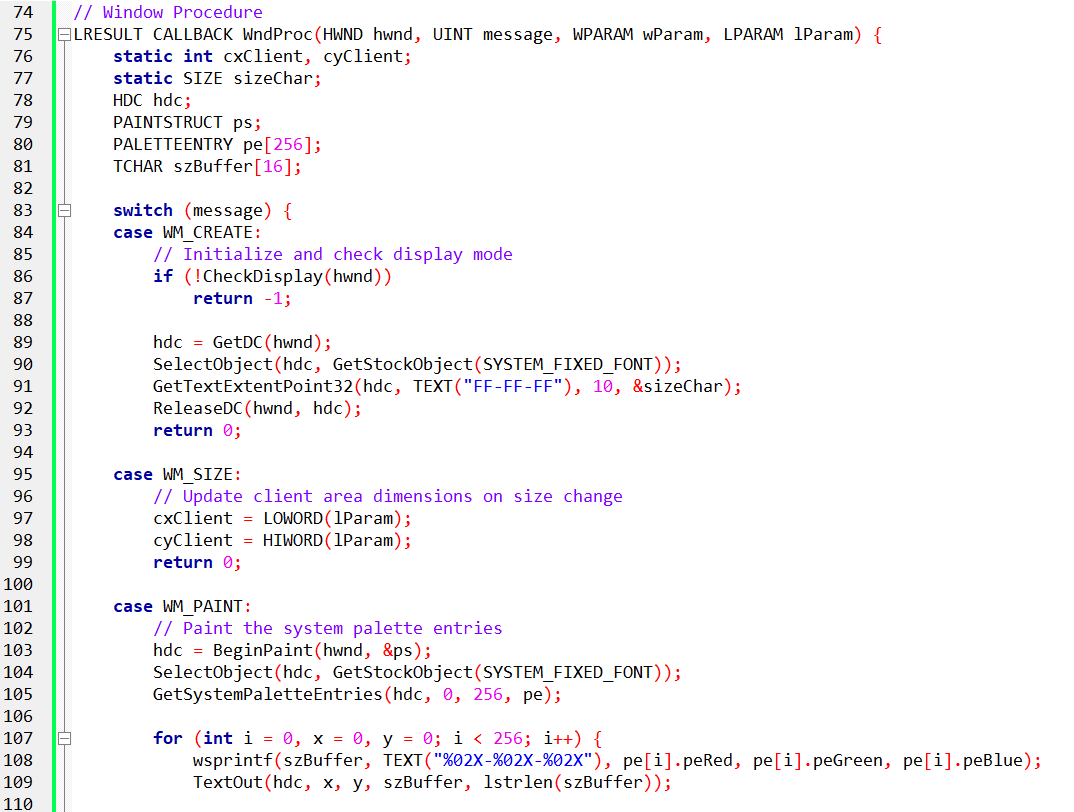
Black, the initial entry in the system palette with all pixel bits set to 0, and white, the final entry with all pixel bits set to 1, remain unchangeable.

To anticipate the outcomes of raster operations on colored objects, one can retrieve the system palette table and inspect the RGB color values associated with different pixel-bit configurations.

This provides a means to predict the impact of raster operations when palette alterations are in play.

SYSPAL1 PROGRAM





Purpose and Overview:

The SYSPAL1 program is designed to display the contents of the system palette in a Windows environment. The system palette, essentially a logical palette, is critical for handling colors across various programs.

The goal is to visualize the RGB values of the system palette, aiding in the debugging of palette-related applications. The program achieves this by utilizing the GetSystemPaletteEntries function, which retrieves the RGB values associated with each color entry in the system palette.

Implementation Details:

The program begins by registering a window class and creating a window. It checks whether the display mode supports a 256-color palette, as this is a prerequisite for the program's operation. The window class and main window creation are standard steps in setting up the graphical user interface.

The heart of the program lies in its Window Procedure (WndProc), where it handles various messages such as WM\_CREATE, WM\_SIZE, WM\_PAINT, WM\_PALETTECHANGED, and WM\_DESTROY.

Initialization (WM\_CREATE):

The WM\_CREATE message initializes the program, checking if the display supports the required 256-color palette. It obtains the device context (hdc) and sets up the font and character dimensions to be used in displaying the RGB values later.

Palette Checking (CheckDisplay):

The CheckDisplay function ensures that the display mode has a 256-color palette. If not, an error message is displayed, and the program exits. This check is crucial for the proper functioning of the subsequent palette-related operations.

Painting the Palette (WM\_PAINT):

The WM\_PAINT message is triggered when the window needs to be redrawn, and it is here that the system palette entries are obtained and displayed. The program uses GetSystemPaletteEntries to fetch the RGB values of all colors in the system palette. It then formats and displays these RGB values as text strings in the client area of the window.

Palette Change Handling (WM\_PALETTECHANGED):

The WM\_PALETTECHANGED message is handled to invalidate the client area when a change in the system palette occurs. This ensures that the display is updated in response to palette modifications.

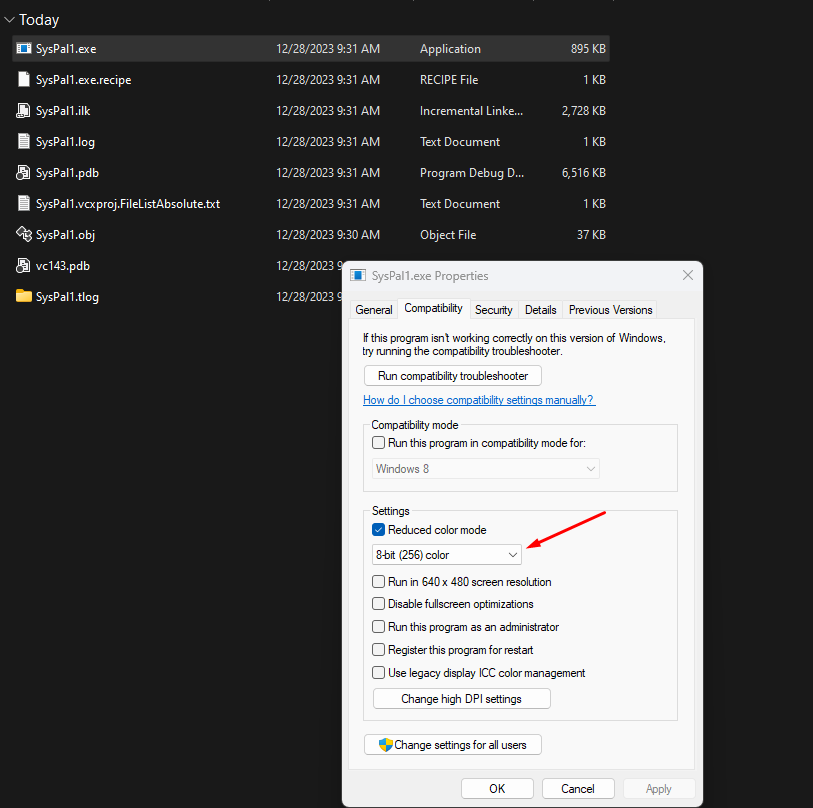
Cleanup and Exit (WM\_DESTROY):

When the user closes the window, the WM\_DESTROY message is processed, leading to the cleaning up of resources and the termination of the program.

Observations and Usage:

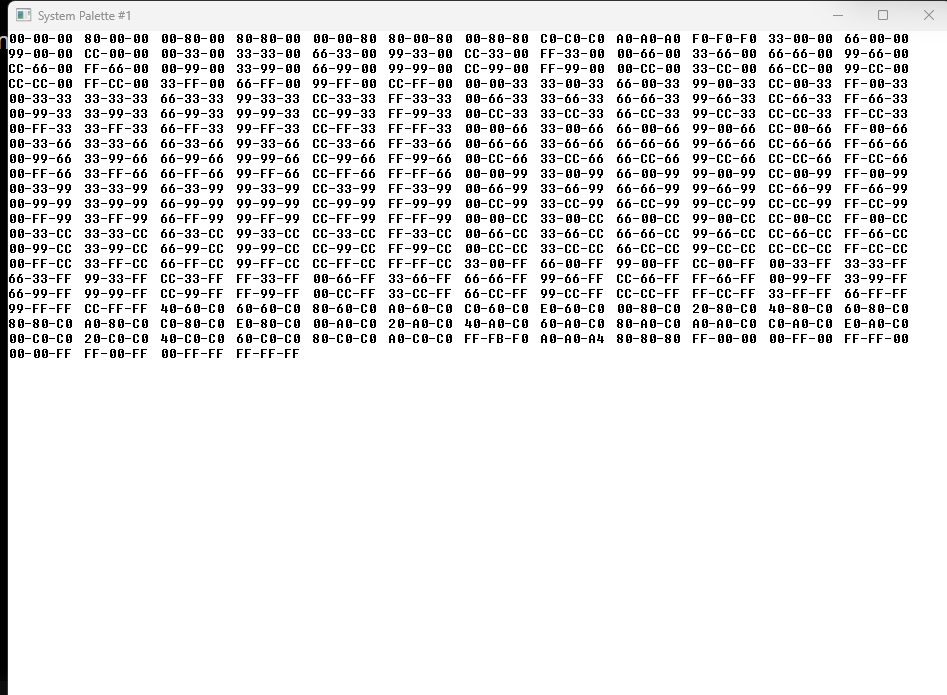
It's important to note that SYSPAL1 specifically caters to environments with a 256-color palette. The program's execution provides a visual representation of the system palette, presenting RGB values in the client area.

This information is valuable for developers working on applications that rely on precise color handling, allowing them to debug and optimize their palette-related functionalities. The program captures the essence of the Windows graphical environment, emphasizing the significance of the system palette in managing color across diverse applications.

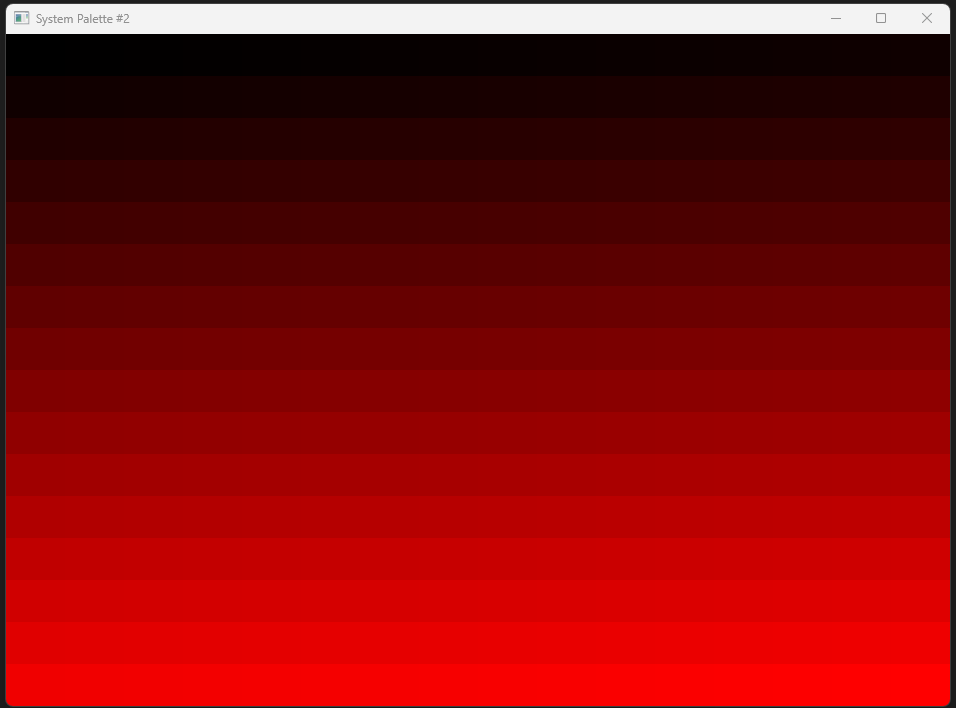


Compile it first, then change the properties because it won’t run the first time.

Output:



SYSPAL2 PROGRAM



Core Functionality:

Visualizing a Custom Palette: The code's primary purpose is to create a window that visually demonstrates a custom 256-color palette, specifically composed of 256 shades of gray.

Window Creation and Palette Setup:

Class Registration: The WinMain function sets the foundation by registering a window class, defining its basic appearance and behavior.

Window Instance and Error Handling: An instance of the window is then created, bearing the title "System Palette #2." The code prioritizes error handling by using the CheckDisplay function to verify if the current display mode supports a 256-color palette, an essential requirement for the intended visualization. If the display mode is incompatible, a user-friendly error message is displayed, and the program gracefully terminates.

Palette Construction: Upon successful window creation, the WM\_CREATE message handler meticulously crafts a custom logical palette:

* An allocated LOGPALETTE structure is used to store 256 palette entries.
* Each entry is filled with a different shade of gray, covering the entire spectrum from pure black (0, 0, 0) to pure white (255, 255, 255).
* The CreatePalette function is then used to create the logical palette, providing a handle for further manipulation.

Orchestrating the Window's Visual Symphony:

* When the window needs to be repainted, the WM\_PAINT message handler takes action.
* It selects the custom palette for the device context (DC) using the SelectPalette function, similar to a painter choosing their preferred colors.
* It then uses the RealizePalette function to accurately map the logical palette to the system palette, ensuring high-quality display and color accuracy.
* A nested loop is used to traverse a 16x16 grid, mimicking an artist's brushstrokes.
* For each grid position, a unique solid brush is created using the custom palette's distinct shades of gray.
* A rectangular canvas is defined within the grid to serve as the area for the brushstroke.
* Finally, the FillRect function is used to apply the chosen gray brush to the canvas, simulating an artist painting on a canvas.

Adapting to Palette Dynamics and Window Closure:

* The WM\_DISPLAYCHANGE message handler keeps track of changes in the display mode.
* It uses the CheckDisplay function to confirm support for the 256-color palette, ensuring a consistent visual experience.
* If compatibility is lost, it gracefully closes the window to maintain visual integrity.
* The WM\_PALETTECHANGED message handler ensures a harmonious color display.
* If another window modifies the system palette, it promptly initiates a repaint by calling InvalidateRect, ensuring the window adapts smoothly to the new palette.
* The WM\_DESTROY message handler takes care of resource cleanup when the window is closed.
* It uses the DeleteObject function to responsibly dispose of the custom palette, demonstrating proper memory management and system hygiene.

Key Takeaways:

The code eloquently showcases palette manipulation in Windows, empowering developers to create custom color schemes that align with their artistic vision.

It underscores the importance of gracefully handling display changes and palette updates to deliver a consistent and visually pleasing experience, akin to an artist adapting to changing lighting conditions.

The code not only reacts to outside influences like changes in the display but also takes charge of its visual appearance.

By carefully creating its own palette of 256 colors, it ensures a consistent and predictable range of colors, regardless of potential changes in the system palette.

This deliberate palette crafting allows the program to act like a careful artist, mixing and choosing its own colors to achieve a desired look.

Each pixel, filled with a specific shade of gray, becomes a subtle note in the overall visual picture. The program orchestrates the interplay of light and dark, creating a sense of depth and texture, much like a skilled artist using shading and tonal variations to bring life to their artwork.

Even though the displayed palette remains the same, the user's interaction with the window adds a dynamic element.

Resizing the window stretches and compresses the grid of colored squares, prompting the viewer to see the same shades of gray differently in various spatial contexts.



This interactive aspect encourages the user to participate in the artistic interpretation, similar to how a museum visitor engages with a painting from different angles.

The core ideas of palette manipulation showcased in the code go beyond grayscale.

By changing the rules used to define the palette entries, artists and developers can unlock a wide range of creative possibilities.

Vibrant rainbow colors, earthy tones, or dreamlike pastel combinations can be brought to life, turning the code into a versatile tool for creating unique visual experiences.

This program can be a starting point for further exploration into color manipulation.

Understanding how pixels, palettes, and code work together offers valuable insights into creating visual stories.

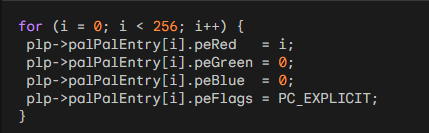
By experimenting with different ways of constructing palettes and adding interactive elements, aspiring artists and programmers can push the boundaries of digital art and find their own path in the ever-changing world of visual expression.

In conclusion, the presented code is more than just a display of a system palette; it reveals a small digital art studio, carefully created and adaptable, ready to be explored and used to create a beautiful combination of colors and pixels. It invites both technical minds and artistic souls to engage in a captivating dance of creativity.

Why is my output red and black?

The code you provided, SYSPAL2, creates a window that displays a grid of colored squares using a custom palette. The colors range from black to red, and the intensity of red increases as you move down the grid. This behavior is intentional and defined in the code.

Here's the relevant section of the code that determines the colors:



In this loop, the program is creating a custom palette with 256 entries. Each entry represents a color, and the red component of each color is set based on the loop index (i). The green and blue components are set to zero, resulting in shades of red.

So, as i increases from 0 to 255, the red component of the color increases, creating a gradient from black to red. This gradient is then displayed in a grid in the window, where the intensity of red becomes more pronounced as you move down the grid.

If you want to modify the color range or experiment with different gradients, you can adjust the logic inside this loop. For example, you could create a gradient that goes from black to a different color or a more complex combination of colors. Experimenting with these values will allow you to create various visual effects based on your preferences.

In the notes, the author explains a technique used in the third version of the SYSPAL program. The technique involves using two GDI functions, BitBlt and StretchBlt, to specify colors directly in pixel bits.

Here are the key points to explain:

GDI functions typically work with color specified as RGB values. The RGB values are converted into pixel bits that correspond to the desired color.

In some video modes (such as 16-bit or 24-bit color modes), the conversion from RGB values to pixel bits is straightforward. However, in other video modes (such as 4-bit or 8-bit color modes), a process called nearest-color search is performed to find the closest matching color in the limited palette.

BitBlt and StretchBlt are two GDI functions that allow specifying color directly in pixel bits. When used in this way, these functions bypass the RGB-to-pixel-bits conversion process.

The behavior of BitBlt and StretchBlt when used to directly specify pixel bits is highly device-dependent. It depends on the capabilities of the video display adapter and the specific video mode in use.

By using BitBlt or StretchBlt to directly specify pixel bits, it becomes possible to display the actual palette lookup table on the video display adapter.

This means that the colors shown on the screen correspond directly to the colors in the palette, without any intermediate color approximation.

Here's a breakdown of the additional points mentioned in the paragraph, focusing on color conversion in GDI and the nearest-color search:

Color Conversion in GDI:

As discussed above, in most cases, GDI functions require color to be specified using RGB values. These RGB values are then converted into pixel bits that represent the corresponding color on the screen.

The conversion process varies depending on the video mode and color depth. In simpler terms, in higher color-depth modes (like 24-bit), the conversion is relatively straightforward, while in lower color-depth modes (like 8-bit or 4-bit), it can be more complex.

Nearest-Color Search:

In limited color modes, where the number of available colors in the display's palette is restricted (such as 4-bit or 8-bit color modes), GDI may need to approximate colors that are not directly available in the palette.

When a color specified in RGB values is not present in the palette, GDI performs a nearest-color search to find the closest matching color available in the palette.

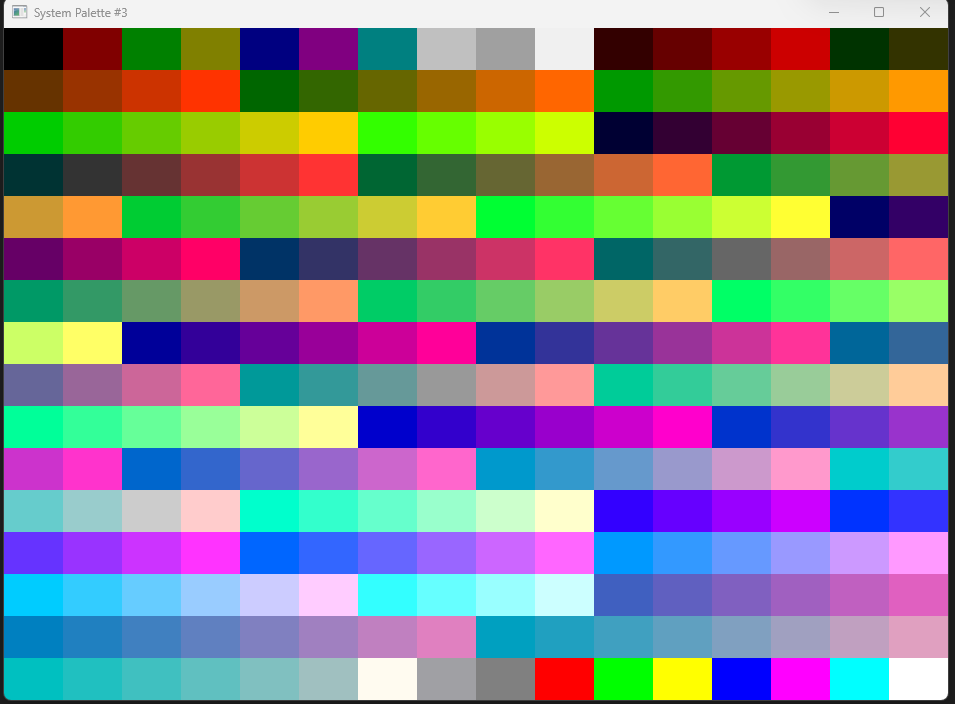
The nearest-color search algorithm determines the best match by comparing the RGB components of the desired color with the RGB components of the colors in the palette. The goal is to find the palette entry that most closely resembles the desired color.

It's important to note that the direct pixel access technique using BitBlt and StretchBlt bypasses the color conversion and nearest-color search process altogether.

Instead of specifying colors using RGB values, these functions allow you to directly manipulate pixel bits, which provides more control over the exact colors displayed on the screen.

However, this technique is highly device-dependent and may not be applicable or produce the desired results in all video modes or display configurations.

SYSPAL3 PROGRAM



The image you sent indeed shows the output of the SYSPAL3 program. It displays all 256 colors available in the system palette, arranged in a neat 16x16 grid. Each pixel in the grid directly corresponds to an entry in the palette, showcasing the full spectrum of colors available for manipulation in your specific video mode.

It's truly fascinating to see how SYSPAL3 bypasses the usual RGB conversion and logical palette management by directly accessing pixel bits. This technique grants deeper insights into the inner workings of the system palette and its connection to the hardware display.

The image itself is visually captivating, with the vibrant colors arranged in a uniform pattern. It serves as a testament to the program's effectiveness in revealing the capabilities of the system palette and the intricate relationship between pixel values and color representation.

Context and Motivation:

Palette Management: In Windows graphics programming, managing color palettes is crucial, especially in limited-color modes (e.g., 256 colors).

RGB Conversion: Most GDI functions specify color in RGB format, which undergoes internal conversion to pixel bits for display.

SYSPAL3's Approach: Explores a direct pixel-based method for accessing and displaying the system palette, bypassing RGB and providing deeper insights into palette mechanics.

Key Features and Steps:

Window Creation:

Establishes a standard window for displaying the palette.

Verifies the presence of a 256-color display mode, as the technique relies on this specific configuration.

Bitmap Construction (WM\_CREATE):

Generates a compact 16x16 bitmap with 8 bits per pixel, serving as a canvas for representing all 256 possible palette entries.

Each pixel value in the bitmap directly maps to an index in the system palette.

Displaying the Palette (WM\_PAINT):

Creates a memory device context, a virtual canvas for graphical operations.

Associates the bitmap with the memory device context, preparing it for display.

Employs the StretchBlt function to enlarge the bitmap to fill the window's client area, effectively showcasing all 256 colors.

During this process, Windows transfers the raw pixel bits from the bitmap to the video hardware, seamlessly accessing the corresponding colors in the palette lookup table.

Palette Synchronization:

Any modifications to the system palette, either by SYSPAL3 itself or other programs, are instantly reflected in the displayed bitmap. This eliminates the need for manual repainting or invalidation of the window's client area, ensuring a dynamic and up-to-date representation of the palette.

Implications and Considerations:

Device Dependence: The technique's reliance on direct pixel manipulation makes it highly specific to 256-color display modes and hardware configurations. It might not function as expected on systems with different color depths or display technologies.

Performance: By circumventing RGB color conversion and logical palette handling, it can potentially offer performance advantages in scenarios where direct pixel access is crucial.

Palette Exploration: SYSPAL3 serves as a valuable tool for exploring the inner workings of the system palette and understanding the relationship between pixel values and color representation in device-dependent graphics programming.

Window Initialization:

Defines a window class and creates a window titled "System Palette #3."

Checks if the display mode supports a 256-color palette through the CheckDisplay function.

Displays an error message and exits if the display mode does not support the required palette.

Bitmap Creation:

During the WM\_CREATE message, creates a 16-by-16 bitmap with 8 bits per pixel.

The bitmap serves as a representation of the system palette, with each pixel corresponding to one of the 256 possible color indices.

Displaying the Palette:

Upon receiving the WM\_PAINT message, initiates the painting process.

Creates a compatible device context (hdcMem) and selects the previously created bitmap into this memory context.

Uses the StretchBlt function to transfer the pixel bits from the bitmap to the hardware display, filling the entire client area.

Dynamic Adaptation:

Unlike SYSPAL2, does not rely on handling WM\_PALETTECHANGED messages to update its display.

Any changes to the system palette are immediately reflected in SYSPAL3's display due to the direct use of the StretchBlt function.

Ensures that the program always represents the current state of the system palette without explicit invalidation.

Program Destruction:

Upon receiving the WM\_DESTROY message, deletes the created bitmap object, freeing up system resources.

Posts a quit message to exit the application.

StretchBlt and Palette Interaction:

The StretchBlt function efficiently transfers the pixel bits from the bitmap to the display, leveraging the system's hardware capabilities.

Allows SYSPAL3 to access the 256 entries in the palette lookup table directly.

Any modifications to the system palette are instantly reflected in the displayed colors without requiring additional handling.

In summary, SYSPAL3's streamlined approach utilizes a bitmap and the StretchBlt function for efficient and dynamic representation of the 256-color system palette, eliminating the need for explicit handling of palette change messages.

PALETTE ANIMATION: A DIFFERENT BREED OF MOTION

While the title might conjure up visions of cartoon characters zipping across your screen, Windows Palette Manager offers a distinct form of animation – much subtler, yet surprisingly effective.

Moving Objects Without Moving Pixels:

Instead of displaying a rapid sequence of bitmaps, palette animation focuses on manipulating the existing visual elements on screen. Imagine drawing your scene, then strategically shifting its colors through palette adjustments.

Certain objects blend into the background, creating the illusion of movement without ever redrawing pixels. This approach boasts impressive speed, ideal for situations requiring smooth and responsive animation.

Palette Preparation – Reserving the Stage:

Creating a palette for animation demands a slight twist compared to previous examples. Each RGB color you plan to adjust during the animation must have its peFlags field set to PC\_RESERVED within the PALETTEENTRY structure.

Remember, normally setting peFlags to 0 allows GDI to merge identical colors from various logical palettes into one system palette entry.

But for animation, we want exclusive control. Setting PC\_RESERVED prevents GDI from merging, ensuring your animation's color changes don't affect other programs or disturb the system palette organization. This isolation keeps the animation swift and predictable.

Bringing the Palette to Life:

During the WM\_PAINT message, standard SelectPalette and RealizePalette calls remain in play. You continue specifying colors using the PALETTEINDEX macro, relying on its logical palette index reference.

The real magic happens in response to WM\_TIMER messages. This is where you control the animation by modifying the palette.

The AnimatePalette function comes into play, taking an array of PALETTEENTRY structures as input. It efficiently updates the necessary entries in both the system palette and the video board hardware's palette table, ensuring a smooth and responsive animation experience.

In essence, palette animation offers a unique approach to moving visuals on screen.

By strategically manipulating the colors of existing elements, you achieve animation effects without the burden of constant redrawing.

This technique shines in scenarios where speed and efficiency are paramount, adding a dynamic touch to your graphical applications.

BOUNCE PROGRAM

The provided code outlines the structure of the BOUNCE program, a Windows application designed to display a bouncing ball.

The program incorporates palette animation, an approach to animation that involves manipulating the palette to alter colors dynamically without redrawing images extensively.

The code is organized into several components, each responsible for a specific aspect of the program's functionality.

*The first portion of the program PalAnim.c*

Header Inclusions and External Declarations:

The program begins with necessary header inclusions, notably the <windows.h> header for Windows-specific functions and types.

Additionally, there are external declarations for functions such as CreateRoutine, PaintRoutine, TimerRoutine, and DestroyRoutine.

These functions handle palette creation, painting, timer-based actions, and destruction, respectively.

Window Procedure (WndProc):

The WndProc function serves as the window procedure for message handling.

It responds to various messages, including WM\_CREATE, WM\_DISPLAYCHANGE, WM\_SIZE, WM\_PAINT, WM\_TIMER, WM\_QUERYNEWPALETTE, WM\_PALETTECHANGED, and WM\_DESTROY.

Each case within this function corresponds to a specific message type and dictates the corresponding actions.

Initialization (WM\_CREATE):

During the WM\_CREATE message, the program checks the display capabilities using the CheckDisplay function.

If the display does not support a 256-color palette, an error message is displayed, and the program exits.

Otherwise, the CreateRoutine function is called to create the initial palette, and the handle to the palette is stored in the hPalette variable.

Display Change Handling (WM\_DISPLAYCHANGE):

In the event of a display change, the program checks the display capabilities again.

If the new display does not support a 256-color palette, the window is destroyed.

This ensures that the program adapts to changes in the display environment.

Window Resizing (WM\_SIZE):

The WM\_SIZE message updates the client area dimensions, storing the width and height in cxClient and cyClient variables, respectively.

Painting (WM\_PAINT):

During the WM\_PAINT message, the program begins painting by obtaining a device context (hdc).

It selects the current palette, realizes the palette, and invokes the PaintRoutine to paint the bouncing ball within the client area. Finally, the painting is concluded with EndPaint.

The subsequent parts of the code handle timer events, palette queries, palette changes, and program destruction.

These components collectively create a framework for a palette-animated bouncing ball application in a Windows environment. Let’s discuss them.

Timer Event Handling (WM\_TIMER):

In response to the WM\_TIMER message, the program retrieves the device context (hdc) for the window.

It then selects the current palette (hPalette) into the device context.

The TimerRoutine function is called, which likely contains the logic for updating the palette or performing actions related to the timer event.

After the timer-related actions are completed, the device context is released using ReleaseDC.

Palette Query (WM\_QUERYNEWPALETTE):

When the program receives a WM\_QUERYNEWPALETTE message, it checks if the palette (hPalette) is valid.

If the palette is valid, the program retrieves the device context (hdc) for the window and selects the current palette into the device context.

The palette is realized in the device context using RealizePalette.

The window is invalidated using InvalidateRect to trigger a repaint, ensuring that the new palette is reflected in the display.

The device context is then released.

Palette Change Notification (WM\_PALETTECHANGED):

Upon receiving a WM\_PALETTECHANGED message, the program checks whether the palette (hPalette) is valid and if the sender of the message is not the current window ((HWND)wParam != hwnd).

If these conditions are met, the program retrieves the device context (hdc) for the window and selects the current palette into the device context.

The palette is realized in the device context using RealizePalette.

The UpdateColors function is called, which likely contains logic for updating colors or taking actions related to palette changes.

The device context is released.

Program Destruction (WM\_DESTROY):

In response to the WM\_DESTROY message, the program initiates cleanup procedures.

The DestroyRoutine function is called, passing the window handle (hwnd) and the palette handle (hPalette).

This function likely handles any necessary cleanup, such as freeing resources associated with the palette.

Finally, the program posts a quit message using PostQuitMessage(0) to terminate the application.

*Let’s now discuss Bounce.c*

CreateRoutine: Crafting the Palette's Canvas

Palette Allocation: Allocates memory for a LOGPALETTE structure, a blueprint for defining a custom palette.

Red Hues: The Palette's Palette: Fills the palette with 34 entries, all sharing a maximum red value (255). The first entry is pure black (0, 0, 0), while the remaining 33 are shades of red (255, G, B).

Green Lights, Red Alert: Sets the peFlags field to PC\_RESERVED for all entries except the last one (index 33), ensuring exclusive control for animation purposes.

Palette Creation and Timer Trigger: Translates the defined palette into a usable HPALETTE object and sets a timer with an ID of 1, initiating the animation process.

PaintRoutine: Bringing the Ball to Life

Background Brush: Uses palette index 33 (presumably a light gray or white shade) to paint the window background, creating a canvas for the ball's movement.

33 Shades of Red: A Ball Symphony: Draws 33 ellipses (balls) arranged in a visually appealing pattern, each colored with a different shade of red from the palette.

Coordinate Calculations: Meticulously positions the balls within the window, ensuring proper placement and spacing for a visually cohesive effect.

TimerRoutine: The Engine of Animation

Ball Tracking and Direction: Maintains two static variables: bLeftToRight (a boolean indicating the ball's movement direction) and iBall (an integer tracking the active ball index).

Whitewash the Past: Sets the green and blue components of the previous ball's palette entry to 255, effectively turning it white and removing its visual presence.

Step Forward (or Back): Updates the iBall index based on the current direction, either incrementing or decrementing it to move the animation forward.

Reverse Course: If the ball reaches the end of its path (index 33 or 0), it reverses direction, creating a continuous bouncing effect.

Red Alert: Sets the green and blue components of the new ball's palette entry to 0, making it stand out in a vibrant red hue.

Animating the Palette: Employs the AnimatePalette function to seamlessly update the colors in the palette, orchestrating the visual transition between balls and producing the smooth bouncing illusion.

DestroyRoutine: Cleaning Up the Playground

Halting the Timer: Stops the timer that drives the animation, preventing unnecessary resource usage when the program exits.

Palette Disposal: Deletes the custom palette, freeing up associated memory and ensuring proper program termination.

Memory Release: Frees the allocated memory block that housed the LOGPALETTE structure, preventing memory leaks.

*Let’s discuss this code a bit more*

PaintRoutine:

The PaintRoutine function is responsible for painting the bouncing balls on the window.

It starts by drawing the window background using a solid brush with the palette index 33. This index was likely reserved for the background color in the CreateRoutine function.

Next, it draws 33 balls, each represented by an ellipse.

The color of each ball is determined by its palette index, ranging from 0 to 32.

The balls are drawn in different positions based on their indices, creating a visually appealing pattern.

The function uses the Ellipse function to draw the individual balls.

TimerRoutine:

The TimerRoutine function handles the logic for animating the bouncing balls.

It maintains a static variable bLeftToRight to keep track of the direction in which the animation is moving.

The variable iBall represents the index of the ball currently undergoing animation.

It updates the color of the previously animated ball to white in the palette to make it disappear.

The bLeftToRight flag and the value of iBall are adjusted to determine the direction and next ball to animate.

The color of the new ball is set to red in the palette, making it visible.

Finally, the AnimatePalette function is called to update the palette and create the animation effect.

DestroyRoutine:

The DestroyRoutine function is responsible for cleanup operations when the window is being destroyed.

It kills the timer associated with the window using KillTimer.

The palette is deleted using DeleteObject.

The memory allocated for the plp structure (LOGPALETTE) is freed using free.

In summary, the PaintRoutine handles the visual representation of bouncing balls, the TimerRoutine manages the animation logic, and the DestroyRoutine takes care of cleanup during program termination. Together, these functions create a palette-animated bouncing ball demonstration in a Windows environment.