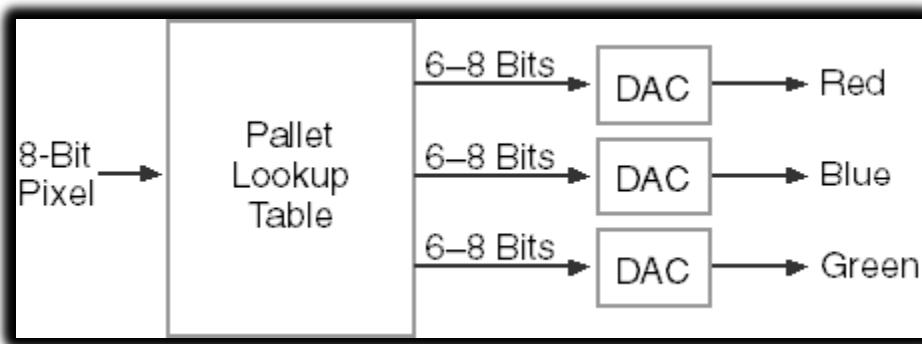


CHAPTER 16: THE PALETTE MANAGER - REVEALING THE WORLD OF 256 COLORS

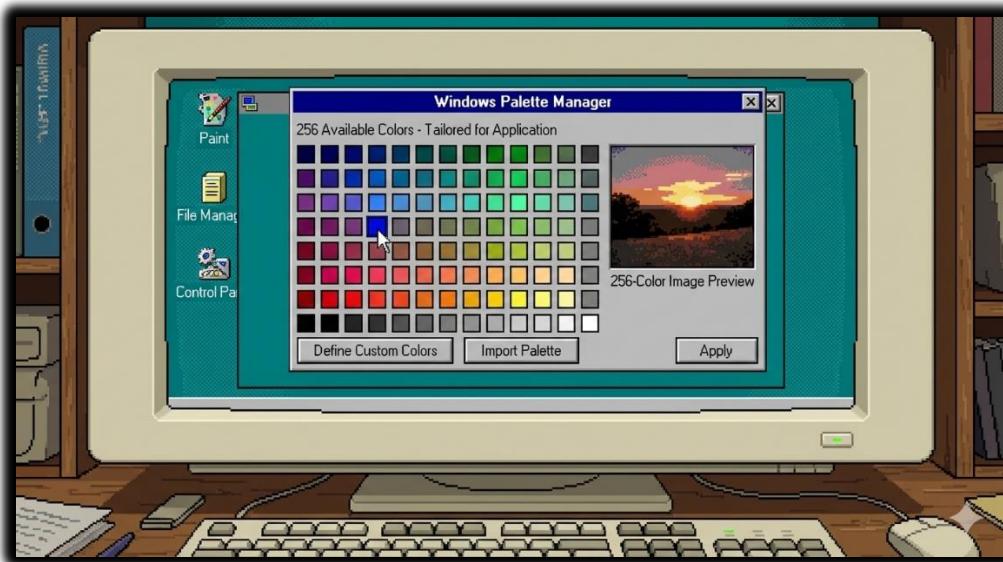
This chapter explores the Windows Palette Manager—a tool that exists largely because of hardware constraints.

On some video systems, color is handled through a palette lookup table. While most modern graphics adapters support high color depths such as 16-bit or 24-bit, certain configurations—especially older laptops or high-resolution modes—are limited to 8 bits per pixel. That means only 256 colors can be displayed at once.



So, what can you really do with just 256 colors? Sixteen colors are clearly too few for realistic images, while thousands or millions make the problem trivial. The challenge lies in that middle ground. With only 256 colors available, every choice matters.

To display real-world images effectively under this limitation, the palette must be tailored to each image. There's no universal "best" set of 256 colors that works for every application—different images demand different color selections.



That's where the Windows Palette Manager comes in. It allows programs running in 8-bit video modes to define exactly which colors they need, making the most of the limited palette.

If your applications always run in higher color modes, you may never need to use the Palette Manager directly. Still, the concepts covered here are useful, especially for rounding out your understanding of bitmap handling and color management.

Palette Manager — Why It Exists 🎨

💻 Hardware Limitation (The Root Problem)

Some video cards **can only display 8 bits per pixel**.

That means:

- **256 total colors**
- Not “millions”
- Not “whatever you want”

Once 256 slots are full → that's it. No extras.

🧠 Why This Is a Problem

Modern thinking:

“Just use RGB, the system will handle it.”

That **does NOT work** in 8-bit mode.

Why?

- You don't get free colors
- Every color must fit inside those 256 slots
- If two apps fight for colors → colors shift, flicker, or look wrong

Palette Manager — Its Actual Job

The **Palette Manager** exists to answer one question:

"Which 256 colors matter most right now?"

It lets an application:

- Declare its **important colors**
- Negotiate with Windows for palette entries
- React when it gains or loses focus

This is **only relevant in 8-bit video modes**.

If you're in 16/24/32-bit color:

 You don't care. Skip this entire topic.

The Color Selection Challenge (Why It's Hard)

In high color modes:

- Windows has a **fixed, universal color model**
- Everyone gets accurate colors

In 256-color mode:

- Every app needs **different colors**
- Photos, UI, gradients, charts — all compete
- You must **choose wisely**

Bad palette = ugly banding, wrong colors, visual chaos.

The 20 Reserved Colors (Important)

In 256-color modes:

- You do NOT get all 256
- 20 colors are reserved by Windows

These are used for:

- Window frames
- Menus
- System UI
- Text and controls

So, your app really gets:

236 colors max

And even those can change when focus shifts.

Why This Still Matters Conceptually

Even if you'll never code for 8-bit video again:

- It explains **why palettes exist**
- It explains **color flashing**
- It explains **why DIB → DDB conversions hurt**
- It teaches **resource-constrained design**

This is real-world engineering thinking.

One-Sentence Mental Model

8-bit video mode is a shared fridge with 256 slots, and Windows already took 20. The Palette Manager decides who gets what's left. That's it.

Pixel Bits	RGB Value	Color Name	Pixel Bits
00000000	00 00 00	Black	11111111
00000001	80 00 00	Dark Red	11111110
00000010	00 80 00	Dark Green	11111101
00000011	80 80 00	Dark Yellow	11111100
00000100	00 00 80	Dark Blue	11111011
00000101	80 00 80	Dark Magenta	11111010
00000110	00 80 80	Dark Cyan	11111001
00000111	C0 C0 C0	Light Gray	11111000
11111111	FF FF FF	White	
11111110	00 FF FF	Cyan	
11111101	FF 00 FF	Magenta	
11111100	00 00 FF	Blue	
11111000	80 80 80	Dark Gray	
11110111	A0 A0 A4	Medium Gray	
11110110	FF FB F0	Cream	
11110101	A6 CA F0	Sky Blue	
11110100	C0 DC C0	Money Green	

The 20 Reserved Colors — What They Really Mean 🎨

Earlier we said:

“Windows already took 20 colors.”

Now here's **what those 20 actually are and why they exist.**

These colors are **hard-reserved** by Windows in **256-color (8-bit) video modes**. Applications **cannot steal them**, override them, or redefine them.

They exist so:

- Windows UI stays readable
- Menus, borders, text, and controls don't randomly change colors
- The system doesn't look broken when apps fight for colors

The First 8: The “Classic VGA” Base

These are the **old-school foundation colors** — straight from early PC graphics:

1. **Black**
2. **Dark Red**
3. **Dark Green**
4. **Dark Yellow**
5. **Dark Blue**
6. **Dark Magenta**
7. **Dark Cyan**
8. **Light Gray**

Think:

Terminal colors, window frames, text, shadows.

These are **non-negotiable**.

The Bright Extremes

At the *other end* of the palette:

9. **White**
10. **Cyan**
11. **Magenta**
12. **Blue**

These are used for:

- Highlights
- Selection states
- UI accents

Windows wants **guaranteed contrast**, so it locks these too.

≋ The Neutral UI Colors

Then come the grays and soft system tones:

13. Dark Gray

14. Medium Gray

15. Cream

16. Sky Blue

17. Money Green

These exist for:

- Dialog backgrounds
- Buttons
- Scrollbars
- List views
- “Default Windows look”

If Windows didn’t reserve these, every app could visually wreck the UI.

❓ The Last 3: Reserved (Hands Off)

18. Reserved

19. Reserved

20. Reserved

Why?

- Future compatibility
- Internal system use
- Historical reasons

You don’t touch them. Period.

The System Palette — The Big Picture

In 256-color mode:

- The **video card has a hardware color table (LUT)**
- Windows mirrors this into the **system palette**
- Total entries: **256**
- Fixed system colors: **20**
- Free for apps: **236**

This palette is **global** — shared by *all* applications.

Logical Palettes (Where Apps Get Involved)

Applications don't directly change hardware colors.

Instead they:

- Create a **logical palette**
- Ask Windows:
“Please map these colors into the system palette”

Windows then:

- Tries its best
- Shuffles colors when apps gain/lose focus

Active Window Priority (Why Colors Flicker)

When multiple apps use palettes:

- **Only one wins**
- That winner is the **active window**

Active window =

-  highlighted title bar
-  foreground app

Its colors take priority.

Background apps:

- Get approximations
- May look wrong
- Often cause **palette flashing**

This is normal behavior in 8-bit mode.

What this section is really saying

The book isn't asking you to switch to **256 colors for nostalgia** or because it's "better." It's doing it **so you can actually see how palettes work**.

In **high-color modes** (16-bit, 24-bit, 32-bit):

- Every pixel stores its *own* color directly
- Palettes are basically ignored
- Windows doesn't need to negotiate colors between programs

So, when you run palette-related code:  **Nothing visible happens**

In **256-color (8-bit) mode**:

- Pixels don't store colors directly
- They store **indexes into a shared palette**
- Windows must constantly manage which program's palette is active

This is where the "interesting" behavior appears:

 **Palette flashing** - When you switch between windows, the system reloads different palettes, causing visible color flicker.

 **Color remapping** - If two programs want different colors in the same palette slots, Windows remaps colors on the fly—sometimes imperfectly.

 **Active-window dominance** - The foreground window gets priority. Its palette is applied more accurately, while background windows may look wrong.

 **Why CreateDIBSection and DDBs matter** - Some bitmap types cooperate with the palette system; others don't. In 256-color mode, the difference becomes obvious and measurable.

Bottom line:

If you don't switch to 256 colors, Windows never enters palette-management mode—so the demonstrations in the chapter simply don't do anything visible.

Key Points

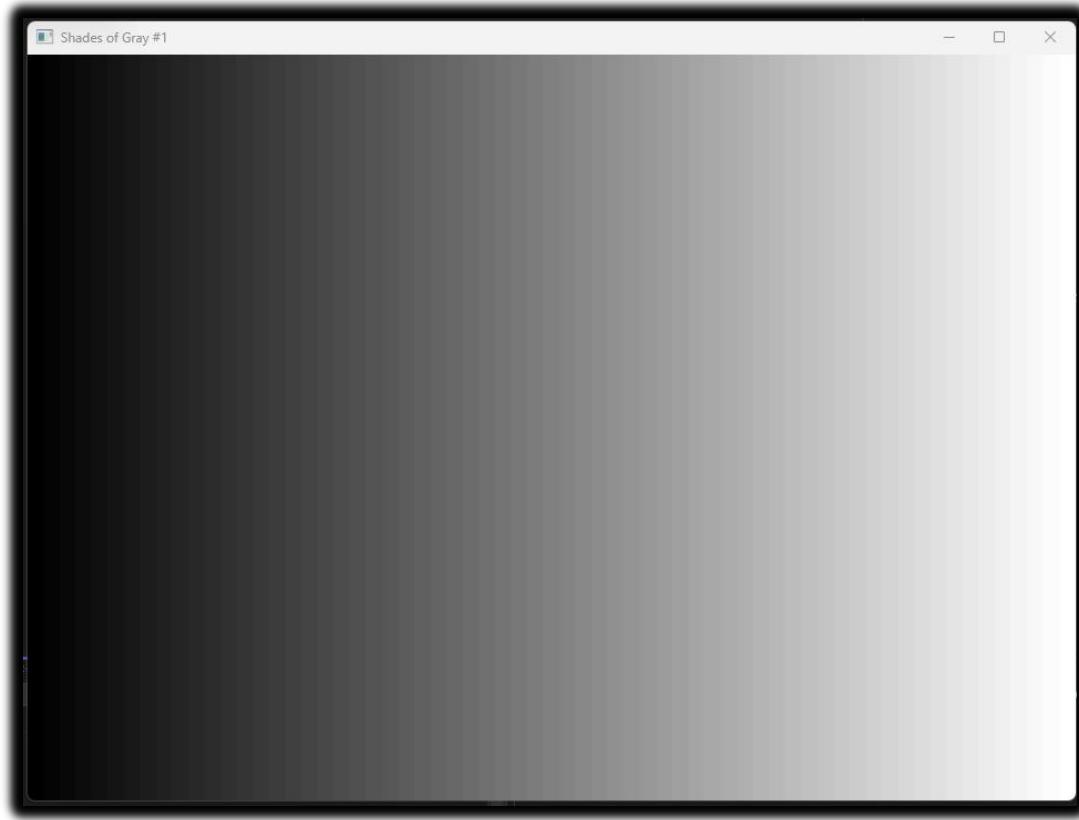
- **System palette** = global 256-color table
- **20 colors are locked** for Windows UI
- **236 colors max** for applications
- **Logical palettes** let apps request colors
- **Active window wins**
- Palette issues only exist in **8-bit modes**

Final Mental Model

256-color mode is musical chairs.
Windows reserves 20 seats.
Apps fight over the rest.
Only the app in focus gets to sit comfortably.

GRAYS1 PROGRAM

```
1 #include <windows.h>
2 LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
3 int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
4     static TCHAR szAppName[] = TEXT("Grays1");
5     HWND hwnd;
6     MSG msg;
7     WNDCLASS wndclass;
8     wndclass.style = CS_HREDRAW | CS_VREDRAW;
9     wndclass.lpszWndProc = WndProc;
10    wndclass.cbClsExtra = 0;
11    wndclass.cbWndExtra = 0;
12    wndclass.hInstance = hInstance;
13    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
14    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
15    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
16    wndclass.lpszMenuName = NULL;
17    wndclass.lpszClassName = szAppName;
18    if (!RegisterClass(&wndclass)) {
19        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
20        return 0;
21    }
22    hwnd = CreateWindow(szAppName, TEXT("Shades of Gray #1"), WS_OVERLAPPEDWINDOW,
23        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
24        NULL, NULL, hInstance, NULL);
25    ShowWindow(hwnd, iCmdShow);
26    UpdateWindow(hwnd);
27    while (GetMessage(&msg, NULL, 0, 0)) {
28        TranslateMessage(&msg);
29        DispatchMessage(&msg);
30    }
31    return msg.wParam;
32}
33
34 LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
35     static int cxClient, cyClient;
36     HBRUSH hBrush;
37     HDC hdc;
38     int i;
39     PAINTSTRUCT ps;
40     RECT rect;
41     switch (message) {
42     case WM_SIZE:
43         cxClient = LOWORD(lParam);
44         cyClient = HIWORD(lParam);
45         return 0;
46     case WM_PAINT:
47         hdc = BeginPaint(hwnd, &ps);
48
49         // Draw the fountain of grays
50         for (i = 0; i < 65; i++) {
51             rect.left = i * cxClient / 65;
52             rect.top = 0;
53             rect.right = (i + 1) * cxClient / 65;
54             rect.bottom = cyClient;
55
56             hBrush = CreateSolidBrush(RGB(min(255, 4 * i), min(255, 4 * i), min(255, 4 * i)));
57             FillRect(hdc, &rect, hBrush);
58             DeleteObject(hBrush);
59         }
60         EndPaint(hwnd, &ps);
61         return 0;
62     case WM_DESTROY:
63         PostQuitMessage(0);
64         return 0;
65     }
66
67     return DefWindowProc(hwnd, message, wParam, lParam);
68 }
```



The **GRAYS1** program draws **65 vertical rectangles**, each one a slightly lighter shade of gray, forming a smooth black-to-white gradient (“fountain”).

That's it.

No palette creation.

No Palette Manager calls.

No logical palettes.

Just **RGB colors + solid brushes**.

1. Why This Program Exists (The Point)

GRAYS1 exists to prove something important:

Even in **256-color (8-bit) mode**, Windows can *appear* to show many more colors than the hardware palette actually has.

How?  **Dithering**

2. What Happens in 256-Color Mode

In 8-bit video modes:

- The system palette has **very few gray entries**, mainly: black, dark gray, light gray, white.
- Yet GRAYS1 shows **65 shades**

Windows achieves this by:

Dithering

- Mixing pixels of available colors
- Creating a *pattern* that your eye perceives as an intermediate shade
- Works well for **filled areas** (like rectangles)

That's why:

- The gradient looks slightly grainy
 - But still smooth enough visually
-

3. Important Distinction (This Is the Gold)

Filled shapes → Windows may dither

Lines and text → No dithering

- They use only pure palette colors

Bitmaps → Usually **approximated**, not dithered

- Often look worse than solid fills in 256-color mode

This explains why:

- Gradients drawn manually can look “okay”
 - But photos and bitmaps look awful in 8-bit modes
-

4. Why No Palette Manager Is Used Here

GRAYS1 deliberately **avoids** the Palette Manager to show:

- What Windows does **by default**
- How far the system can go *without* palette control
- The limits of relying on dithering alone

The **next program (GRAYS2)** exists because:

Dithering is not enough if you care about accurate colors.

That's where palettes come in.

5. Key Takeaways (Short, Honest)

- GRAYS1 draws 65 gray shades using plain RGB brushes
- No palettes, no color management
- In 256-color mode, Windows uses **dithering** to fake extra shades
- Dithering works for filled areas, not for text or bitmaps
- This program sets the stage for **why the Palette Manager exists**

That's all the reader needs.

GRAYS2 PROGRAM

```
1 #include <windows.h>
2 LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
3 int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
4     static TCHAR szAppName[] = TEXT("Grays2");
5     HWND hWnd;
6     MSG msg;
7     WNDCLASS wndclass;
8     wndclass.style = CS_HREDRAW | CS_VREDRAW;
9     wndclass.lpszWndProc = WndProc;
10    wndclass.cbClsExtra = 0;
11    wndclass.cbWndExtra = 0;
12    wndclass.hInstance = hInstance;
13    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
14    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
15    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
16    wndclass.lpszMenuName = NULL;
17    wndclass.lpszClassName = szAppName;
18
19    if (!RegisterClass(&wndclass)) {
20        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
21        return 0;
22    }
23    hWnd = CreateWindow(szAppName, TEXT("Shades of Gray #2"), WS_OVERLAPPEDWINDOW,
24        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
25        NULL, NULL, hInstance, NULL);
26
27    ShowWindow(hWnd, iCmdShow);
28    UpdateWindow(hWnd);
29    while (GetMessage(&msg, NULL, 0, 0)) {
30        TranslateMessage(&msg);
31        DispatchMessage(&msg);
32    }
33    return msg.wParam;
34}
35
36 LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
37     static HPALETTE hPalette;
38     static int cxClient, cyClient;
39     HBRUSH hBrush;
40     HDC hdc;
41     int i;
42     LOGPALETTE* plp;
43     PAINTSTRUCT ps;
44     RECT rect;
45     switch (message) {
46     case WM_CREATE:
47         // Set up a LOGPALETTE structure and create a palette
48         plp = malloc(sizeof(LOGPALETTE) + 64 * sizeof(PALETTEENTRY));
49         plp->palVersion = 0x0300;
50         plp->palNumEntries = 65;
51         for (i = 0; i < 65; i++) {
52             plp->palPalEntry[i].peRed = (BYTE)min(255, 4 * i);
53             plp->palPalEntry[i].peGreen = (BYTE)min(255, 4 * i);
54             plp->palPalEntry[i].peBlue = (BYTE)min(255, 4 * i);
55             plp->palPalEntry[i].peFlags = 0;
56         }
57         hPalette = CreatePalette(plp);
58         free(plp);
59         return 0;
60     case WM_SIZE:
61         cxClient = LOWORD(lParam);
62         cyClient = HIWORD(lParam);
63         return 0;
64     case WM_PAINT:
65         hdc = BeginPaint(hwnd, &ps);
66         // Select and realize the palette in the device context
67         SelectPalette(hdc, hPalette, FALSE);
68         RealizePalette(hdc);
69
70         // Draw the fountain of gray
71         for (i = 0; i < 65; i++) {
72             rect.left = i * cxClient / 64;
73             rect.top = 0;
74             rect.right = (i + 1) * cxClient / 64;
75             rect.bottom = cyClient;
76             hBrush = CreateSolidBrush(PALETTERGB(min(255, 4 * i), min(255, 4 * i), min(255, 4 * i)));
77             FillRect(hdc, &rect, hBrush);
78             DeleteObject(hBrush);
79         }
80         EndPaint(hwnd, &ps);
81         return 0;
82     case WM_QUERYNEWPALETTE:
83         if (!hPalette)
84             return FALSE;
85
86         hdc = GetDC(hwnd);
87         SelectPalette(hdc, hPalette, FALSE);
88         RealizePalette(hdc);
89         InvalidateRect(hwnd, NULL, TRUE);
90         ReleaseDC(hwnd, hdc);
91         return TRUE;
92
93     case WM_PALETTECHANGED:
94         if (!hPalette || (HWND)wParam == hwnd)
95             break;
96
97         hdc = GetDC(hwnd);
98         SelectPalette(hdc, hPalette, FALSE);
99         RealizePalette(hdc);
100        UpdateColors(hdc);
101        ReleaseDC(hwnd, hdc);
102        break;
103
104     case WM_DESTROY:
105         DeleteObject(hPalette);
106         PostQuitMessage(0);
107         return 0;
108     }
109
110     return DefWindowProc(hwnd, message, wParam, lParam);
111 }
```

GRAYS2 is the “palette-aware” version of GRAYS1.

GRAYS1 relied on **dithering**.

GRAYS2 says: “*No tricks — give me real colors.*”

So instead of letting Windows fake grays, GRAYS2 **explicitly asks for them** using the **Palette Manager**.

I. The Core Idea

GRAYS2 creates a **logical palette with 65 gray colors** and tells Windows to use *those exact colors* when drawing.

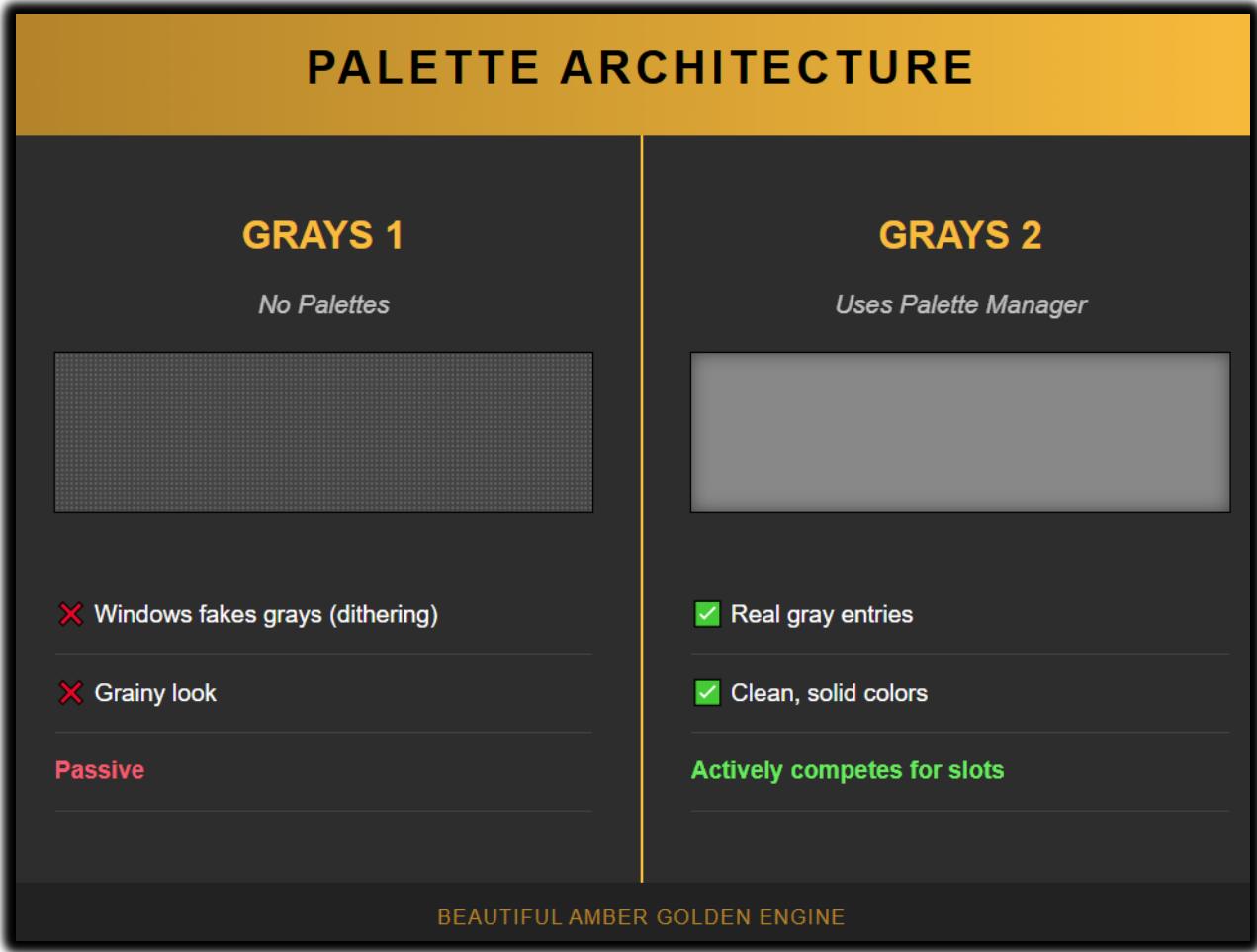
Everything else in the program supports that goal.

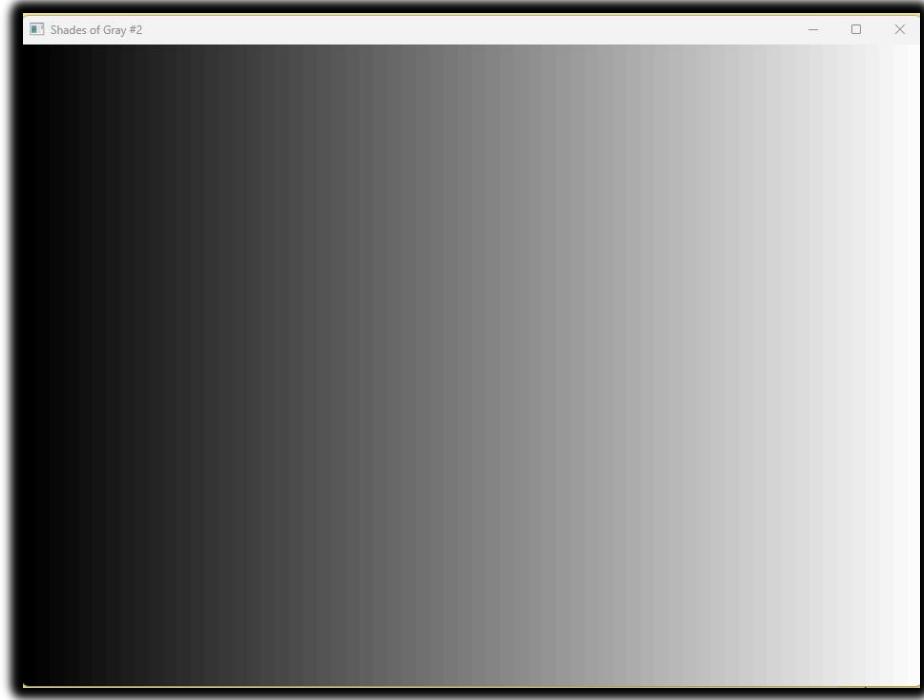
II. What GRAYS2 Does

- Builds a **logical palette** with 65 shades of gray
- Selects that palette into the device context
- Realizes the palette so Windows maps it into the system palette
- Draws the same gray “fountain” as GRAYS1 — but **without dithering**

That’s the whole story.

III. What's Different from GRAYS1 (Important)





GRAYS2 **takes control** instead of accepting whatever colors Windows gives it.

IV. Why the Palette Messages Matter

You'll see these messages:

- WM_QUERYNEWPALETTE
- WM_PALETTECHANGED

You don't need to memorize them.

Just know this:

- **Multiple apps can fight over the palette**
- Windows gives priority to the **active window**
- These messages exist so your app can re-assert its colors when needed

That's it.

V. PALETTERRGB — The One Subtle Rule

When a logical palette is selected:

- **RGB()** → may get remapped or approximated
- **PALETTERRGB()** → tells Windows:

“Use *this* palette entry, not the closest guess”

So GRAYS2 uses **PALETTERRGB** to ensure its grays come from *its own palette*, not the system defaults.

This is the difference between:

- “Something gray”
- “**My exact gray**”

VI. What Happens on Non-256 Color Systems?

Nothing special.

- Palette management is ignored
- Windows behaves like GRAYS1
- The program still works

So GRAYS2 is **safe**, not fragile.

VII. Key Takeaways (No Fluff)

- GRAYS2 exists to show **real palette control**
- It replaces dithering with **explicit gray entries**
- Logical palettes let apps define their own colors
- The active window gets palette priority
- PALETTERRGB ensures your colors come from *your* palette

LOGICAL PALETTE DEFINITION

```
1 // LOGPALETTE structure definition
2 typedef struct {
3     WORD palVersion;
4     WORD palNumEntries;
5     PALETTEENTRY palPalEntry[1];
6 } LOGPALETTE, *PLOGPALETTE;
7 // PALETTEENTRY structure definition
8 typedef struct {
9     BYTE peRed;
10    BYTE peGreen;
11    BYTE peBlue;
12    BYTE peFlags;
13 } PALETTEENTRY, *PPALETTEENTRY;
14
15 // Creating a logical palette in GRAYS2
16 LOGPALETTE* plp = malloc(sizeof(LOGPALETTE) + 64 * sizeof(PALETTEENTRY));
17 plp->palVersion = 0x0300;
18 plp->palNumEntries = 65;
19 for (int i = 0; i < 65; i++) {
20     plp->palPalEntry[i].peRed = (BYTE)min(255, 4 * i);
21     plp->palPalEntry[i].peGreen = (BYTE)min(255, 4 * i);
22     plp->palPalEntry[i].peBlue = (BYTE)min(255, 4 * i);
23     plp->palPalEntry[i].peFlags = 0;
24 }
25 hPalette = CreatePalette(plp);
26 free(plp);
27 // Selecting and realizing the logical palette in WM_PAINT
28 case WM_PAINT:
29     hdc = BeginPaint(hwnd, &ps);
30     // Select and realize the palette in the device context
31     SelectPalette(hdc, hPalette, FALSE);
32     RealizePalette(hdc);
33     // Draw the fountain of grays
34     for (i = 0; i < 65; i++) {
35         // Using PALETTERGB to specify color from logical palette
36         hBrush = CreateSolidBrush(PALETTERGB(min(255, 4 * i), min(255, 4 * i), min(255, 4 * i)));
37         FillRect(hdc, &rect, hBrush);
38         DeleteObject(hBrush);
39     }
40     EndPaint(hwnd, &ps);
41     return 0;
```

A **logical palette** is simply a **list of colors** your program asks Windows to use.

That's it.

You're not drawing pixels yet.

You're saying:

"Here are the colors I care about. Please make room for them."

I. LOGPALETTE — What It Really Is

LOGPALETTE is just a **container**:

- How many colors you want
- What each color's RGB value is

Nothing magical.

Each color is stored as a **PALETTEENTRY**, which is just:

- Red
- Green
- Blue

No drawing happens here.

This is **color registration**, not rendering.

II. What GRAYS2 Does with It

GRAYS2 creates:

- **65 gray colors**
- From black → white

Then it hands that list to Windows using CreatePalette.

At this point:

- The colors exist
- But Windows is **not using them yet**

III. When the Palette Becomes Active

During painting (WM_PAINT):

1. GRAYS2 selects its palette into the device context
2. It calls **RealizePalette**

This tells Windows:

"Map my colors into the system palette now."

If GRAYS2 is the **active window**, Windows tries to honor those colors.

IV. Why PALETTERGB Is Used

This part matters.

When a palette is active:

- `RGB()` → “closest match”
- `PALETTERGB()` → “use my palette entry”

GRAYS2 uses **PALETTERGB** so the rectangles:

- Pull colors from its logical palette
- Not from Windows' default 20 system colors

That's how it avoids dithering.

V. Drawing the Gray Fountain

Once the palette is active:

- GRAYS2 draws rectangles
- Each rectangle uses one gray entry
- Colors are **real**, not simulated

Same visual idea as GRAYS1 —
but now it's **palette-driven**, not hacked.

VI. One-Line Summary

GRAYS2 works because it defines its own colors, tells Windows to use them, and then draws using those exact palette entries.

That's the mental model.

No need to read the code twice.

No need to memorize structures.

What's Important Here 🎯

I. Active window ALWAYS wins

This is the big rule.

- Only **one app** gets palette priority at a time
- That app = **the active window**
- When focus changes → **palette reshuffle happens**

If your app loses focus, **your colors may get remapped**.

👉 This is why old apps used to “flash” colors when you Alt-Tab.

II. System palette is shared

There is **one system palette**.

- If two apps ask for the **same RGB color**
→ Windows uses **one entry**
- If your color matches one of the **20 reserved system colors**
→ Windows maps directly to it
- If no slots are free
→ Windows picks the **closest available color**

You don't fully control colors in 8-bit mode.

You negotiate.

III. PC_NOCOLLAPSE = “don't merge my color”

This flag tells Windows:

“Try NOT to merge this color with others.”

Important nuance:

- It's a **request**, not a guarantee
- Helps prevent your palette entries from collapsing into shared ones
- Still limited by available system palette slots

Use it when color accuracy matters.

IV. The Two Messages You MUST Understand 🧠

🍇 WM_QUERYNEWPALETTE

When your window is about to become active

Your job:

- Select your palette
- Realize it
- Refresh colors

Return value:

- TRUE → “I changed the palette”
- FALSE → “Nothing happened”

This is where your app **takes control**.

🍇 WM_PALETTECHANGED

When another app messed with the palette

Rules:

- Ignore it if **you caused it**
- Otherwise:
 - ✓ Re-select your palette
 - ✓ Re-realize it

This is damage control.

V. Why GRAYS2 Uses UpdateColors 🚀

Instead of repainting everything:

- UpdateColors remaps existing pixels
- Faster
- Less flicker
- Preserves the look while inactive

Smart move for palette apps.

VI. Mental Model (Remember This)

In 256-color mode, colors are political.

Whoever has focus gets priority, others compromise.

- Active window wins palette priority
- System palette is shared
- Same colors get merged
- WM_QUERYNEWPALETTE = take control
- WM_PALETTECHANGED = recover
- PC_NOCOLLAPSE = ask Windows nicely

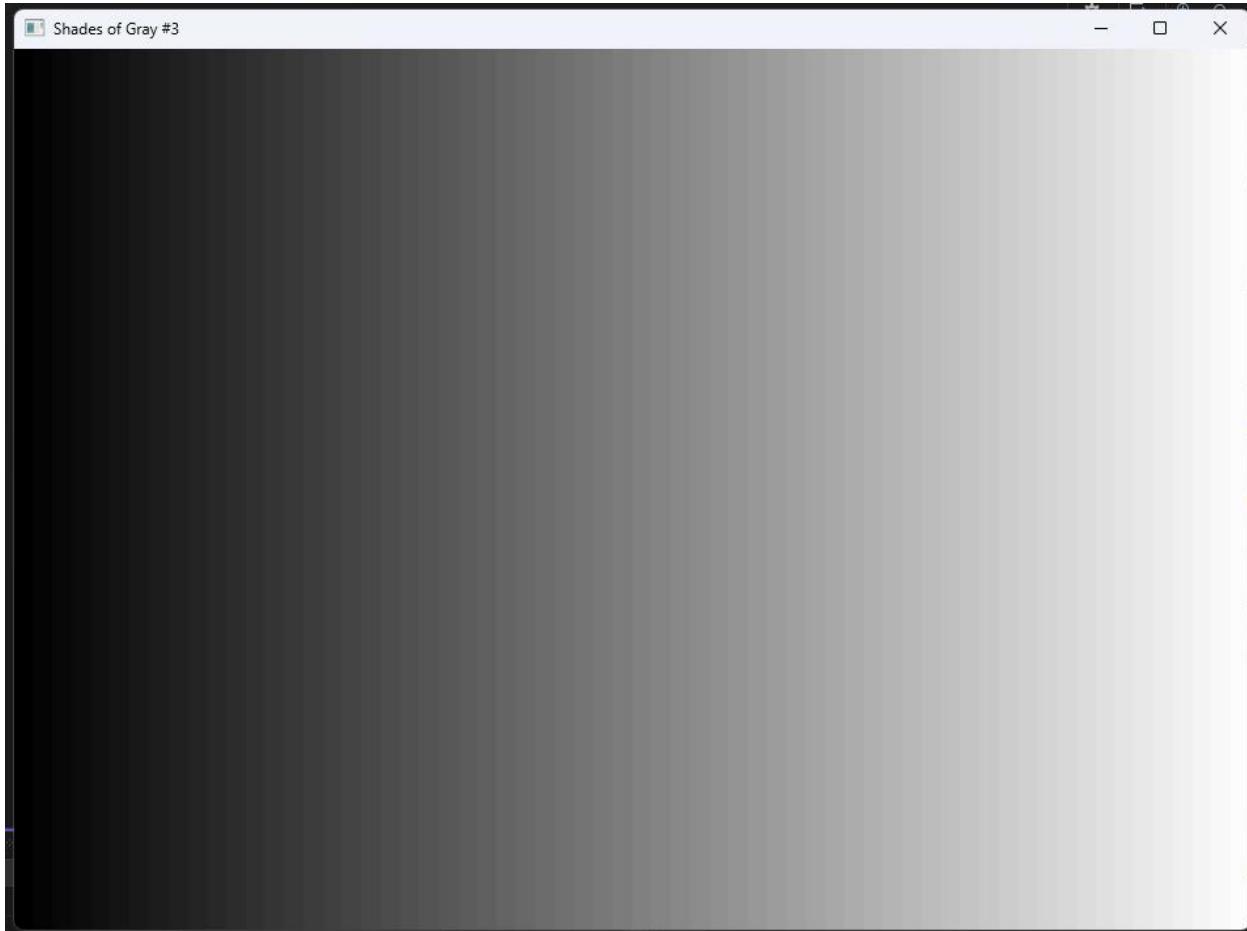
That's it.

Everything else is just plumbing.

GRAYS3 PROGRAM

```
1 // LOGPALETTE structure definition
2 typedef struct {
3     WORD palVersion;
4     WORD palNumEntries;
5     PALETTEENTRY palPalEntry[1];
6 } LOGPALETTE, *PLOGPALETTE;
7
8 // PALETTEENTRY structure definition
9 typedef struct {
10     BYTE peRed;
11     BYTE peGreen;
12     BYTE peBlue;
13     BYTE peFlags;
14 } PALETTEENTRY, *PPALETTEENTRY;
15
16 // Creating a logical palette in GRAYS3
17 LOGPALETTE* plp = malloc(sizeof(LOGPALETTE) + 64 * sizeof(PALETTEENTRY));
18 plp->palVersion = 0x0300;
19 plp->palNumEntries = 65;
20
21 for (int i = 0; i < 65; i++) {
22     plp->palPalEntry[i].peRed = (BYTE)min(255, 4 * i);
23     plp->palPalEntry[i].peGreen = (BYTE)min(255, 4 * i);
24     plp->palPalEntry[i].peBlue = (BYTE)min(255, 4 * i);
25     plp->palPalEntry[i].peFlags = 0;
26 }
27
28 hPalette = CreatePalette(plp);
29 free(plp);
30
31 // WM_PAINT message processing in GRAYS3
32 case WM_PAINT:
33     hdc = BeginPaint(hwnd, &ps);
34
35     // Select and realize the palette in the device context
36     SelectPalette(hdc, hPalette, FALSE);
37     RealizePalette(hdc);
38
39     // Draw the fountain of grays using PALETTEINDEX
40     for (i = 0; i < 65; i++) {
41         rect.left = i * cxClient / 64;
42         rect.top = 0;
43         rect.right = (i + 1) * cxClient / 64;
44         rect.bottom = cyClient;
45
46         hBrush = CreateSolidBrush(PALETTEINDEX(i));
47         FillRect(hdc, &rect, hBrush);
48         DeleteObject(hBrush);
49     }
50
51     EndPaint(hwnd, &ps);
52     return 0;
53
54 // Handling WM_QUERYNEWPALETTE and WM_PALETTECHANGED
55 case WM_QUERYNEWPALETTE:
56     if (!hPalette)
57         return FALSE;
58
59     hdc = GetDC(hwnd);
60     SelectPalette(hdc, hPalette, FALSE);
61     RealizePalette(hdc);
62     InvalidateRect(hwnd, NULL, FALSE);
63     ReleaseDC(hwnd, hdc);
64     return TRUE;
65
66 case WM_PALETTECHANGED:
67     if (!hPalette || (HWND)wParam == hwnd)
68         break;
69
70     hdc = GetDC(hwnd);
71     SelectPalette(hdc, hPalette, FALSE);
72     RealizePalette(hdc);
73     UpdateColors(hdc);
74     ReleaseDC(hwnd, hdc);
75     break;
```

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_QUERYPALLETTE 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:

```

#include <windows.h>

// Function to draw the fountain of grays
void DrawFountainOfGrays(HDC hdc, int cxClient, int cyClient) {
    for (int i = 0; i < 65; i++) {
        RECT rect = {
            .left = i * cxClient / 64,
            .top = 0,
            .right = (i + 1) * cxClient / 64,
            .bottom = cyClient
        };

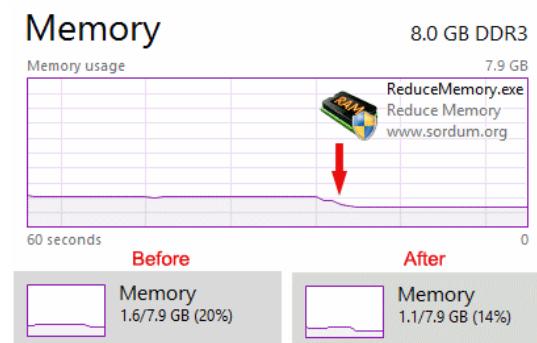
        HBRUSH hBrush = CreateSolidBrush(PALETTEINDEX(i));
        FillRect(hdc, &rect, hBrush);
        DeleteObject(hBrush);
    }
}

```

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.



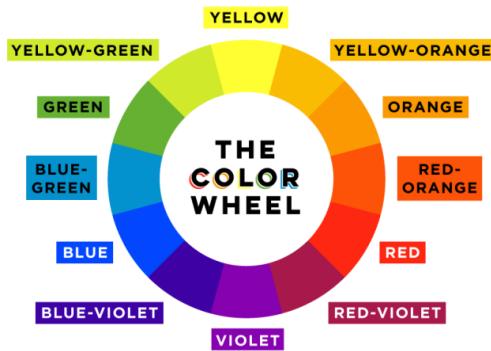
Improved Performance: Since palette indices directly map to entries in the logical palette, color selection and rendering are faster compared to the computation involved in matching RGB values.



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:

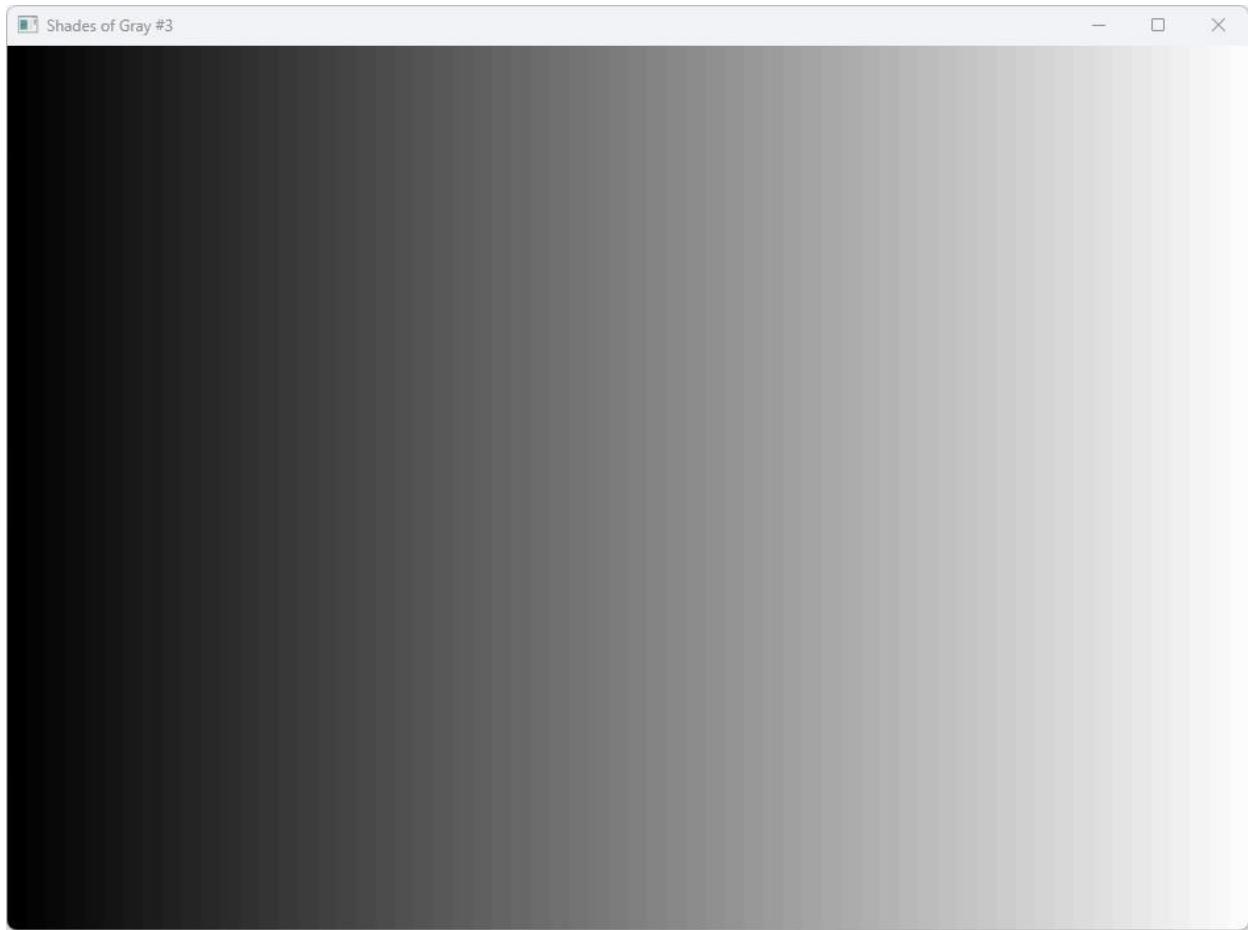
Causes Windows to realize the logical palette in the device context by mapping colors to the system palette.

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.

```
1 // Define PALETTEENTRY structure
2 PALETTEENTRY pe;
3
4 // Iterate through the system palette entries
5 for (int i = 0; i < GetDeviceCaps(hdc, SIZEPALETTE); ++i) {
6     // Retrieve one system palette entry at a time
7     GetSystemPaletteEntries(hdc, i, 1, &pe);
8
9     // Process the retrieved palette entry
10    // (You can use pe.peRed, pe.peGreen, and pe.peBlue for RGB values)
11 }
```

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

```
1 // SYSPAL1 - Displays the contents of the system palette
2
3 #include <windows.h>
4
5 // Function to check if the display supports a 256-color palette
6 BOOL CheckDisplay(HWND hwnd);
7
8 // Window Procedure
9 LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
10
11 // Application entry point
12 int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
13     HWND hwnd;
14     MSG msg;
15     WNDCLASS wndclass;
16
17     wndclass.style = CS_HREDRAW | CS_VREDRAW;
18     wndclass.lpfnWndProc = WndProc;
19     wndclass.cbClsExtra = 0;
20     wndclass.cbWndExtra = 0;
21     wndclass.hInstance = hInstance;
22     wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
23     wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
24     wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
25     wndclass.lpszMenuName = NULL;
26     wndclass.lpszClassName = TEXT("SysPal1");
27
28     // Register the window class
29     if (!RegisterClass(&wndclass)) {
30         MessageBox(NULL, TEXT("This program requires Windows NT!"), TEXT("SysPal1"), MB_ICONERROR);
31         return 0;
32     }
33
34     // Create the main window
35     hwnd = CreateWindow(TEXT("SysPal1"), TEXT("System Palette #1"), WS_OVERLAPPEDWINDOW,
36                         CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
37                         NULL, NULL, hInstance, NULL);
38
39     if (!hwnd)
40         return 0;
41
42     // Show and update the window
43     ShowWindow(hwnd, iCmdShow);
44     UpdateWindow(hwnd);
45
46     // Message loop
47     while (GetMessage(&msg, NULL, 0, 0)) {
48         TranslateMessage(&msg);
49         DispatchMessage(&msg);
50     }
51
52     return msg.wParam;
53 }
54
55 // Function to check if the display supports a 256-color palette
56 BOOL CheckDisplay(HWND hwnd) {
57     HDC hdc;
58     int iPalSize;
59
60     hdc = GetDC(hwnd);
61     iPalSize = GetDeviceCaps(hdc, SIZEPALETTE);
62     ReleaseDC(hwnd, hdc);
63
64     // Display an error message if not 256 colors
65     if (iPalSize != 256) {
66         MessageBox(hwnd, TEXT("This program requires that the video display mode have a 256-color palette."),
67                    TEXT("SysPal1"), MB_ICONERROR);
68         return FALSE;
69     }
70
71     return TRUE;
72 }
```

```
74 // Window Procedure
75 LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
76     static int cxClient, cyClient;
77     static SIZE sizeChar;
78     HDC hdc;
79     PAINTSTRUCT ps;
80     PALETTEENTRY pe[256];
81     TCHAR szBuffer[16];
82
83     switch (message) {
84     case WM_CREATE:
85         // Initialize and check display mode
86         if (!CheckDisplay(hwnd))
87             return -1;
88
89         hdc = GetDC(hwnd);
90         SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
91         GetTextExtentPoint32(hdc, TEXT("FF-FF-FF"), 10, &sizeChar);
92         ReleaseDC(hwnd, hdc);
93         return 0;
94
95     case WM_SIZE:
96         // Update client area dimensions on size change
97         cxClient = LOWORD(lParam);
98         cyClient = HIWORD(lParam);
99         return 0;
100
101    case WM_PAINT:
102        // Paint the system palette entries
103        hdc = BeginPaint(hwnd, &ps);
104        SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
105        GetSystemPaletteEntries(hdc, 0, 256, pe);
106
107        for (int i = 0, x = 0, y = 0; i < 256; i++) {
108            wsprintf(szBuffer, TEXT("%02X-%02X-%02X"), pe[i].peRed, pe[i].peGreen, pe[i].peBlue);
109            TextOut(hdc, x, y, szBuffer, lstrlen(szBuffer));
110
111            if ((x += sizeChar.cx) + sizeChar.cx > cxClient)
112                x = 0;
113
114            if ((y += sizeChar.cy) > cyClient)
115                break;
116        }
117
118        EndPaint(hwnd, &ps);
119        return 0;
120
121    case WM_PALETTECHANGED:
122        // Invalidate client area on palette change
123        InvalidateRect(hwnd, NULL, FALSE);
124        return 0;
125
126    case WM_DESTROY:
127        // Clean up and exit
128        PostQuitMessage(0);
129        return 0;
130    }
131
132    return DefWindowProc(hwnd, message, wParam, lParam);
133
134 }
```

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.

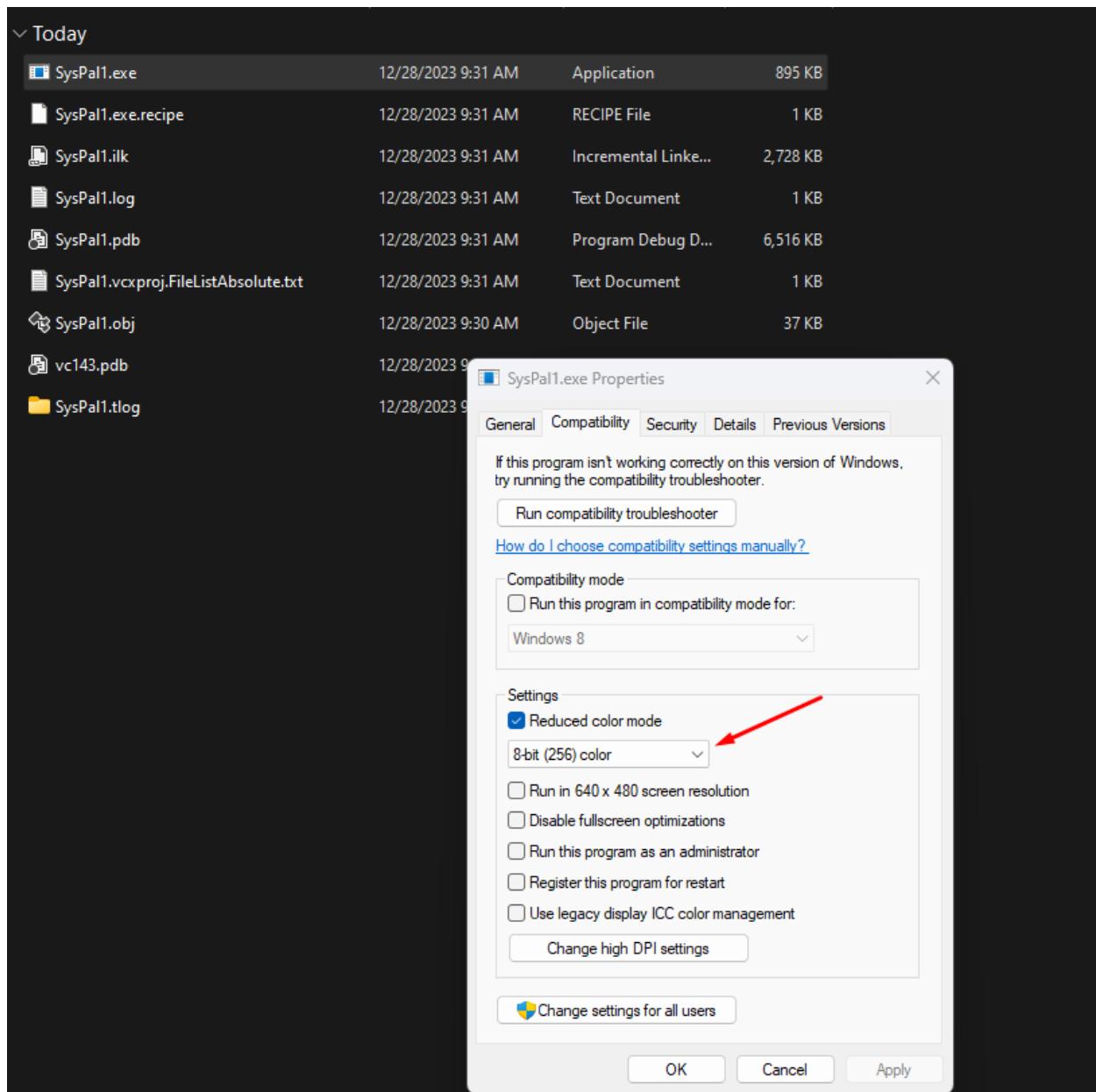
Cleanup and Exit (WM_DESTROY):

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:

```
System Palette #1
00-00-00 80-00-00 00-80-00 80-80-00 00-00-80 80-00-80 00-80-80 C0-C0-C0 A0-A0-A0 F0-F0-F0 33-00-00 66-00-00
99-00-00 CC-00-00 00-33-00 33-33-00 66-33-00 99-33-00 CC-33-00 FF-33-00 00-66-00 33-66-00 66-66-00 99-66-00
CC-66-00 FF-66-00 00-99-00 33-99-00 66-99-00 99-99-00 CC-99-00 FF-99-00 00-CC-00 33-CC-00 66-CC-00 99-CC-00
CC-CC-00 FF-CC-00 33-FF-00 66-FF-00 99-FF-00 CC-FF-00 00-00-33 33-00-33 66-00-33 99-00-33 CC-00-33 FF-00-33
00-33-33 33-33-33 66-33-33 99-33-33 CC-33-33 FF-33-33 00-66-33 33-66-33 66-66-33 99-66-33 CC-66-33 FF-66-33
00-99-33 33-99-33 66-99-33 99-99-33 CC-99-33 FF-99-33 00-CC-33 33-CC-33 66-CC-33 99-CC-33 CC-CC-33 FF-CC-33
00-FF-33 33-FF-33 66-FF-33 99-FF-33 CC-FF-33 FF-FF-33 00-00-66 33-00-66 66-00-66 99-00-66 CC-00-66 FF-00-66
00-33-66 33-33-66 66-33-66 99-33-66 CC-33-66 FF-33-66 00-66-66 33-66-66 66-66-66 99-66-66 CC-66-66 FF-66-66
00-99-66 33-99-66 66-99-66 99-99-66 CC-99-66 FF-99-66 00-CC-66 33-CC-66 66-CC-66 99-CC-66 CC-CC-66 FF-CC-66
00-FF-66 33-FF-66 66-FF-66 99-FF-66 CC-FF-66 FF-FF-66 00-00-99 33-00-99 66-00-99 99-00-99 CC-00-99 FF-00-99
00-33-99 33-33-99 66-33-99 99-33-99 CC-33-99 FF-33-99 00-66-99 33-66-99 66-66-99 99-66-99 CC-66-99 FF-66-99
00-99-99 33-99-99 66-99-99 99-99-99 CC-99-99 FF-99-99 00-CC-99 33-CC-99 66-CC-99 99-CC-99 CC-CC-99 FF-CC-99
00-FF-99 33-FF-99 66-FF-99 99-FF-99 CC-FF-99 FF-FF-99 00-00-CC 33-00-CC 66-00-CC 99-00-CC CC-00-CC FF-00-CC
00-33-CC 33-33-CC 66-33-CC 99-33-CC CC-33-CC FF-33-CC 00-66-CC 33-66-CC 66-66-CC 99-66-CC CC-66-CC FF-66-CC
00-99-CC 33-99-CC 66-99-CC 99-99-CC CC-99-CC FF-99-CC 00-CC-CC 33-CC-CC 66-CC-CC 99-CC-CC CC-CC-CC FF-CC-CC
00-FF-CC 33-FF-CC 66-FF-CC 99-FF-CC CC-FF-CC FF-FF-CC 33-00-FF 66-00-FF 99-00-FF CC-00-FF 00-33-FF 33-33-FF
66-33-FF 99-33-FF CC-33-FF FF-33-FF 00-66-FF 33-66-FF 66-66-FF 99-66-FF CC-66-FF FF-66-FF 00-99-FF 33-99-FF
66-99-FF 99-99-FF CC-99-FF FF-99-FF 00-CC-FF 33-CC-FF 66-CC-FF 99-CC-FF CC-CC-FF FF-CC-FF 33-FF-FF 66-FF-FF
99-FF-FF CC-FF-FF 40-60-C0 60-60-C0 80-60-C0 A0-60-C0 C0-60-C0 E0-60-C0 00-80-C0 20-80-C0 40-80-C0 60-80-C0
80-80-C0 A0-80-C0 C0-80-C0 E0-80-C0 00-A0-C0 20-A0-C0 40-A0-C0 60-A0-C0 80-A0-C0 A0-A0-C0 C0-A0-C0 E0-A0-C0
00-C0-C0 20-C0-C0 40-C0-C0 60-C0-C0 80-C0-C0 A0-C0-C0 FF-FB-F0 A0-A0-A4 80-80-80 FF-00-00 00-FF-00 FF-FF-00
00-00-FF FF-00-FF 00-FF-FF FF-FF-FF
```

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:

```
for (i = 0; i < 256; i++) {
    plp->palPalEntry[i].peRed = i;
    plp->palPalEntry[i].peGreen = 0;
    plp->palPalEntry[i].peBlue = 0;
    plp->palPalEntry[i].peFlags = PC_EXPLICIT;
}
```

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.

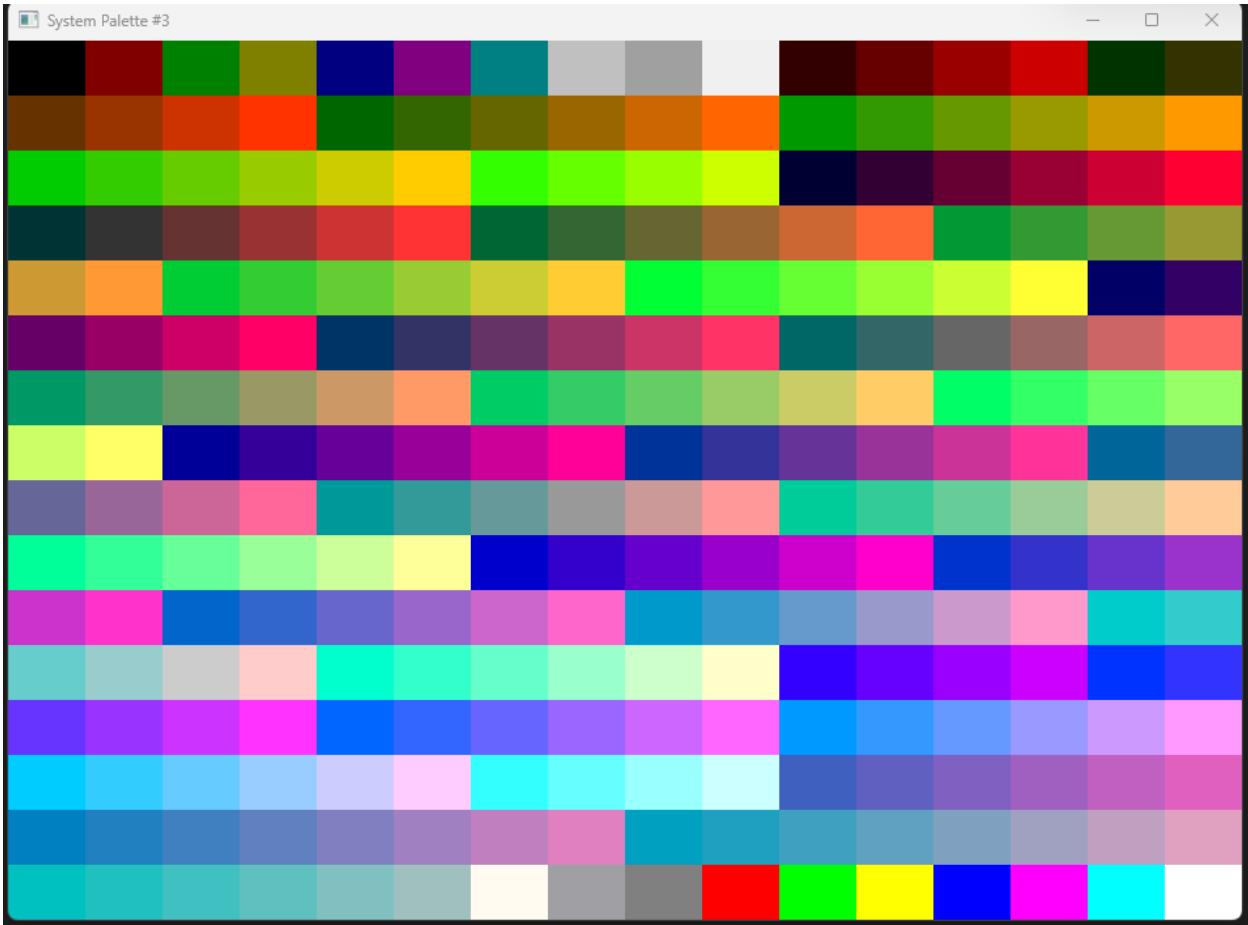
The nearest-color search algorithm determines the best match by comparing the RGB components of the desired color with the RGB components of the colors in the palette. The goal is to find the palette entry that most closely resembles the desired color.

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.

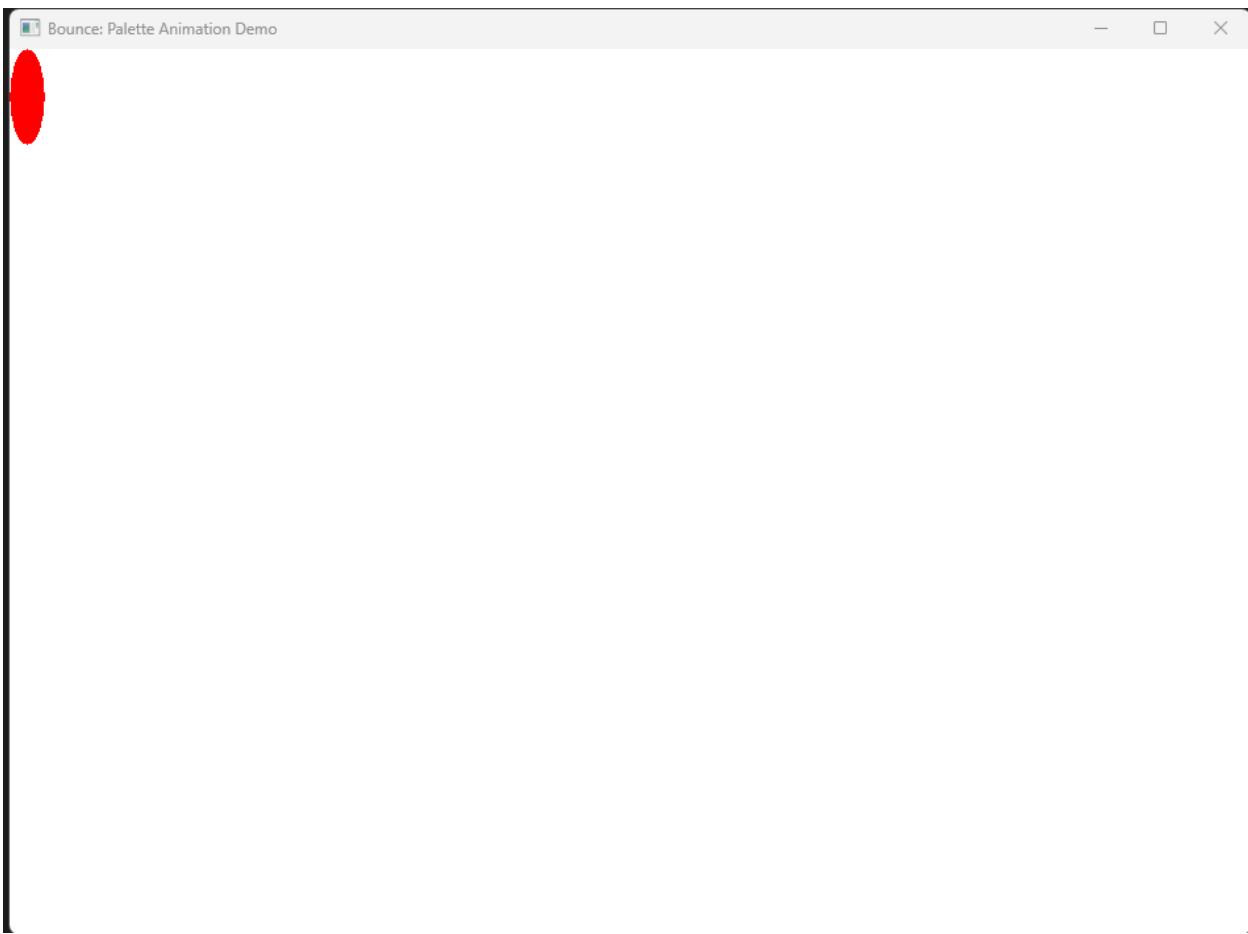
Animating the Palette: Employs the AnimatePalette function to seamlessly update the colors in the palette, orchestrating the visual transition between balls and producing the smooth bouncing illusion.

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: Freed 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.

PALETTE ANIMATION FUNDAMENTALS:

Palette-Driven Animation: A technique that creates the illusion of movement by strategically manipulating colors within a palette, rather than redrawing entire images.

Key Requirements:

256-Color Video Mode: Essential for palette animation to function effectively.

Logical Palette: A custom palette designed for the animation, defining the available colors.

AnimatePalette Function: The core function that modifies entries within the logical palette, triggering visual changes.

CreateRoutine function:

```
HPALETTE CreateRoutine(HWND hwnd) {
    // Allocate memory for the logical palette
    plp = malloc(sizeof(LOGPALETTE) + 33 * sizeof(PALETTEENTRY));

    // Initialize palette entries with red hues and PC_RESERVED flags
    plp->palVersion = 0x0300;
    plp->palNumEntries = 34;
    for (int i = 0; i < 34; i++) {
        plp->palPalEntry[i].peRed = 255;
        plp->palPalEntry[i].peGreen = (i == 0 ? 0 : 255);
        plp->palPalEntry[i].peBlue = (i == 0 ? 0 : 255);
        plp->palPalEntry[i].peFlags = (i == 33 ? 0 : PC_RESERVED);
    }

    // Create the logical palette and set a timer for animation
    HPALETTE hPalette = CreatePalette(plp);
    SetTimer(hwnd, ID_TIMER, 50, NULL);

    return hPalette;
}
```

PaintRoutine function:

```
void PaintRoutine(HDC hdc, int cxClient, int cyClient) {
    // Draw background using palette index 33
    // ...

    // Draw 33 balls in different positions using shades of red
    // ...
}
```

TimerRoutine function:

```
void TimerRoutine(HDC hdc, HPALETTE hPalette) {
    // Make the previous ball white
    plp->palPalEntry[iBall].peGreen = 255;
    plp->palPalEntry[iBall].peBlue = 255;

    // Update ball position and direction
    // ...

    // Make the new ball red
    plp->palPalEntry[iBall].peGreen = 0;
    plp->palPalEntry[iBall].peBlue = 0;

    // Animate the palette
    AnimatePalette(hPalette, 0, 33, plp->palPalEntry);
}
```

Optimization Considerations:

Targeted Palette Updates: Instead of modifying all 33 palette entries, focus on the specific entries that change (the previous and new ball positions).

Using a Single PALETTEENTRY Structure: Create a single structure to hold the color information for the current ball, reducing overhead.

Palette Animation Suitability:

While a bouncing ball might not be the most efficient demonstration of palette animation, this technique excels in scenarios that involve repetitive patterns of movement.

To troubleshoot the issue with your bouncing ball program not displaying the expected animation, follow these steps:

Palette Animation Support:

Confirm that your display mode supports palette animation. Palette animation relies on the use of a palette, and not all display modes support palettes. Ensure that you are running the program on a system that supports palette animation.

Logical Palette Initialization:

Check the initialization of the logical palette in the CreateRoutine function. Ensure that the logical palette is correctly created, and the entries are initialized with the appropriate colors for the bouncing ball.

Timer Interval:

Verify the timer interval in the SetTimer function call. The interval determines how frequently the WM_TIMER message is triggered, affecting the animation speed. Adjust the interval to see if it has an impact on the animation.

Palette Animation Call:

Inspect the AnimatePalette function call in the TimerRoutine. Ensure that it is correctly updating the palette entries and that the parameters passed to AnimatePalette are accurate.

Painting Routine:

Examine the PaintRoutine function. Confirm that the background is drawn using the palette index corresponding to the background color, and the balls are drawn with the correct palette indices.

Debugging Output:

Insert debug output statements or use a debugger to print or inspect variable values during runtime. This can help identify any issues with variable values or logic flow.

Error Handling:

Check for error messages or return values from functions like CreatePalette, SetTimer, and AnimatePalette. Handle any errors that may be occurring during runtime.

Memory Allocation:

Ensure that memory allocation for the LOGPALETTE structure is successful. Check for any errors or warnings related to memory allocation.

Code Review:

Review the code for typos, syntax errors, or logical errors. Pay attention to details in variable names, comparisons, and calculations.

Run with Other Programs:

As mentioned in the explanation, run the bouncing ball program alongside SYSPAL2 or SYSPAL3 to observe how the palette animation interacts with other programs.

Testing Environment:

Test the program on different Windows environments to ensure compatibility.

Operating System Compatibility:

Verify that the program is compatible with the operating system you are using. Some functionalities may behave differently across Windows versions.

By systematically checking these aspects and providing more context about your implementation, we can get to the bottom of why your bouncing ball isn't animating as expected.

FADER PROGRAM

```

1  /*-----|
2   FADER.C -- Palette Animation Demo
3   (c) Charles Petzold, 1998
4   -----*/
5
6 #include <windows.h>
7
8 #define ID_TIMER 1
9
10 TCHAR szAppName [] = TEXT ("Fader") ;
11 TCHAR szTitle [] = TEXT ("Fader: Palette Animation Demo") ;
12
13 static LOGPALETTE lp ;
14
15 HPALETTE CreateRoutine (HWND hwnd)
16 {
17     HPALETTE hPalette ;
18
19     lp.palVersion      = 0x0300 ;
20     lp.palNumEntries   = 1 ;
21     lp.palPalEntry[0].peRed  = 255 ;
22     lp.palPalEntry[0].peGreen = 255 ;
23     lp.palPalEntry[0].peBlue = 255 ;
24     lp.palPalEntry[0].peFlags = PC_RESERVED ;
25
26     hPalette = CreatePalette (&lp) ;
27
28     SetTimer (hwnd, ID_TIMER, 50, NULL) ;
29     return hPalette ;
30 }
31
32 void PaintRoutine (HDC hdc, int cxClient, int cyClient)
33 {
34     static TCHAR szText [] = TEXT (" Fade In and Out ") ;
35     int         x, y ;
36     SIZE        sizeText ;
37
38     SetTextColor (hdc, PALETTEINDEX (0)) ;
39     GetTextExtentPoint32 (hdc, szText, lstrlen (szText), &sizeText) ;
40
41     for (x = 0 ; x < cxClient ; x += sizeText.cx)
42     for (y = 0 ; y < cyClient ; y += sizeText.cy)
43     {
44         TextOut (hdc, x, y, szText, lstrlen (szText)) ;
45     }
46     return ;
47 }
48
49 void TimerRoutine (HDC hdc, HPALETTE hPalette)
50 {
51     static BOOL bFadeIn = TRUE ;
52
53     if (bFadeIn)
54     {
55         lp.palPalEntry[0].peRed  -= 4 ;
56         lp.palPalEntry[0].peGreen -= 4 ;
57
58         if (lp.palPalEntry[0].peRed == 3)
59             bFadeIn = FALSE ;
60     }
61     else
62     {
63         lp.palPalEntry[0].peRed  += 4 ;
64         lp.palPalEntry[0].peGreen += 4 ;
65
66         if (lp.palPalEntry[0].peRed == 255)
67             bFadeIn = TRUE ;
68     }
69
70     AnimatePalette (hPalette, 0, 1, lp.palPalEntry) ;
71     return ;
72 }
73
74 void DestroyRoutine (HWND hwnd, HPALETTE hPalette)
75 {
76     KillTimer (hwnd, ID_TIMER) ;
77     DeleteObject (hPalette) ;
78     return ;
79 }

```

1. Program Overview:

The FADER program, demonstrated in Figure above, is another example of palette animation. Unlike the BOUNCE program, FADER focuses on a single palette entry to create a fade-in and fade-out effect. Similar to other palette animation demos, this program requires the PALANIM.C file.

2. Palette Initialization:

In the CreateRoutine function, the program initializes a logical palette (lp) with a single palette entry. The palette entry is initially set to white (RGB values of 255 for each component), and the PC_RESERVED flag is applied. The CreatePalette function is then called to create the palette, and a timer is set with an interval of 50 milliseconds.

3. Painting Routine:

The PaintRoutine function is responsible for drawing the text "Fade In and Out" repeatedly on the window's client area. The text is drawn using the palette index 0, which corresponds to the single entry in the logical palette.

4. Timer Routine for Fade Effect:

The TimerRoutine function manages the palette animation by altering the RGB values of the single palette entry. The program implements a fade-in and fade-out effect by incrementing and decrementing the RGB values. The AnimatePalette function is then called to update the palette dynamically.

5. Fade-In and Fade-Out Logic:

Fade-In:

The program checks if it's currently in the fade-in phase. If true, it decreases the red and green components of the palette entry. Once the red component reaches 3 (near black), the fade-in phase concludes, and the program switches to the fade-out phase.

Fade-Out:

In the fade-out phase, the red and green components are incremented until the red component reaches 255 (white). At this point, the fade-out phase concludes, and the program switches back to the fade-in phase.

6. Palette Animation Call:

The AnimatePalette function is used to apply the changes to the palette. It takes the palette handle, a range of entries to update (in this case, just one entry at index 0), and the array of PALETTEENTRY structures containing the modified RGB values.

7. Cleanup on Program Termination:

The DestroyRoutine function handles the cleanup tasks. It kills the timer, deletes the palette object, and ensures proper resource management.

The FADER program showcases a creative use of palette animation, emphasizing the simplicity of manipulating a single palette entry to achieve dynamic visual effects. If you have any specific questions or if there's an aspect you'd like to explore further, feel free to ask!

Here are a few additional aspects we can discuss about the FADER program:

1. Palette Animation Techniques:

Color Blending: Explore different methods of color blending for smoother transitions during the fade-in and fade-out phases.

Multiple Palette Entries: Experiment with using more than one palette entry for varied effects, such as simultaneous fading of multiple colors.

2. Enhancements and Variations:

Text Effects: Modify the PaintRoutine to implement various text effects, such as rotation, scaling, or changing font styles.

Background Changes: Introduce background changes or patterns during the animation to create a more dynamic visual experience.

3. User Interaction:

Mouse or Keyboard Interaction: Implement user interaction to control the animation, allowing users to pause, resume, or manipulate the animation speed.

4. Optimizations and Performance:

Optimizing Timer Interval: Fine-tune the timer interval to achieve a balance between smooth animation and system performance.

Memory Management: Explore ways to optimize memory usage, especially if handling a large number of palette entries.

5. Integration with Other Programs:

Combine with SYSPAL Programs: Run the FADER program alongside other palette animation programs like SYSPAL2 or SYSPAL3 to observe interactions and visual effects.

6. Dynamic Text Content:

Text Content Changes: Dynamically change the text content during the animation to display messages or create visual storytelling.

7. Debugging and Troubleshooting:

Debug Output: Implement debug output statements or use a debugger to trace the program's execution and understand how variables change over time.

Error Handling: Enhance error handling mechanisms to gracefully handle unexpected scenarios.

8. Cross-Platform Considerations:

Compatibility: Consider adapting the program for cross-platform compatibility or different Windows versions.

9. Educational Insights:

Teaching Tool: Discuss how the FADER program can serve as an educational tool for understanding palette animation concepts, RGB color model, and Windows graphics programming.

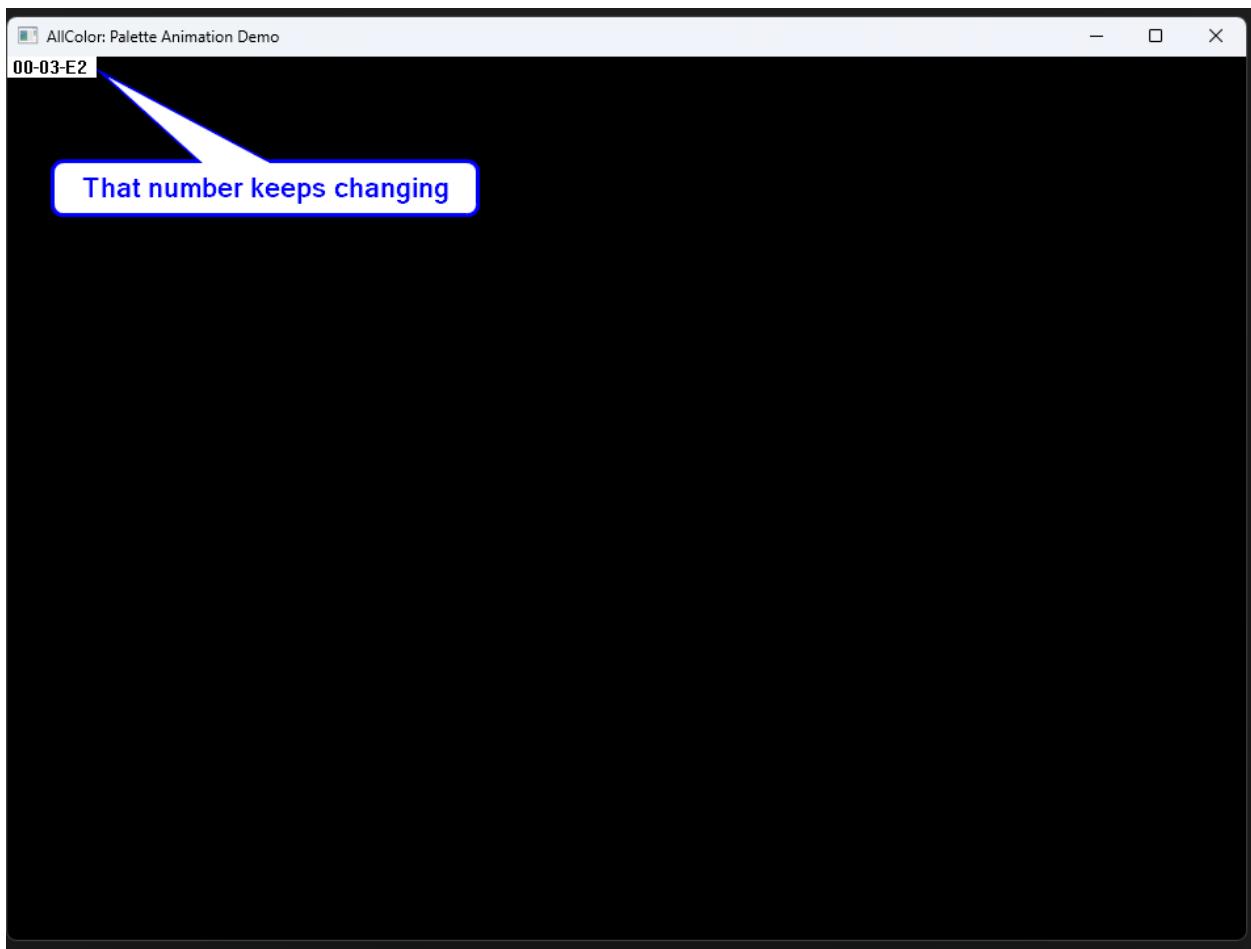
ALLCOLORS PROGRAM

```
1  /*-----  
2   ALLCOLOR.C -- Palette Animation Demo  
3   (c) Charles Petzold, 1998  
4   -----*/  
5  
6  #include <windows.h>  
7  #define ID_TIMER    1  
8  
9  TCHAR szAppName [] = TEXT ("AllColor") ;  
10 TCHAR szTitle [] = TEXT ("AllColor: Palette Animation Demo") ;  
11 static int      iIncr ;  
12 static PALETTEENTRY pe ;  
13  
14 HPALETTE CreateRoutine (HWND hwnd)  
15 {  
16     HDC      hdc ;  
17     HPALETTE hPalette ;  
18     LOGPALETTE lp ;  
19     // Determine the color resolution and set iIncr  
20     hdc = GetDC (hwnd) ;  
21     iIncr = 1 << (8 - GetDeviceCaps (hdc, COLORRES) / 3) ;  
22     ReleaseDC (hwnd, hdc) ;  
23     // Create the logical palette  
24     lp.palVersion      = 0x0300 ;  
25     lp.palNumEntries   = 1 ;  
26     lp.palPalEntry[0].peRed = 0 ;  
27     lp.palPalEntry[0].peGreen = 0 ;  
28     lp.palPalEntry[0].peBlue = 0 ;  
29     lp.palPalEntry[0].peFlags = PC_RESERVED ;  
30  
31     hPalette = CreatePalette (&lp) ;  
32     // Save global for less typing  
33     pe = lp.palPalEntry[0] ;  
34     SetTimer (hwnd, ID_TIMER, 10, NULL) ;  
35     return hPalette ;  
36 }  
37
```

```

38 void DisplayRGB (HDC hdc, PALETTEENTRY * ppe)
39 {
40     TCHAR szBuffer [16] ;
41     wsprintf (szBuffer, TEXT (" %02X-%02X-%02X "), ppe->peRed, ppe->peGreen, ppe->peBlue) ;
42     TextOut (hdc, 0, 0, szBuffer, strlen (szBuffer)) ;
43 }
44
45 void PaintRoutine (HDC hdc, int cxClient, int cyClient)
46 {
47     HBRUSH hBrush ;
48     RECT rect ;
49     //Draw Palette Index 0 on entire window
50     hBrush = CreateSolidBrush (PALETTEINDEX (0)) ;
51     SetRect (&rect, 0, 0, cxClient, cyClient) ;
52     FillRect (hdc, &rect, hBrush) ;
53     DeleteObject (SelectObject (hdc, GetStockObject (WHITE_BRUSH))) ;
54     // Display the RGB value
55     DisplayRGB (hdc, &pe) ;
56     return ;
57 }
58
59 void TimerRoutine (HDC hdc, HPALETTE hPalette)
60 {
61     static BOOL bRedUp = TRUE, bGreenUp = TRUE, bBlueUp = TRUE ;
62     // Define new color value
63     pe.peBlue += (bBlueUp ? iIncr : -iIncr) ;
64     if (pe.peBlue == (BYTE) (bBlueUp ? 0 : 256 - iIncr))
65     {
66         pe.peBlue = (bBlueUp ? 256 - iIncr : 0) ;
67         bBlueUp ^= TRUE ;
68         pe.peGreen += (bGreenUp ? iIncr : -iIncr) ;
69         if (pe.peGreen == (BYTE) (bGreenUp ? 0 : 256 - iIncr))
70         {
71             pe.peGreen = (bGreenUp ? 256 - iIncr : 0) ;
72             bGreenUp ^= TRUE ;
73             pe.peRed += (bRedUp ? iIncr : -iIncr) ;
74             if (pe.peRed == (BYTE) (bRedUp ? 0 : 256 - iIncr))
75             {
76                 pe.peRed = (bRedUp ? 256 - iIncr : 0) ;
77                 bRedUp ^= TRUE ;
78             }
79         }
80     }
81     // Animate the palette
82     AnimatePalette (hPalette, 0, 1, &pe) ;
83     DisplayRGB (hdc, &pe) ;
84     return ;
85 }
86
87 void DestroyRoutine (HWND hwnd, HPALETTE hPalette)
88 {
89     KillTimer (hwnd, ID_TIMER) ;
90     DeleteObject (hPalette) ;
91     return ;
92 }

```



Colorful Prelude:

The ALLCOLOR program, showcased in Figure 16-10, is designed to exhibit a spectrum of colors sequentially, providing a visual representation of the palette's capabilities.

Palette Initialization:

The program begins by creating a logical palette in the CreateRoutine function. It dynamically determines the color resolution of the system, setting the variable iIncr to reflect the color depth. The logical palette consists of a single entry initialized to black (0, 0, 0) and marked as PC_RESERVED.

```
LOGPALETTE lp;
lp.palVersion = 0x0300;
lp.palNumEntries = 1;
lp.palPalEntry[0].peRed = 0;
lp.palPalEntry[0].peGreen = 0;
lp.palPalEntry[0].peBlue = 0;
lp.palPalEntry[0].peFlags = PC_RESERVED;
```

Displaying RGB Information:

The DisplayRGB function prints the current RGB values of the palette entry at the top left corner of the window. It utilizes the TextOut function to display the hexadecimal representation of the RGB values.

```
void DisplayRGB(HDC hdc, PALETTEENTRY *ppe) {
    /* Display RGB values at the top left corner */
    TCHAR szBuffer[16];
    wsprintf(szBuffer, TEXT(" %02X-%02X-%02X "), ppe->peRed, ppe->peGreen, ppe->peBlue);
    TextOut(hdc, 0, 0, szBuffer, lstrlen(szBuffer));
}
```

Painting Routine:

In the PaintRoutine function, the program fills the entire window with the color represented by palette index 0. This serves as the background against which the changing colors will be displayed. The DisplayRGB function is then called to show the RGB values at the top left corner.

```
void PaintRoutine(HDC hdc, int cxClient, int cyClient) {
    HBRUSH hBrush;
    RECT rect;

    /* Draw Palette Index 0 on the entire window */
    hBrush = CreateSolidBrush(PALETTEINDEX(0));
    SetRect(&rect, 0, 0, cxClient, cyClient);
    FillRect(hdc, &rect, hBrush);
    DeleteObject(SelectObject(hdc, GetStockObject(WHITE_BRUSH)));

    /* Display the RGB value */
    DisplayRGB(hdc, &pe);
}
```

Color Transition - TimerRoutine:

The TimerRoutine function handles the color transition. It incrementally adjusts the RGB values of the palette entry, creating a smooth transition from one color to another. The AnimatePalette function is then called to update the logical palette with the new color.

```
void TimerRoutine(HDC hdc, HPALETTE hPalette) {
    // ... (Color transition logic)

    /* Animate the palette and display updated RGB value */
    AnimatePalette(hPalette, 0, 1, &pe);
    DisplayRGB(hdc, &pe);
}
```

Continuous Animation:

The program sets up a timer with a 10-millisecond interval to trigger the TimerRoutine function periodically. This continuous execution results in a seamless animation of color changes.

```
SetTimer(hwnd, ID_TIMER, 10, NULL);
```

The number:

The number displayed at the top left corner of the window in the ALLCOLOR program serves as a **real-time representation of the RGB (Red, Green, Blue) values** of the current color being displayed. This dynamic display is achieved through a series of intricate processes that unfold as the program executes.

The **journey begins with the initialization of the logical palette** in the CreateRoutine function. Here, the program gauges the color resolution of the system, determining the variable iIncr accordingly.

Subsequently, a logical palette with a single entry is created, initially set to black (0, 0, 0), and marked as PC_RESERVED.

The responsibility of showcasing the RGB values falls on the DisplayRGB function, which employs the TextOut function to print the RGB values in hexadecimal format at the window's top left corner.

These RGB values are obtained from the [PALETTEENTRY structure \(pe\)](#), representing the current color in the palette.

The crux of the color transition lies in the TimerRoutine function, triggered at regular intervals by the WM_TIMER message.

This function orchestrates a [dynamic change in the RGB values](#), creating a fluid transition from one color to another.

The process involves [incrementing or decrementing the blue, green, and red components](#) based on boolean flags and reversing directions when reaching boundaries.

The AnimatePalette function is then summoned to update the logical palette with the recalibrated color, ensuring a smooth and continuous animation.

Simultaneously, the DisplayRGB function is called within TimerRoutine to refresh and exhibit the updated RGB values, providing a visual insight into the changing color.

The [numbers displayed at the top left corner](#) constantly change as a result of a fascinating interplay between RGB values. This mesmerizing dance is achieved through a well-planned process that involves setting up the color palette, smoothly transitioning between colors, and updating the display in real-time.

The program's intricate design ensures that the colors keep shifting, creating a captivating visual experience. This captivating display is made possible by a timer that controls the rhythm of color changes and the DisplayRGB function, which presents the evolving colors on the screen.

AllColor and Fader Routines:

The [ALLCOLOR routine is structurally similar to the FADER routine](#).

In the [CreateRoutine function](#), ALLCOLOR initializes a palette with a single entry that represents the color black (all RGB fields set to 0).

In the [PaintRoutine function](#), ALLCOLOR creates a solid brush using PALETTEINDEX(0) and fills the entire client area with that brush, effectively coloring it black.

The [TimerRoutine](#) in ALLCOLOR brings animation to the palette by smoothly changing the color of the PALETTEENTRY and invoking AnimatePalette.

The [color transition](#) is achieved in a gradual manner. Initially, the blue value is incremented until it reaches its maximum value.

Then, the green value is incremented while progressively decrementing the blue value.

The increments and decrements of the red, green, and blue color values are determined by the iIncr variable, which is calculated during CreateRoutine based on the COLORRES value returned by GetDeviceCaps.

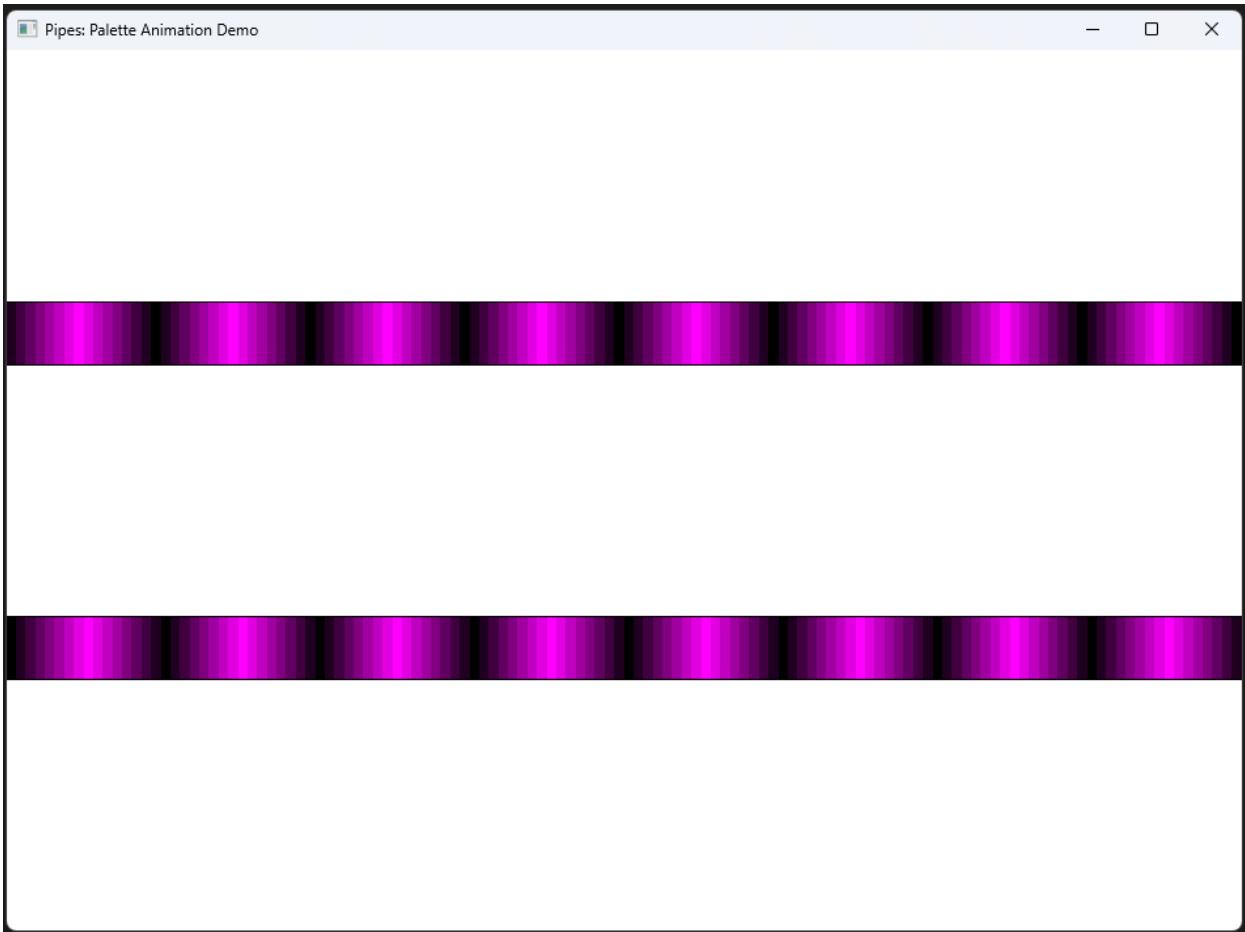
For example, if GetDeviceCaps returns 18, iIncr is set to 4, which is the lowest value necessary to achieve the full range of colors.

Additionally, ALLCOLOR displays the current RGB color value in the upper left corner of the client area. Initially added for testing purposes, this feature proved to be useful and was kept in the final implementation.

UNVEILING THE DYNAMICS OF PIPES: A PALETTE ANIMATION ODYSSEY

In the realm of engineering applications, the PIPES program stands as a testament to the power of animation in elucidating complex processes.

This palette animation demonstration, created by Charles Petzold in 1998, ingeniously visualizes fluid flow through pipes, transforming a static representation into a dynamic, easily comprehensible showcase.



1. Palette Initialization:

The heart of PIPES lies in its palette initialization process, a symphony of colors orchestrated to represent fluid movement. The `CreateRoutine` function dynamically allocates memory for a `LOGPALETTE` structure, housing 16 palette entries. These entries are strategically designed to convey the fluid's progression through the pipes. The color variations, ranging from dark to light, create a visually intuitive representation.

2. Painting the Canvas:

The `PaintRoutine` function serves as the artistic brushstroke, painting the canvas of the client area. It meticulously draws the background and interiors of two horizontal pipes. The color gradient within each pipe, determined by the palette entries, gives the illusion of fluid moving in opposite directions.

3. Crafting the Animation:

As the PIPES program comes to life, the TimerRoutine function takes center stage. Animated palette manipulation becomes the maestro, smoothly transitioning between palette entries. This dynamic shift in colors simulates the continuous flow of fluid through the pipes. The rhythmic updating of the palette captures the essence of motion, making the demonstration both informative and engaging.

4. Palette Dance Continues:

With each tick of the timer, AnimatePalette elegantly animates the color transitions. The fluid's movement is not just visualized but felt through the carefully selected palette entries. The cyclical nature of the palette updates ensures a seamless and captivating animation, making the program a compelling tool for conveying dynamic engineering processes.

5. A Palette Symphony:

In essence, PIPES is a symphony of palette entries, where each note represents a moment in the fluid's journey. The combination of color, motion, and symbolism creates a powerful educational tool. Unlike a static representation, this palette animation provides clarity and insight, turning the complexities of fluid dynamics into a visually digestible masterpiece.

6. The Grand Finale:

As the program gracefully concludes, the DestroyRoutine function takes its bow. Timers are halted, palettes are bid farewell, and memory is gracefully released. PIPES, in its elegant simplicity, leaves behind a legacy as a testament to how animation can elevate the understanding of intricate engineering processes.

The PIPES program showcases the versatility of palette animation for illustrating dynamic processes, specifically the flow of fluids in this case. This program employs a two-pronged approach: creating a visually appealing pipe layout and simulating fluid movement through animation.

Building the Pipes:

- **Palette Palette:** PIPES utilizes 16 palette entries, though fewer might suffice. These entries form a gradient, transitioning from red-green at the center to pure red and pure green at the edges. This color scheme symbolizes the direction of the fluid flow in each pipe.
- **Dual Pipes, Different Directions:** The client area features two horizontal pipes positioned above and below each other. Fluid in the top pipe flows from left to right (red to green), while the bottom pipe exhibits the opposite flow (green to red).

Animating the Flow:

- **Palette Manipulation:** The animation hinges on modifying the 16 palette entries with each timer tick. The TimerRoutine function increments an index variable that specifies the starting point for modifying the palette. This essentially shifts the gradient, creating the illusion of fluid movement.
- **Visual Cues:** To further depict the flow, the program draws rectangle segments within each pipe using specific palette indices based on the animation stage. This creates a "wave-like" effect, reinforcing the perception of fluid motion.

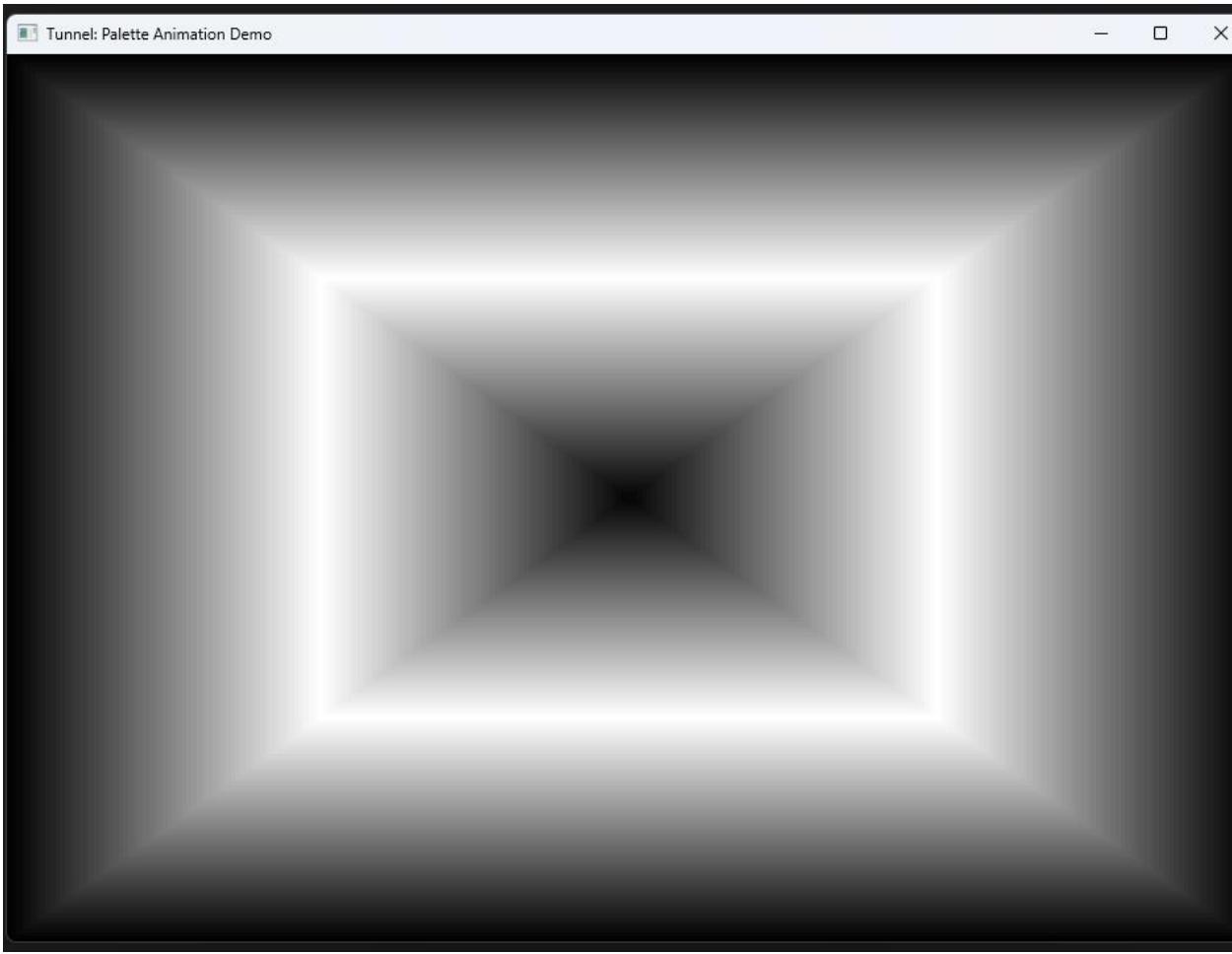
Beyond PIPES:

While PIPES demonstrates palette animation for simulating fluid flow, the technique's potential extends beyond this specific application. Engineering can benefit from palette animation in numerous ways, such as:

- **Visualizing heat transfer:** Gradual palette transitions can portray the spread of heat across a surface.
- **Depicting stress distribution:** Shifting color palettes can indicate areas of high and low stress in a mechanical structure.
- **Illustrating electrical currents:** Dynamically changing colors can represent the flow of electricity through a circuit.

By leveraging palette animation in conjunction with carefully chosen color schemes and visual elements, engineers can effectively communicate complex processes in a clear and intuitive manner.

TUNNEL PROGRAM



The TUNNEL program is a palette animation demo developed by Charles Petzold in 1998. The purpose of this program is to create a visually appealing effect that simulates traveling through a tunnel using 64 moving gray shades within the 128 palette entries. The program utilizes the Windows API and C programming language to achieve this animation.

Palette Initialization:

The program begins by initializing a palette to be used for animation. It allocates memory for a LOGPALETTE structure and sets its version and the number of palette entries. In this case, 128 palette entries are used. The program then iterates through these entries, setting the red, green, and blue components of each entry to create a gradient of gray shades. The peFlags field is set to PC_RESERVED, indicating that the color is reserved and should not be modified.

Animation Logic:

The heart of the animation lies in the TimerRoutine function, which is called periodically using a timer. This function increments the color level (iLevel) in a cyclic manner, creating the effect of smoothly transitioning through the 128 palette entries. The AnimatePalette function is then called to update the display with the new palette entries, creating the illusion of movement through the tunnel.

Painting the Tunnel:

The PaintRoutine function is responsible for painting the tunnel on the window. It uses a series of rectangles, each filled with a solid brush of a specific palette index. The palette index determines the color of the rectangle based on the current state of the palette entries. By varying the position and size of these rectangles, the program achieves the visual representation of a tunnel.

Cleanup:

Finally, the DestroyRoutine function is called when the program is exiting. It deallocates resources, including stopping the timer, deleting the palette, and freeing the memory allocated for the LOGPALETTE structure.

In summary, the TUNNEL program is a creative demonstration of palette animation, utilizing a carefully crafted palette and timer-based color transitions to create the immersive illusion of traveling through a tunnel. The use of gray shades and the cyclic animation give the program a visually captivating effect.

PACKED DIBS AND PALETTE POWER: DISPLAYING REAL-WORLD IMAGES

The next chapter of our palette exploration dives into using them with "real-world" images under 8-bit video modes. This involves delving into the world of [packed DIBs](#), a special type of memory block holding both image data and color information. To assist us, we'll utilize functions defined in the PACKEDIB files, specifically focusing on three key aspects:

On page 685 of the book, there is the packedDIB.c and .h files which are going to be custom creations to be used to customize the upcoming programs.

1. Extracting Palette Information:

These functions help us navigate the packed DIB structure and extract crucial details about the palette it uses. This includes:

- [PackedDibGetColorsUsed](#): Retrieves the number of colors actually used in the image, which might be less than the total entries in the color table.
- [PackedDibGetNumColors](#): Determines the actual number of entries in the color table.
- [PackedDibGetColorTablePtr](#): Locates the memory address of the color table within the packed DIB.
- [PackedDibGetColorTableEntry](#): Fetches a specific color entry from the table using its index.

2. Building a Logical Palette:

With the gathered information, we can construct a logical palette that Windows understands. The [PackedDibCreatePalette function](#) performs this magic, taking the packed DIB's color table and generating a corresponding logical palette object. This object encapsulates the color mapping information needed to properly display the image.

3. Leveraging the Palette for Display:

The created logical palette can then be used in conjunction with GDI functions to render the image on the screen. This allows our program to display real-world images stored as packed DIBs with their intended colors, despite the limitations of an 8-bit video mode.

Beyond the Code:

The PACKEDIB functions and palette manipulation techniques showcased here offer a powerful toolkit for working with image data in 8-bit environments. This opens doors to various applications, such as:

- Displaying resource images embedded within programs.
- Loading and showing external image files in custom formats.
- Implementing basic image viewers or editors with limited color capabilities.

In summary, these code snippets demonstrate a comprehensive set of functions to handle packed DIBs and palettes. The provided functionalities are essential for managing color information and creating logical palettes when dealing with real-world images in scenarios with limited color depth, such as 8-bit video modes. This groundwork is crucial for subsequent programs that involve displaying real-world images under such constraints.

KeyNotes:

1. Function Ordering:

Petzold organizes the functions in a "bottom-up" order, where each function relies on information obtained from earlier functions. This approach ensures that the functions are used in a logical sequence, with dependencies handled appropriately. For example, information obtained about the packed DIB's size and structure in earlier functions is crucial for subsequent functions to operate correctly.

2. Limitations of the Provided Functions:

Petzold acknowledges that the set of functions provided is not intended to be exhaustive for working with packed DIBs. He highlights that he hasn't included functions for certain operations, such as retrieving individual pixels (PackedDibGetPixel), due to their potential inefficiency and slowness resulting from nested function calls.

3. Concerns about Function Efficiency:

Petzold raises concerns about the efficiency of certain operations, indicating that functions like PackedDibGetPixel would involve numerous nested calls and could be inefficient and

slow. This hints at the potential performance drawbacks of certain approaches to working with packed DIBs.

4. Alternative Approaches:

Petzold hints at a better approach to working with packed DIBs, which he promises to describe later in the chapter. This suggests that there might be alternative and more efficient methods for certain operations, and the reader can expect further insights or recommendations as they progress through the chapter.

5. Handling OS/2-Compatible DIBs:

The note mentions that many functions require different processing for OS/2-compatible DIBs. The functions frequently check the size of the first field in the BITMAPINFO structure to determine whether it corresponds to the size of the BITMAPCOREHEADER structure. This highlights the need for conditional processing based on the type of DIB being handled.

6. PackedDibCreatePalette Function:

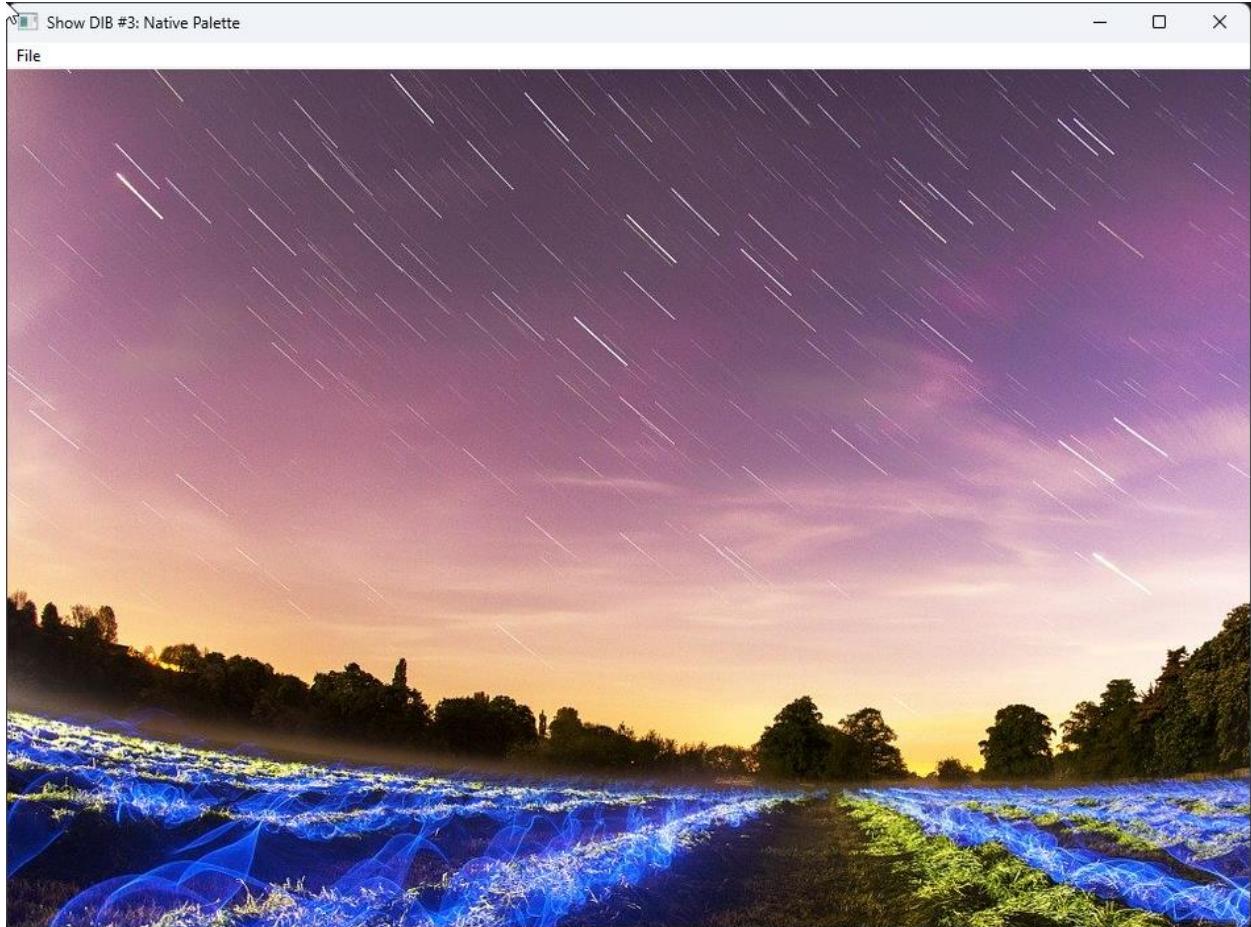
The final function, `PackedDibCreatePalette`, is highlighted as particularly interesting. This function creates a palette using the color table in the DIB. Importantly, if the DIB lacks a color table (as is the case for DIBs with 16, 24, or 32 bits per pixel), no palette is created. The palette created from the DIB color table is referred to as the DIB's native palette.

In summary, these notes provide context and guidance for using the provided functions to work with packed DIBs. On [page 685 of the book](#), the PACKEDIB.C and .H files have been introduced, containing custom code written by Charles Petzold. These `custom functions` are designed to handle packed DIBs (Device Independent Bitmaps) and are intended for customization in upcoming programs.

Therefore, what we have created are custom functions for managing packed DIBs and generating logical palettes to be utilized in subsequent programs. They address limitations, efficiency concerns, and foreshadow alternative approaches to enhance the handling of packed DIBs in the subsequent parts of the chapter.

Let's move on to the first program showdib3 which will use these function and header file packedDIB.c and .h ... I downloaded .bmp files and then loaded them and also must have is the afresx.h header file.

SHOWDIB3 PROGRAM



The SHOWDIB3 program, as outlined in the provided code snippets, is [designed to display a Device Independent Bitmap \(DIB\)](#) with its native palette in a Windows graphical user interface.

The program begins by [defining a window procedure \(WndProc\)](#) and necessary variables, including a pointer to a packed DIB (`pPackedDib`), a handle to the palette (`hPalette`), and information about the client area (`cxClient` and `cyClient`). Additionally, the program uses the `OPENFILENAME` structure to handle file opening dialogues.

Upon initialization, the [window class is registered](#), and a window is created with the title "Show DIB #3: Native Palette." The main functionality of the program involves loading a DIB from a file, creating a logical palette from its color table, and displaying the DIB using the native palette.

When the user selects "Open" from the File menu, the program triggers the File Open dialog, allowing the user to choose a bitmap file. The selected file is then loaded as a packed DIB into memory using the custom PackedDibLoad function. Subsequently, the program creates a logical palette from the DIB's color table using the PackedDibCreatePalette function.

The painting process involves setting up the device context, selecting the palette, and using SetDIBitsToDevice to display the DIB. If a palette is associated with the DIB, the program ensures proper palette realization and updates the colors. The program handles palette changes using messages like WM_QUERYNEWPALETTE and WM_PALETTECHANGED.

In the SHOWDIB3 program, the main thing to watch out for is the handling of logical palettes associated with the loaded DIB. The program goes beyond simply loading and displaying a DIB; it takes into account the creation and utilization of a logical palette. Here are key points to note:

Palette Handling:

The program creates a logical palette (hPalette) from the color table of the loaded DIB using the PackedDibCreatePalette function. This step is crucial for accurate color representation, especially when dealing with images that have a limited color palette.

Palette realization is managed through messages such as WM_QUERYNEWPALETTE and WM_PALETTECHANGED. This ensures that the program responds appropriately when the system palette changes.

Palette-Dependent Display:

During painting (WM_PAINT), the program selects the logical palette into the device context and uses RealizePalette to ensure proper color mapping. This step is essential for displaying the DIB with its native palette, preserving the intended colors of the image.

Dynamic Palette Update:

The program dynamically updates the palette in response to changes using the WM_PALETTECHANGED message. When the program loses focus and regains it, the palette is re-realized to reflect any modifications in the system palette.

Difference from Normal DIB Handling:

Unlike a normal program that loads DIBs without considering color palettes, SHOWDIB3 pays special attention to palette creation and utilization. This is particularly important for images with a color table, as it ensures that the displayed colors closely match the original ones.

In summary, the unique aspect of SHOWDIB3 is its focus on handling logical palettes associated with DIBs. It goes beyond the basics of loading and displaying images, incorporating palette management to provide accurate and vibrant color representation. This is especially relevant when dealing with images that rely on a limited set of colors.

Let's explain this a bit more...

1. Native Palette Focus:

SHOWDIB3 prioritizes displaying the image using its original palette embedded in the DIB file. This ensures accurate color representation, unlike a standard program that might rely on the default system palette.

2. Logical Palette Creation:

It actively creates a logical palette based on the DIB's color table. This mapping allows the program to translate the image's colors to the available hardware palette on the system, minimizing color errors.

3. Palette Management:

SHOWDIB3 properly handles the logical palette throughout its execution. It selects and realizes the palette when necessary, responds to system palette changes, and updates colors accordingly. This ensures color consistency across different events.

4. Packed DIBs:

It likely utilizes functions specific to "packed DIBs," a special format for storing both image data and color information efficiently. This might differ from programs dealing with standard uncompressed DIBs.

5. Memory Management:

SHOWDIB3 carefully frees memory allocated for the DIB data and the logical palette when they're no longer needed. This helps prevent memory leaks, a common issue in image-handling programs.

In essence, SHOWDIB3 goes beyond simply displaying a DIB. It focuses on accurately representing the image's colors by actively managing its native palette and ensuring a smooth integration with the system's color capabilities.

The **SHOWDIB3** program pays close attention to memory management to ensure efficient resource usage. The window procedure carefully handles memory allocation and deallocation, preventing memory leaks and conflicts. It stores the packed DIB pointer and palette as static variables, ensuring their accessibility throughout the program's execution.

To accurately represent colors, **SHOWDIB3** gives priority to using the DIB's native palette. When opening a file, it creates a logical palette from the DIB's color table using the PackedDibCreatePalette function. During the WM_PAINT message, it selects and realizes the palette before drawing the image using SetDIBitsToDevice. This ensures proper color mapping and faithful color reproduction.

For **DIBs without color tables (such as 16-bit, 24-bit, and 32-bit DIBs)**, SHOWDIB3 recognizes that displaying them without a palette in 8-bit video modes would result in a limited color range, typically limited to the standard 20 colors.

To address this limitation, the program offers two potential solutions. One approach is to create an all-purpose palette that can accommodate a wide variety of images. Although relatively simple, this method may not provide the highest level of color accuracy for specific images.

The other approach involves dynamic color analysis, where the pixel data within the DIB is analyzed to determine the most suitable colors for display. This technique requires more computational effort but can potentially offer superior color fidelity.

The choice between the two approaches depends on the specific requirements of the application and the desired level of color accuracy.

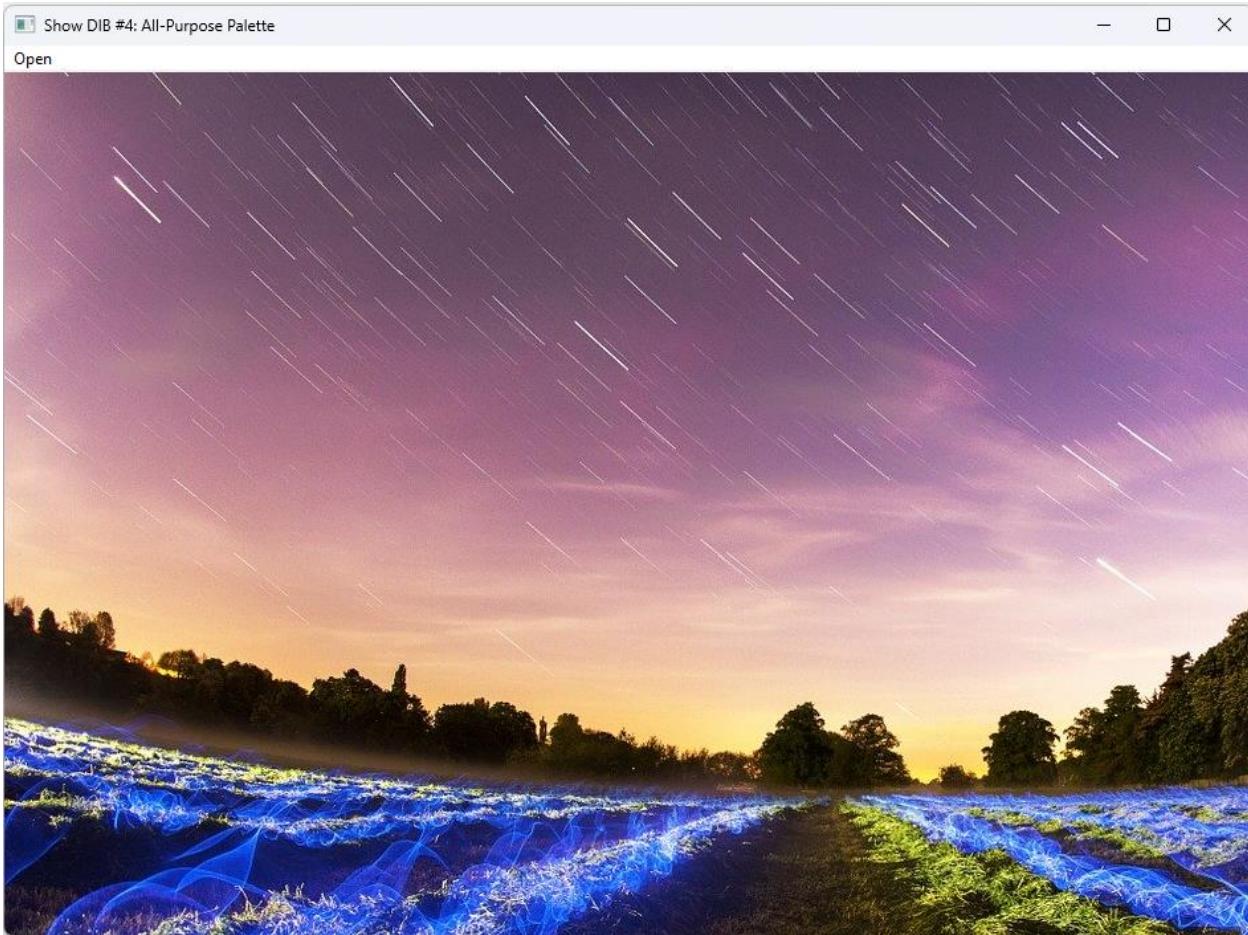
For general-purpose image viewing, an all-purpose palette might be sufficient.

However, for professional image editing or tasks that demand precise color representation, dynamic color analysis could be the preferred option.

The author hints at the possibility of exploring the dynamic color analysis technique in a later chapter, suggesting a more advanced approach to handling DIBs without color tables.

SHOWDIB4 PROGRAM

The SHOWDIB4 program introduces an innovative approach to palette management compared to its predecessor, SHOWDIB3. The primary focus is on constructing an "all-purpose" palette that it utilizes for displaying all DIBs loaded into the program. Let's delve into the notable features and changes in depth:



All-Purpose Palette Creation:

SHOWDIB4 introduces the function `CreateAllPurposePalette` that dynamically generates an all-purpose palette suitable for a wide variety of images. This palette consists of 247 entries, carefully chosen to cover a range of colors. The palette creation process is meticulously designed to avoid duplicating or conflicting with the standard 20 colors.

Palette Handling Enhancement:

Unlike SHOWDIB3, which relies on creating a palette specific to each DIB, SHOWDIB4 takes a more generic approach by using a single, versatile palette for all images. This is a departure from the native palette focus in SHOWDIB3. The palette created in `CreateAllPurposePalette` covers a diverse spectrum of colors, ensuring compatibility with various image types.

Simplified Palette Management:

With the adoption of an all-purpose palette, the program simplifies palette management. There's no need to create and update individual palettes for each loaded DIB. This streamlining enhances efficiency and reduces the complexity associated with palette realization and handling.

Unified Color Representation:

SHOWDIB4 aims for a consistent and unified color representation across different images. By using the all-purpose palette, it ensures that the displayed colors maintain a certain level of consistency, regardless of the specific color characteristics of each loaded DIB.

Palette Initialization:

The creation of the all-purpose palette occurs during the WM_CREATE message handling. This ensures that the palette is ready for use as soon as the program starts, and it remains constant throughout its execution.

Menu and UI Similarities:

The structure of the program, including menu options and the user interface, remains largely similar to SHOWDIB3. This continuity allows users familiar with the previous version to adapt seamlessly to the new palette management strategy.

In essence, [SHOWDIB4 distinguishes itself by adopting a unified approach to palette management](#) with the creation of an all-purpose palette. This shift simplifies the handling of color information, making the program more versatile and suitable for displaying a diverse range of images while maintaining a consistent and well-defined color palette.

In the [process of handling the WM_CREATE message](#), SHOWDIB4 efficiently manages its palette by calling the CreateAllPurposePalette function. This palette remains persistently available throughout the program's execution and is appropriately destroyed when handling the WM_DESTROY message.

The assurance of the palette's continuous existence eliminates the need for constant checks during the processing of WM_PAINT, WM_QUERYNEWPALETTE, or WM_PALETTECHANGED messages.

The [CreateAllPurposePalette function](#) is noteworthy for generating a logical palette with 247 entries, surpassing the typical 236 entries in the system palette. Despite this excess, 15

entries are intentionally duplicated or mapped to colors within the standard 20 reserved colors, providing a buffer for flexibility and convenience.

The palette creation process begins by establishing 31 gray shades, strategically aligning with certain entries in the standard 20 reserved colors. Subsequently, the function generates colors with various combinations of red, green, and blue values, resulting in a total of 216 colors.

While eight of these duplicate standard 20 colors and another four replicate previously calculated gray shades, Windows intelligently avoids placing duplicate entries in the system palette if the peFlags field of the PALETTEENTRY structure is set to 0.

It's important to note that, for practical applications not concerned with optimizing palettes for 16-bit, 24-bit, or 32-bit DIBs, utilizing the DIB color table for displaying 8-bit DIBs might be a preferred approach.

However, SHOWDIB4 deviates from this convention, using its all-purpose palette for all image types. This deliberate choice aligns with SHOWDIB4's role as a demonstration program, allowing users to compare its display of 8-bit DIBs with that of SHOWDIB3.

As an interesting experiment, modifying the CreateAllPurposePalette function in SHOWDIB4, such as by reducing the logical palette size, reveals that, when a palette is selected into a device context, Windows exclusively utilizes the colors from the selected palette, excluding any colors from the standard 20-color palette. This behavior underscores the level of control and precision achievable in color rendering when manipulating logical palettes in Windows programming.

SHOWDIB5 PROGRAM

SHOWDIB5 introduces the utilization of the halftone palette provided by the Windows API, showcasing its application in conjunction with the bitmap stretching mode known as HALFTONE. Let's delve into the notable aspects of this program:

Halftone Palette Integration:

In the WM_CREATE message processing, SHOWDIB5 stands out by creating the halftone palette using the CreateHalftonePalette function. This palette is designed to be versatile

and adaptable for various images. The program takes advantage of this palette throughout its execution.

Dynamic Stretching with HALFTONE Mode:

One of the key differences in SHOWDIB5 lies in its approach to displaying images. Instead of using the traditional SetDIBitsToDevice method seen in previous examples, SHOWDIB5 employs the StretchDIBits function. This allows for dynamic stretching of the image, offering better control over the scaling process.

Halftone Stretch Mode Setup:

Prior to performing the stretching operation, SHOWDIB5 sets the stretching mode to HALFTONE using SetStretchBltMode. This mode enhances the quality of the stretched image by employing a halftone pattern. The SetBrushOrgEx function is also used to set the brush origin to (0,0), ensuring proper alignment during the stretching process.

Palette Handling in Stretching:

During the WM_PAINT message processing, the program selects and realizes the halftone palette before executing the StretchDIBits function. This palette management ensures that the stretched image maintains a high-quality appearance, with colors accurately represented.

Comparison with SHOWDIB4:

SHOWDIB5's use of the halftone palette showcases an alternative approach compared to SHOWDIB4, which relies on a custom all-purpose palette. The choice between these palettes can impact how images are displayed, especially in scenarios where dynamic stretching is involved.

Palette Update on Events:

Similar to previous examples, SHOWDIB5 responds to palette-related messages, such as WM_QUERYNEWPALETTE and WM_PALETTECHANGED. These messages trigger palette updates, ensuring consistent and accurate color representation, especially when the program loses or gains focus.

The SHOWDIB5 program, akin to SHOWDIB4, diverges from utilizing the color table within the DIB, opting instead for a palette suitable for a broad spectrum of images.

However, in contrast to the custom all-purpose palette of SHOWDIB4, SHOWDIB5 leverages the logical palette provided by Windows, acquired through the CreateHalftonePalette function.

The **halftone palette generated** by CreateHalftonePalette, while not significantly more intricate than the one crafted by CreateAllPurposePalette in SHOWDIB4, proves to be a pivotal component when coupled with specific operations.

Notably, when the program sets the stretching mode using SetStretchBltMode to HALFTONE and adjusts the brush origin with SetBrushOrgEx before displaying the DIB through StretchDIBits, the outcomes surpass expectations.

This strategic combination of operations, particularly the application of the HALFTONE stretching mode, yields remarkable improvements in color accuracy, especially for flesh tones.

The **halftone palette, when employed in conjunction with the specified stretching mode**, introduces a form of dithering pattern. This pattern, derived from the colors of the halftone palette, enhances the approximation of the original image's colors on 8-bit video boards. The result is a more faithful representation of subtle color variations, particularly crucial in capturing the nuances of skin tones.

However, it's crucial to acknowledge that this enhanced **color fidelity comes at a cost**—increased processing time. The utilization of a dithering pattern for improved color approximation involves additional computational overhead.

This trade-off highlights the balance between achieving optimal visual results and considering the efficiency of image processing, a common consideration in graphics programming.

In essence, SHOWDIB5 showcases the significance of thoughtful palette management and strategic utilization of Windows-provided palettes in conjunction with advanced stretching modes.

The program's ability to enhance color accuracy, particularly for challenging elements like flesh tones, exemplifies the nuanced decisions involved in achieving optimal visual rendering in graphics programming.

It demonstrates the integration of the halftone palette for enhanced image stretching, providing a different perspective on color rendering and image quality compared to previous examples.

The dynamic nature of stretching, coupled with thoughtful palette management, makes this program a valuable exploration of advanced techniques in DIB handling.

SHOWDIB6 PROGRAM

SHOWDIB6 is a Windows program that exemplifies the utilization of palette indices when displaying an 8-bit Device-Independent Bitmap (DIB). It employs the DIB_PAL_COLORS flag in functions like SetDIBitsToDevice and takes advantage of palette indices, offering a performance boost by sidestepping the need for a nearest-color search.

Palette Indices and DIB_PAL_COLORS:

The DIB_PAL_COLORS flag allows the color table in the BITMAPINFO structure to be interpreted as 16-bit indices into a logical palette, rather than RGB color values.

This logical palette is the one currently selected in the device context given as the first argument to the function. In the case of SHOWDIB6, this logical palette is created from the DIB's color table.

When DIB_PAL_COLORS is used, the color table in the DIB is assumed to contain indices pointing to entries in the logical palette.

This eliminates the need for Windows to perform a nearest-color search, which is typically required when displaying images in an 8-bit video mode. Instead, the program can directly map DIB pixel values to device pixels using the palette indices.

Program Workflow:

Loading and Displaying DIB:

- The program prompts the user to open a bitmap file (BMP).
- It loads the DIB into memory and creates a logical palette from the DIB color table.
- The program then replaces the RGB color values in the DIB color table with WORD indices starting from 0.

Displaying DIB with Palette Indices:

- During painting (WM_PAINT), the logical palette is selected into the device context.
- The DIB is then displayed using SetDIBitsToDevice with the DIB_PAL_COLORS flag.
- This process takes advantage of the palette indices, avoiding the need for a nearest-color search.

Palette Management:

- The program handles palette-related messages such as WM_QUERYNEWPALETTE and WM_PALETTECHANGED to ensure proper palette realization and updates.

Optimization and Considerations:

- **Performance Improvement:** The use of DIB_PAL_COLORS provides a performance improvement, especially when displaying 8-bit DIBs in an 8-bit video mode. By utilizing palette indices, the program bypasses the nearest-color search, potentially resulting in faster image rendering.
- **Limitation with Palette Creation:** This technique is feasible when creating a palette directly from the color table of the DIB. If an all-purpose palette were used, a manual nearest-color search would be necessary, making the approach less practical.
- **Caution on Saving and Clipboard Usage:** If palette indices are employed, it's crucial to replace the color table in the DIB before saving it to disk. Additionally, DIBs containing palette indices should be handled carefully, and it's recommended to restore RGB color values before saving to ensure compatibility.

Importance of DIB_PAL_COLORS:

- It signifies to Windows that the color table contains palette indices, not RGB values.
- This optimizes performance by eliminating nearest-color searches, as Windows directly maps indices to device colors.

Key Considerations and Best Practices:

- Use palette indices primarily when working with DIBs that have limited color palettes (e.g., 8-bit DIBs).
- Create the palette directly from the DIB's color table to avoid manual nearest-color searches.
- Revert to RGB color values before saving DIBs to disk or placing them on the clipboard to ensure compatibility.
- Consider using palette indices only for display purposes and reverting to RGB colors afterward for consistency.

Additional Insights from Feedback:

- Provide a visual representation of the DIB structure and palette concept for better understanding.
- Discuss potential drawbacks of using palette indices, such as reduced color accuracy in certain scenarios.
- Explore alternative color management techniques, such as halftoning, for handling images with large color palettes.
- Offer guidance on choosing between RGB values and palette indices based on image characteristics and display requirements.

SHOWDIB7 PROGRAM

SHOWDIB7, much like SHOWDIB6, is a Windows program that focuses on the utilization of palettes in conjunction with Device-Independent Bitmaps (DIBs). However, SHOWDIB7 introduces a new approach by [demonstrating how to convert a DIB into a Device-Dependent Bitmap \(DDB\)](#) using the CreateDIBitmap function. This alteration results in a change in the method of displaying the image.

Conversion to DDB:

In SHOWDIB7, the primary deviation from SHOWDIB6 lies in the conversion of the DIB to a DDB. Instead of directly using SetDIBitsToDevice, SHOWDIB7 employs CreateDIBitmap to create a DDB from the DIB. This transition allows for more flexibility in handling and displaying the bitmap.

Bitmap Conversion Process:

Loading and Displaying:

- The user opens a bitmap file, similar to SHOWDIB6.
- The program loads the DIB into memory and creates a logical palette from the DIB's color table.
- It then creates a DDB using CreateDIBitmap, initializing it with information from the packed DIB.

Palette Management:

- The logical palette is created and selected into the device context, as in SHOWDIB6.
- Palette-related messages are handled for proper realization and updates.

Displaying the DDB:

- During painting (WM_PAINT), the DDB is displayed using BitBlt.
- The DDB is compatible with the device context, allowing efficient copying to the screen.

Advantages of DDB Conversion:

- **Flexibility in Displaying:** Creating a DDB provides greater flexibility in displaying the image. BitBlt allows for efficient copying of the DDB to the screen, and it opens up possibilities for additional manipulations.
- **Compatibility:** DDBs are compatible with standard GDI functions, offering a seamless integration into the Windows graphical environment.
- **Resource Management:** The DIB memory can be freed after creating the DDB, potentially optimizing resource usage.

Changes from SHOWDIB6:

The transition from SetDIBitsToDevice to CreateDIBitmap represents a shift in strategy. Instead of directly rendering the DIB on the screen, SHOWDIB7 takes an intermediary step by creating a DDB. This not only provides more options for handling and displaying the image but also aligns with the standard GDI practices for working with bitmaps.

Nearest-Color Searches:

SHOWDIB6 replaces DIB colors with palette indices to optimize nearest-color searches.

SHOWDIB7 uses DIB_RGB_COLORS when creating the DDB, indicating that colors are already in RGB format, eliminating the need for index-based optimization.

Key Considerations:

- Converting to a DDB can improve performance for certain operations but might not be necessary in all cases.
- Handling palettes appropriately is crucial for accurate color display, especially when working with DDBs.
- Memory management is important to avoid leaks and ensure efficient resource usage.
- The choice between using palette indices or RGB colors depends on the specific requirements of the application and the target display device.

Key Points Not Yet Discussed:

Importance of Palette Selection and Realization:

- Selecting and realizing the palette before creating the DDB ensures accurate color mapping.
- This is crucial because DDBs directly use device colors, and the palette dictates how DIB colors are translated.
- Failing to do so could result in incorrect or limited colors in the resulting DDB.

Memory Management Considerations:

- DIBs can be freed after DDB creation, while DDBs must be retained for later display.
- This difference stems from DIBs being **standalone data structures**, while DDBs are GDI objects managed by Windows.

Clipboard Handling Best Practices:

- Copying DIBs to the clipboard is generally preferred for compatibility and flexibility.
- If DDB copying is necessary, selecting and realizing the palette ensures proper conversion to a DIB based on the current system palette.

Understanding GetObject:

- This function retrieves information about a GDI object, including its width and height in this case.
- It's essential for determining the size of the bitmap when performing operations like BitBlt.

Additional Insights:

Trade-offs Between DIBs and DDBs:

- DIBs offer portability and flexibility, while DDBs can be faster for certain operations and compatible with DDB-specific functions.
- The choice depends on application requirements and performance needs.

Palette Implications for Color Accuracy:

- Using the correct palette throughout DIB loading, DDB creation, and display is critical for ensuring accurate color representation.
- This is especially important when working with limited-color devices or images with custom palettes.

Importance of BitBlt for DDB Display:

This function efficiently copies bitmaps between device contexts, essential for displaying DDBs on the screen or other devices.

Summary:

SHOWDIB7 demonstrates DIB-to-DDB conversion, palette handling for accurate color mapping, memory management considerations for different bitmap types, and best practices for clipboard interactions.

Understanding these concepts is vital for working effectively with bitmaps in Windows programming contexts.

SHOWDIB8 PROGRAM

DIB Sections: Unlocking Performance and Direct Pixel Access:

Memory-Mapped Bitmaps: DIB sections are distinctive in their memory management. They're memory-mapped GDI bitmap objects, meaning their pixel data resides in a specified memory region, providing direct access for efficient manipulations. This eliminates the need for intermediate buffers or data copying, potentially boosting performance.

Creation and Handling: SHOWDIB8 employs CreateDIBSection to generate a DIB section from the loaded DIB. It obtains a pointer to the pixel data using pBits and directly accesses it for operations like copying. This contrasts with packed DIBs, where memory management is manual, and DDBs, where pixel data is managed internally by Windows.

Palette Management: While DIB sections offer direct pixel access, they still require appropriate palette handling to ensure accurate color representation. SHOWDIB8 demonstrates this by selecting and realizing the palette associated with the DIB, similar to previous programs.

Key Considerations When Choosing Bitmap Representations:

Performance vs. Compatibility: DIB sections often excel in performance for operations involving frequent pixel access, such as image processing tasks. However, packed DIBs generally offer wider compatibility across different platforms and file formats. DDBs, while not directly addressed in SHOWDIB8, can be advantageous for certain device-specific operations.

Clipboard Operations: When working with the clipboard, packed DIBs are generally favored due to their broader compatibility with other applications.

Unlocking the Potential of DIB Sections:

Image Processing Optimizations: DIB sections enable direct pixel manipulation without intermediate copies, potentially accelerating image processing tasks like filtering, transformations, and adjustments.

Custom Memory Management: For specialized memory needs or optimization strategies, DIB sections offer greater flexibility in memory management compared to packed DIBs or DDBs.

Let's discuss this program in-depth:

SHOWDIB8 is a Windows program that demonstrates the usage of palettes with Device-Independent Bitmap (DIB) sections. It explores the conversion of a DIB to a DIB section, highlighting the benefits of this approach in handling bitmap data. Let's delve into the key aspects of SHOWDIB8:

DIB to DIB Section Conversion:

File Loading:

- Similar to previous programs, the user can open a bitmap file using the File Open dialog.
- The packed DIB is loaded into memory, and the logical palette is created from the DIB's color table.

DIB Section Creation:

- A DIB section is created using the CreateDIBSection function.
- The function returns a handle to the newly created DIB section, and a pointer (pBits) to the memory location where the DIB bits are stored.

Bit Copying:

- The bits from the packed DIB are copied to the DIB section using CopyMemory.
- This step ensures that the DIB section contains the pixel data from the original DIB.

Palette Management:

- A logical palette is created from the original DIB.
- The palette is associated with the device context, allowing for proper color realization during painting.

Displaying the DIB Section:

- During the painting process (WM_PAINT), the DIB section is displayed on the window using BitBlt.
- Palette management ensures that colors are accurately represented.

Advantages of DIB Section:

- **Direct Memory Access:** DIB sections provide direct access to the bitmap's pixel data through a pointer (pBits), facilitating efficient manipulation and updates.
- **Palette Compatibility:** The logical palette associated with the DIB section ensures proper color mapping and representation.

Changes from Previous Versions:

DIB Section Usage: Unlike SHOWDIB6 and SHOWDIB7, which focused on DDBs, SHOWDIB8 specifically emphasizes DIB sections. This choice is driven by the benefits offered by DIB sections, especially in scenarios where direct access to pixel data is crucial.

Memory Management: SHOWDIB8 manages memory more directly, leveraging the capabilities of DIB sections for pixel data storage and manipulation.

Overall Significance:

SHOWDIB8 showcases an advanced approach to handling bitmaps by utilizing DIB sections.

The program emphasizes the benefits of direct memory access and efficient color representation, offering insights into an alternative method of working with bitmap data in Windows graphics programming.

This versatility is particularly valuable in scenarios where low-level control over pixel data is essential.

SHOWDIB8 showcases DIB sections as a powerful bitmap representation for Windows programming, balancing performance with flexibility.

Understanding the strengths and trade-offs of DIB sections, packed DIBs, and DDBs empowers developers to make informed choices based on specific application requirements and performance goals.

By effectively leveraging DIB sections, developers can optimize image handling and processing tasks in their Windows applications.

FINAL WORDS ABOUT THE PROGRAMS 7&8 ABOVE:

The WM_PAINT processing in both SHOWDIB7 and SHOWDIB8 serves the common purpose of displaying the bitmap stored in the static variables hBitmap and hPalette onto the client area. While the overall structure is similar, the crucial distinction lies in how these programs handle the File Open menu command and the subsequent processing of the loaded bitmap.

Common WM_PAINT Processing:

Retaining Bitmap and Palette Handles: Both programs maintain static variables for the bitmap handle (hBitmap) and the logical palette handle (hPalette).

Palette Management: The logical palette is selected into the device context and realized, ensuring accurate color representation during painting.

Bitmap Display:

- The dimensions of the bitmap are obtained using the GetObject function.
- A memory device context is created, and the bitmap is selected into it.
- The BitBlt function is used to display the bitmap on the client area.

Distinctions in File Open Processing:

SHOWDIB7:

Palette Handling: After obtaining the packed DIB pointer, SHOWDIB7 selects the palette into a video device context and realizes it.

The [CreateDIBitmap function](#) is then called, requiring the palette to be associated with the device context.

SHOWDIB8:

Palette Independence:

- SHOWDIB8, after obtaining the packed DIB pointer, directly calls CreateDIBSection without the need to select the palette into a device context.
- The first argument to CreateDIBSection (device context handle) is used only if the DIB_PAL_COLORS flag is employed.

Pixel Copying:

- The pixel bits are copied from the packed DIB to the memory location returned by CreateDIBSection.
- PackedDibCreatePalette is called to create a palette, leveraging information from the packed DIB.

Significance of SHOWDIB8 Approach:

Direct Bitmap Section Creation:

- SHOWDIB8 directly utilizes CreateDIBSection, bypassing the need to convert the DIB to a device-dependent format.
- The program gains efficiency by avoiding unnecessary conversions.

Palette Independence:

- The absence of palette selection in SHOWDIB8 signifies that CreateDIBSection inherently deals with DIBs without relying on device-dependent palettes.

Efficient Memory Handling:

- Pixel bits are directly copied to the memory location created by CreateDIBSection, streamlining the process.

Dynamic Palette Creation:

- Palette creation in SHOWDIB8 is done through PackedDibCreatePalette, allowing flexibility in handling palette information.

Conclusion:

SHOWDIB7 and SHOWDIB8 share commonalities in displaying bitmaps but diverge in their approaches to handling palettes during the loading of a new bitmap.

SHOWDIB8's strategy, centered around CreateDIBSection and dynamic palette creation, showcases a more direct and efficient method for dealing with DIBs, particularly in scenarios where palette independence and direct memory access are prioritized.

BUILDING A ROBUST DIB LIBRARY IN C: BEYOND PACKED DIBS

The journey through GDI bitmaps, DIBs, and the Windows Palette Manager has equipped us with the knowledge to construct a robust library for working with bitmaps in C. While the previous approach using packed DIBs offered simplicity, its performance bottlenecks, particularly for "get pixel" and "set pixel" operations, necessitate a more efficient solution.

This is where the DIB section shines. By abandoning the packed DIB model and leveraging DIB sections, we can achieve:

- **Performance Boost:** Direct pixel access eliminates intermediate data copies, significantly speeding up image processing tasks.
- **Flexibility:** Similar to packed DIBs, DIB sections allow manipulation of bitmap pixels in a device-independent manner.

- **Windows NT Optimization:** DIB sections are particularly efficient under Windows NT, making them a preferable choice for modern applications.

Here's how we can build a C library around this approach:

1. Defining the HDIB Handle:

```
typedef void * HDIB;
```

This simple declaration defines an opaque handle of type HDIB that acts as a reference to a DIB section object. The specifics of its internal structure remain hidden, adhering to the "none of your business!" principle.

2. The DIBInfo Structure:

```
typedef struct {
    HDC hdc;           // Device context associated with the DIB section
    HBITMAP hBitmap;  // Handle to the DIB section bitmap object
    HBITMAP hOldBitmap; // Previous bitmap handle saved for restoration
    int cx;            // Bitmap width
    int cy;            // Bitmap height
    int cBitsPerPixel; // Color depth in bits per pixel
    int cBytesPerRow;  // Number of bytes per row in the bitmap memory
    BYTE *pBits;       // Pointer to the raw bitmap data
} DIBInfo;
```

This structure encapsulates relevant information about the DIB section, including the associated device context, bitmap handle, dimensions, color depth, bytes per row, and most importantly, a direct pointer (pBits) to the raw bitmap data. This enables efficient pixel access through functions like DibGetPixel and DibSetPixel.

3. Library Functions:

Here are some example functions for manipulating DIB sections:

DibCreate(int cx, int cy, int cBitsPerPixel): Creates a new DIB section with the specified width, height, and color depth. This function allocates memory for the bitmap data and returns an HDIB handle referencing the DIBInfo structure.

DibGetPixel(HDIB hdib, int x, int y): Uses the hdib handle to access the DIBInfo structure and retrieves the pixel value at the specified coordinates (x, y) within the bitmap data (pointed to by pBits). The function can then return the color value based on the color depth format.

DibSetPixel(HDIB hdib, int x, int y, DWORD color): Similar to DibGetPixel, this function sets the pixel at the specified coordinates to the provided color value, manipulating the raw bitmap data directly through the pBits pointer.

DibDestroy(HDIB hdib): Release resources associated with the DIB section, freeing memory and deleting the bitmap object. This should be called when the DIB is no longer needed.

4. Advantages of this Approach:

Faster Pixel Access: Direct manipulation of raw bitmap data through pBits eliminates expensive data copies and context switching, significantly speeding up pixel operations compared to packed DIBs.

Simplified Memory Management: The library functions handle memory allocation and deallocation for the DIB section, reducing the burden on the programmer.

Flexibility and Control: The internal structure of the DIBInfo can be adapted to include additional information or functionalities as needed.

5. Conclusion:

By leveraging DIB sections and encapsulating them within a well-defined C library, we can achieve efficient and flexible bitmap manipulation in Windows applications. This approach balances performance advantages with clear and structured function interfaces, empowering developers to write robust image processing and graphics applications.

Note: While the code examples are presented in code boxes, remember that this is a conceptual explanation. The actual implementation of the library functions would involve detailed error checking, memory management considerations, and platform-specific adaptations.

Additional Considerations for Effective DIB Handling:

File-Mapping Fields: Out of Scope: The dshSection and dsOffset fields within DIBSECTION, primarily used for file-mapping objects, lie beyond the realm of DIBHELP and can be safely disregarded in this context.

Structure Ordering: Macro-Friendly: The strategic placement of ppRow at the structure's forefront simplifies macro definitions, enabling convenient access to row pointers. It's like positioning the most frequently used tool within easy reach.

Error Prevention with iSignature: By diligently checking the iSignature field, DIBHELP functions can thwart potential errors stemming from invalid structure pointers. This vigilance serves as a watchful guard against mishaps.

In essence, DIBSTRUCT serves as a meticulously crafted blueprint, guiding developers through the intricate world of DIB manipulation within the DIBHELP library. By mastering its intricacies, you'll unlock the power to create, modify, and unleash the full potential of device-independent bitmaps in your Windows applications.

DIBSTRUCT: THE CORNERSTONE OF DIBHELP

```
typedef struct {
    PBYTE* ppRow;           // array of row pointers
    int iSignature;         // = "Dib "
    HBITMAP hBitmap;        // handle returned from CreateDIBSection
    BYTE* pBits;            // pointer to bitmap bits
    DIBSECTION ds;          // DIBSECTION structure
    int iRShift[3];          // right-shift values for color masks
    int iLShift[3];          // left-shift values for color masks
} DIBSTRUCT, *PDIBSTRUCT;
```

Accelerated Pixel Access with ppRow:

This array of pointers acts as a fast lane to individual pixel rows, enabling direct manipulation without complex calculations. It's like having an address book for every pixel row, streamlining image processing operations.

An array of pointers, each pointing to a row of pixels in the DIB. These pointers facilitate a faster method for accessing DIB pixel bits, with the rows arranged from the top row first.

Integrity Check with iSignature:

This field safeguards against erroneous pointer usage by confirming the validity of DIBSTRUCT instances. Think of it as a security guard ensuring you're working with the correct structure.

A signature check field set to the binary equivalent of the text string "Dib." This serves as a validity check for pointers to the structure.

Handle to a DIB's Potential:

hBitmap This handle, bestowed by CreateDIBSection, unlocks a DIB's capabilities for diverse bitmap operations. It's like the key to a treasure chest filled with graphical possibilities.

The bitmap handle returned from the CreateDIBSection function. It behaves similarly to handles of GDI bitmap objects and allows for operations on the DIB.

Unlocking Pixel Data with pBits:

This pointer grants unfettered access to the raw pixel data within the DIB section, empowering efficient manipulations. It's like holding a direct line to the heart of a DIB's visual information.

A pointer to the bitmap bits. This memory block is controlled by the operating system, and applications have access to it. It is automatically freed when the bitmap handle is deleted.

DIBSECTION:

A Wealth of Information: This embedded structure holds crucial details about the DIB's anatomy, including its dimensions, color depth, and compression scheme. It's like a detailed blueprint revealing a DIB's inner workings.

A DIBSECTION structure obtained by passing the bitmap handle to the GetObject function. It contains information about the DIB, such as dimensions, color masks, and compression. This provides a convenient way to retrieve detailed information about the DIB.

Mastering Color Masks with iRShift and iLShift:

These arrays hold the keys to unlocking color information in 16-bit and 32-bit DIBs with BI_BITFIELDS compression. They're like skilled decoders, translating pixel values into their true RGB colors.

Arrays storing right and left shift values for color masks in 16-bit and 32-bit DIBs. These values are used to manipulate and extract color components from the DIB pixels.

DIBHELP.C

The DIBHELP.C file presented in Figure 16–20 provides a set of functions for handling Device-Independent Bitmaps (DIBs) in a Windows environment. Let's delve into the details of these information functions and understand their workings.

1. DIBSTRUCT Structure:

The file begins by defining a structure called DIBSTRUCT, which encapsulates essential information about a DIB. It includes fields such as pointers to rows of pixels, a signature check, a bitmap handle, a pointer to bitmap bits, a DIBSECTION structure, and arrays for right and left shift values for color masks.

2. DibIsValid Function:

The DibIsValid function checks the validity of a given HDIB (Handle to a DIB) by ensuring that it is not NULL, the memory it points to is readable, and the signature matches an expected value.

3. DibBitmapHandle Function:

The DibBitmapHandle function returns the handle to the DIB section bitmap object from the provided HDIB. It first checks the validity of the HDIB using the DibIsValid function.

4. DibWidth, DibHeight, DibBitCount Functions:

These functions, namely DibWidth, DibHeight, and DibBitCount, provide information about the dimensions and bit count of the DIB, respectively. They retrieve values from the DIBSECTION structure within the DIBSTRUCT.

5. DibRowLength Function:

The DibRowLength function calculates and returns the number of bytes per row of pixels in the DIB, considering padding for alignment.

6. DibNumColors, DibMask, DibRShift, DibLShift Functions:

These functions provide information about the color-related aspects of the DIB, such as the number of colors, color masks, and shift values. They access the relevant fields in the DIBSECTION structure.

7. DibCompression Function:

The DibCompression function retrieves the value of the biCompression field from the BITMAPINFOHEADER structure within the DIBSECTION.

8. DibIsAddressable Function:

The DibIsAddressable function checks whether the DIB is compressed or not by examining the compression type in the DIBSECTION.

9. DibInfoHeaderSize, DibMaskSize, DibColorSize, DibInfoSize, DibBitsSize, DibTotalSize Functions:

These functions calculate the sizes of various components of the DIB section as they would appear in a packed DIB. They aid in converting the DIB section to a packed DIB and saving DIB files.

10. DibInfoHeaderPtr, DibMaskPtr, DibBitsPtr Functions:

These functions return pointers to different components of the DIB section, facilitating easy access to header information, color masks, and bitmap bits.

11. DibGetColor, DibSetColor Functions:

The DibGetColor and DibSetColor functions retrieve and set entries in the DIB color table, respectively. They leverage device context operations for interacting with the color table.

Here's a comprehensive explanation of the information functions in DIBHELP.C:

Safeguarding Data Integrity:

DibIsValid: This function acts as a gatekeeper, ensuring that only valid DIBSTRUCT pointers are used for accessing DIB information. This prevents potential errors and crashes, enhancing the robustness of the library.

Streamlined Access to Essential Information:

Functions for Basic Attributes: DibWidth, DibHeight, DibBitCount, and DibRowLength provide direct access to fundamental DIB properties, including dimensions, color depth, and pixel row lengths. These are crucial for understanding and manipulating DIB content.

DibNumColors: Determines the number of colors in the DIB's color table, essential for various color-related operations.

Handling Compression and Pixel Accessibility:

DibCompression: Reveals the compression technique used in the DIB, guiding appropriate handling methods.

DibIsAddressable: Determines whether direct pixel access is possible based on the compression type. This is crucial for efficient pixel manipulation.

Memory Allocation and File Operations:

Size-Related Functions: DibInfoHeaderSize, DibMaskSize, DibColorSize, DibInfoSize, DibBitsSize, and DibTotalSize calculate the sizes of various DIB components, aiding in memory allocation and file management.

Direct Pointers for Efficient Manipulation:

Pointer-Providing Functions: DibInfoHeaderPtr, DibMaskPtr, and DibBitsPtr offer direct access to the BITMAPINFOHEADER, color masks, and pixel data, respectively, enabling efficient editing and processing of these structural elements.

Color Table Management:

DibGetColor and DibSetColor: Retrieve and modify individual color entries within the DIB's color table, facilitating color customization. These functions act as wrappers around the more fundamental GetDIBColorTable and SetDIBColorTable functions.

Indirect Access to BITMAPINFO:

No Direct Pointer: While most DIB components are accessible through pointers, the BITMAPINFO structure (information header plus color table) is not directly available in DIB sections. This is due to the underlying memory management of DIB sections.

Indirect Access: The color table can be accessed indirectly through DibGetColor and DibSetColor, and the BITMAPINFOHEADER is available through DibInfoHeaderPtr.

DibCopyToInfo: This function, explained later, allocates a new BITMAPINFO structure and fills it with DIB information, but it doesn't provide a pointer to the existing structure in memory.

Key Takeaways:

- The information functions in DIBHELP.C offer a comprehensive suite of tools for understanding, navigating, and interacting with DIB sections.
- They prioritize data integrity through validation and handle various compression formats effectively.
- They provide direct access to most DIB components, with indirect access for the BITMAPINFO structure.
- These functions serve as a foundation for efficient DIB manipulation within the DIBHELP library.

In summary, these functions collectively provide a comprehensive set of tools for extracting various attributes of a DIB and manipulating its components. The DIBSTRUCT structure serves as a central container for organizing DIB-related information, and the functions utilize this structure to offer a convenient interface for DIB handling in Windows programming.

Let's now move on to the next code portion for dibhelp.c part 2... pg 725

DIBHELP.C PART 2

12. DibPixelPtr Function:

The DibPixelPtr function is designed to return a pointer to the pixel at a specified position (x, y) in the DIB. Before providing the pointer, it performs checks to ensure that the DIB is addressable and that the specified coordinates are within the valid range. The returned pointer allows direct manipulation of the pixel bits.

13. DibGetPixel Function:

The DibGetPixel function retrieves the pixel value at a given position (x, y) within the DIB. It utilizes the previously defined DibPixelPtr function to obtain the pointer to the pixel. The pixel value is then extracted based on the bit count of the DIB, employing different bit manipulation operations for various bit depths (1, 4, 8, 16, 24, and 32 bits). This function is essential for reading pixel values.

14. DibSetPixel Function:

In contrast to DibGetPixel, the DibSetPixel function is responsible for setting a pixel value at a specified position (x, y) within the DIB. Similar to DibGetPixel, it uses DibPixelPtr to get the pointer to the target pixel. The function then modifies the pixel bits based on the bit count and the provided pixel value, facilitating the manipulation of individual pixels.

15. DibGetPixelColor and DibSetPixelColor Functions:

These functions, DibGetPixelColor and DibSetPixelColor, extend the pixel manipulation capabilities by allowing the retrieval and setting of pixel colors, respectively. The DibGetPixelColor function obtains the color information at a specified position (x, y) within the DIB. It considers various scenarios, such as different bit depths and color table indexing. On the other hand, the DibSetPixelColor function sets the pixel color at a specified position, again taking into account the bit depth and adjusting the pixel bits accordingly. These functions provide a convenient way to work with pixel colors directly.

16. Color Handling in DibGetPixelColor and DibSetPixelColor:

The color handling in these functions is adaptable to the DIB's characteristics. For instance, in the case of 24-bit bitmaps, it directly uses the pixel bits as the RGB color. For 32-bit bitmaps with BI_RGB compression, it interprets the pixel bits as an RGBQUAD structure. In scenarios involving masks and shifts, the functions perform the necessary bit manipulations to extract or set the RGB components of the pixel color.

In summary, this section of DIBHELP.C provides functions for direct manipulation of pixel bits and colors within a DIB. These functions enhance the flexibility of DIB handling in applications, allowing for efficient reading and writing of pixel values and colors at specific positions in the bitmap. The modular design of these functions contributes to the overall ease of use and maintainability of DIB-related operations.

DibPixelPtr Function Explanation:

The [DibPixelPtr function serves as the foundation for pixel manipulation](#) within the DIB. Its primary purpose is to retrieve a pointer to the byte where a specific pixel is stored. By utilizing the ppRow field of the DIBSTRUCT structure, which points to the rows of pixels in the DIB, this function calculates the address of the desired pixel. The returned pointer points to the leftmost pixel of the top row for (0, 0), and it can be adjusted for any given (x, y) position. The function ensures the DIB's addressability and validates the coordinates, returning NULL if the DIB is compressed or if the coordinates reference an area outside the DIB.

```
BYTE* DibPixelPtr(HDIB hdib, int x, int y) {
    if (!DibIsAddressable(hdib))
        return NULL;

    if (x < 0 || x >= DibWidth(hdib) || y < 0 || y >= DibHeight(hdib))
        return NULL;

    return (((PDIBSTRUCT)hdib)->ppRow)[y] + (x * DibBitCount(hdib) >> 3);
}
```

DibGetPixel and DibSetPixel Functions:

Following DibPixelPtr, the functions DibGetPixel and DibSetPixel **leverage the pointer obtained to either retrieve or set the pixel value at a specified position**. These functions seamlessly handle different bit depths, casting the pointer to the appropriate data size for 8-bit, 16-bit, 24-bit, and 32-bit DIBs. However, for 1-bit and 4-bit DIBs, additional masking and shifting operations are necessary to manipulate individual pixels.

```
DWORD DibGetPixel(HDIB hdib, int x, int y) {
    PBYTE pPixel;
    if (!(pPixel = DibPixelPtr(hdib, x, y)))
        return 0;

    // Bit-depth specific operations to extract pixel value
    // ...

    return 0; // Placeholder return value
}

BOOL DibSetPixel(HDIB hdib, int x, int y, DWORD dwPixel) {
    PBYTE pPixel;
    if (!(pPixel = DibPixelPtr(hdib, x, y)))
        return FALSE;

    // Bit-depth specific operations to set pixel value
    // ...

    return TRUE;
}
```

DibGetColor Function:

The **DibGetColor** function complements the pixel manipulation by obtaining the pixel color as an RGBQUAD structure. It handles 1-bit, 4-bit, and 8-bit DIBs by retrieving the color from the DIB color table. For 16-bit, 24-bit, and 32-bit DIBs, it involves masking and shifting the pixel value to derive an RGB color.

```
BOOL DibGetColor(HDIB hdib, int index, RGBQUAD* prgb) {
    // Color retrieval logic based on DIB bit depth
    // ...

    return TRUE; // Placeholder return value
}
```

DibSetColor Function:

Conversely, the **DibSetColor** function enables setting a pixel value from an RGBQUAD structure. This function is specifically defined for 16-bit, 24-bit, and 32-bit DIBs.

```
BOOL DibSetColor(HDIB hdib, int x, int y, RGBQUAD* prgb) {
    // Pixel value setting logic based on DIB bit depth
    // ...

    return TRUE; // Placeholder return value
}
```

These functions collectively facilitate the direct manipulation of pixels and colors within a DIB, offering flexibility and efficiency in handling various bit depths and compression formats.

DIBHELP.C PART 3

Now we're going in-depth to more advanced levels for dibhelp.c...Part 3 pg. 730

Let's break down the code into sections for discussion:

Shift Calculation Functions:

The MaskToRShift and MaskToLShift functions play a critical role in handling color masks within the DIBHELP.C code.

These static functions are responsible for calculating the shift values necessary for proper bit manipulation.

Specifically, MaskToRShift determines the number of bits a color mask needs to be shifted to the right, while MaskToLShift calculates the leftward shift.

These shift values are essential for correctly extracting color information from the DIB and are particularly relevant in scenarios where color masks are used, such as in the case of BI_BITFIELDS compression.

DibCreateFromInfo Function:

The DibCreateFromInfo function stands as a cornerstone in the creation of DIB sections.

It employs the CreateDIBSection API to establish a DIB, allocates memory for the associated DIBSTRUCT, and sets up the row pointer.

Beyond these fundamental tasks, it also undertakes the responsibility of handling color masks and bitfields.

This includes calculating shift values for color masks, a crucial aspect when dealing with DIBs that utilize BI_BITFIELDS compression.

The function provides a robust foundation for creating and initializing DIB structures, ensuring their integrity and usability.

DibDelete Function:

The DibDelete function serves a pivotal role in memory management within the DIBHELP.C code.

Its primary function is to release all memory associated with a given DIB section.

Before initiating the deletion process, the function performs a validity check to ensure that the DIB is, indeed, valid.

This **validation step** is crucial in **preventing potential issues** that could arise from attempting to delete an invalid or corrupted DIB.

By encapsulating memory deallocation and validation logic, DibDelete contributes to the overall robustness and reliability of the DIB handling mechanism.

DibCreate Function:

The DibCreate function is tasked with the **creation of an HDIB (Handle to DIB)** based on explicit arguments provided, such as width, height, bit count, and color count.

This function **encapsulates the process of allocating memory** for the BITMAPINFO structure, setting its parameters based on the provided arguments, and invoking the DibCreateFromInfo function.

The use of **DibCreateFromInfo ensures a consistent and standardized approach to DIB creation**, leveraging a well-established mechanism for initializing DIB structures.

This function provides a convenient and flexible interface for developers to generate DIBs tailored to specific requirements.

DibCopyToInfo Function:

The DibCopyToInfo function plays a crucial role in **constructing a BITMAPINFO structure** based on an existing Device-Independent Bitmap (DIB).

This function is integral in scenarios where a developer needs to obtain a comprehensive representation of the DIB's attributes.

It adeptly **handles various aspects of DIB information**, including color masks and color table copying.

The primary purpose is to **encapsulate the complexities associated with extracting pertinent details from the DIB**, ultimately returning a well-formed BITMAPINFO structure.

This function serves as a valuable tool for developers seeking to interact with DIBs at a higher level, providing a **standardized interface** for obtaining essential information.

DibCopy Function:

The DibCopy function facilitates the **creation of a new DIB section** derived from an existing one.

One notable feature is its **flexibility in allowing the potential swapping of** the DIB's width and height.

This function **relies on the DibCopyToInfo function** to generate the BITMAPINFO structure, ensuring a consistent and standardized approach to the creation of the new DIB section.

By leveraging DibCopyToInfo, the function benefits from a well-established mechanism for capturing the necessary information from the source DIB, providing a convenient way to duplicate DIBs while offering an option for adjusting their dimensions.

This flexibility enhances the utility of the function in various scenarios, from straightforward duplication to more intricate transformations.

DibCopyToPackedDib Function:

The **DibCopyToPackedDib function serves a dual purpose**, primarily used for both saving DIBs to a file and transferring DIBs to the clipboard.

The function excels in copying DIB information to a packed DIB format, encapsulating color masks, color table, and pixel bits.

Notably, it **provides the option to allocate memory** with the GMEM_SHARE flag, enhancing compatibility and sharing capabilities.

This function is **particularly valuable in scenarios where the goal is** to store or transmit DIBs in a compact and standardized format.

By encapsulating the complexities of packing DIB information, the function streamlines the process of handling DIBs in various contexts, from storage to clipboard operations.

DibCopyFromPackedDib Function:

The DibCopyFromPackedDib function is a pivotal component when dealing with clipboard operations involving Device-Independent Bitmaps (DIBs).

Its primary function is to reconstruct an HDIB (Handle to DIB) from a packed DIB format.

This involves copying the pixel bits and utilizing the information encapsulated in the packed DIB.

The function acts as a bridge between different representations of DIBs, facilitating seamless integration and manipulation of image data.

By intelligently handling the reconstruction process, it simplifies the task of pasting DIBs from the clipboard, ensuring that the HDIB retains all necessary attributes for further processing.

DibFileLoad and DibFileSave Functions:

The tandem of DibFileLoad and DibFileSave functions introduces file I/O operations to the program, enabling the reading and writing of DIBs to and from files.

DibFileLoad reads a DIB from a specified file, employing file access operations to retrieve both the header and pixel data.

This function is crucial for scenarios where pre-existing DIBs need to be loaded into the program.

On the other hand, DibFileSave performs the reverse operation by saving a DIB to a file.

It utilizes file-writing mechanisms to store the DIB's structure and pixel information persistently.

These functions extend the program's utility, allowing developers to seamlessly integrate DIBs with external storage, retrieval, and sharing mechanisms.

DibCopyToDdb Function:

The DibCopyToDdb function is tailored for [efficient screen displays](#), particularly when working with device-dependent bitmaps (DDBs).

It orchestrates the copying of a DIB to a DDB for the purpose of display, leveraging a provided palette for color accuracy.

This function [becomes invaluable when developers need to optimize the rendering of DIBs](#) on the screen, as DDBs are typically more efficient for direct rendering on display devices.

By seamlessly integrating with the Windows Graphics Device Interface (GDI), DibCopyToDdb enhances the program's capabilities in visualizing DIBs in a manner that is both efficient and visually accurate.

Let's keep going...

The initial segment of the DIBHELP.C file introduces two essential functions for deriving left-shift and right-shift values from color masks, specifically tailored for 16-bit and 32-bit DIBs.

These functions were previously detailed in the Color Masking section of Chapter 15.

Moving forward, the focal point becomes the [DibCreateFromInfo function](#), the sole function in DIBHELP responsible for invoking CreateDIBSection and allocating memory for the DIBSTRUCT structure.

All other creation and copy functions within the codebase route through this central function. The [function takes a single parameter](#)—a pointer to a BITMAPINFO structure.

While the color table must exist, it doesn't necessarily need to be filled with valid values.

Following the CreateDIBSection call, the function meticulously initializes all fields of the DIBSTRUCT structure.

It's noteworthy that distinct logic is implemented for [bottom-up and top-down DIBs](#) when setting the values of the ppRow field, signifying the pointers to the DIB row addresses. The first element of ppRow invariably represents the top row of the DIB.

The subsequent function, [DibDelete](#), serves a dual purpose. It not only deletes the bitmap created in DibCreateFromInfo but also releases the memory allocated during that function's execution.

In contrast to [DibCreateFromInfo](#), the DibCreate function is more likely to be invoked by application programs. Its initial three arguments define the pixel width, height, and the number of bits per pixel, while the last argument allows flexibility for specifying a custom color table size or opting for the default size by setting it to 0.

[DibCopy](#), a function responsible for creating a new DIB section from an existing one, relies on the DibCreateInfo function for memory allocation of a BITMAPINFO structure and populating it with the necessary data.

A [BOOL argument in DibCopy](#) determines whether the DIB width and height should be swapped when crafting the new DIB section, providing a level of flexibility for certain use cases.

The tandem of DibCopyToPackedDib and DibCopyFromPackedDib functions finds its purpose in facilitating the transfer of DIBs, particularly for operations involving the clipboard.

The DibFileLoad function reads a DIB section from a DIB file, while DibFileSave performs the complementary operation of saving a DIB to a file.

Concluding the functionalities is the [DibCopyToDdb function](#), designed for efficient screen displays.

It creates a GDI bitmap object from a DIB, requiring handles to the current palette and the program's window.

The window handle serves to obtain a device context into which the palette is selected and realized, allowing the function to subsequently invoke CreateDIBitmap. This functionality was demonstrated in the SHOWDIB7 program earlier in the chapter.

For those who don't know english  let me educate you 

Meticulously:

The word "meticulously" means [to do something with great attention to detail](#), precision, and thoroughness. When something is done meticulously, it implies that care and precision have been taken to ensure every aspect is handled with great accuracy and thoroughness.

Tandem:

The word "tandem" refers to **two things or elements that work closely together** or in coordination with each other. It suggests a partnership or collaboration between the two entities, often emphasizing their interdependence or joint functioning. In the given context, "DibCopyToPackedDib" and "DibCopyFromPackedDib" are described as a tandem, indicating that these two functions work together in a complementary manner to facilitate the transfer of DIBs.

DIBHELP HEADER FILES AND MACROS

DIBHELP.H is a header file that accompanies the DIBHELP.C file and **provides various functions and macros related to handling Device Independent Bitmaps (DIBs)**.

The header file defines the **HDIB handle as a void pointer**, shielding the internal structure of the DIB from direct access by applications. It also includes declarations for all the functions present in the DIBHELP.C file.

However, the **header file also introduces a set of macros that are designed to optimize the performance of certain operations involving DIB pixels**. These macros are specific to different bit counts, and they provide a way to access and manipulate pixels in a more efficient manner.

For instance, the header file introduces macros such as DibPixelPtr1, DibPixelPtr4, DibPixelPtr8, and so on.

These **macros allow direct access to the memory location of a pixel** based on its coordinates (x, y) in the DIB, depending on the bit count of the DIB.

By using these macros, one can bypass function calls and directly manipulate the pixel values, potentially improving performance.

Similarly, the **header file provides macros like DibGetPixel1, DibGetPixel4, DibGetPixel8**, and their corresponding set functions (e.g., DibSetPixel1, DibSetPixel4, DibSetPixel8).

These **macros enable quick retrieval and modification of pixel values** without the need for function calls. Again, these macros are specific to different bit counts and are designed to enhance performance by eliminating function call overhead.

In summary, the [DIBHELP.H header file](#) augments the [DIBHELP.C file](#) by introducing macros that optimize pixel access and manipulation operations for different bit counts in Device Independent Bitmaps. These macros allow for faster pixel handling by eliminating function calls and directly accessing the relevant memory locations.

THE HEADER FILE

The header file provides several sets of macros for different bit counts, allowing efficient access and manipulation of pixels in Device Independent Bitmaps (DIBs). Here are the key sets of macros and their purposes:

DibPixelPtr Macros:

[DibPixelPtr1](#), [DibPixelPtr4](#), [DibPixelPtr8](#), [DibPixelPtr16](#), [DibPixelPtr24](#), [DibPixelPtr32](#): These macros provide a way to obtain a pointer to the memory location of a specific pixel in the DIB based on its coordinates (x, y) and the bit count of the DIB.

For example, `DibPixelPtr8(hdib, x, y)` returns a pointer to the memory location of the pixel at coordinates (x, y) in an 8-bit DIB.

DibGetPixel Macros:

[DibGetPixel1](#), [DibGetPixel4](#), [DibGetPixel8](#), [DibGetPixel16](#), [DibGetPixel24](#), [DibGetPixel32](#): These macros allow quick retrieval of the pixel value at a specific location in the DIB, based on its coordinates (x, y) and the bit count of the DIB.

For instance, DibGetPixel24(hdib, x, y) returns the RGB value of the pixel at coordinates (x, y) in a 24-bit DIB.

DibSetPixel Macros:

DibSetPixel1, DibSetPixel4, DibSetPixel8, DibSetPixel16, DibSetPixel24, DibSetPixel32:

These macros enable the modification of the pixel value at a specific location in the DIB, based on its coordinates (x, y) and the bit count of the DIB.

For example, DibSetPixel8(hdib, x, y, p) sets the pixel at coordinates (x, y) in an 8-bit DIB to the value 'p'.

By utilizing these macros, **developers can directly access and manipulate pixels in the DIB** without the need for function calls. This can lead to improved performance by reducing the overhead associated with function calls and enabling more efficient pixel handling.

It's worth noting that **the macros assume certain conditions and have limited error checking**. They are designed to be used carefully, ensuring that the provided coordinates and bit count match the characteristics of the DIB. Hence, it's important to use these macros in a controlled and appropriate manner to avoid unexpected behavior or memory access issues.

Here are some examples of the macros provided by the DIBHELP.H header file:

DibPixelPtr Macros:

- **DibPixelPtr1(hdib, x, y):** Returns a pointer to the memory location of the pixel at coordinates (x, y) in a 1-bit DIB.
- **DibPixelPtr8(hdib, x, y):** Returns a pointer to the memory location of the pixel at coordinates (x, y) in an 8-bit DIB.
- **DibPixelPtr24(hdib, x, y):** Returns a pointer to the memory location of the pixel at coordinates (x, y) in a 24-bit DIB.

DibGetPixel Macros:

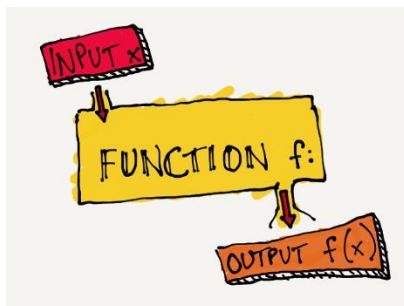
- **DibGetPixel1(hdib, x, y):** Retrieves the pixel value at coordinates (x, y) in a 1-bit DIB.
- **DibGetPixel8(hdib, x, y):** Retrieves the pixel value at coordinates (x, y) in an 8-bit DIB.
- **DibGetPixel24(hdib, x, y):** Retrieves the RGB value of the pixel at coordinates (x, y) in a 24-bit DIB.

DibSetPixel Macros:

- **DibSetPixel1(hdib, x, y, p):** Sets the pixel at coordinates (x, y) in a 1-bit DIB to the value 'p'.
- **DibSetPixel8(hdib, x, y, p):** Sets the pixel at coordinates (x, y) in an 8-bit DIB to the value 'p'.
- **DibSetPixel24(hdib, x, y, r, g, b):** Sets the RGB value of the pixel at coordinates (x, y) in a 24-bit DIB to the specified (r, g, b) values.

Now, let's discuss how the macros in the DIBHELP.H header file improve the performance of pixel operations:

Elimination of function call overhead: The macros allow direct access to pixel values and memory locations, bypassing function calls. This eliminates the overhead associated with function calls, resulting in faster pixel operations.



Efficient memory access: The macros provide optimized memory access by directly calculating the memory location of a pixel based on its coordinates and the bit count of the DIB. This avoids unnecessary memory lookups and improves performance.



Reduced code size: By using macros instead of function calls, the resulting code can be more concise and compact. This can lead to improved cache utilization and instruction pipelining, enhancing overall performance.



Now, let's discuss the purpose of the HDIB handle and why it is defined as a void pointer:

The HDIB handle is a type definition used to represent a handle to a Device Independent Bitmap (DIB). It is defined as a void pointer (`void*`) to provide flexibility and avoid exposing the internal structure of the DIB to applications.

By defining HDIB as a void pointer, the specific structure of the DIB is hidden from the application code. This abstraction allows the implementation of the DIB handling functions to change and evolve without impacting the application code that uses the HDIB handle.

The void pointer allows the handle to be used as a generic pointer that can be cast to the appropriate DIB structure type when needed. This enables compatibility with different DIB formats and simplifies the interface for working with DIBs.

In summary, the HDIB handle is defined as a void pointer to provide flexibility, abstraction, and compatibility in working with Device Independent Bitmaps. It allows the DIB handling functions to operate on different DIB formats without exposing the internal structure to the application code.

How can I determine the bit count of a given DIB? Are there any limitations or considerations when using these macros? Can you provide an example of how these macros can be used in practice? What are some other benefits of using the DibPixelPtr macros for pixel operations? Can you provide an example of how the HDIB handle is used in practice? How does the HDIB handle differ from other types of handles used in programming?

To determine the bit count of a given Device Independent Bitmap (DIB), you can use the biBitCount member of the BITMAPINFOHEADER structure. The biBitCount member specifies the number of bits per pixel in the DIB. For example, if the biBitCount is 8, it indicates an 8-bit DIB.

When using the macros provided in the DIBHELP.H header file, there are a few limitations and considerations to keep in mind:

- **Ensure the correct bit count:** The macros are specific to different bit counts (e.g., DibPixelPtr8 for an 8-bit DIB). It's crucial to use the appropriate macros that match the bit count of the DIB you are working with. Using the wrong macro may lead to incorrect results or memory access violations.
- **Coordinate validity:** The macros assume that the provided coordinates (x, y) are within the valid range of the DIB's dimensions. It's important to ensure that the coordinates fall within the DIB's width and height to avoid accessing out-of-bounds memory locations.

Here's an example of how these macros can be used in practice:

```
1 // Assuming the DIB has been loaded and the HDIB handle is available
2
3 // Get the pixel value at coordinates (x, y) in an 8-bit DIB
4 unsigned char pixelValue = DibGetPixel8(hdib, x, y);
5
6 // Set the pixel at coordinates (x, y) in a 24-bit DIB to the RGB value (r, g, b)
7 DibSetPixel24(hdib, x, y, r, g, b);
```

Some benefits of using the DibPixelPtr macros for pixel operations include:

- **Direct memory access:** The macros provide a pointer directly to the memory location of a pixel, allowing efficient and direct manipulation of pixel values without the need for intermediate steps.

- **Performance optimization:** By bypassing function calls and accessing pixels directly, the macros can improve the performance of pixel operations, especially in scenarios where a large number of pixels need to be processed.

Regarding the usage of HDIB handles in practice, here's an example:

```
// Create an HDIB handle
HDIB hdib = CreateDIB(...);

// Use the HDIB handle to perform various operations on the DIB
DibSetPixel8(hdib, x, y, p);
unsigned char pixelValue = DibGetPixel8(hdib, x, y);

// Release the HDIB handle when it's no longer needed
DestroyDIB(hdib);
```

The *HDIB handle* is used to represent a handle to a DIB. It provides an abstraction that encapsulates the internal structure of the DIB, allowing operations to be performed on the DIB without exposing its implementation details. The HDIB handle can be used to pass DIB information between functions and manage the lifetime of the DIB.

The *HDIB handle differs from other types of handles* used in programming in that it is specific to DIBs. Other types of handles, such as file handles or window handles, are typically used to represent different resources or objects in a system. Each handle type has its own purpose and may have specific functions and operations associated with it. The HDIB handle specifically represents a handle to a DIB and provides functions and macros tailored for DIB manipulation.

DIBBLE PROGRAM

This is going to be long, let's break it down into sections, so that we can cover it in-depth.



Dibble program in action.mp4

```

1  /*-----*
2   [ DIBBLE.C -- Bitmap and Palette Program
3   (c) Charles Petzold, 1998
4   -----*/
5
6  #include <windows.h>
7  #include "dibhelp.h"
8  #include "dibpal.h"
9  #include "dibconv.h"
10 #include "resource.h"
11
12 #define WM_USER_SETSCROLLS    (WM_USER + 1)
13 #define WM_USER_DELETEDIB    (WM_USER + 2)
14 #define WM_USER_DELETEPAL    (WM_USER + 3)
15 #define WM_USER_CREATEPAL    (WM_USER + 4)
16
17 LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
18
19 TCHAR szAppName[] = TEXT ("Dibble");
20
21 int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
22                     PSTR szCmdLine, int iCmdShow)
23 {
24     HACCEL hAccel;
25     HWND hwnd;
26     MSG msg;
27     WNDCLASS wndclass;
28
29     wndclass.style      = CS_HREDRAW | CS_VREDRAW;
30     wndclass.lpfnWndProc = WndProc;
31     wndclass.cbClsExtra = 0;
32     wndclass.cbWndExtra = 0;
33     wndclass.hInstance = hInstance;
34     wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
35     wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
36     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
37     wndclass.lpszMenuName = szAppName;
38     wndclass.lpszClassName = szAppName;
39
40     if (!RegisterClass (&wndclass))
41     {
42         MessageBox (NULL, TEXT ("This program requires Windows NT!"),
43                     szAppName, MB_ICONERROR);
44         return 0;
45     }
46
47     hwnd = CreateWindow (szAppName, szAppName,
48                          WS_OVERLAPPEDWINDOW | WM_VSCROLL | WM_HSCROLL,
49                          CW_USEDEFAULT, CW_USEDEFAULT,
50                          CW_USEDEFAULT, CW_USEDEFAULT,
51                          NULL, NULL, hInstance, NULL);
52
53     ShowWindow (hwnd, iCmdShow);
54     UpdateWindow (hwnd);
55
56     hAccel = LoadAccelerators (hInstance, szAppName);
57
58     while (GetMessage (&msg, NULL, 0, 0))
59     {
60         if (!TranslateAccelerator (hwnd, hAccel, &msg))
61         {
62             TranslateMessage (&msg);
63             DispatchMessage (&msg);
64         }
65     }
66     return msg.wParam;
67 }

```

The program **defines a window procedure (WndProc)** and several constants, including window messages (WM_USER_SETSCROLLS, WM_USER_DELETEDIB, WM_USER_DELETEPAL, WM_USER_CREATEPAL).

It also **defines the application name (szAppName)** and the entry point (WinMain).

The **WndProc function handles messages for the main window**. The window class is registered, and a window is created with scrollbars. The main loop processes messages, and accelerator keys are translated using LoadAccelerators and TranslateAccelerator.

The program involves working with bitmaps and palettes, as suggested by the header files (dibhelp.h, dibpal.h, dibconv.h) and resource file (resource.h).

The code mentions a requirement for Windows NT, indicating that it may have been developed for an older version of Windows. It uses standard Windows API functions for window creation, message handling, and resource loading.

Let's move to the second part of `dibble.c` file...

```

69  /*-----  

70   DisplayDib: Displays or prints DIB actual size or stretched  

71   depending on menu selection  

72   -----*/  

73  

74  int DisplayDib (HDC hdc, HBITMAP hBitmap, int x, int y,  

75                  int cxClient, int cyClient,  

76                  WORD wShow, BOOL fHalftonePalette)  

77  {  

78      BITMAP bitmap ;  

79      HDC hdcMem ;  

80      int cxBitmap, cyBitmap, iReturn ;  

81  

82      GetObject (hBitmap, sizeof (BITMAP), &bitmap) ;  

83      cxBitmap = bitmap.bmWidth ;  

84      cyBitmap = bitmap.bmHeight ;  

85  

86      SaveDC (hdc) ;  

87  

88      if (fHalftonePalette)  

89          SetStretchBltMode (hdc, HALFTONE) ;  

90      else  

91          SetStretchBltMode (hdc, COLORONCOLOR) ;  

92  

93      hdcMem = CreateCompatibleDC (hdc) ;  

94      SelectObject (hdcMem, hBitmap) ;  

95  

96      switch (wShow)  

97      {  

98          case IDM_SHOW_NORMAL:  

99              if (fHalftonePalette)  

100                 iReturn = StretchBlt (hdc, 0, 0,  

101                             min (cxClient, cxBitmap - x),  

102                             min (cyClient, cyBitmap - y),  

103                             hdcMem, x, y,  

104                             min (cxClient, cxBitmap - x),  

105                             min (cyClient, cyBitmap - y),  

106                             SRCCOPY);  

107          else  

108              iReturn = BitBlt (hdc, 0, 0,  

109                      min (cxClient, cxBitmap - x),  

110                      min (cyClient, cyBitmap - y),  

111                      hdcMem, x, y, SRCCOPY) ;  

112          break ;  

113  

114          case IDM_SHOW_CENTER:  

115              if (fHalftonePalette)  

116                  iReturn = StretchBlt (hdc, (cxClient - cxBitmap) / 2,  

117                                         (cyClient - cyBitmap) / 2,  

118                                         cxBitmap, cyBitmap,  

119                                         hdcMem, 0, 0, cxBitmap, cyBitmap, SRCCOPY);  

120          else  

121              iReturn = BitBlt (hdc, (cxClient - cxBitmap) / 2,  

122                                 (cyClient - cyBitmap) / 2,  

123                                 cxBitmap, cyBitmap,  

124                                 hdcMem, 0, 0, SRCCOPY) ;  

125          break ;  

126  

127          case IDM_SHOW_STRETCH:  

128              iReturn = StretchBlt (hdc, 0, 0, cxClient, cyClient,  

129                                    hdcMem, 0, 0, cxBitmap, cyBitmap, SRCCOPY) ;  

130          break ;  

131  

132          case IDM_SHOW_ISOSTRETCH:  

133              SetMapMode (hdc, MM_ISOTROPIC) ;  

134              SetWindowExtEx (hdc, cxBitmap, cyBitmap, NULL) ;  

135              SetViewportExtEx (hdc, cxClient, cyClient, NULL) ;  

136              SetWindowOrgEx (hdc, cxBitmap / 2, cyBitmap / 2, NULL) ;  

137              SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;  

138  

139              iReturn = StretchBlt (hdc, 0, 0, cxBitmap, cyBitmap,  

140                                    hdcMem, 0, 0, cxBitmap, cyBitmap, SRCCOPY) ;  

141          break ;  

142      }  

143      DeleteDC (hdcMem) ;  

144      RestoreDC (hdc, -1) ;  

145      return iReturn ;  

146 }

```

The DisplayDib function is designed to display or print a Device Independent Bitmap (DIB) in a specified manner based on the menu selection.

It takes various parameters, including the device context (hdc), the bitmap handle (hBitmap), position coordinates (x and y), client area dimensions (cxClient and cyClient), a flag indicating the display mode (wShow), and a boolean flag for halftone palette usage (fHalftonePalette).

The function begins by retrieving information about the bitmap using the GetObject function, storing the width and height in cxBitmap and cyBitmap, respectively.

A device context (hdcMem) is then created as a memory-compatible DC, and the bitmap is selected into this DC using SelectObject. The subsequent operations are determined by the value of wShow, which corresponds to different display modes.

IDM_SHOW_NORMAL:

This mode displays the bitmap either stretched or normally, depending on whether the halftone palette is used or not. It uses either StretchBlt or BitBlt to copy the bitmap from the memory DC to the destination DC, considering the specified coordinates and dimensions.

IDM_SHOW_CENTER:

In this mode, the bitmap is displayed at the center of the client area. Similar to the normal mode, it uses either StretchBlt or BitBlt to copy the bitmap with appropriate calculations for centering.

IDM_SHOW_STRETCH:

The bitmap is stretched to fit the entire client area in this mode, using StretchBlt.

IDM_SHOW_ISOSTRETCH:

This mode employs isotropic stretching, setting the mapping mode to isotropic and adjusting the window and viewport extents and origins. The bitmap is then stretched using StretchBlt.

Finally, the memory DC is deleted, and the device context is restored to its original state. The function returns the result of the last GDI operation (iReturn), which can be used to check for success or failure.

The overall purpose of this function is to provide flexibility in displaying a bitmap in different ways, such as normal, centered, stretched, or isotropically stretched, based on user preferences and menu selections. The choice between halftone and color-on-color modes influences the quality of the displayed image, with halftone generally providing smoother results in stretched scenarios.

The third part of dibble.c...

```
152     HDIB DibFlipHorizontal (HDIB hdibSrc)
153     {
154         HDIB hdibDst ;
155         int cx, cy, x, y ;
156
157         if (!DibIsAddressable (hdibSrc))
158             return NULL ;
159
160         if (NULL == (hdibDst = DibCopy (hdibSrc, FALSE)))
161             return NULL ;
162
163         cx = DibWidth (hdibSrc) ;
164         cy = DibHeight (hdibSrc) ;
165
166         for (x = 0 ; x < cx ; x++)
167             for (y = 0 ; y < cy ; y++)
168             {
169                 DibSetPixel (hdibDst, x, cy - 1 - y, DibGetPixel (hdibSrc, x, y)) ;
170             }
171         return hdibDst ;
172     }
```

The **DibFlipHorizontal** function is responsible for creating a horizontally flipped version of a given DIB (Device Independent Bitmap) represented by the `hdibSrc` parameter. The resulting flipped DIB is returned as a new DIB (`hdibDst`).

The function begins by **checking if the source DIB is addressable** using the `DibIsAddressable` function. If the source DIB is not addressable (e.g., due to memory issues or an invalid DIB structure), the function returns `NULL`.

Next, it **allocates memory for the destination DIB (`hdibDst`) using the `DibCopy` function**. This function creates a copy of the source DIB, and the `FALSE` parameter indicates that a deep copy (including pixel data) is desired.

The **dimensions of the DIB (width and height)** are then obtained using the `DibWidth` and `DibHeight` functions.

The core operation of the function involves **iterating through each pixel of the source DIB and copying it to the corresponding position in the destination DIB**, but with a horizontal flip. The nested for loops iterate over each column (`x`) and row (`y`) of the source DIB.

For each pixel at coordinates (x, y) in the source DIB, the function uses the [DibGetPixel](#) function to retrieve the pixel's color information.

It then employs the DibSetPixel function to set the color information at the mirrored position (x, cy - 1 - y) in the destination DIB.

The mirroring is achieved by subtracting the current y coordinate from the total height (cy - 1 - y), effectively flipping the pixel horizontally.

After [iterating through all pixels in the source DIB](#), the function returns the resulting horizontally flipped DIB (hdibDst).

This function essentially [performs a simple image transformation](#) by flipping the pixels of the source DIB horizontally to create a mirrored version.

It [demonstrates basic pixel manipulation techniques](#) using functions like DibGetPixel and DibSetPixel. Keep in mind that this implementation might not be optimized for performance, and more efficient algorithms could be employed for large images.

The fourth part of the dibble.c file...

```
174  /*-----  
175   DibRotateRight: Calls optimized DibSetPixelx and DibGetPixelx  
176   -----*/  
177  
178 HDIB DibRotateRight (HDIB hdibSrc)  
179 {  
180     HDIB hdibDst ;  
181     int cx, cy, x, y ;  
182  
183     if (!DibIsAddressable (hdibSrc))  
184         return NULL ;  
185  
186     if (NULL == (hdibDst = DibCopy (hdibSrc, TRUE)))  
187         return NULL ;  
188  
189     cx = DibWidth (hdibSrc) ;  
190     cy = DibHeight (hdibSrc) ;  
191  
192     switch (DibBitCount (hdibSrc))  
193     {  
194         case 1:  
195             for (x = 0 ; x < cx ; x++)  
196                 for (y = 0 ; y < cy ; y++)  
197                     DibSetPixel1 (hdibDst, cy - y - 1, x,  
198                         DibGetPixel1 (hdibSrc, x, y)) ;  
199             break ;  
200  
201         case 4:  
202             for (x = 0 ; x < cx ; x++)  
203                 for (y = 0 ; y < cy ; y++)  
204                     DibSetPixel4 (hdibDst, cy - y - 1, x,  
205                         DibGetPixel4 (hdibSrc, x, y)) ;  
206             break ;  
207  
208         case 8:  
209             for (x = 0 ; x < cx ; x++)  
210                 for (y = 0 ; y < cy ; y++)  
211                     DibSetPixel8 (hdibDst, cy - y - 1, x,  
212                         DibGetPixel8 (hdibSrc, x, y)) ;  
213             break ;  
214  
215         case 16:  
216             for (x = 0 ; x < cx ; x++)  
217                 for (y = 0 ; y < cy ; y++)  
218                     DibSetPixel16 (hdibDst, cy - y - 1, x,  
219                         DibGetPixel16 (hdibSrc, x, y)) ;  
220             break ;  
221  
222         case 24:  
223             for (x = 0 ; x < cx ; x++)  
224                 for (y = 0 ; y < cy ; y++)  
225                     DibSetPixel24 (hdibDst, cy - y - 1, x,  
226                         DibGetPixel24 (hdibSrc, x, y)) ;  
227             break ;  
228  
229         case 32:  
230             for (x = 0 ; x < cx ; x++)  
231                 for (y = 0 ; y < cy ; y++)  
232                     DibSetPixel32 (hdibDst, cy - y - 1, x,  
233                         DibGetPixel32 (hdibSrc, x, y)) ;  
234             break ;  
235     }  
236     return hdibDst ;  
237 }
```

The **DibRotateRight** function is designed to rotate a given DIB (Device Independent Bitmap) represented by the `hdibSrc` parameter by 90 degrees to the right. The resulting rotated DIB is returned as a new DIB (`hdibDst`).

As with the previous function, the first step involves **checking if the source DIB is addressable** using the `DibIsAddressable` function. If the source DIB is not addressable, the function returns NULL.

Next, it **allocates memory for the destination DIB (`hdibDst`)** using the `DibCopy` function. This function creates a copy of the source DIB, and the TRUE parameter indicates that a deep copy (including pixel data) is desired.

The **dimensions of the DIB (width and height)** are then obtained using the `DibWidth` and `DibHeight` functions.

The **function then enters a switch statement** based on the bit count of the source DIB (`DibBitCount(hdibSrc)`). This switch statement handles different bit depths, and for each bit depth, it uses nested for loops to iterate over each pixel of the source DIB.

Within the nested loops, **it calls specialized set and get pixel functions** (`DibSetPixel1`, `DibGetPixel1` for 1-bit, `DibSetPixel4`, `DibGetPixel4` for 4-bit, and so on) to perform the pixel manipulation required for the rotation.

The **set pixel functions are used to write pixels to the destination DIB**, and the get pixel functions retrieve the corresponding pixel from the source DIB.

The **rotation involves moving pixels from their original positions** in the source DIB to new positions in the destination DIB, resulting in a 90-degree clockwise rotation.

After processing all pixels based on the bit depth, the **function returns the rotated DIB (`hdibDst`)**.

This function showcases a technique for rotating a bitmap, demonstrating different strategies for handling various bit depths efficiently.

It's important to note that this implementation uses specialized set and get pixel functions optimized for different bit depths, which can contribute to better performance.

The fifth part of dibble.c file... Palette menu

WNDPROC FUNCTION IN DEPTH

The provided code segment is part of the [WndProc function](#), which is the window procedure for the application. This function handles various messages that the window receives and contains logic for responding to those messages.

Initialization in WM_CREATE:

In the WM_CREATE case, the [code initializes several variables and structures](#). It sets up the OPENFILENAME structure for file open and save dialogs, specifying the filter for bitmap files. This is used later in the application for opening and saving image files.

Handling WM_DISPLAYCHANGE:

In the WM_DISPLAYCHANGE case, the [code sends custom messages](#) (WM_USER_DELETEPAL and WM_USER_CREATEPAL) to delete and recreate the palette when the display changes. This ensures that the application adjusts to changes in display settings.

Handling WM_SIZE and WM_USER_SETSCROLLS:

The WM_SIZE case [calculates and saves the client area's width and height](#). The subsequent WM_USER_SETSCROLLS case sets up the scroll bars based on the dimensions of the DIB (Device Independent Bitmap) being displayed. If the display mode is not normal or the DIB is not available, it [hides the scroll bars](#). If the wParam is true, it resets the scroll bar positions.

Handling WM_VSCROLL and WM_HSCROLL:

The WM_VSCROLL and WM_HSCROLL [cases manage vertical and horizontal scrolling](#), respectively. They use the ScrollWindow function to scroll the content within the window based on user actions, such as clicking on scroll buttons or dragging the scroll thumb. These messages are processed to update the scroll bar positions and adjust the displayed portion of the image accordingly.

WM_INITMENUPOPUP:

The WM_INITMENUPOPUP case is responsible for enabling or graying out specific menu items based on the application's state. It dynamically adjusts the menu options' availability depending on whether there is a loaded DIB (Device Independent Bitmap). For instance, it enables or grays out options related to file operations, editing, conversions, and palette operations based on the presence of a DIB and its properties.

WM_COMMAND - File Operations:

The WM_COMMAND case processes various menu commands, focusing on file-related operations. For example:

- **IDM_FILE_OPEN:** Opens a File Open dialog to load a DIB from a file. It then deletes the existing DIB and palette, loads the new DIB, resets scroll bars, and recreates the palette and device-dependent bitmap (DDB).
- **IDM_FILE_SAVE:** Opens a File Save dialog to save the current DIB to a file.
- **IDM_FILE_PRINT:** Prints the current DIB. It obtains the printer device context, checks if it can handle bitmaps, and sends the DIB to the printer.

WM_COMMAND - Edit Operations:

- **IDM_EDIT_COPY** and **IDM_EDIT_CUT:** Copy and cut operations to the clipboard. If the operation is cut, it also deletes the current DIB.
- **IDM_EDIT_PASTE:** Pastes a DIB from the clipboard, creating a new DIB and updating the display accordingly.

These operations demonstrate the basic file and clipboard functionality of the application. The code checks for valid conditions, handles user input, and updates the application state accordingly.

The use of the clipboard is implemented, allowing users to copy, cut, and paste DIBs between the application and other programs.

In-depth, the code reflects a responsive and interactive user interface, providing a range of file and editing capabilities with consideration for different states of the application.

It employs Windows API functions for file dialogs, clipboard operations, and printer interactions. The logic ensures a smooth user experience by handling potential errors and keeping the interface synchronized with the application's internal state.

This portion of the code, covered within the WM_COMMAND case, handles various user commands initiated through menu selections. Let's delve into the functionality provided by each case.

IDM_EDIT_PASTE:

This case is responsible for handling the paste operation. If there is data in the clipboard in the form of a packed DIB, the existing DIB and palette are deleted. Subsequently, the packed DIB is converted into an HDIB (handle to a device-independent bitmap), and a new palette and DDB are created. The scroll bars are then reset, and the window is invalidated for redrawing.

IDM_EDIT_ROTATE and IDM_EDIT_FLIP:

These cases handle the rotation and horizontal flip operations, respectively. If successful, they delete the current DIB and associated bitmap, replace them with the rotated or flipped version, update the scroll bars, and invalidate the window for redrawing. In case of a failure due to insufficient memory, an appropriate error message is displayed.

IDM_SHOW_NORMAL, IDM_SHOW_CENTER, IDM_SHOW_STRETCH, IDM_SHOW_ISOSTRETCH:

These cases handle different display options. They check and uncheck the corresponding menu items, update the show mode (wShow), reset the scroll bars, and invalidate the window for redrawing.

IDM_CONVERT_01 to IDM_CONVERT_32:

These cases handle color depth conversion. They accumulate the color depth value (iConvert), initiate the conversion, delete the existing DIB, create a new one with the

specified color depth, recreate the palette, and invalidate the window for redrawing. If the conversion fails due to insufficient memory, an error message is displayed.

IDM_APP_ABOUT:

This case displays an informational message box about the application, mentioning its name, copyright information, and the year it was created.

In-depth, this section of the code deals with [diverse user actions](#), from clipboard operations to image transformations and display options.

It ensures [dynamic updates](#) to the application state, reflects changes in the UI, and provides appropriate feedback in case of errors.

The [modular design](#) allows for easy extension and maintenance of the code.

The following section of the code primarily handles palette-related operations in response to menu commands, as well as some user-defined messages. Let's break down the functionality:

WM_COMMAND for Palette Items:

When a [palette-related menu item is selected](#), the existing palette is deleted, and the cursor changes to an hourglass.

The corresponding palette creation function is then called based on the selected menu item.

After palette creation, the cursor is restored to an arrow, the menu item is checked, and the window is [invalidated for redrawing](#). The selected palette is also used to create a new device-dependent bitmap (DDB).

- [IDM_PAL_DIBTABLE](#): Creates a palette based on the DIB's color table.
- [IDM_PAL_HALFTONE](#): Creates a halftone palette.
- [IDM_PAL_ALLPURPOSE](#): Creates an all-purpose palette.
- [IDM_PAL_GRAY2](#) to [IDM_PAL_GRAY256](#): Creates grayscale palettes with varying levels.
- [IDM_PAL_RGB222](#) to [IDM_PAL_RGB488](#): Creates RGB palettes with different bit depths.

- **IDM_PAL_OPT_POP4** to **IDM_PAL_OPT_MEDCUT**: Creates palettes using various optimization algorithms.

WM_USER_DELETEDIB:

This **user-defined message deletes the existing DIB**. It is used in preparation for loading a new DIB, pasting from the clipboard, or other operations that involve acquiring a new image.

WM_USER_DELETEPAL:

This **user-defined message deletes the existing palette**. It is used in preparation for defining a new palette. If a bitmap is associated with the current palette, it is also deleted.

WM_USER_CREATEPAL:

This **user-defined message creates a new palette based on the current DIB**. If wParam is TRUE, it also creates a new device-dependent bitmap (DDB) using the current palette. The specific palette creation method depends on the capabilities of the device context (hdc).

In summary, this **part of the code manages the creation and deletion of palettes**, responds to user commands related to palettes, and ensures the UI is updated accordingly.

The **use of user-defined messages** (WM_USER_DELETEDIB, WM_USER_DELETEPAL, WM_USER_CREATEPAL) adds clarity to the code structure, making it easier to understand and maintain.

The **remaining portion of the Dibble program** handles the WM_PAINT, WM_QUERYNEWPALETTE, WM_PALETTECHANGED, and WM_DESTROY messages, finalizing the application's functionality.

WM_PAINT:

In response to the WM_PAINT message, **the program begins painting by obtaining the device context (hdc) using BeginPaint**. If a palette is present (hPalette), it is selected and

realized to ensure proper color mapping. The DIB or DDB is then displayed on the window using the DisplayDib function. Finally, the painting is concluded with EndPaint.

WM_QUERYNEWPALETTE:

This message is triggered when the window is queried about accepting a new palette. If a palette is present (hPalette), it is selected, realized, and the window is invalidated to trigger a repaint with the new colors.

WM_PALETTECHANGED:

This message is sent when the system palette has changed. If a palette is present (hPalette), it is selected, realized, and the colors are updated using the UpdateColors function. This ensures that the colors in the application are consistent with the system palette.

WM_DESTROY:

Upon receiving the WM_DESTROY message, the program performs cleanup operations. It deletes the DIB, DDB, and palette objects if they exist. The PostQuitMessage function is then called to signal the termination of the message loop.

The WndProc function appears to be a central part of the application's logic, responsible for managing the window and user interactions.

It's the core of the Dibble program, responding to various messages to handle user input, update the display, and manage resources such as DIBs and palettes. It orchestrates the behavior of the program's main window in response to user actions and system events.

DIBBLE.RC FILE BREAKDOWN

The provided excerpt is from the resource script (DIBBLE.RC) of the Dibble program, defining the menu structure and accelerators. Let's break down its functionality:

Menu Structure:

The menu is divided into several pop-up menus, such as "File," "Edit," "Show," "Palette," "Gray Shades," "Uniform Colors," "Optimized," "Convert," and "Help."

Each pop-up menu contains specific menu items related to the corresponding functionality of the program.

File Menu:

Contains items for opening, saving, printing, viewing properties, and exiting the application.

Accelerators (Ctrl+O, Ctrl+S, Ctrl+P) are associated with file-related actions for quick access.

Edit Menu:

Includes options for cutting, copying, pasting, deleting, flipping, and rotating.

Accelerators (Ctrl+X, Ctrl+C, Ctrl+V, Delete) are defined for common editing operations.

Show Menu:

Provides options for displaying the image in different ways, such as actual size, centered, stretched to the window, and isotropically stretched.

Palette Menu:

Offers various palette options for displaying the image, including none, DIB color table, halftone, all-purpose, gray shades, uniform colors, and optimized algorithms.

Gray Shades and Uniform Colors Submenus:

Submenus under "Gray Shades" and "Uniform Colors" present different predefined grayscale and color palette options, respectively.

Optimized Submenu:

Contains options for optimized color palette algorithms, including the popularity algorithm with different bit depths and the median cut algorithm.

Convert Menu:

Provides options for converting the image to different color depths, ranging from 1 bit per pixel to 32 bits per pixel.

Help Menu:

Contains a single item, "About," providing information about the Dibble program.

Accelerators:

Defines [keyboard shortcuts \(accelerators\)](#) for some common actions, such as copying (Ctrl+C), opening (Ctrl+O), printing (Ctrl+P), saving (Ctrl+S), pasting (Ctrl+V), deleting (Delete), and cutting (Ctrl+X).

IDM Constants:

Assigns numerical identifiers to menu items for easy reference in the code.

To summarise, the resource script defines the structure of the program's menu, including various options related to file operations, editing, display, palette selection, conversion, and help.

[Accelerators](#) are provided for quick keyboard access to common actions. The IDM constants facilitate the identification of menu items in the program's code.

RESOURCE.H FOR DIBBLE PROGRAM

The excerpt from [RESOURCE.H](#) provides numerical identifiers ([IDM constants](#)) for various menu items used in the Dibble program.

These [identifiers are crucial for referencing menu items](#) in the program's code. The comments in the code describe the purpose of each constant, such as file operations,

editing options, display modes, palette selections, conversion choices, and help-related actions.

Now, let's dive into the notes regarding the DIBBLE program's functionality and key static variables in its WndProc function:

Conversion and Palette Handling Files:

- DIBBLE utilizes two sets of files, DIBCONV (DIBCONV.C and DIBCONV.H) and DIBPAL (DIBPAL.C and DIBPAL.H).
- DIBCONV files handle the conversion between different pixel formats, such as converting from 24 bits per pixel to 8 bits per pixel.
- DIBPAL files are responsible for creating palettes used in displaying the images.

Static Variables in WndProc:

The WndProc function in DIBBLE maintains three crucial static variables:

- **hdib (HDIB handle):** Represents the handle for a DIB (Device-Independent Bitmap). It is obtained from various functions in the DIBHELP module.
- **hPalette (HPALETTE handle):** Represents the handle for a palette. It is obtained from functions in DIBPAL or the standard CreateHalftonePalette function.
- **hBitmap (HBITMAP handle):** Represents the handle for a bitmap. It is obtained from the DibCopyToDdb function in DIBHELP.C. This handle accelerates screen displays, especially in 256-color video modes.

Dynamic Recreation of Handles:

The notes highlight that the HBITMAP handle (hBitmap) needs to be re-created under certain conditions:

- Whenever the program creates a new DIB Section.
- Whenever the program creates a different palette.

Functional Overview:

The notes suggest approaching the understanding of DIBBLE functionally rather than sequentially. This implies focusing on the functional aspects and interactions of different modules rather than a strict chronological order of code execution.

In essence, the notes provide insights into the modular structure of DIBBLE, the purpose of DIBCONV and DIBPAL files, and the critical static variables (hdib, hPalette, and hBitmap) in the WndProc function, emphasizing the importance of **re-creating the HBITMAP handle under specific circumstances** for optimal performance.

MORE NOTES ON DIBBLE PROGRAM FOR UNDERSTANDING

File Loading and Saving:

DIBBLE facilitates the loading and saving of DIB files through the following mechanisms:

Loading (File Open):

- In response to the IDM_FILE_LOAD command, DIBBLE utilizes common file dialog boxes using GetOpenFileName.
- Prior to loading a new DIB, DIBBLE deletes the existing HDIB, palette, and bitmap objects. This cleanup involves sending a WM_USER_DELETEDIB message, processed by calls to DibDelete and DeleteObject.
- Subsequently, DIBBLE calls DibFileLoad from the DIBHELP module, and it sends WM_USER_SETSCROLLS and WM_USER_CREATEPAL messages to reset scroll bars

and create a new palette. The WM_USER_CREATEPAL message also involves creating a new DDB from the DIB section.

Saving (File Save):

- When handling the IDM_FILE_SAVE command, DIBBLE directly calls DibFileSave to save the current DIB.

Display, Scrolling, and Printing:

- DIBBLE provides various options for displaying DIBs, scrolling within the client area, and printing:

Display Options:

- DIBBLE's menu allows users to choose between different display options, such as actual size, centered, stretched to fill the client area, or isotropically stretched while maintaining the aspect ratio.

DisplayDib Function:

- During the WM_PAINT message and File Print command processing, DIBBLE calls the DisplayDib function.
- DisplayDib employs BitBlt and StretchBlt for display rather than SetDIBitsToDevice and StretchDIBits.
- The bitmap handle passed during WM_PAINT is created by DibCopyToDdb during WM_USER_CREATEPAL, ensuring compatibility with the video device context.
- For File Print command processing, DisplayDib is called with the DIB section handle from DibBitmapHandle in DIBHELP.C.

fHalftonePalette Variable:

- DIBBLE maintains a static BOOL variable named fHalftonePalette, set to TRUE if hPalette is obtained from CreateHalftonePalette.
- This variable influences the behavior of the DisplayDib function. If fHalftonePalette is TRUE, StretchBlt is used even for actual size display, and the DIB section handle is passed during WM_PAINT, not the bitmap handle.

Scrolling:

- For the first time in sample programs, DIBBLE introduces scrolling for DIBs in the client area.
- Scroll bars are visible only when the DIB is displayed in actual size.

- WndProc passes the current scroll bar positions to the DisplayDib function during WM_PAINT.

These functionalities enhance the versatility of DIBBLE in handling DIB files, providing flexible display options, and introducing scrolling capabilities within the client area. The use of fHalftonePalette ensures proper display and printing with halftone palettes.

Clipboard Operations:

Cut and Copy:

- DIBBLE employs the DibCopyToPackedDib function in DIBHELP for the Cut and Copy menu items.
- This function consolidates all components of the DIB into a single memory block for clipboard operations.

Paste:

- For the first time in the sample programs, DIBBLE introduces pasting a DIB from the clipboard.
- Pasting involves a call to DibCopyFromPackedDib, replacing the existing HDIB, palette, and bitmap stored by the window procedure.

Flipping and Rotating:

Flip:

- The Flip menu option utilizes the DibFlipHorizontal function in DIBBLE.C.
- This function obtains an exact copy of the DIB using DibCopy and then enters a loop to flip the pixels upside down.
- It calls DibGetPixel and DibSetPixel from DIBHELP.C during the pixel copy process.

Rotate:

- The Rotate menu option, specifically Rotate Right, uses the DibRotateRight function.
- This function calls DibCopy with a second argument set to TRUE, flipping the width and height of the original DIB to create a new DIB.
- Pixel bits are not copied by DibCopy; instead, the function uses six different loops for each possible DIB pixel width (1 bit, 4 bit, 8 bit, 16 bit, 24 bit, and 32 bit).
- Notably, DibRotateRight employs the faster DibGetPixel and DibSetPixel macros from DIBHELP.H.

Additional Insights:

- ✓ Both flipping and rotating functions highlight the usage of DibGetPixel and DibSetPixel for pixel manipulation.
- ✓ The speed difference between using functions and macros is illustrated, with the Rotate function opting for macros to enhance performance.
- ✓ DIBBLE, being a demonstration program, showcases these operations for educational purposes, although a [full-fledged application might implement additional transformations](#) directly.
- ✓ These functionalities enhance DIBBLE's capabilities, allowing users to [perform basic image transformations](#) and clipboard operations seamlessly.

DIBPAL.C PROGRAM, HEADER AND EXPLANATION

```
// Header file for DIBPAL.C

HPALETTE DibPalDibTable(HDIB hdib);
HPALETTE DibPalAllPurpose(void);
HPALETTE DibPalUniformGrays(int iNum);
HPALETTE DibPalUniformColors(int iNumR, int iNumG, int iNumB);
HPALETTE DibVga(void);
HPALETTE DibPalPopularity(HDIB hdib, int iRes);
HPALETTE DibPalMedianCut(HDIB hdib, int iRes);
```

Notes:

- ✓ These functions provide various palette creation options for DIBBLE.
- ✓ They are used for displaying DIBs on 256-color video displays.
- ✓ The halftone palette is created separately using a Windows function call.

Function Breakdown:

- [DibPalDibTable](#): Creates a palette directly from the DIB's color table.
- [DibPalAllPurpose](#): Creates a general-purpose palette with a wide range of colors.
- [DibPalUniformGrays](#): Creates a palette with a specified number of evenly spaced gray shades.
- [DibPalUniformColors](#): Creates a palette with evenly distributed colors across red, green, and blue components.
- [DibPalVga](#): Creates a palette based on the standard VGA color set.
- [DibPalPopularity](#): Creates a palette optimized for the DIB's colors, emphasizing frequently used colors.
- [DibPalMedianCut](#): Creates a palette using a median cut algorithm for color quantization, aiming for accurate color representation.

The first portion of the code DibPal.c...

```

Hpalette DibPalDibTable(HDIB hdib)
{
    Hpalette hPalette;
    int i, iNum;
    LOGPALETTE *plp;
    RGBQUAD rgb;

    if (0 == (iNum = DibNumColors(hdib)))
        return NULL;

    plp = malloc(sizeof(LOGPALETTE) + (iNum - 1) * sizeof(PALETTEENTRY));
    plp->palVersion = 0x0300;
    plp->palNumEntries = iNum;

    for (i = 0; i < iNum; i++)
    {
        DibGetColor(hdib, i, &rgb);
        plp->palPalEntry[i].peRed = rgb.rgbRed;
        plp->palPalEntry[i].peGreen = rgb.rgbGreen;
        plp->palPalEntry[i].peBlue = rgb.rgbBlue;
        plp->palPalEntry[i].peFlags = 0;
    }

    hPalette = CreatePalette(plp);
    free(plp);

    return hPalette;
}

```

The provided portion of the DIBPAL.C code defines the [DibPalDibTable function](#), responsible for creating a palette from the color table of a Device Independent Bitmap (DIB) in the DIBBLE application.

The [DibPalDibTable function](#) begins by [taking an HDIB \(Handle to DIB\)](#) as its parameter. This handle likely represents a DIB structure, which is a common image format used in Windows programming.

The [function initializes variables](#), including hPalette, which will hold the created palette. It then determines the number of colors in the DIB using the DibNumColors function, a utility function likely present in the dibhelp.h file.

If the DIB has zero colors (indicating an issue), the function returns NULL, signifying the failure to create a palette. Otherwise, it proceeds to allocate memory for a LOGPALETTE structure, which is a data structure used to define a color palette. The size of the structure is calculated based on the number of colors in the DIB.

A [LOGPALETTE structure](#) contains information such as the palette version, the number of entries in the palette, and an array of PALETTEENTRY structures, each representing a color

in the palette. In this case, the PALETTEENTRY structures are filled with RGB values obtained from the DIB's color table.

The [function iterates over each color in the DIB's color table](#), retrieves the RGB components using DibGetColor, and stores them in the corresponding PALETTEENTRY structure. The peFlags member is set to zero, indicating no special attributes for each color.

After populating the LOGPALETTE structure, the function creates a palette using the CreatePalette function, passing the initialized structure. Finally, the allocated memory for the LOGPALETTE structure is freed using free.

Here's a breakdown of the code again:

- ❖ It declares the necessary variables, including hPalette (the resulting palette handle), i and iNum (used for iteration), plp (a pointer to a LOGPALETTE structure), and rgb (to store RGB color information).
- ❖ It checks the number of colors in the DIB using DibNumColors function. If the number is 0, indicating no colors or an invalid DIB, it returns NULL.
- ❖ It allocates memory for the plp structure, which will store the color information for the palette. The size of the memory block is calculated based on the number of colors in the DIB.
- ❖ It sets the palVersion to 0x0300 and palNumEntries to the number of colors in the DIB in the plp structure.
- ❖ It iterates over each color entry in the DIB using a loop. For each entry, it retrieves the color information using DibGetColor, and assigns the corresponding RGB values to the palPalEntry array of plp.
- ❖ It sets peFlags to 0 for each color entry.
- ❖ It creates a palette using the CreatePalette function, passing the plp structure.
- ❖ It frees the memory allocated for plp using free.
- ❖ It returns the resulting palette handle (hPalette).

The second portion of the code...

```

HPALETTE DibPalAllPurpose(void)
{
    HPALETTE hPalette;
    int i, incr, R, G, B;
    LOGPALETTE *plp;

    // Allocate memory for the LOGPALETTE structure
    plp = malloc(sizeof(LOGPALETTE) + 246 * sizeof(PALETTEENTRY));
    plp->palVersion = 0x0300; // Set the palette version
    plp->palNumEntries = 247; // Set the number of palette entries

    // The first loop calculates 31 gray shades with a peculiar pattern
    for (i = 0, G = 0, incr = 8; G <= 0xFF; i++, G += incr)
    {
        // Set RGB values for the gray shades
        plp->palPalEntry[i].peRed = (BYTE)G;
        plp->palPalEntry[i].peGreen = (BYTE)G;
        plp->palPalEntry[i].peBlue = (BYTE)G;
        plp->palPalEntry[i].peFlags = 0;

        // Alternate the increment value in a pattern
        incr = (incr == 9 ? 8 : 9);
    }

    // The second loop is responsible for creating 216 entries with RGB combinations
    for (R = 0; R <= 0xFF; R += 0x33)
        for (G = 0; G <= 0xFF; G += 0x33)
            for (B = 0; B <= 0xFF; B += 0x33)
            {
                // Set RGB values for the 216 entries
                plp->palPalEntry[i].peRed = (BYTE)R;
                plp->palPalEntry[i].peGreen = (BYTE)G;
                plp->palPalEntry[i].peBlue = (BYTE)B;
                plp->palPalEntry[i].peFlags = 0;
                i++;
            }

    // Create the palette using the LOGPALETTE structure
    hPalette = CreatePalette(plp);

    // Free the allocated memory for the LOGPALETTE structure
    free(plp);

    // Return the created palette
    return hPalette;
}

```

The **DibPalAllPurpose** function within the DIBPAL.C file is responsible for creating a versatile palette suitable for a broad range of images.

The resulting palette contains 247 entries, and it strategically combines 31 gray shades and 216 RGB combinations. To achieve this, the function uses a LOGPALETTE structure, allocating memory for the structure and the corresponding PALETTEENTRY entries.

The first part of the function focuses on generating 31 gray shades with a distinct pattern.

- ❖ The loop iterates through varying shades of gray, setting the RGB values accordingly. An interesting feature is the alternation of the increment value between 8 and 9, creating a specific pattern in the gray shades.

The second part of the function is dedicated to creating 216 RGB entries.

- ❖ Three nested loops traverse through different values of red, green, and blue components, respectively. For each combination, the RGB values are set, expanding the palette with a diverse set of colors. It's noteworthy that these combinations avoid duplicates with the standard 20 colors.

After populating the LOGPALETTE structure with the desired entries, the function calls the CreatePalette function to generate the palette. Once the palette is created, the function deallocates the previously allocated memory for the LOGPALETTE structure to prevent memory leaks.

In summary, **DibPalAllPurpose** is designed to provide a versatile palette that accommodates a wide array of images by incorporating a thoughtfully chosen set of gray shades and RGB combinations. The resulting palette is well-suited for applications that require a diverse yet balanced color representation.

The third portion of the program...

```
HPALETTE DibPalUniformGrays(int iNum)
{
    HPALETTE hPalette;
    int i;
    LOGPALETTE* plp;

    plp = malloc(sizeof(LOGPALETTE) + (iNum - 1) * sizeof(PALETTEENTRY));
    plp->palVersion = 0x0300;
    plp->palNumEntries = iNum;

    for (i = 0; i < iNum; i++)
    {
        plp->palPalEntry[i].peRed = plp->palPalEntry[i].peGreen = plp->palPalEntry[i].peBlue
= (BYTE)(i * 255 / (iNum - 1));
        plp->palPalEntry[i].peFlags = 0;
    }

    hPalette = CreatePalette(plp);
    free(plp);

    return hPalette;
}
```

The function DibPalUniformGrays creates a palette of iNum grays that are uniformly spaced. It uses the LOGPALETTE structure to store the palette entries.

Here's a breakdown of the code:

- ❖ It declares the necessary variables, including hPalette (the resulting palette handle), i (used for iteration), and plp (pointer to the LOGPALETTE structure).
- ❖ It allocates memory for the plp structure, taking into account the number of palette entries needed.
- ❖ It sets the palVersion to 0x0300 and palNumEntries to iNum in the plp structure.
- ❖ It loops over i from 0 to iNum - 1 and assigns the same value to peRed, peGreen, and peBlue in the palPalEntry array, creating uniformly spaced grays. The value is calculated based on the loop index, ranging from 0 to 255.
- ❖ It sets peFlags to 0 for each palette entry.
- ❖ It creates a palette using the CreatePalette function, passing the plp structure.
- ❖ It frees the allocated memory for plp using free.
- ❖ It returns the resulting palette handle (hPalette).
- ❖ This function can be used to create a palette with a desired number of uniformly spaced gray shades for various applications, such as image processing or color mapping.

The **DibPalUniformGrays** function in the DIBPAL.C file is designed to create a palette consisting of a specified number of uniformly spaced gray shades.

The function takes an integer parameter **iNum** representing the desired number of gray shades in the palette. It dynamically allocates memory for a **LOGPALETTE** structure along with the necessary **PALETTEENTRY** entries.

In the implementation, the function uses a **loop** to iteratively calculate and assign **RGB values** for each gray shade in the palette.

The **RGB components—red, green, and blue—are set to the same value**, resulting in various shades of gray. The calculation ensures that the gray shades are uniformly spaced across the entire spectrum from black to white.

Once the **LOGPALETTE** structure is populated with the specified number of gray shades, the function calls the **CreatePalette** function to generate the actual palette. Following the creation of the palette, the function frees the dynamically allocated memory to prevent memory leaks.

In summary, **DibPalUniformGrays** is a utility function that facilitates the creation of a palette with a user-defined number of uniformly spaced gray shades.

This function can be **beneficial in scenarios where a gradient of gray tones is needed**, offering a simple and efficient way to generate such a palette for diverse imaging applications.

The fourth portion of the code...

```
HPALETTE DibPalUniformColors(int iNumR, int iNumG, int iNumB)
{
    HPALETTE hPalette;
    int i, iNum, R, G, B;
    LOGPALETTE* plp;

    iNum = iNumR * iNumG * iNumB;
    plp = malloc(sizeof(LOGPALETTE) + (iNum - 1) * sizeof(PALETTEENTRY));
    plp->palVersion = 0x0300;
    plp->palNumEntries = iNumR * iNumG * iNumB;
    i = 0;

    for (R = 0; R < iNumR; R++)
    {
        for (G = 0; G < iNumG; G++)
        {
            for (B = 0; B < iNumB; B++)
            {
                plp->palPalEntry[i].peRed = (BYTE)(R * 255 / (iNumR - 1));
                plp->palPalEntry[i].peGreen = (BYTE)(G * 255 / (iNumG - 1));
                plp->palPalEntry[i].peBlue = (BYTE)(B * 255 / (iNumB - 1));
                plp->palPalEntry[i].peFlags = 0;
                i++;
            }
        }
    }

    hPalette = CreatePalette(plp);
    free(plp);

    return hPalette;
}
```

The [DibPalUniformColors](#) function creates a palette of colors by combining different levels of red, green, and blue (RGB) components. The resulting palette will have iNumR levels of red, iNumG levels of green, and iNumB levels of blue.

Here's a breakdown of the code:

- ❖ It declares the necessary variables, including hPalette (the resulting palette handle), i, iNum, R, G, and B.
- ❖ It calculates the total number of colors iNum by multiplying the levels of red, green, and blue: $iNum = iNumR * iNumG * iNumB$.
- ❖ It allocates memory for the plp structure, taking into account the number of palette entries needed.
- ❖ It sets the palVersion to 0x0300 and palNumEntries to $iNumR * iNumG * iNumB$ in the plp structure.
- ❖ It initializes i to 0.
- ❖ It uses nested loops to iterate over all combinations of the red, green, and blue values. For each combination, it assigns the respective RGB values to the corresponding palette entry in the plp structure. The RGB values are calculated based on the current loop indices and the specified number of levels for each component.
- ❖ It sets peFlags to 0 for each palette entry.
- ❖ It creates a palette using the CreatePalette function, passing the plp structure.
- ❖ It frees the allocated memory for plp using free.
- ❖ It returns the resulting palette handle (hPalette).

This [function can be used to create a palette with a desired number of colors](#) by specifying the number of levels for each RGB component. The resulting palette can be used for various purposes, such as computer graphics, image processing, or color mapping

Repeating for clarity:

The [DibPalUniformColors function in the DIBPAL.C file](#) is designed to generate a palette containing a specified number of colors based on the input parameters iNumR, iNumG, and iNumB. These parameters represent the desired number of color variations along the red, green, and blue axes, respectively.

The [function begins by calculating the total number of colors \(iNum\)](#) that will be present in the palette, which is the product of the individual variations along each axis. It then dynamically allocates memory for a LOGPALETTE structure along with the required number of PALETTEENTRY entries to store the RGB values for each color.

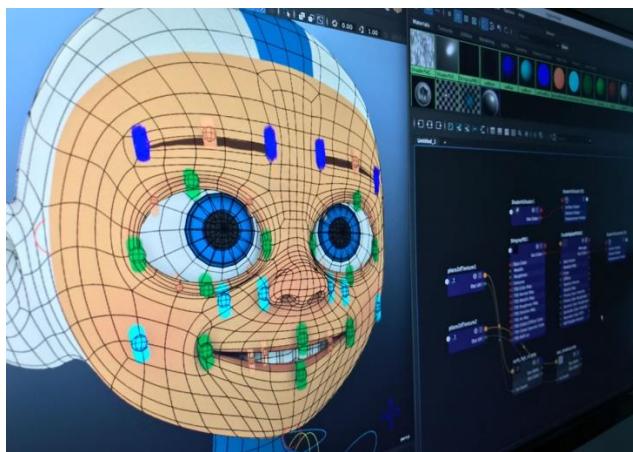
The implementation utilizes nested loops to iterate over the specified variations along the red, green, and blue axes. Within these loops, RGB values are calculated based on the current iteration and the total number of variations along each axis. The calculated RGB values are then assigned to the corresponding PALETTEENTRY structure in the LOGPALETTE.

After populating the LOGPALETTE structure with the RGB values for all the colors, the function calls the CreatePalette function to generate the actual palette. Following the creation of the palette, the function frees the dynamically allocated memory to prevent memory leaks.

In summary, DibPalUniformColors is a utility function that facilitates the creation of a palette with a specified number of colors, allowing for variations along the red, green, and blue axes. This function can be useful in scenarios where a diverse color palette is needed, providing a straightforward way to generate such palettes for various graphical applications.

The DibPalUniformColors function can be helpful in various use cases where you need to generate a palette of colors with specific characteristics. Here are a few examples:

Computer Graphics: This function can be used in computer graphics applications where you need to generate a palette of colors for rendering images, animations, or graphical user interfaces. By specifying the number of levels for each RGB component, you can create a palette that meets your specific color requirements.



Data Visualization: When visualizing data, it can be useful to have a palette of colors that are evenly distributed and easily distinguishable. This function allows you to create such palettes by specifying the number of levels for each RGB component. You can use these

palettes to represent different data categories or levels in a visually appealing and informative way.



Color Mapping: In image processing or computer vision applications, you may need to map colors from one range to another. The resulting palette can be used as a lookup table (LUT) for color mapping operations. By associating each color index with a specific RGB value, you can efficiently remap colors in an image or apply color transformations.

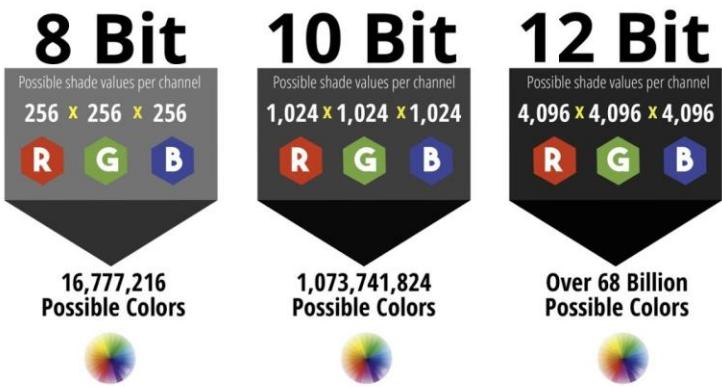


When [using the resulting palette](#) in a computer graphics application, you typically associate each color index with a pixel value or a specific graphical element. The palette can be used to determine the color of each element based on its index.

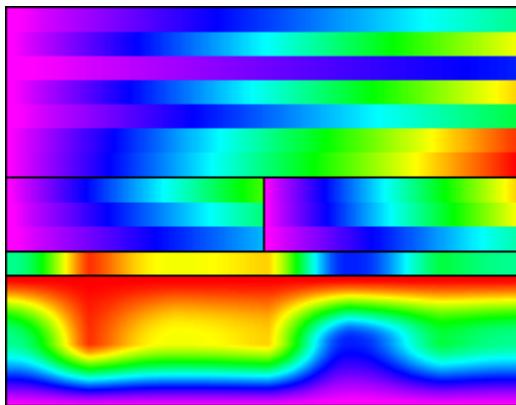
For example, if you're [rendering an image using indexed color mode](#), you would assign a color index to each pixel of the image, and then use the palette to look up the RGB value associated with that index to determine the pixel's color.

It's important to keep in mind some limitations and considerations when using this function:

Limited Color Depth: The resulting palette is limited by the number of levels specified for each RGB component. If you use a small number of levels, the generated palette may not capture the full range of colors, resulting in a reduced color depth.



Palette Size: The size of the resulting palette is determined by the product of the number of levels for each RGB component. Creating palettes with a large number of colors may consume significant memory resources.



Color Repetition: Depending on the combination of levels for each RGB component, you may encounter color repetition in the resulting palette. This means that different color indices may have the same RGB values, resulting in limited color variation.



Compatibility: The usage of palettes in computer graphics applications may vary depending on the specific graphics library or framework you are using. Ensure that the palette creation and usage are compatible with your chosen graphics API.



Consider these factors based on your specific requirements and the limitations of the graphics system you are working with to ensure optimal results when using the DibPalUniformColors function.

The fifth portion of the code...

```

/* Creates a palette based on standard 16 VGA colors */
HPALETTE DibPalVga(void)
{
    static RGBQUAD rgb[16] = {
        {0x00, 0x00, 0x00, 0x00}, // Black
        {0x00, 0x00, 0x80, 0x00}, // Blue
        {0x00, 0x80, 0x00, 0x00}, // Green
        {0x00, 0x80, 0x80, 0x00}, // Cyan
        {0x80, 0x00, 0x00, 0x00}, // Red
        {0x80, 0x00, 0x80, 0x00}, // Magenta
        {0x80, 0x80, 0x00, 0x00}, // Brown
        {0x80, 0x80, 0x80, 0x00}, // Light Gray
        {0xC0, 0xC0, 0xC0, 0x00}, // Dark Gray
        {0x00, 0x00, 0xFF, 0x00}, // Light Blue
        {0x00, 0xFF, 0x00, 0x00}, // Light Green
        {0x00, 0xFF, 0xFF, 0x00}, // Light Cyan
        {0xFF, 0x00, 0x00, 0x00}, // Light Red
        {0xFF, 0x00, 0xFF, 0x00}, // Light Magenta
        {0xFF, 0x00, 0x00, 0x00}, // Yellow
        {0xFF, 0xFF, 0xFF, 0x00} // White
    };

    HPALETTE hPalette;
    int i;
    LOGPALETTE* plp;

    plp = malloc(sizeof(LOGPALETTE) + 15 * sizeof(PALETTEENTRY));
    plp->palVersion = 0x0300;
    plp->palNumEntries = 16;

    for (i = 0; i < 16; i++)
    {
        plp->palPalEntry[i].peRed = rgb[i].rgbRed;
        plp->palPalEntry[i].peGreen = rgb[i].rgbGreen;
        plp->palPalEntry[i].peBlue = rgb[i].rgbBlue;
        plp->palPalEntry[i].peFlags = 0;
    }

    hPalette = CreatePalette(plp);
    free(plp);

    return hPalette;
}

```

The [DibPalVga](#) function creates a palette based on the standard 16 VGA colors. Each color is represented by an RGBQUAD structure that contains the red, green, blue, and reserved components. The function then creates a palette using the LOGPALETTE structure and the color data.

Here's a breakdown of the code:

- ❖ It declares the necessary variables, including `rgb` (the array of `RGBQUAD` structures), `hPalette` (the resulting palette handle), `i`, and `plp` (the pointer to the `LOGPALETTE` structure).
- ❖ It initializes the `rgb` array with the 16 VGA colors by specifying the RGB values for each color.
- ❖ It allocates memory for the `plp` structure, taking into account the number of palette entries needed (16 in this case).
- ❖ It sets the `palVersion` to `0x0300` and `palNumEntries` to 16 in the `plp` structure.
- ❖ It uses a loop to iterate over the 16 VGA colors. For each color, it assigns the respective RGB values to the corresponding palette entry in the `plp` structure. The `peFlags` field is set to 0 for each palette entry.
- ❖ It creates a palette using the `CreatePalette` function, passing the `plp` structure.
- ❖ It frees the allocated memory for `plp` using `free`.
- ❖ It returns the resulting palette handle (`hPalette`).

This function is specifically designed to [create a palette based on the standard 16 VGA colors](#). The resulting palette can be used in computer graphics applications to achieve a consistent and compatible color representation across different systems or platforms that support VGA colors.

In-Depth notes:

The [DibPalVga function in the DIBPAL.C file is designed to create a palette based on the standard 16 VGA colors](#). This palette is commonly used in graphics applications to ensure compatibility with a broad range of display devices. The function utilizes a predefined array of `RGBQUAD` structures named `rgb` to represent the RGB values of the 16 VGA colors.

The [rgb array contains 16 entries](#), each representing one of the standard VGA colors. Each entry consists of four components: `rgbRed`, `rgbGreen`, `rgbBlue`, and a reserved component set to 0. These components define the intensity of the red, green, and blue channels for each color. The colors include black, blue, green, cyan, red, magenta, yellow, white, and various shades of gray.

The [function begins by dynamically allocating memory for a LOGPALETTE structure](#) along with space for 15 `PALETTEENTRY` structures. The `LOGPALETTE` structure is used to store information about the palette, including the version and the number of entries. In this case, it specifies a version of `0x0300` and a total of 16 entries.

A [loop is then used to populate the PALETTEENTRY structures](#) within the `LOGPALETTE` with the RGB values from the `rgb` array. The function assigns the red, green, and blue

components to the corresponding fields and sets the reserved component to 0. This loop iterates over the 16 standard VGA colors.

After filling the LOGPALETTE structure, the function calls the CreatePalette function to create the actual palette based on the specified information. The resulting palette represents the standard VGA colors.

Finally, the dynamically allocated memory is freed using the free function to prevent memory leaks.

In summary, DibPalVga is a utility function that generates a palette containing the standard 16 VGA colors, making it suitable for applications that require compatibility with a broad range of display devices.

Some examples of applications that commonly use the standard 16 VGA colors palette include:

Retro Games: Many classic video games from the 1980s and early 1990s were designed to run on systems with VGA graphics capabilities. These games often utilized the standard 16 VGA colors palette to achieve a consistent and recognizable visual style.



Emulators: Emulators that simulate vintage computer systems or game consoles often include support for the standard 16 VGA colors palette. This allows users to experience the authentic graphics of the original systems on modern hardware.



Terminal Emulators: Terminal emulators, which are programs that simulate text-based terminals, may use the standard 16 VGA colors palette for displaying text and colored output. This palette provides a simple and widely supported set of colors for terminal applications.



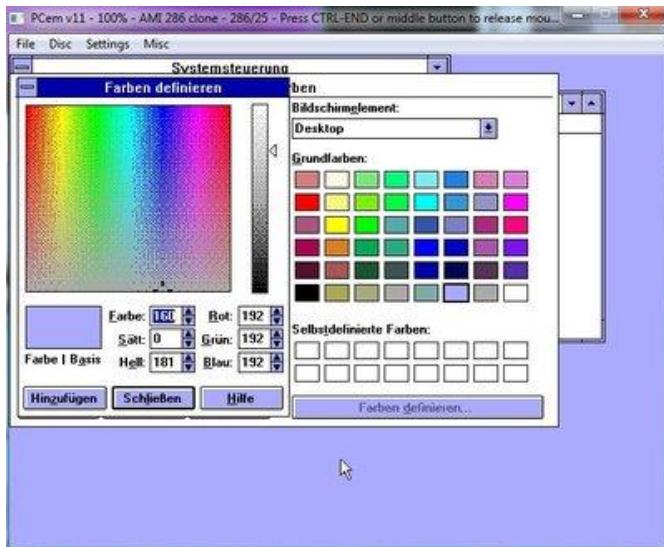
```
itsfoss@itsfoss-B250M-DS3H:~$ sudo apt update
Hit:1 https://brave-browser-apt-release.s3.brave.com stable InRelease
Hit:2 http://ppa.launchpad.net/sandromani/gimagerreader/ubuntu focal InRelease
Hit:3 http://archive.ubuntu.com/ubuntu focal InRelease
Hit:4 http://archive.canonical.com/ubuntu focal InRelease
Ign:5 http://packages.linuxmint.com ulyssa InRelease
Hit:6 http://security.ubuntu.com/ubuntu focal-security InRelease
Hit:7 http://archive.ubuntu.com/ubuntu focal-updates InRelease
Hit:8 http://packages.linuxmint.com ulyssa Release
Hit:9 https://repo.protonvpn.com/debian unstable InRelease
Hit:10 http://archive.ubuntu.com/ubuntu focal-backports InRelease
Building package lists... Done
Building dependency tree
Reading state information... Done
All packages are up to date.
itsfoss@itsfoss-B250M-DS3H:~$
```

Graphics Demos: Graphics demos or visualizations that aim to recreate the look and feel of retro computer graphics may rely on the standard 16 VGA colors palette to achieve an authentic retro aesthetic.



While the standard 16 VGA colors palette has its benefits, there are some limitations and drawbacks to consider:

Limited Color Range: The standard 16 VGA colors palette consists of a fixed set of colors. It does not support the full range of colors available in modern graphics systems. Consequently, using this palette restricts the available color choices and can lead to a limited color range in graphics applications.

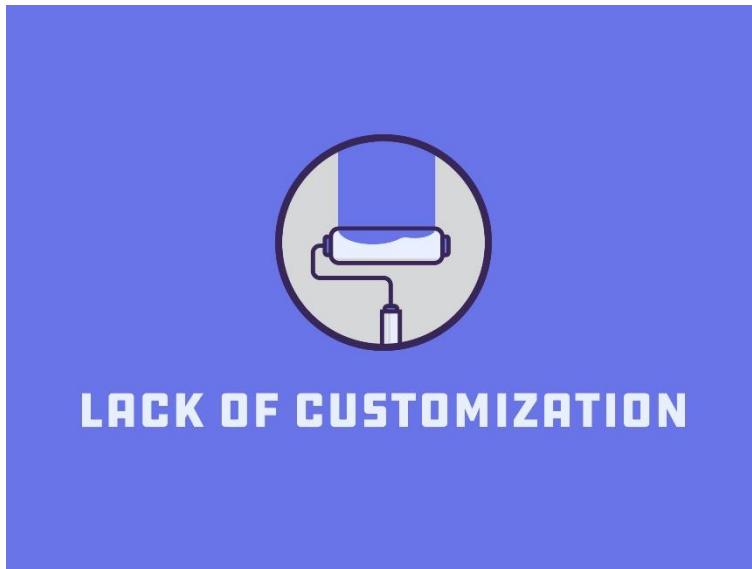


Incompatibility with High-Color or True-Color Modes: The standard 16 VGA colors palette is primarily designed for 8-bit color modes. If a graphics application is running in a high-color or true-color mode, the use of the standard VGA palette may not provide optimal color representation.



TRUE COLOR TECHNOLOGY

Lack of Customization: The colors in the standard VGA palette cannot be easily modified or customized. If an application requires specific or unique colors beyond the standard 16 VGA colors, an alternative palette or color model may be more suitable.



An example of how the DibPalVga function would be called in a graphics application using the Windows API is as follows:

```
#include <windows.h>

int main()
{
    HPALETTE hPalette;
    // ... other code ...

    // Create the VGA palette
    hPalette = DibPalVga();

    // Use the palette in your graphics application
    // ... code to select and use the palette ...

    // Cleanup
    DeleteObject(hPalette);

    return 0;
}
```

In this example, the [DibPalVga](#) function is called to create the VGA palette, and the resulting palette handle (hPalette) is stored.

The application can then use the palette by selecting it into a device context or applying it to the desired graphics objects. Finally, the palette is released using DeleteObject to free the associated resources.

The sixth portion of the code...

```

HPALETTE DibPalPopularity(HDIB hdib, int iRes) {
    HPALETTE hPalette;
    int i, iArraySize, iEntry, iCount, iIndex, iMask, R, G, B;
    int *piCount;
    LOGPALETTE *plp;

    if (DibBitCount(hdib) < 16) return NULL;
    if (iRes < 3 || iRes > 8) return NULL;

    iArraySize = 1 << (3 * iRes);
    iMask = (1 << iRes) - 1;

    if (NULL == (piCount = calloc(iArraySize, sizeof(int)))) return NULL;

    AccumColorCounts(hdib, piCount, iRes);

    plp = malloc(sizeof(LOGPALETTE) + 235 * sizeof(PALETTEENTRY));
    plp->palVersion = 0x0300;

    for (iEntry = 0; iEntry < 236; iEntry++) {
        for (i = 0, iCount = 0; i < iArraySize; i++) {
            if (piCount[i] > iCount) {
                iCount = piCount[i];
                iIndex = i;
            }
        }

        if (iCount == 0) break;

        R = (iMask & iIndex) << (8 - iRes);
        G = (iMask & (iIndex >> iRes)) << (8 - iRes);
        B = (iMask & (iIndex >> (iRes + iRes))) << (8 - iRes);

        plp->palPalEntry[iEntry].peRed = (BYTE)R;
        plp->palPalEntry[iEntry].peGreen = (BYTE)G;
        plp->palPalEntry[iEntry].peBlue = (BYTE)B;
        plp->palPalEntry[iEntry].peFlags = 0;
        piCount[iIndex] = 0;
    }

    plp->palNumEntries = iEntry;

    hPalette = CreatePalette(plp);

    free(piCount);
    free(plp);

    return hPalette;
}

```

1. Validity Checks:

The function first ensures that the provided DIB has a compatible color depth (16 bits or higher) and that the specified color resolution (iRes) falls within the valid range of 3 to 8 bits per color channel.

2. Color Counting Array Preparation:

It calculates the required size of an array to store color counts based on the color resolution (iArraySize). It creates a mask (iMask) to isolate color components within pixel values. It allocates memory for the color count array (piCount) and initializes it to zeros.

3. Color Frequency Counting:

The function calls AccumColorCounts (not shown) to analyze the DIB's pixels and increment the corresponding counts in the piCount array. This tracks how often each color appears in the image.

4. Palette Construction:

- ❖ It allocates memory for a LOGPALETTE structure, which holds palette entries.
- ❖ It iterates through the piCount array to identify the most frequently used colors:
- ❖ For each palette entry to be created:
 - ❖ It scans piCount to find the color with the highest count.
 - ❖ It extracts the red, green, and blue components of the color using the iMask.
 - ❖ It creates a PALETTEENTRY structure with the extracted color values and adds it to the LOGPALETTE.
 - ❖ It resets the count for the added color to zero in piCount to avoid duplicates.

5. Palette Creation and Cleanup:

- ❖ The function sets the final number of palette entries in the LOGPALETTE.
- ❖ It calls CreatePalette to generate a Windows palette handle (hPalette) from the LOGPALETTE data.
- ❖ It frees the allocated memory for piCount and plp.
- ❖ It returns the created palette handle.

This function aims to [create an optimized palette](#) by prioritizing the most prevalent colors in the DIB, leading to better color representation when displaying the image on systems with limited color palettes.

The seventh portion of the code...

```
/* Structures used for implementing median cut algorithm */
typedef struct {
    int Rmin, Rmax, Gmin, Gmax, Bmin, Bmax;
} MINMAX;

typedef struct {
    int iBoxCount;
    RGBQUAD rgbBoxAv;
} BOXES;

/* FindAverageColor: In a box */
static int FindAverageColor(int *piCount, MINMAX mm, int iRes, RGBQUAD *prgb) {
    int R, G, B, iR, iG, iB, iTotal, iCount;

    // Initialize some variables
    iTotal = iR = iG = iB = 0;

    // Loop through all colors in the box
    for (R = mm.Rmin; R <= mm.Rmax; R++) {
        for (G = mm.Gmin; G <= mm.Gmax; G++) {
            for (B = mm.Bmin; B <= mm.Bmax; B++) {
                // Get the number of pixels of that color
                iCount = piCount[PACK_RGB(R, G, B, iRes)];

                // Weight the pixel count by the color value
                iR += iCount * R;
                iG += iCount * G;
                iB += iCount * B;
                iTotal += iCount;
            }
        }
    }

    // Find the average color
    prgb->rgbRed = (BYTE)((iR / iTotal) << (8 - iRes));
    prgb->rgbGreen = (BYTE)((iG / iTotal) << (8 - iRes));
    prgb->rgbBlue = (BYTE)((iB / iTotal) << (8 - iRes));

    // Return the total number of pixels in the box
    return iTotal;
}
```

The code snippet provided implements the `FindAverageColor` function, which calculates the average color within a given box using the [median cut algorithm](#). Here's an explanation of the code:

Structures:

- The code defines two structures: `MINMAX` and `BOXES`.
- `MINMAX` is used to represent the dimensions (minimum and maximum values) of a color box.
- `BOXES` is used for storing information about each color box during the median cut algorithm.

FindAverageColor function:

This function calculates the average color within a given box based on the color distribution stored in `piCount`.

It takes the following parameters:

- `piCount`: An array storing the count of pixels for each color in the box.
- `mm`: A `MINMAX` structure representing the dimensions of the box.
- `iRes`: The color resolution, which determines the number of bits per channel.
- `prgb`: A pointer to the `RGBQUAD` structure where the average color will be stored.

Color Calculation:

The function initializes variables `iTotal`, `iR`, `iG`, and `iB` to zero.

It then [loops through all colors within the box](#), iterating over each possible combination of red (R), green (G), and blue (B) values.

Within the nested loops, the function retrieves the pixel count (`iCount`) for the current color from the `piCount` array.

It then weights the pixel count by the color value and [accumulates the weighted sums](#) of red (`iR`), green (`iG`), and blue (`iB`) values, as well as the total pixel count (`iTotal`).

Average Color Calculation:

After the nested loops, the [function calculates the average color](#) by dividing the accumulated sums of red, green, and blue values by the total pixel count.

The color values are shifted left by $8 - iRes$ bits to adjust for the color resolution and then cast to BYTE type before assigning them to the prgb structure.

Return:

The function returns the total number of pixels in the box (iTTotal).

The [FindAverageColor function](#) is part of a larger implementation of the median cut algorithm, which involves dividing color space into boxes based on the color distribution in the image.

This function calculates the average color within each box, which can be used for various purposes such as color quantization or palette creation.

[Here's more information about the MINMAX structure and an overview of how the median cut algorithm works:](#)

Purpose of the MINMAX structure:

The MINMAX structure is used in the code to represent the dimensions (minimum and maximum values) of a color box.

It contains six members: Rmin, Rmax, Gmin, Gmax, Bmin, and Bmax, which correspond to the minimum and maximum values for the red, green, and blue color channels.

The purpose of this structure is to define the boundaries of a color box in the RGB color space.

Median cut algorithm:

The median cut algorithm is a color quantization algorithm that divides the color space into smaller boxes based on the distribution of colors in an image.

The goal is to reduce the number of distinct colors while maintaining the overall visual quality as much as possible.

Here's an overview of how the algorithm works:

Initialization:

Start with a single-color box that encompasses the entire color space.

Selection of the cutting dimension:

Choose the color channel (red, green, or blue) that has the largest range within the current color box.

This channel will be used to split the color box.

Finding the median:

Sort the colors within the box based on the selected color channel.

Find the median color, which is the color that divides the sorted colors into two equal halves.

Splitting the color box:

Divide the current color box into two smaller boxes along the selected color channel, using the median color as the splitting point.

The two resulting boxes will each have a narrower range along the selected color channel.

Repeat:

Repeat steps 2-4 for each of the resulting boxes until a desired number of color boxes or color levels is reached.

Average color calculation:

Once the desired number of color boxes is obtained, calculate the average color within each box using the FindAverageColor function (as shown in the code snippet).

The average color represents the dominant color for each box.

Color quantization:

Replace the colors in the image with the average colors of the corresponding color boxes.

This reduces the number of distinct colors in the image while preserving the overall visual appearance.

The median cut algorithm is a popular technique used in various applications, such as image compression, color reduction, and palette generation. It provides a way to approximate the original image with a reduced number of colors while minimizing visual degradation.

CUTBOX FUNCTION IN-DEPTH

The code provided is an implementation of the CutBox function, which is a recursive function used within the median cut algorithm to divide a given color box into smaller boxes. Let's break down the code and explain its functioning in more detail.

The CutBox function takes several parameters as input:

- **piCount**: An array storing the count of pixels for each color in the box.
- **iBoxCount**: The total number of pixels in the current box.
- **mm**: A MINMAX structure representing the dimensions (minimum and maximum values) of the current box.
- **iRes**: The color resolution, which determines the number of bits per channel.
- **iLevel**: The nesting level of the current box.
- **pboxes**: A pointer to the BOXES structure where information about each color box will be stored.
- **piEntry**: A pointer to an integer representing the index of the current box in the pboxes array.

Now, let's dive into the functioning of the CutBox function step by step:

Checking for termination conditions:

The function first checks if the current box is empty (`iBoxCount == 0`). If so, it simply returns, as there is nothing to divide.

Next, it checks if the nesting level has reached 8 or if the box contains only one pixel. In either case, the algorithm has reached a leaf node, and the average color within the box is calculated using the `FindAverageColor` function.

The average color and the number of pixels in the box are then stored in the pboxes array, and the piEntry index is incremented.

Splitting along the blue channel:

If the termination conditions are not met, the algorithm proceeds to check if the blue channel has the largest range within the current box ($mm.Bmax - mm.Bmin > mm.Rmax - mm.Rmin$ and $mm.Bmax - mm.Bmin > mm.Gmax - mm.Gmin$).

If the blue channel is the largest, the algorithm initializes a counter iCount and enters a loop that iterates over the blue values within the box ($B = mm.Bmin; B < mm.Bmax; B++$).

Within the nested loops, the algorithm accumulates the pixel count for each successive blue value, iterating over all possible combinations of red (R), green (G), and blue (B) values.

If the accumulated pixel count iCount is greater than or equal to half the total number of pixels in the box ($iCount \geq iBoxCount/2$), the algorithm breaks out of the loop, indicating that the splitting point along the blue channel has been found.

The algorithm then recursively calls the CutBox function to divide the current box into two new boxes: one with the blue values ranging from $mm.Bmin$ to B , and the other with the blue values ranging from $B + 1$ to $mm.Bmax$. The new box counts and dimensions (mmNew) are updated accordingly.

Splitting along the red channel:

If the blue channel is not the largest, the algorithm checks if the red channel has the largest range ($mm.Rmax - mm.Rmin > mm.Gmax - mm.Gmin$).

If the red channel is the largest, the algorithm follows a similar procedure as in the previous step. It initializes iCount and enters a loop over the red values within the box ($R = mm.Rmin; R < mm.Rmax; R++$).

Within the nested loops, it accumulates the pixel count for each successive red value, iterating over all possible combinations of red (R), green (G), and blue (B) values.

If the accumulated pixel count iCount is greater than or equal to half the total number of pixels in the box, the loop is terminated, and the splitting point along the red channel is found.

The algorithm then recursively calls the CutBox function to split the current box into two new boxes along the red channel, similar to the previous step.

Splitting along the green channel:

If neither the blue channel nor the red channel has the largest range, the algorithm proceeds to split along the green channel.

It follows a similar procedure as in the previous steps, accumulating the pixel count for each successive green value and finding the splitting point along the green channel.

The current box is divided into two new boxes along the green channel, and the CutBox function is recursively called for both new boxes.

The CutBox function continues to recursively divide the color boxes until the termination conditions are met, resulting in a hierarchical structure of nested boxes representing different regions of the color space. The average color within each box is calculated and stored, providing a means to approximate the average color of a region in the image.

Overall, the **CutBox function implements the key logic of the median cut algorithm**, which recursively partitions a color space based on the distribution of colors in a given image. By dividing the color space into smaller boxes, the algorithm aims to represent the image with a reduced number of dominant colors while preserving visual quality.

Tell me more. What is the purpose of the 'FindAverageColor' function in the 'CutBox' algorithm? How does the 'CutBox' function determine the splitting point along the blue channel? Can you explain how the 'CutBox' function handles the case when neither the blue nor the red channel has the largest range?

Let's delve into more details about the FindAverageColor function, how the CutBox function determines the splitting point along the blue channel, and how it handles the case when neither the blue nor the red channel has the largest range.

FindAverageColor function:

The purpose of the FindAverageColor function is to calculate the average color within a given box. It takes the following parameters:

- **piCount:** An array storing the count of pixels for each color in the box.
- **mm:** The dimensions (minimum and maximum values) of the box.
- **iRes:** The color resolution, which determines the number of bits per channel.
- **rgbBoxAv:** A pointer to an RGB structure where the average color will be stored.

The function iterates over all possible combinations of red (R), green (G), and blue (B) values within the box's dimensions.

For each combination, it calculates the corresponding index in the piCount array using the PACK_RGB macro, which packs the color components into a single integer.

It then accumulates the pixel count for that color combination by accessing the piCount array.

After iterating over all color combinations, the function divides the accumulated counts by the total number of pixels in the box to calculate the average color.

Finally, it stores the average color in the rgbBoxAv structure.

Splitting along the blue channel:

To determine the splitting point along the blue channel, the CutBox function iterates over the blue values within the box's range ($B = \text{mm.Bmin}; B < \text{mm.Bmax}; B++$).

Within each iteration, it accumulates the pixel count for each successive blue value by nested loops iterating over the red (R) and green (G) values within their respective ranges.

The algorithm keeps track of the accumulated pixel count in the iCount variable.

If the accumulated count iCount is greater than or equal to half the total number of pixels in the box ($iCount \geq iBoxCount/2$), it means that the splitting point along the blue channel has been found.

At that point, the algorithm breaks out of the loop, indicating that further iterations are not needed.

The algorithm then proceeds to divide the current box into two new boxes: one with the blue values ranging from mm.Bmin to the current value of B, and the other with the blue values ranging from B + 1 to mm.Bmax.

The new box counts and dimensions (mmNew) are updated accordingly, and the CutBox function is recursively called for both new boxes.

Handling the case when neither the blue nor the red channel has the largest range:

When neither the blue nor the red channel has the largest range, the algorithm determines that the green channel must have the largest range.

Similar to the previous steps, the CutBox function iterates over the green values within the box's range ($G = \text{mm.Gmin}; G < \text{mm.Gmax}; G++$).

Within each iteration, it accumulates the pixel count for each successive green value by nested loops iterating over the blue (B) and red (R) values within their respective ranges.

The algorithm keeps track of the accumulated pixel count in the iCount variable.

If the accumulated count iCount is greater than or equal to half the total number of pixels in the box, it indicates that the splitting point along the green channel has been found.

The algorithm breaks out of the loop, and the current box is divided into two new boxes along the green channel, similar to the previous steps.

The new box counts and dimensions (mmNew) are updated accordingly, and the CutBox function is recursively called for both new boxes.

By iteratively splitting the color boxes based on the largest channel range, the CutBox algorithm effectively partitions the color space, allowing the median cut algorithm to approximate the dominant colors in the image efficiently.

Tenth portion of the code...

```

1  /* Compare routine for qsort */
2  static int Compare(const BOXES *pbox1, const BOXES *pbox2)
3  {
4      return pbox1->iBoxCount - pbox2->iBoxCount;
5  }
6
7  /* DibPalMedianCut: Creates palette based on median cut algorithm */
8  HPALETTE DibPalMedianCut(HDIB hdib, int iRes)
9  {
10     BOXES boxes[256];
11     HPALETTE hPalette;
12     int i, iArraySize, iCount, R, G, B, iTotCount, iDim, iEntry = 0;
13     int *piCount;
14     LOGPALETTE *plp;
15     MINMAX mm;
16
17     // Validity checks
18     if (DibBitCount(hdib) < 16)
19         return NULL;
20     if (iRes < 3 || iRes > 8)
21         return NULL;
22
23     // Accumulate counts of pixel colors
24     iArraySize = 1 << (3 * iRes);
25     if (NULL == (piCount = calloc(iArraySize, sizeof(int))))
26         return NULL;
27     AccumColorCounts(hdib, piCount, iRes);
28
29     // Find the dimensions of the total box
30     iDim = 1 << iRes;
31     mm.Rmin = mm.Gmin = mm.Bmin = iDim - 1;
32     mm.Rmax = mm.Gmax = mm.Bmax = 0;
33     iTotCount = 0;
34
35     for (R = 0; R < iDim; R++)
36         for (G = 0; G < iDim; G++)
37             for (B = 0; B < iDim; B++)
38                 if ((iCount = piCount[PACK_RGB(R, G, B, iRes)]) > 0)
39                 {
40                     iTotCount += iCount;
41                     if (R < mm.Rmin) mm.Rmin = R;
42                     if (G < mm.Gmin) mm.Gmin = G;
43                     if (B < mm.Bmin) mm.Bmin = B;
44                     if (R > mm.Rmax) mm.Rmax = R;
45                     if (G > mm.Gmax) mm.Gmax = G;
46                     if (B > mm.Bmax) mm.Bmax = B;
47                 }
48
49     // Cut the first box (iterative function).
50     // On return, the boxes structure will have up to 256 RGB values,
51     // one for each of the boxes, and the number of pixels in
52     // each box.
53     // The iEntry value will indicate the number of non-empty boxes.
54     CutBox(piCount, iTotCount, mm, iRes, 0, boxes, &iEntry);
55     free(piCount);
56
57     // Sort the RGB table by the number of pixels for each color
58     qsort(boxes, iEntry, sizeof(BOXES), Compare);
59
60     plp = malloc(sizeof(LOGPALETTE) + (iEntry - 1) * sizeof(PALETTEENTRY));
61     if (plp == NULL)
62         return NULL;
63
64     plp->palVersion = 0x0300;
65     plp->palNumEntries = iEntry;
66
67     for (i = 0; i < iEntry; i++)
68     {
69         plp->palPalEntry[i].peRed = boxes[i].rgbBoxAv.rgbRed;
70         plp->palPalEntry[i].peGreen = boxes[i].rgbBoxAv.rgbGreen;
71         plp->palPalEntry[i].peBlue = boxes[i].rgbBoxAv.rgbBlue;
72         plp->palPalEntry[i].peFlags = 0;
73     }
74
75     hPalette = CreatePalette(plp);
76     free(plp);
77
78     return hPalette;
79 }

```

The **Compare** function is a comparison routine used by the `qsort` function to sort the `BOXES` array based on the number of pixels (`iBoxCount`) in each box. It returns the difference between the `iBoxCount` of two `BOXES` structures.

The **DibPalMedianCut** function is the main function that creates a palette based on the Median Cut algorithm. It takes an input image (`hdib`) and the desired resolution (`iRes`) as parameters and returns an `HPALETTE` handle representing the created palette.

Validity checks are performed to ensure that the input image has at least 16 bits per pixel and that the resolution (`iRes`) is within the range of 3 to 8. If any of these checks fail, the function returns `NULL`.

The **AccumColorCounts** function is called to accumulate the counts of pixel colors in the input image. It takes the input image, an array `piCount` to store the color counts, and the resolution (`iRes`) as parameters. This step prepares the data for the Median Cut algorithm.

An array of `BOXES` structures called `boxes` is created to store the resulting color boxes. Each `BOXES` structure represents a box in the color space and contains information about the box's dimensions and the average color within the box.

Memory is allocated for an integer array `piCount` with a size `iArraySize` to store the accumulated color counts. The size of the array is calculated as $2^{(3 * iRes)}$ since each color component (R, G, B) can have 2^{iRes} possible values.

The dimensions of the total color space (the entire image) are determined by iterating over all possible values of R, G, and B. The minimum and maximum values of R, G, and B are tracked using the `mm` structure. The total count of pixels (`iTotCount`) is also calculated.

The **CutBox** function is called to perform the iterative process of dividing the color space into smaller boxes.

It takes the accumulated color counts (`piCount`), the total count of pixels (`iTotCount`), the dimensions of the total box (`mm`), the resolution (`iRes`), and initial parameters for the iteration.

The function populates the boxes array with the resulting boxes and updates the `iEntry` variable to indicate the number of non-empty boxes.

The `piCount` array is freed since it is no longer needed.

The `qsort` function is called to sort the boxes array based on the number of pixels (`iBoxCount`) in each box. The `Compare` function is used as the comparison routine for sorting.

Memory is allocated for a `LOGPALETTE` structure `plp` to store the RGB values of the palette entries. The number of entries is set to `iEntry`.

The `LOGPALETTE` structure is populated with the RGB values from the boxes array.

The [CreatePalette](#) function is called to create a palette based on the LOGPALETTE structure. The resulting palette handle is stored in hPalette. Memory allocated for plp is freed. The function returns the created palette handle (hPalette).

The [DibPalMedianCut](#) function implements the Median Cut algorithm to create a palette based on an input image. It divides the color space into smaller boxes, sorts the boxes based on the number of pixels in each box, and creates a palette using the average colors from the sorted boxes.

SUMMARY

In summary, the provided information discusses various aspects related to palette creation and displaying DIBs (Device Independent Bitmaps) in different color modes. Here are the key points:

Palette Creation: The code implements different functions for palette creation. The DibPalDibTable function creates a palette from the DIB's color table. If the DIB doesn't have a color table, the CreateHalftonePalette function is called to create a halftone palette. The DibPalAllPurpose function creates an all-purpose palette that can be used for displaying images in 256-color modes.



Controlling Colors in 256-Color Mode: When displaying bitmaps in 256-color modes, the palette used by Windows determines the colors used for displaying the image. By creating and realizing a specific palette, you can control the colors used. For example, creating a palette with shades of gray can result in a high-contrast "chalk and charcoal" effect.

000000	005f00	008700	00af00	00d700	00ff00	5fff00	5fd700	5faf00	5f8700	5f5f00	5f0000
016	022	028	034	040	046	082	076	070	064	058	052
00005f	005f5f	00875f	00af5f	00d75f	00ff5f	5fff5f	5fd75f	5faf5f	5f875f	5f5f5f	5f005f
017	023	029	035	041	047	083	077	071	065	059	053
000087	005f87	008787	00af87	00d787	00ff87	5fff87	5fd787	5faf87	5f8787	5f5f87	5f0087
018	024	030	036	042	048	084	078	072	066	060	054
0000af	005faf	0087af	00afaf	00d7af	00ffaf	5fffff	5fd7af	5fafaf	5f87af	5f5faf	5f00af
019	025	031	037	043	049	085	079	073	067	061	055
0000d7	005fd7	0087d7	00afd7	00d7d7	00ffd7	5ffffd7	5fd7d7	5fafd7	5f87d7	5f5fd7	5f00d7
020	026	032	038	044	050	086	080	074	068	062	056
0000ff	005fff	0087ff	00afff	00d7ff	00ffff	5fffff	5fd7ff	5fafff	5f87ff	5f5fff	5f00ff
021	027	033	039	045	051	087	081	075	069	063	057
8700ff	875fff	8787ff	87afff	87d7ff	87ffff	afffff	afd7ff	afafff	af87ff	af5fff	af00ff
093	099	105	111	117	123	159	153	147	141	135	129
8700d7	875fd7	8787d7	87af7d	87d7d7	87ffd7	aффd7	afd7d7	afaf7d	af87d7	af5fd7	af00d7
092	098	104	110	116	122	158	152	146	140	134	128
8700af	875faf	8787af	87afaf	87d7af	87ffaf	aффaf	afd7af	afafaf	af87af	af5faf	af00af
091	097	103	109	115	121	157	151	145	139	133	127
870087	875f87	878787	87af87	87d787	87ffd87	aфф87	afd787	afaf87	af8787	af5f87	af0087
090	096	102	108	114	120	156	150	144	138	132	126
87005f	875f5f	87875f	87af5f	87d75f	87ffd5f	aфф5f	afd75f	afaf5f	af875f	af5f5f	af005f
089	095	101	107	113	119	155	149	143	137	131	125
870000	875f00	878700	87af00	87d700	87ff00	aфф00	afd700	afaf00	af8700	af5f00	af0000
088	094	100	106	112	118	154	148	142	136	130	124
d70000	d75f00	d78700	dfa000	dfdf00	dfff00	ffff00	ffdf00	ffaф00	ff8700	ff5f00	ff0000
160	166	172	178	184	190	226	220	214	208	202	196
d7005f	d75f5f	d7875f	dfa5f	dfdf5f	dfff5f	ffff5f	ffdf5f	ffaф5f	ff875f	ff5f5f	ff005f
161	167	173	179	185	191	227	221	215	209	203	197
d70087	d75f87	d78787	dfa87	dfdf87	dfff87	ffff87	ffdf87	ffaф87	ff8787	ff5f87	ff0087
162	168	174	180	186	192	228	222	216	210	204	198
d700af	d75faf	d787af	dfafaf	dfdfaf	dfffaf	ffffaf	ffdfaf	ffaфaf	ff87af	ff5faf	ff00af
163	169	175	181	187	193	229	223	217	211	205	199
d700d7	d75fd7	d787d7	dfafdf	dfdfdf	dfffdf	ffffdf	ffdfdf	ffaфdf	ff87df	ff5fdf	ff00df
164	170	176	182	188	194	230	224	218	212	206	200
d700ff	d75fff	d787ff	dfafff	dfdfff	dfffff	ffffff	ffdfff	ffaфff	ff87ff	ff5fff	ff00ff
165	171	177	183	189	195	231	225	219	213	207	201
080808	121212	1clclc	262626	303030	3a3a3a	444444	4e4e4e	585858	626262	6c6c6c	767676
232	233	234	235	236	237	238	239	240	241	242	243
eeeeee	e4e4e4	dadada	d0d0d0	c6c6c6	bcbcbc	b2b2b2	a8a8a8	9e9e9e	949494	8a8a8a	808080
255	254	253	252	251	250	249	248	247	246	245	244
000000	800000	008000	808000	000080	800080	008080	c0c0c0	xterm-256-color chart			
000	001	002	003	004	005	006	007				
808080	ff0000	00ff00	ffff00	0000ff	ff00ff	00ffff	ffffff				
008	009	010	011	012	013	014	015				

xterm-256-color chart

Spooned carefully into SVG by Jason Miknis in 2012
 this SVG is in the public domain
<https://gist.github.com/2868891> · contains the dataset in yaml

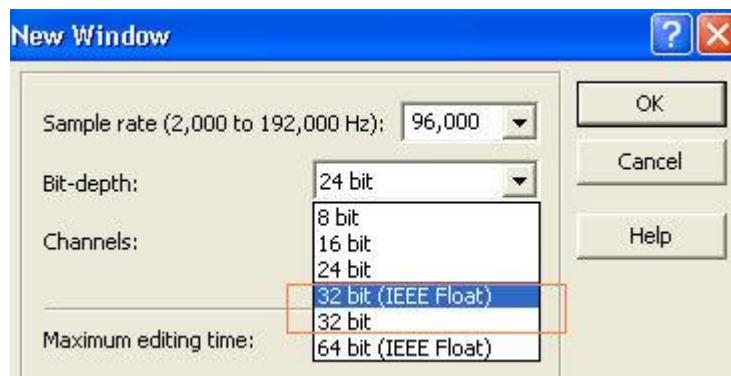
Gray Shades: Different numbers of gray shades can be used in a palette to achieve different effects. Using two shades of gray (black and white) creates high contrast. Adding more shades of gray improves the image quality, reducing contouring artifacts. The number of shades depends on the display equipment, with 64 gray shades often considered the limit for most devices.

GRAY COLOR NAMES

GRAY #828282	FOSSIL #787276	MINK #88807B	PEARL RIVER #D9DDDC
ABALONE #D6CFC7	HARBOR GRAY #C7C6C1	SMOKE #BEBDB8	THUNDER #BDB7AB
PEWTER #999DAO	STEEL #777B7E	STONE #877F7D	IRON #48494B
RHINO #B9BBB6	TROUT #97978F	SEAL #818380	LAVA #808588
SHADOW #363636	ASH #544C4A	ANCHOR #3E424B	CHARCOAL #222021

GRAF1X.COM

Challenges with 16-bit, 24-bit, and 32-bit DIBs in 8-bit Mode: Displaying higher bit-depth DIBs in 8-bit color modes presents challenges. Creating an all-purpose palette or using the halftone palette can be insufficient for accurately representing the colors in the image. The process of matching each pixel to the closest static color in the palette can be time-consuming and result in loss of quality.



Optimal Palette: To display higher bit-depth DIBs in 8-bit mode, converting them to 8-bit DIBs with an "optimal palette" is recommended. An optimal palette is a palette of 256

colors that closely approximates the color range of the original image. Quantization techniques, such as color image quantization, can be used to find an optimal palette.



DibPalCreateUniformColors:

This function [creates a palette based on uniform ranges of RGB primaries](#). One approach involves using 8 levels of red and green and 4 levels of blue.

The resulting palette consists of all possible combinations of these color values, resulting in 256 colors.

Another approach is to use 6 levels of red, green, and blue, [resulting in a palette of 216 colors](#). DIBBLE provides these options and others.

The "Popularity" Algorithm:

The [popularity algorithm involves finding the 256 most common RGB color values](#) in the bitmap by examining all the pixels.

These common colors are then used to create the palette.

However, [using the full 24 bits per color can be memory-intensive](#), so using only the most significant bits (e.g., 6 bits) is often sufficient. This reduces the memory usage while still providing a reasonable number of colors.

The "Median Cut" Algorithm:

The DibPalMedianCut function implements Paul Heckbert's "median cut" algorithm. This algorithm works by **envisioning the RGB color space as a cube** and finding the box that encloses all the pixels in the image.

The **algorithm recursively splits the boxes based on the longest dimension**, creating smaller boxes. Eventually, 256 boxes are obtained, each containing approximately an equal number of pixels. The RGB color values of the pixels within each box are averaged to determine the palette colors.

Octree Quantization:

Octree quantization is another **technique for creating an optimal palette**. It involves representing the RGB color space as an octree data structure.

The octree is recursively divided into smaller cubes until each cube contains a limited number of colors. The average color of each cube is then used for the palette. This technique provides good results but is not explicitly discussed in the provided information.

DIBCONV.C PROGRAM

```
1  HDIB DibConvert(HDIB hdibSrc, int iBitCountDst)
2  {
3      HDIB hdibDst;
4      HPALETTE hPalette;
5      int i, x, cy, cx, iBitCountSrc, cColors;
6      PALETTEENTRY pe;
7      RGBQUAD rgb;
8      WORD wNumEntries;
9
10     cx = DibWidth(hdibSrc);
11     cy = DibHeight(hdibSrc);
12     iBitCountSrc = DibBitCount(hdibSrc);
13
14     if (iBitCountSrc == iBitCountDst)
15         return NULL;
16
17     // DIB with color table to DIB with larger color table:
18     if ((iBitCountSrc < iBitCountDst) && (iBitCountDst <= 8))
19     {
20         cColors = DibNumColors(hdibSrc);
21         hdibDst = DibCreate(cx, cy, iBitCountDst, cColors);
22
23         for (i = 0; i < cColors; i++)
24         {
25             DibGetColor(hdibSrc, i, &rgb);
26             DibSetColor(hdibDst, i, &rgb);
27         }
28
29         for (x = 0; x < cx; x++)
30             for (y = 0; y < cy; y++)
31             {
32                 DibSetPixel(hdibDst, x, y, DibGetPixel(hdibSrc, x, y));
33             }
34     }
35     // Any DIB to DIB with no color table
36     else if (iBitCountDst >= 16)
37     {
38         hdibDst = DibCreate(cx, cy, iBitCountDst, 0);
39
40         for (x = 0; x < cx; x++)
41             for (y = 0; y < cy; y++)
42             {
43                 DibGetPixelColor(hdibSrc, x, y, &rgb);
44                 DibSetPixelColor(hdibDst, x, y, &rgb);
45             }
46     }
47     // DIB with no color table to 8-bit DIB
48     else if (iBitCountSrc >= 16 && iBitCountDst == 8)
49     {
50         hPalette = DibPalMedianCut(hdibSrc, 8);
51         GetObject(hPalette, sizeof(WORD), &wNumEntries);
52         hdibDst = DibCreate(cx, cy, 8, wNumEntries);
53
54         for (i = 0; i < (int)wNumEntries; i++)
55         {
56             GetPaletteEntries(hPalette, i, 1, &pe);
57             rgb.rgbRed = pe.peRed;
58             rgb.rgbGreen = pe.peGreen;
59             rgb.rgbBlue = pe.peBlue;
60             rgb.rgbReserved = 0;
61             DibSetColor(hdibDst, i, &rgb);
62         }
63
64         for (x = 0; x < cx; x++)
65             for (y = 0; y < cy; y++)
66             {
67                 DibGetPixelColor(hdibSrc, x, y, &rgb);
68                 DibSetPixel(hdibDst, x, y, GetNearestPaletteIndex(hPalette, RGB(rgb.rgbRed, rgb.rgbGreen, rgb.rgbBlue)));
69             }
70
71         DeleteObject(hPalette);
72     }
73     // Any DIB to monochrome DIB
74     else if (iBitCountDst == 1)
75     {
76         hdibDst = DibCreate(cx, cy, 1, 0);
77         hPalette = DibPalUniformGrays(2);
78
79         for (i = 0; i < 2; i++)
80         {
81             GetPaletteEntries(hPalette, i, 1, &pe);
82             rgb.rgbRed = pe.peRed;
83             rgb.rgbGreen = pe.peGreen;
84             rgb.rgbBlue = pe.peBlue;
85             rgb.rgbReserved = 0;
86             DibSetColor(hdibDst, i, &rgb);
87         }
88
89         for (x = 0; x < cx; x++)
90             for (y = 0; y < cy; y++)
91             {
92                 DibGetPixelColor(hdibSrc, x, y, &rgb);
93                 DibSetPixel(hdibDst, x, y, GetNearestPaletteIndex(hPalette, RGB(rgb.rgbRed, rgb.rgbGreen, rgb.rgbBlue)));
94             }
95
96         DeleteObject(hPalette);
97     }
98     // All non-monochrome DIBs to 4-bit DIB
99     else if (iBitCountSrc >= 8 && iBitCountDst == 4)
100    {
101        hdibDst = DibCreate(cx, cy, 4, 0);
102        hPalette = DibPalVga();
103
104        for (i = 0; i < 16; i++)
105        {
106            GetPaletteEntries(hPalette, i, 1, &pe);
107            rgb.rgbRed = pe.peRed;
108            rgb.rgbGreen = pe.peGreen;
109            rgb.rgbBlue = pe.peBlue;
110            rgb.rgbReserved = 0;
111            DibSetColor(hdibDst, i, &rgb);
112        }
113
114        for (x = 0; x < cx; x++)
115            for (y = 0; y < cy; y++)
116            {
117                DibGetPixelColor(hdibSrc, x, y, &rgb);
118                DibSetPixel(hdibDst, x, y, GetNearestPaletteIndex(hPalette, RGB(rgb.rgbRed, rgb.rgbGreen, rgb.rgbBlue)));
119            }
120
121        DeleteObject(hPalette);
122    }
123    // Should not be necessary
124    else
125        hdibDst = NULL;
126
127    return hdibDst;
128 }
```

To see zoom and the code will be very clear or go to page 770 of the book.

```
1  /* DIBCONV.H header file for DIBCONV.C */  
2  
3  HDIB DibConvert(HDIB hdibSrc, int iBitCountDst);
```

The given code is a function named "DibConvert" that converts a device-independent bitmap (DIB) from one format to another. Let's break down the code and explain each section in paragraphs.

The function takes two parameters: "hdibSrc" is the handle to the source DIB that needs to be converted, and "iBitCountDst" is the desired bit count of the destination DIB.

The first step in the code is to retrieve some information about the source DIB, such as its width, height, and bit count. This information is necessary for the conversion process.

The code then checks if the bit count of the source DIB is the same as the desired bit count of the destination DIB. If they are the same, it means no conversion is needed, and the function returns NULL.

Next, the code checks different conversion scenarios based on the bit counts of the source and destination DIBs.

DIB with color table to DIB with a larger color table:

If the bit count of the source DIB is less than the bit count of the destination DIB (but both are 8 bits or less), the code creates a new destination DIB with the desired bit count and copies the color table from the source DIB to the destination DIB. Then, it iterates over each pixel of the source DIB and copies it to the corresponding location in the destination DIB.

Any DIB to DIB with no color table:

If the destination DIB has 16 bits or more, which means it does not have a color table, the code creates a new destination DIB with the desired bit count and no color table. It then iterates over each pixel of the source DIB, retrieves the color of the pixel, and sets the corresponding color in the destination DIB.

DIB with no color table to 8-bit DIB:

If the bit count of the source DIB is 16 bits or more and the destination DIB has a bit count of 8, the code creates a new destination DIB with an 8-bit color depth.

It uses the "DibPalMedianCut" function to generate a palette for the destination DIB based on the colors in the source DIB. The code retrieves the palette entries, sets them in the destination DIB, and then performs a similar pixel-by-pixel conversion as in the previous scenarios.

Any DIB to monochrome DIB:

If the destination DIB has a bit count of 1, the code creates a new destination DIB with a monochrome color depth. It uses the "DibPalUniformGrays" function to generate a grayscale palette with two entries. The code retrieves the palette entries, sets them in the destination DIB, and performs the pixel conversion similar to the previous scenarios.

All non-monochrome DIBs to 4-bit DIB:

If the source DIB has a bit count of 8 or more and the destination DIB has a bit count of 4, the code creates a new destination DIB with a 4-bit color depth. It uses the "DibPalVga" function to generate a 16-color VGA palette. The code retrieves the palette entries, sets them in the destination DIB, and performs the pixel conversion.

Finally, if none of the above conversion scenarios apply, which should not be necessary, the function sets the destination DIB handle to NULL.

In summary, the "DibConvert" function handles various scenarios for converting DIBs between different color depths and formats. It takes into account color tables, palettes, and pixel-by-pixel conversion to achieve the desired conversion.

Tell me more. What are the different scenarios in which the "DibConvert" function is used? Can you explain how the function handles the conversion from a DIB with a color table to a DIB with a larger color table? How does the function handle the conversion from any DIB to a monochrome DIB?

The "DibConvert" function is used to convert device-independent bitmaps (DIBs) from one format to another. It covers several scenarios based on the bit count of the source and destination DIBs. The function supports the following conversions:

DIB with a color table to a DIB with a larger color table:

This scenario occurs when the [bit count of the source DIB is less](#) than the bit count of the destination DIB, and both DIBs have 8 bits or less. The function creates a new destination DIB with the desired bit count and the same color table size as the source DIB.

It then [copies the color table entries](#) from the source DIB to the destination DIB. Next, it iterates over each pixel of the source DIB, retrieves its color, and sets the corresponding pixel in the destination DIB.

Any DIB to a DIB with no color table:

If the [destination DIB has a bit count of 16 or more](#), it indicates that the destination DIB does not have a color table. In this case, the function creates a new destination DIB with the desired bit count and no color table.

It then [iterates over each pixel of the source DIB](#), retrieves the color of the pixel, and sets the corresponding color in the destination DIB.

DIB with no color table to a monochrome DIB:

When the destination DIB has a bit count of 1, it implies that the desired conversion is to a monochrome DIB. The function creates a new destination DIB with a bit count of 1 and no color table.

It uses the "[DibPalUniformGrays](#)" function to generate a grayscale palette with two entries. The function then retrieves the palette entries, sets them in the destination DIB, and performs a pixel-by-pixel conversion.

For each pixel in the source DIB, it retrieves the color, [converts it to the nearest grayscale value](#), and sets the corresponding pixel in the destination DIB.

These are just a few examples of the conversion scenarios covered by the "DibConvert" function. The function also handles other conversions, such as converting from a DIB with no color table to an 8-bit DIB, converting any non-monochrome DIB to a 4-bit DIB, and handling cases where the bit counts of the source and destination DIBs do not match.

The provided code snippet has a declaration of the function DibConvert in the header file DIBCONV.H. Let's break down its components:

HDIB: This is a type definition representing a handle to a Device-Independent Bitmap (DIB). In Windows programming, handles are used to reference objects or resources.

DibConvert: This is the name of the function being declared.

(HDIB hdibSrc, int iBitCountDst): These are the parameters of the DibConvert function.

HDIB hdibSrc: This parameter represents the handle to the source DIB that is to be converted.

int iBitCountDst: This parameter is an integer representing the desired bit count for the destination DIB.

HDIB: This is the return type of the DibConvert function, indicating that the function returns a handle to a DIB.

In summary, this header file provides a declaration for a function called DibConvert that takes a handle to a source DIB (hdibSrc) and an integer representing the desired bit count for the destination DIB (iBitCountDst).

The function is expected to return a handle to the converted DIB. The actual implementation of this function would be found in the corresponding source file (presumably DIBCONV.C). This function is part of a larger program or library for handling device-independent bitmaps and their conversions.

And off we go to the final chapters of the book. This was the largest topic, dealing with bitmaps, but it was really fun...