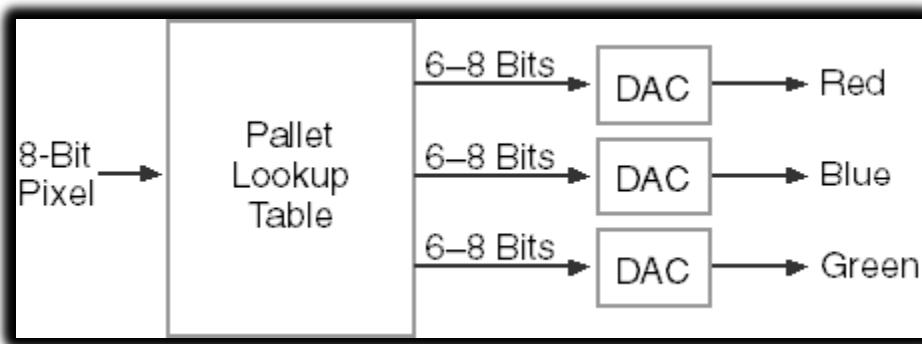


# CHAPTER 16: THE PALETTE MANAGER - REVEALING THE WORLD OF 256 COLORS

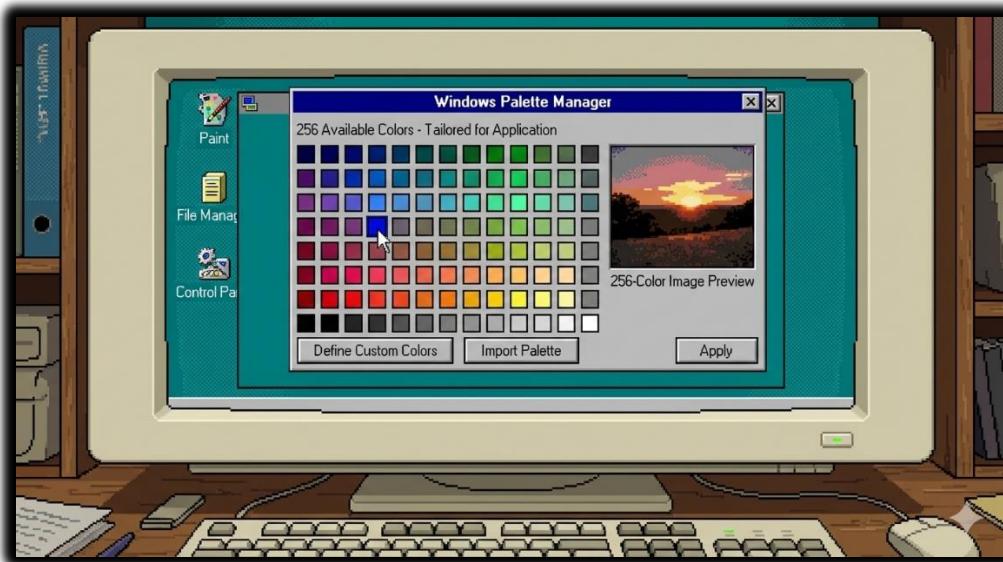
This chapter explores the Windows Palette Manager—a tool that exists largely because of hardware constraints.

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**

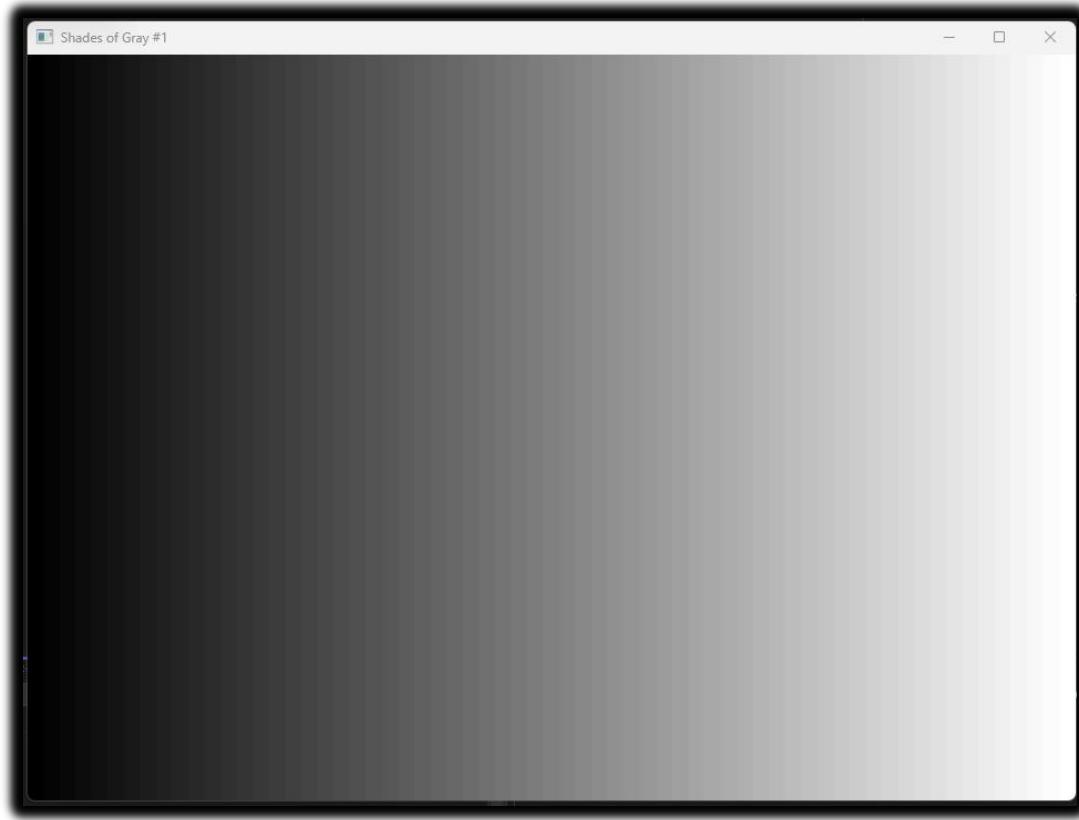
- **System palette** = global 256-color table
- **20 colors are locked** for Windows UI
- **236 colors max** for applications
- **Logical palettes** let apps request colors
- **Active window wins**
- Palette issues only exist in **8-bit modes**

### **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.

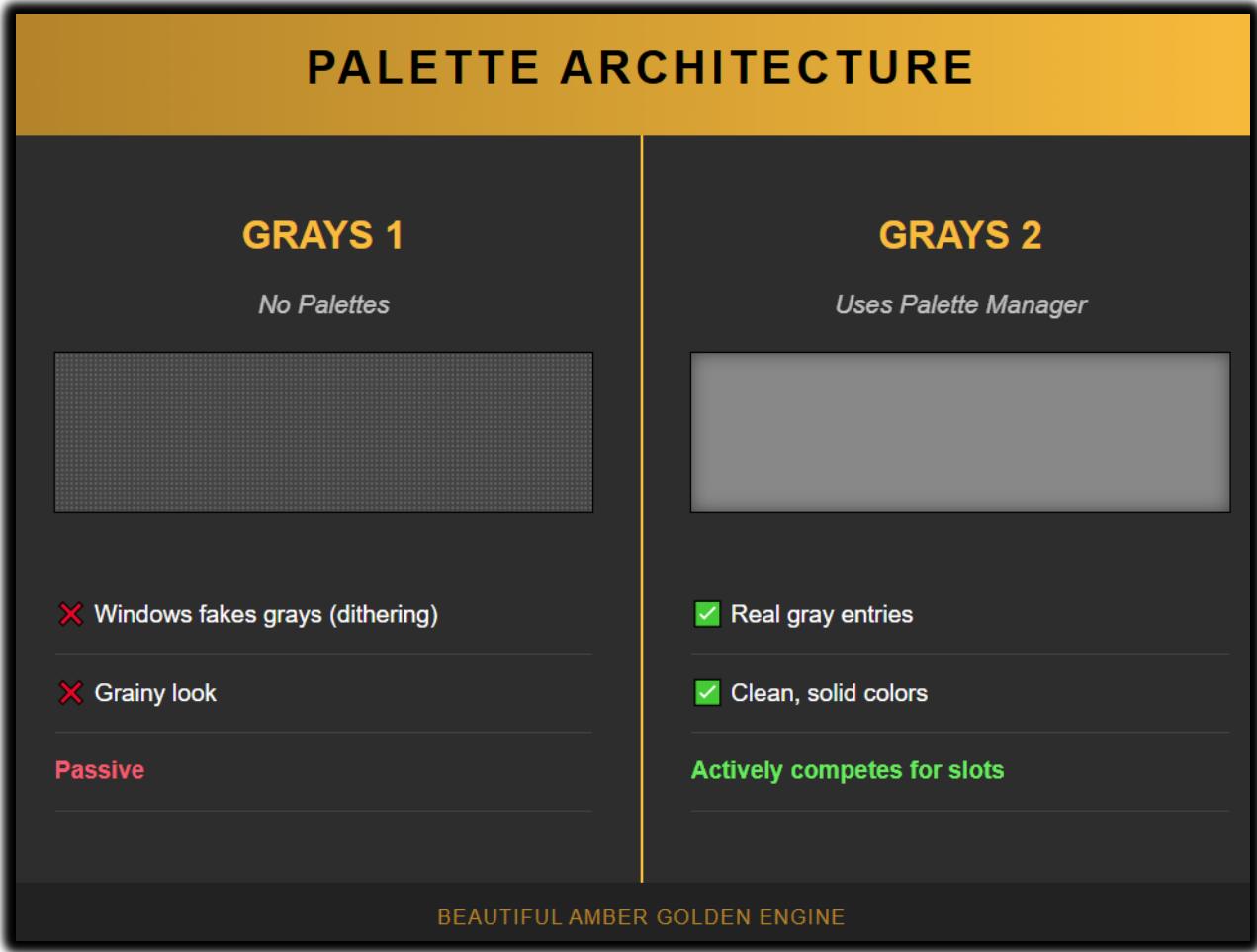
Everything else in the program supports that goal.

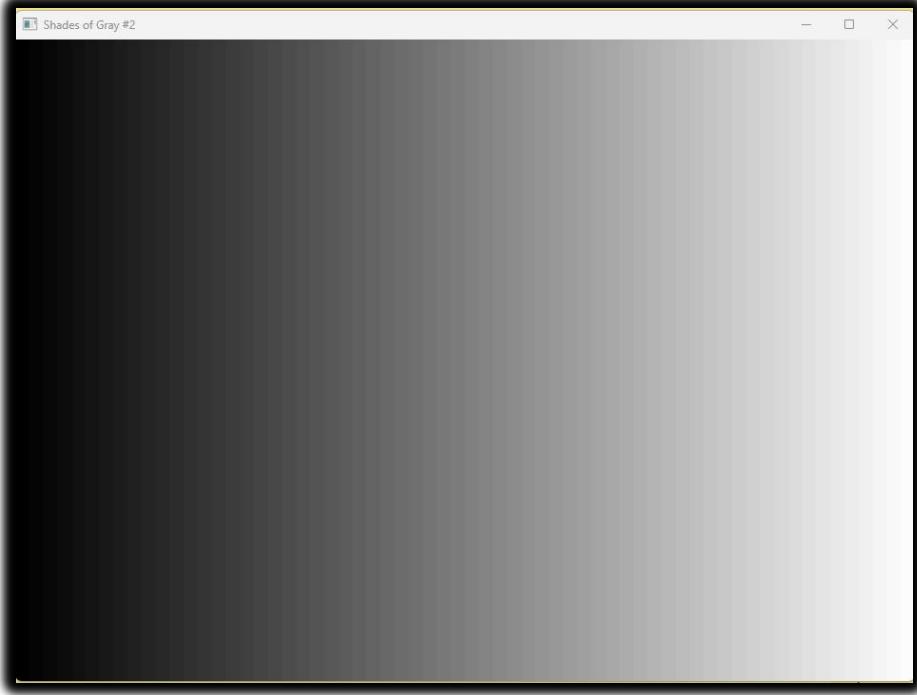
## II. What GRAYS2 Does

- 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.

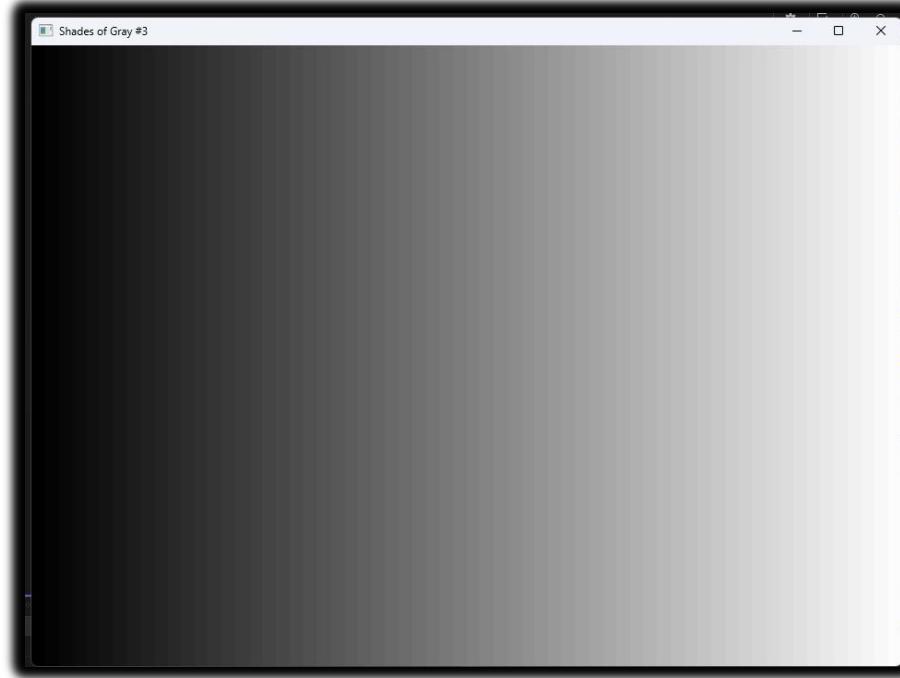
Everything else is just plumbing.

## 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;
```

## GRAYS3 PROGRAM

In the GRAYS3 program, several changes were introduced, focusing on how colors are managed and displayed using the Palette Manager.



GRAYS3 stops using RGB values and starts using palette indexes.

That's it. Everything else flows from that.

### I. RGB vs Palette Index (the big shift)

#### GRAYS2:

- You say: "**Give me this RGB color**"
- Windows says: "*Hmm... what's the closest color I can give you?*"
- That costs time (nearest-color search)

#### GRAYS3:

- You say: "**Give me color #32 from my palette**"
- Windows says: "*Done.*"
- No guessing, no searching
- Palette index = faster + exact

## II. PALETTEINDEX = direct access

- PALETTEINDEX(i) means:

*"Use entry i from the currently selected logical palette"*

Example:

- PALETTEINDEX(0) → black
- PALETTEINDEX(32) → mid gray
- PALETTEINDEX(64) → white

No RGB math. No approximation.

## III. Logical palette is still the same

GRAYS3 still creates:

- A logical palette
- 65 gray shades
- Using LOGPALETTE

Nothing new here.

The **difference is how colors are used**, not how the palette is built.

## IV. Why this is faster

Using RGB:

- Windows must **search** the palette
- Decide closest match
- Remap pixels

Using palette index:

- Windows **already knows the slot**
- Just use it

### Zero conversion cost

That's why GRAYS3 is better.

## V. Palette messages (same as before)

GRAYS3 still handles:

- WM\_QUERYNEWPALETTE → when window becomes active
- WM\_PALETTECHANGED → when another app changes the palette

Nothing new here — **same rules as GRAYS2.**

## VI. UpdateColors instead of repaint 🎨

When palette changes:

- GRAYS3 calls UpdateColors
- Instead of repainting everything

Why?

- Faster
- Less flicker
- Keeps existing pixels

Smart, efficient choice.

## VII. What if the video card doesn't support palettes?

Then:

- Palette Manager is ignored
- App behaves like **GRAYS1**
- Everything still works

No crash. No special handling needed.

## VIII. Why query palette support (GetDeviceCaps)?

Before doing palette tricks, Windows lets you ask:

- Does this device support palettes? (RC\_PALETTE)
- How many entries exist? (SIZEPALETTE)
- How many are reserved? (NUMRESERVED)
- How precise are the colors? (COLORRES)

Why this matters:

- If hardware can only do **64 gray levels**
- Asking for **128 grays is pointless**

Smart programs adapt.

## IX. Why palette indices beat RGB (summary)

Using palette indices gives you:

- Faster drawing
- Exact colors
- Less CPU work
- Better behavior in 256-color mode
- Legacy compatibility

This is **the real lesson of GRAYS3**.

*GRAYS3 is faster than GRAYS2 because it uses palette indexes instead of RGB guesses.*

# UNDERSTANDING RASTER OPERATIONS (ROPS)

## Raster Operations (ROPs) — What You Actually Need to Know

A ROP is **not** a color operation.

A ROP is:

**bitwise math on pixel bits**

Not RGB values.

Not palette entries.

Just raw binary data.

ROPs operate on the *numbers* stored in video memory. Any color you see comes later, after those numbers are interpreted. This is the root cause of every strange or unexpected effect you'll see when palettes are involved.

---

### Binary vs. Ternary

#### I. Binary ROP (SetROP2)

Uses **two inputs**:

- What you draw (pen or brush)
- What's already on the screen (destination)

Common examples:

- XOR
- AND
- OR
- INVERT

 Typically used for lines, borders, cursors, sizing rectangles, and other temporary UI elements.

---

## II. Ternary ROP (BitBlt)

Uses **three inputs**:

- Source bitmap
- Brush
- Destination bitmap

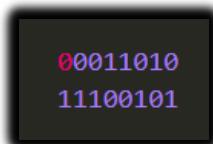
👉 More complex, more flexible, and mainly used for bitmap transfers (blitting).

---

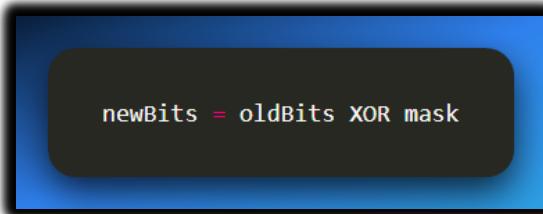
## III. The CORE problem (this is the killer point)

ROPs operate on pixel bits, not colors.

Pixel bits are just numbers:



Windows applies the ROP like this:



Only *after that* does Windows interpret those bits:



⚠ If the palette changes, the *same bits* now map to a **different color**.

The math didn't change.  
The meaning of the result did.

---

## IV. Why GRAYS2 / GRAYS3 show random colors ⚡

Example scenario:

- Windows draws a sizing border
- It uses the **INVERT** ROP
- The pixel bits flip

Before: **bits = 00110010 → gray**

After invert: **bits = 11001101 → palette entry #205**

Palette entry #205 might be:

- Purple
- Green
- Neon pink
- Complete garbage

→ ROPs don't know.

→ ROPs don't care.

They flipped bits. That's their entire job.

---

## V. Reserved colors exist for ONE reason

Windows reserves **20** palette entries:

- First 10
- Last 10

Why?

 To keep ROPs usable and predictable.

Especially important:

- **Black** (all bits = 0)
- **White** (all bits = 1)

These two colors are guaranteed to remain stable across palette changes.

Everything else?

 No promises.

---

## VI. The ONLY guarantee in ROP land

ROPs are **always reliable** with black and white.

Why?

Black = 00000000

White = 11111111

Bitwise operations stay meaningful and predictable.

Any other color?

- Depends on the palette
- Depends on the bit pattern
- Depends on what other applications are doing

---

## VII. Why palette changes break ROP meaning

If you:

- Modify reserved colors ✗
- Aggressively remap logical palettes ✗
- Use “fancy” colors with XOR ✗

Then:

- INVERT no longer means “invert the color”
- XOR no longer means “toggle visibility”
- Borders turn psychedelic

This is **not a bug**.

This is **expected behavior**.

---

## VIII. Practical rules (this is what matters)

### ✓ DO:

- Use black and white for ROP-based drawing
- Respect the 20 reserved palette entries
- Expect strange results when mixing XOR with palette colors

### ✗ DON'T:

- Expect ROPs to preserve color meaning
- Use palette-heavy graphics with ROP tricks blindly
- Modify reserved colors if you care about stability

---

## IX. Why Windows still uses ROPs

Because they are:

- Fast
- Device-level
- Palette-independent *at the bit level*

Perfect for:

- Drag outlines
- Selection rectangles
- Caret blinking
- Temporary UI effects

Not for:

- Color-accurate graphics
- Photo rendering

**Truth:** *ROPs flip bits, not colors — palettes decide later what those bits mean.*

---

# 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."), TEXT("SysPal1"), MB_ICONERROR);
67         return FALSE;
68     }
69
70     return TRUE;
71 }
72
73 }
```

```
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 }
```

**SYSPAL1** is a small utility that displays the contents of the **system palette** in a Windows environment.

The system palette is a shared logical palette that Windows uses to manage colors across all running applications in 8-bit display modes.

The purpose of the program is simple: **make the system palette visible**. By showing the RGB values of every palette entry, SYSPAL1 helps developers understand how colors are assigned and changed, which is useful when debugging palette-dependent applications.

The program uses the `GetSystemPaletteEntries` function to retrieve the RGB values for each entry in the system palette and displays them directly in the window.

The program follows a standard Windows application structure.

It registers a window class, creates a main window, and verifies that the display is running in a **256-color (8-bit)** mode.

Since the system palette only matters in this mode, the program will not run correctly otherwise.

Once the window is created, all core behavior is handled inside the window procedure (WndProc), which responds to messages such as:

- WM\_CREATE
- WM\_SIZE
- WM\_PAINT
- WM\_PALETTECHANGED
- WM\_DESTROY

When the window is created, the program checks whether the display supports a 256-color palette.

This step is mandatory—without it, the palette data is meaningless.

During initialization, the program also obtains a device context (HDC) and sets up the font and character metrics that will later be used to display the RGB values in a readable format.

The `CheckDisplay` function verifies that the system is running in an 8-bit color mode. If the display does not support a 256-color palette, the program displays an error message and exits.

This safeguard ensures that all palette operations performed later behave as expected.

## **Painting the Palette (WM\_PAINT)**

The core functionality of SYSPAL1 lives in the WM\_PAINT handler.

When the window needs to be redrawn, the program calls GetSystemPaletteEntries to retrieve the RGB values for every entry in the system palette. Each entry is then formatted as text and displayed in the client area of the window.

The result is a live, readable listing of the system palette's contents.

## **Responding to Palette Changes (WM\_PALETTECHANGED)**

When another application modifies the system palette, Windows sends a WM\_PALETTECHANGED message. SYSPAL1 responds by invalidating its client area, forcing a repaint.

This ensures the displayed palette values always reflect the current state of the system palette.

When the window is closed, the program releases any allocated resources and terminates cleanly.

## **Usage Notes and Observations**

SYSPAL1 is intended **only** for environments running in 256-color mode. Its output provides a direct view into how Windows assigns and manages system palette entries.

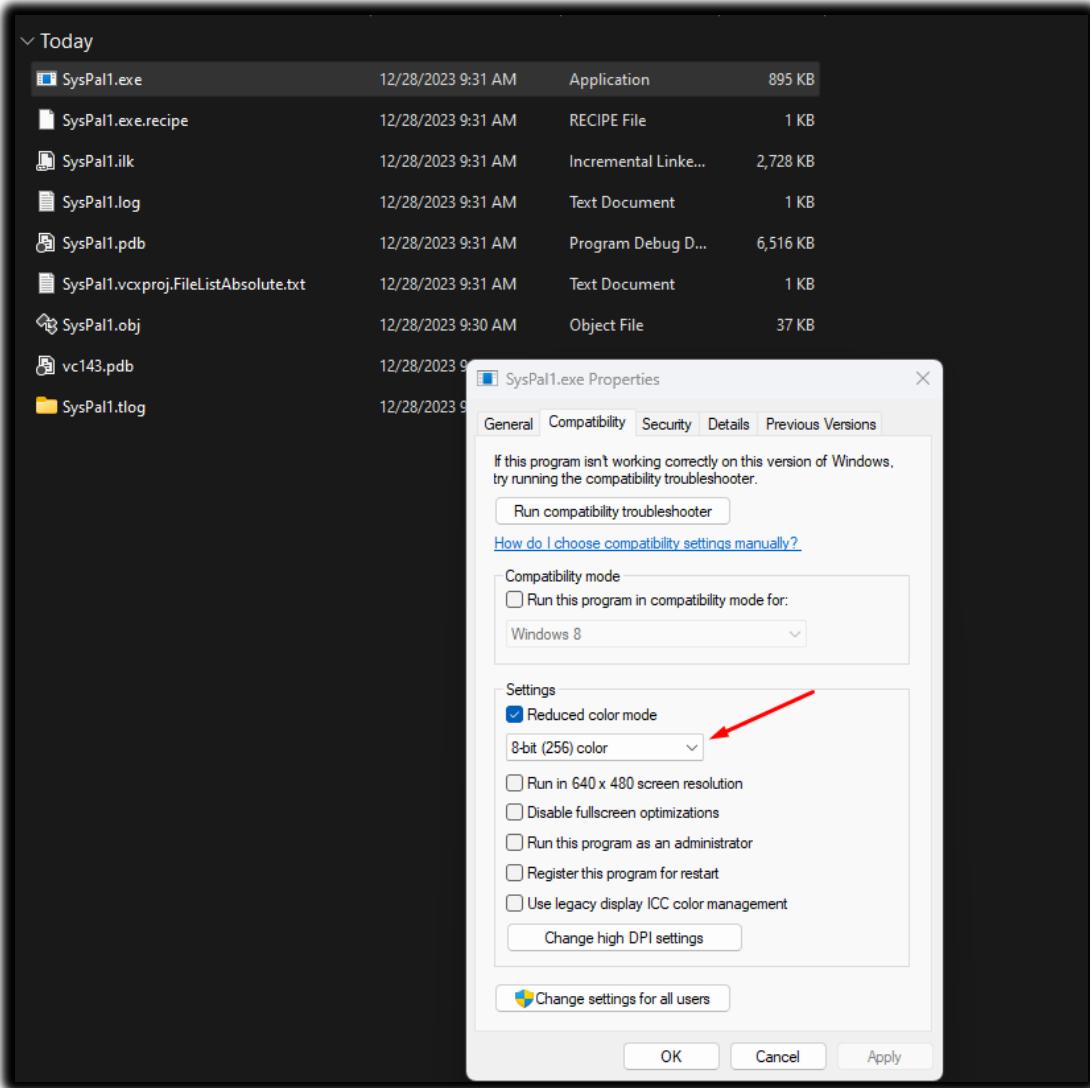
This makes the program especially useful for developers working with:

- Palette-based graphics
- Logical palettes
- Color-sensitive rendering in 8-bit modes

To run the program successfully:

1. **Compile it first**
2. Open the program's **Display Properties**
3. Switch the display to **8-bit (256 colors)**
4. Run the program again

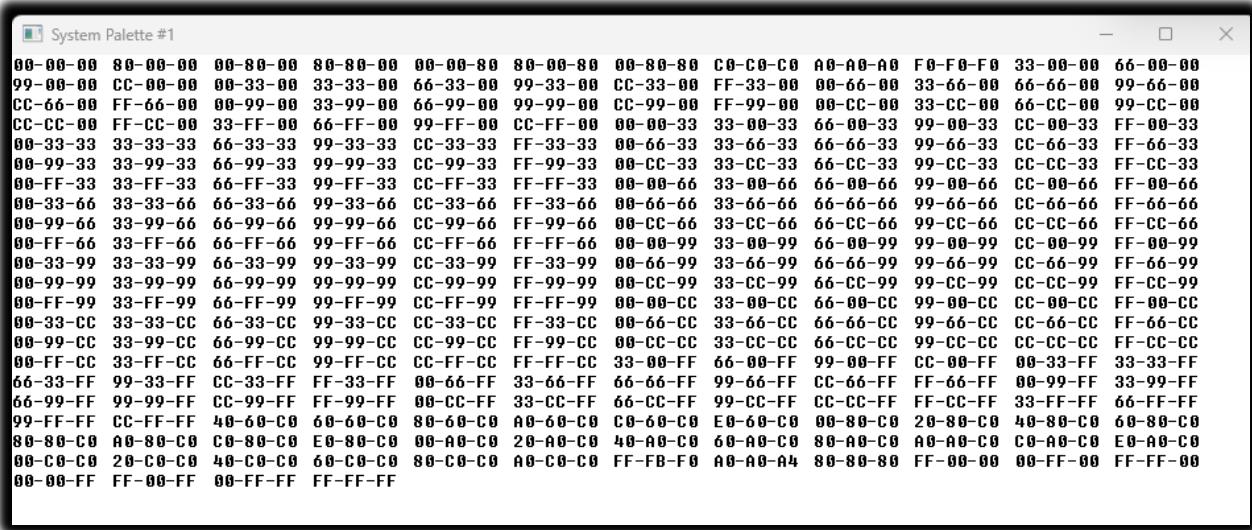
The program will not display meaningful output the first time if the system is not already in 256-color mode.



## Output

The output window displays a list of palette entries along with their corresponding RGB values, providing a real-time snapshot of the system palette.

This visual feedback reinforces how Windows manages shared colors and highlights why palette-aware applications must handle color selection carefully.



System Palette #1

Color Name	RGB Values
00-00-00	00-00-00
99-00-00	CC-00-00
CC-66-00	FF-66-00
CC-CC-00	FF-CC-00
00-33-33	33-33-33
00-99-33	33-99-33
00-FF-33	33-FF-33
00-33-66	33-33-66
00-99-66	33-99-66
00-FF-66	33-FF-66
00-33-99	33-33-99
00-99-99	33-99-99
00-FF-99	33-FF-99
00-33-CC	33-33-CC
00-99-CC	33-99-CC
00-FF-CC	33-FF-CC
66-33-FF	99-33-FF
66-99-FF	99-99-FF
99-FF-FF	CC-99-FF
99-FF-FF	48-60-C0
80-80-C0	A0-80-C0
C0-80-C0	E0-80-C0
00-C0-C0	00-A0-C0
20-C0-C0	20-A0-C0
40-C0-C0	40-A0-C0
60-C0-C0	60-A0-C0
80-C0-C0	A0-C0-C0
FF-FB-F0	A0-A0-A4
00-00-FF	FF-00-FF
00-FF-FF	FF-FF-FF

## SYSPAL2 PROGRAM



SYSPAL2 exists to **visualize the system palette directly** in a 256-color (8-bit) video mode.

That's it.

Not art.

Not gradients for beauty.

Not teaching painting.

It's a **palette inspection tool**.

---

## I. Why 256-Color Mode Matters

SYSPAL2 only makes sense in **8-bit display modes**.

Why?

- Pixel values are **palette indices**, not RGB colors
- What you see depends entirely on the **system palette**
- Higher color modes bypass most palette logic

If the display isn't 256-color:

- 👉 the program exits
- 👉 nothing meaningful can be shown

## II. Logical Palette: The Only Important Idea

The program creates a **logical palette with 256 entries**.

Key points:

- Each entry maps to **one possible pixel value (0-255)**
- The program controls **every palette index**
- This removes ambiguity caused by system or app palettes

This guarantees:

Pixel value **N** always maps to **palette entry N**

That's the entire reason the palette is created.

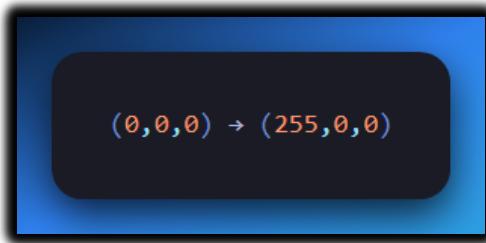
### III. Why the Output Might Be Red & Black

If the output is **black → red**, that's not a bug.

It means:

- Only the **red channel** was populated
- Green and blue were left at zero

So, the palette is:



Same mechanism — different channel choice.

Change the palette data, not the drawing code.

### IV. WM\_PAINT: What Matters (and What Doesn't)

What matters:

- The palette is **selected**
- The palette is **realized**
- Pixels are drawn using **palette indices**

What does NOT matter:

- How rectangles are looped
- Grid size
- Brush creation details

Those are **implementation noise**.

## V. Palette Change Handling (Critical)

Two things matter here:

### 1. Other apps can change the system palette

When that happens:

- Your colors may no longer match your expectations

### 2. SYSPAL2 responds by repainting

This ensures:

- The logical palette is re-mapped
- Display stays consistent

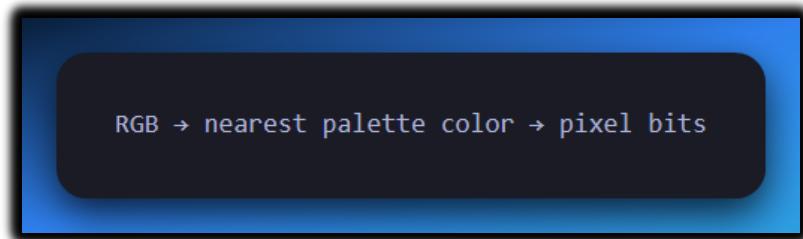
This is **mandatory behavior** for palette-aware apps.

## VI. Why BitBlt / StretchBlt Are Mentioned

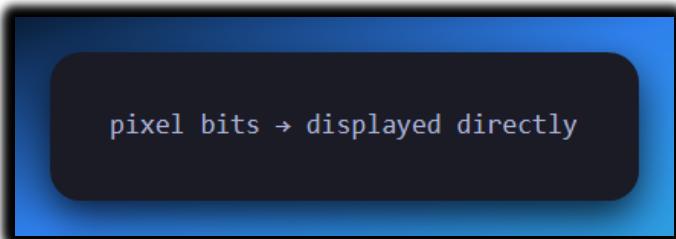
This is NOT about copying images.

This is about **bypassing RGB → palette conversion**.

Normally:



With BitBlt / StretchBlt (in certain modes):



Why this matters:

- Let's you **see the real palette table**
- No color approximation
- No nearest-color search

 This is **device-dependent**

 Not portable

 Mostly educational / diagnostic

## VII. Nearest-Color Search (Only One Sentence Needed)

In limited color modes:

*If a requested RGB color isn't in the palette, Windows picks the closest match.*

That's it. No algorithm breakdown needed.

SYSPAL2 teaches three real lessons:

1. **Pixel values ≠ colors** in 8-bit modes
2. **Palettes control meaning**, not drawing APIs
3. **Direct pixel access exposes hardware reality**

Everything else is decoration.

## VIII. Final Developer Takeaway

SYSPAL2 is not a demo app.

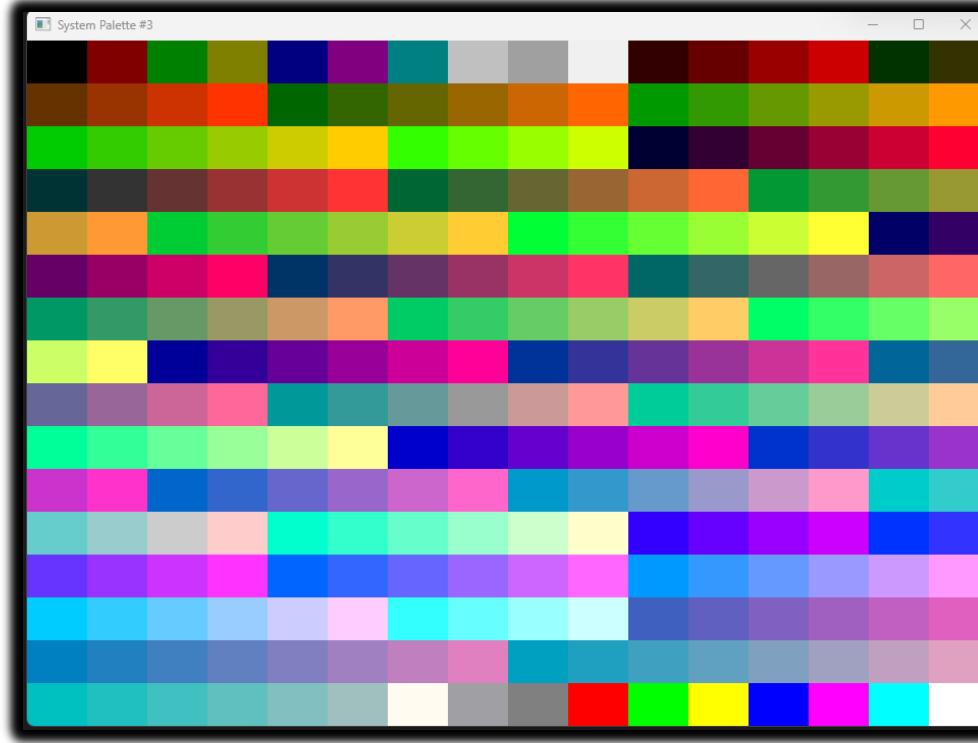
It's a **palette truth machine**.

It shows:

- What pixel values really map to
- How fragile color meaning is in 8-bit modes
- Why palette discipline matters

If someone understands that, they understand the chapter.

## SYSPAL3 PROGRAM



SYSPAL3 shows the **real system palette** by **drawing pixel indices directly**.

Not RGB.

Not logical palettes.

Not color matching.

Just: **pixel value → palette lookup table → screen**

---

### I. Why SYSPAL3 Exists (Difference from SYSPAL2)

SYSPAL2:

- Uses logical palettes
- Depends on palette realization
- Needs palette-change handling

SYSPAL3 is a palette X-ray.

- **Bypasses all that**
- Displays palette entries **exactly as hardware sees them**

---

## II. The One Key Technique

### 8-bit bitmap + StretchBlt

That's the whole trick.

Why it works:

- Bitmap pixels store values 0–255
- Those values are sent **unchanged** to the video hardware
- Each value indexes directly into the system palette

No RGB conversion

No nearest-color search

No palette merging

---

## III. Why the Output Is a 16×16 Grid

- 256 possible pixel values
- $16 \times 16 = 256$
- Each square = one palette index

That layout is just visualization convenience — not logic.

---

## IV. Why It Updates Automatically

Important difference vs SYSPAL2:

SYSPAL3:

- **Does NOT need WM\_PALETTECHANGED**
- **Does NOT repaint manually**

Why?

- StretchBlt always pulls current palette values
- If the system palette changes, the display reflects it instantly

This only works because pixel bits are copied directly, not interpreted

---

## V. Device Dependence (Non-Negotiable)

This only works when:

- Display mode is **8-bit (256 colors)**
- Hardware supports palette lookup

In higher color modes:

- Pixel values no longer map to palette indices
- Program becomes meaningless

That's expected.

---

## VI. Why StretchBlt (Not BitBlt)

StretchBlt:

- Scales the tiny 16×16 bitmap to window size
- Keeps pixel values intact
- Avoids recomputing or recoloring anything

The scaling is visual only — **color meaning stays untouched**.

---

## VII. What SYSPAL3 Teaches

SYSPAL3 proves:

1. Pixel values can be used **as colors**
2. Palettes are **hardware lookup tables**
3. RGB is optional — sometimes misleading
4. GDI can work **below the color abstraction**

This is **low-level graphics reality**.

---

## VIII. Palette Animation — The Real Idea

Palette animation is changing colors **without changing pixels**.

Nothing moves.

Only palette entries change.

The screen updates instantly because pixels already reference those entries.

It's faster than a normal animation.

Normal animation:

- Redraw pixels
- Recalculate colors
- Heavy CPU work

Palette animation:

- Modify palette entries
- Hardware updates colors
- Pixels stay untouched

Speed comes from **not redrawing**.

---

## IX. PC\_RESERVED (Only Reason It Exists)

Setting PC\_RESERVED:

- Prevents GDI from merging colors
- Gives **exclusive control** of those palette entries
- Ensures animation doesn't affect other apps

No reservation = broken animation.

---

## X. AnimatePalette (What It Really Does)

- Replaces palette entries in place
- Updates system palette
- Updates hardware palette
- Triggers instant visual change
- Used typically on: WM\_TIMER

When does Palette Animation make sense?

Useful when:

- 256-color mode
- Repeating color cycles
- Fire, water, blinking, flowing effects

Useless when:

- True-color modes
- Modern GPUs
- Complex images

---

## XI. Final Takeaway

SYSPAL3 is not about drawing.

It's about **seeing the truth**:

- Pixels are numbers.
- Colors are interpretations.
- Palettes are indirection tables.
- Hardware decides the final color.

If you get that — you've outgrown beginner GDI.

## BOUNCE PROGRAM

This program performs a "Magic Trick."

Instead of moving the ball (which requires erasing and redrawing pixels, which is slow), it draws the ball in **every possible position** at the start.

Then, it uses **Palette Animation** to turn the colors on and off.

- It makes 32 balls "Invisible" (White).
  - It makes 1 ball "Visible" (Red).
  - To move the ball, it just swaps the colors.
- 

### Part 1: The Skeleton (PalAnim.c)

This file handles the boring Windows stuff. It keeps the window open and listens for messages.

#### WM\_CREATE:

- Checks if your screen supports Palettes (256 colors). If not, it quits.
- Calls CreateRoutine to set up our magic colors.

#### WM\_PAINT:

- Calls PaintRoutine to draw the picture once.

#### WM\_TIMER:

- The heartbeat of the animation.
- Calls TimerRoutine to swap the colors.

#### WM\_QUERYNEWPALETTE & WM\_PALETTECHANGED:

- Standard "Good Neighbor" rules. If another window changes the system colors, we ensure our palette stays loaded so our animation doesn't look weird.

---

## Part 2: The Magic Logic (Bounce.c)

This is where the actual work happens. It uses 3 custom functions.

### I. CreateRoutine (The Setup)

We need a custom palette with 34 slots.

- **The Flag (PC\_RESERVED):** This is the most important line. We mark these colors as PC\_RESERVED. This tells Windows: "*Do not use these slots for other programs, and let me change them instantly later.*"
- **The Layout:**
  - ✓ **Slots 0-32:** Reserved for the balls. Initially, we make them all Red.
  - ✓ **Slot 33:** The Background (White).

### II. PaintRoutine (The "Invisible" Drawing)

This function draws the balls **once**. It never runs again.

- **Background:** Paints the whole screen White (Index 33).
- **The Balls:** It draws 33 ellipses in a sine-wave pattern across the screen.
  - ✓ The 1st ball uses Palette Index 0.
  - ✓ The 2nd ball uses Palette Index 1.
  - ✓ The 33rd ball uses Palette Index 32.
- **Visual Result:** You see a trail of red balls across the screen.

### III. TimerRoutine (The Switch)

This runs every few milliseconds to create the illusion of movement.

#### 1. Erase the Old Ball:

- ✓ It takes the current ball's Index (e.g., Slot 5).
- ✓ It changes the color in that slot to **White**.
- ✓ *Result:* The ball is still there, but it matches the background, so it becomes invisible.

#### 2. Draw the New Ball:

- ✓ It moves to the next Index (e.g., Slot 6).
- ✓ It changes the color in that slot to **Red**.
- ✓ *Result:* The next ball in the line suddenly appears.

#### 3. AnimatePalette:

- ✓ This function pushes these color changes to the video card instantly.
- ✓ **Zero Repaint:** We did *not* call InvalidateRect or BitBlt. The pixels on the screen never changed; only the definition of "Color #5" changed.

### IV. DestroyRoutine (The Cleanup)

- Stops the timer.
- Deletes the palette handles to free up memory.

### V. Summary: Why is this cool?

In standard animation, to move a ball, you have to:

1. Calculate 1,000 pixels.
2. Send 1,000 pixels to the Video Card.

In **Palette Animation**, you:

1. Change 1 RGB value in the Palette.
2. The Video Card updates the whole screen instantly.

It is incredibly fast and CPU-efficient, which is why it was used for games like *Doom* (for glowing lights) and *Windows 95* (for the loading bar).

Palette animation is an **illusion of motion without redrawing pixels**. Change colors in the palette. Pixels referencing those colors update instantly.

## Requirements

- **256-color display mode** (palette needed).
- **Logical palette** with entries for all animated colors.
- **AnimatePalette()** — updates the palette entries in real time.

## Core Routines

- **CreateRoutine** → sets up the logical palette
- **PaintRoutine** → draws the static scene (pixels reference palette indices)
- **TimerRoutine** → triggered by WM\_TIMER, calls AnimatePalette to update moving elements

## Optimization Tips

- Only update palette entries that **change** (e.g., previous & new positions of bouncing ball)
- Use a **single PALETTEENTRY structure** for updates to reduce overhead
- Avoid unnecessary palette updates — **targeted changes = faster animation**

Troubleshooting Palette Animation. Check these systematically:

1. **Display mode:** confirm 256-color support
2. **Palette creation:** logical palette initialized correctly (CreateRoutine)
3. **Timer interval:** adjust SetTimer interval to control speed
4. **AnimatePalette call:** verify correct indices & colors are being updated
5. **Painting routine:** confirm background & objects use correct palette indices
6. **Debugging:** inspect variables & runtime values for errors
7. **Error handling:** check return values from CreatePalette, SetTimer, AnimatePalette
8. **Memory allocation:** ensure LOGPALETTE is allocated successfully
9. **Code review:** watch for typos, wrong variable names, or logic errors
10. **Interference:** test alongside other programs to see palette interactions
11. **Environment & OS:** test across Windows versions to confirm behavior

---

## Key Points

- Palette animation works best with **repetitive, color-cycling patterns**, like water, fire, or conveyor belts
- Bouncing ball demo is simple but can be inefficient if too many palette entries are updated
- Palette animation = **change colors, not pixels.**
- AnimatePalette() + targeted palette entries + timer = smooth animation in 256-color mode.

# 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 }
```

Palette animation demo focusing on **fade-in/fade-out** of a single color.

### Palette Initialization (CreateRoutine)

- Creates a **logical palette with 1 entry** (white, RGB 255,255,255)
- Uses PC\_RESERVED to control this entry exclusively
- Calls CreatePalette
- Starts a **timer** (50 ms interval)

### Painting (PaintRoutine)

- Draws text "Fade In and Out"
- Uses **palette index 0** (the single entry)

### Fade Logic (TimerRoutine)

- **Fade-In:** decreases RGB values toward black
- **Fade-Out:** increases RGB values back to white
- Switches between phases automatically
- Calls AnimatePalette to update the palette entry dynamically

### Cleanup (DestroyRoutine)

- Stops the timer
- Deletes the palette object
- Frees resources

---

### Tips & Enhancements

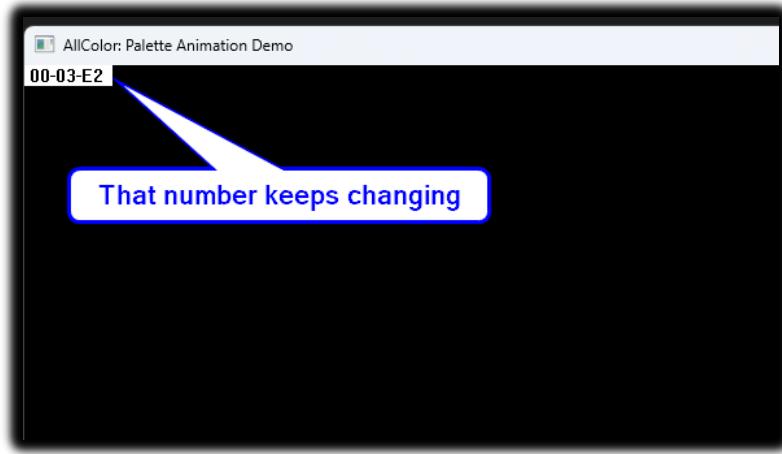
- **Multiple palette entries:** fade several colors simultaneously
  - **Text effects:** rotate, scale, or change font during fade
  - **User control:** pause, resume, or adjust speed
  - **Performance:** optimize timer interval & memory usage
  - **Integration:** combine with SYSPAL2/3 to see palette interactions
  - **Dynamic content:** change text dynamically for messaging or effects
-

 **Developer takeaway:**

The FADER program shows that **even a single palette entry can create smooth animation**. Use AnimatePalette + timer + logical palette entries for efficient visual effects without redrawing pixels.

# 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     SelectObject (hdc, hBrush) ;  
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 }
```



Demonstrates a **continuous color spectrum** using a single palette entry, showing dynamic RGB transitions.

### Palette Initialization (CreateRoutine)

- Determines system color resolution (iIncr) using GetDeviceCaps(COLORRES)
- Creates a **logical palette with 1 entry**, initially black (0,0,0)
- Sets PC\_RESERVED for exclusive control
- Calls CreatePalette
- Starts a **10 ms timer** for smooth updates

### Painting (PaintRoutine)

- Fills **entire client area** with color from palette index 0
- Calls DisplayRGB to show current RGB values at top-left corner

### Color Transition (TimerRoutine)

- Incrementally adjusts RGB components to cycle through colors
  - ✓ **Blue** increases → max
  - ✓ **Green** increases while **Blue** decreases → max
  - ✓ **Red** adjusts as needed, all based on iIncr
- Calls AnimatePalette to apply changes to the palette
- Calls DisplayRGB to refresh on-screen RGB display
- Boolean flags manage **direction reversal** at boundaries for smooth looping

## Continuous Animation

- Timer ensures **real-time updates**
- RGB values display dynamically at top-left corner for monitoring

## Comparison to FADER

- Structurally similar: **single palette entry + AnimatePalette**
  - FADER fades one color in/out; ALLCOLOR cycles through the full spectrum
  - Both rely on **timer-driven palette updates** for animation
- 

## Notes for Developers

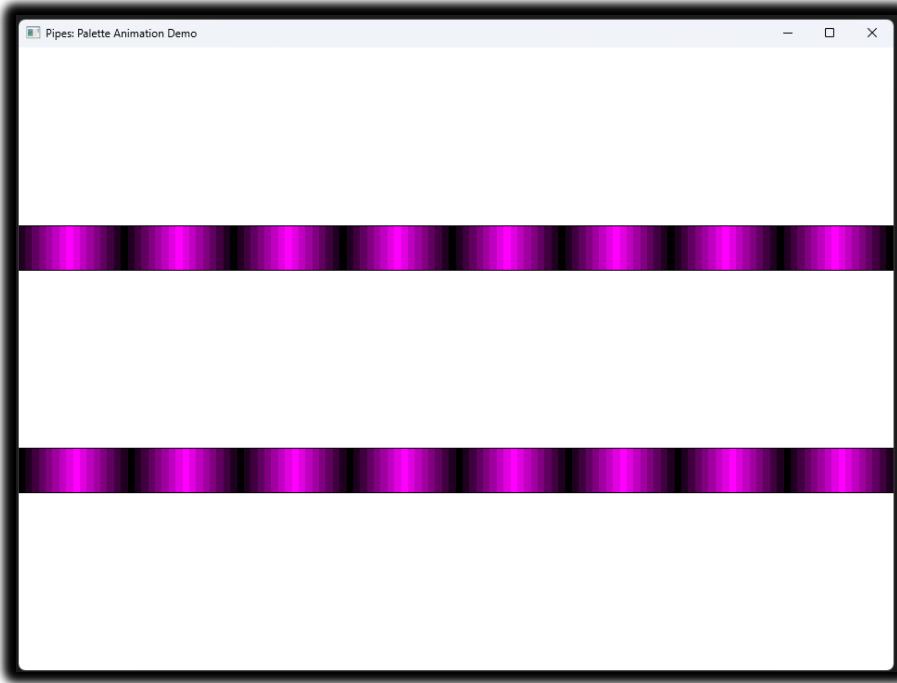
- iIncr = step size for RGB changes, determined by display color depth
- PALETTEINDEX(0) is key for consistent palette-driven drawing
- Efficient because **no pixels are redrawn**—only the palette entry changes
- Displaying RGB values is optional, mostly for **debugging/testing**

### Takeaway:

ALLCOLOR shows **smooth palette-based color transitions** using a single palette entry. It's a practical example of how timers + AnimatePalette can create **dynamic visual effects without redrawing graphics**, ideal for exploring palette animation concepts.

---

## 🛠️ PIPES: PALETTE ANIMATION IN ACTION



The **PIPES program**, created by Charles Petzold in 1998, is a classic example of using **palette animation** to simulate motion—in this case, fluid flowing through pipes.

Instead of drawing every frame manually, it cleverly shifts color entries in a palette to give the illusion of movement.

---

### I. Palette Initialization: Setting the Stage

- **Logical palette creation** happens in CreateRoutine.
- Allocates **16 color entries**, each representing a stage in the fluid's movement.
- Colors are arranged from **dark → light**, visually showing progression.
- Think of it like assigning different shades of paint along the pipe: each shade moves forward with every timer tick.

🎯 **Key takeaway:** The palette itself *is* the animation engine. By changing colors instead of redrawing graphics, the program runs faster and smoother.

---

## II. Painting the Canvas: Drawing the Pipes

Done in PaintRoutine.

Draws two **horizontal pipes** (top and bottom) as rectangles.

Fills each pipe with colors from the palette to show flow direction:

- **Top pipe:** left → right (red → green)
- **Bottom pipe:** right → left (green → red)

Real-world analogy: Imagine pouring colored water into two straws in opposite directions—the colors naturally indicate which way the fluid is moving.

---

## III. Crafting the Animation: Making Colors Move

TimerRoutine drives the animation.

Every timer tick:

- Updates **palette indices**
- Shifts the color gradient slightly forward

Result: continuous flow illusion, without touching the pipe graphics themselves.

 Insight: This is palette-level animation—a WinAPI trick to move color instead of shapes. Super efficient for 90s hardware.

---

## IV. Palette Dance: AnimatePalette in Action

- AnimatePalette is called every tick.
- Cycles through the 16 palette entries to simulate motion.
- The effect is **cyclical**, meaning the animation loops smoothly.

Think of it like a **carousel of colors**: the pipes stay still, but the “fluid” moves because the colors rotate.

---

## V. Palette Symphony: Why It Works

- Combines **color, timing, and symbolism** to convey movement.
  - Even without physics or particles, your brain interprets the gradient shift as fluid flow.
  - Educationally powerful: engineers can visualize processes without complex simulations.
- 

## VI. Clean Exit: DestroyRoutine

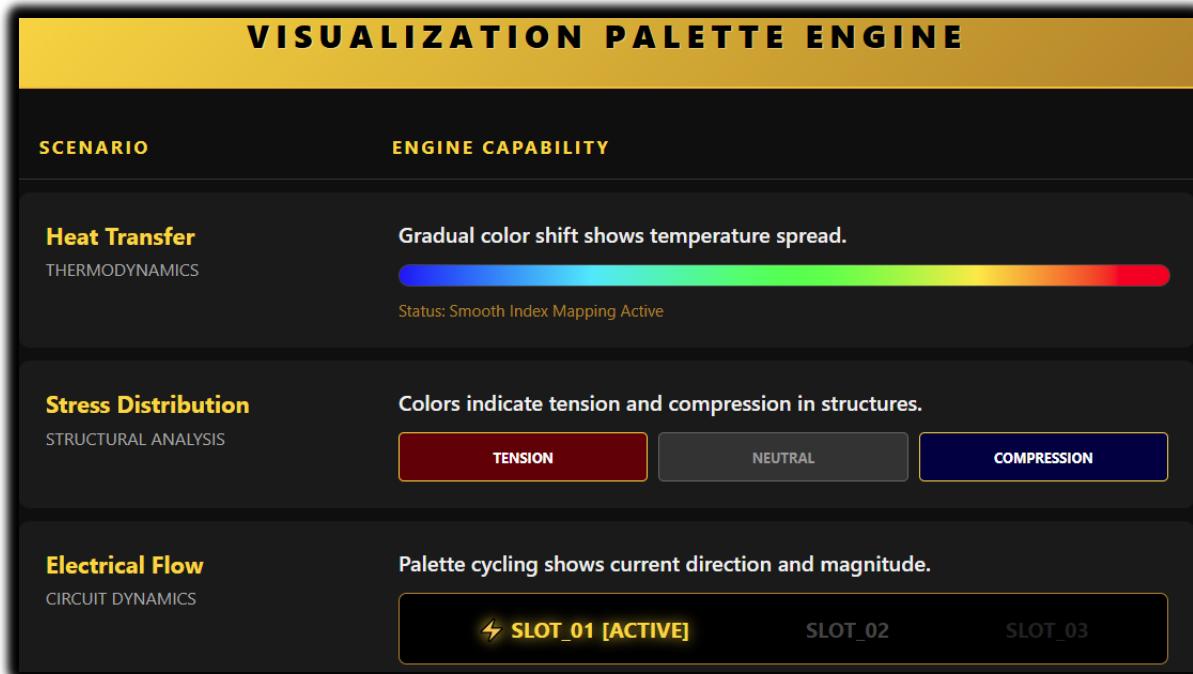
- Stops timers
- Frees palette memory
- Closes the program gracefully

⚡ **Important:** Always release GDI objects in WinAPI to avoid leaks.

---

## VII. Beyond PIPES: Palette Animation Uses

While PIPES is about fluid, the technique scales to other engineering visualizations:



## Summary

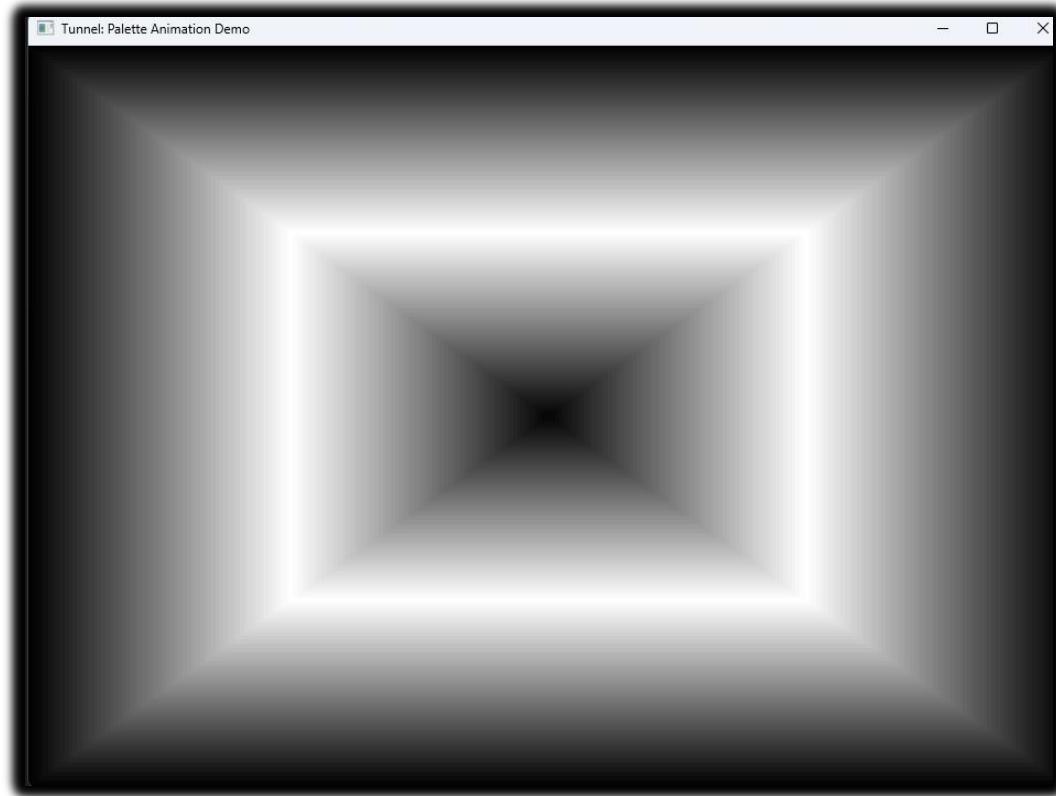
PIPES isn't just a demo—it's a lesson in **efficiency and clarity**:

- **Palette = motion engine**
- **Timer = animation driver**
- **GDI rectangles = static canvas**

This setup demonstrates how simple color manipulation can turn static graphics into **dynamic, educational animations**.

---

## TUNNEL PROGRAM



**TUNNEL**, by Charles Petzold (1998), is a WinAPI demo that creates the illusion of **traveling through a tunnel** using **palette animation**. Instead of moving graphics, it shifts colors to simulate motion.

## I. Palette Initialization

- Allocates a **LOGPALETTE** with **128 entries**.
- Creates **64 shades of gray** to form the tunnel gradient.
- These entries are reserved (PC\_RESERVED) to prevent accidental changes.

Think of it as **painting the tunnel in grayscale**, then rotating the shades to simulate motion.

## II. Painting the Tunnel

- PaintRoutine draws the tunnel using **rectangles**.
- Each rectangle uses a **palette index** to determine its color.
- By varying rectangle size and position, the tunnel appears to **stretch into the distance**.

Imagine stacking gray bands smaller and smaller toward the center of the screen—the eye sees depth.

## III. Animation Logic

- Driven by a **timer (TimerRoutine)**.
- Increments a **level index (iLevel)** cyclically.
- Calls AnimatePalette to **shift the gray shades**, creating the illusion of moving forward through the tunnel.

Key trick: **the tunnel isn't moving; the colors are**. This saves processing power and looks smooth.

DestroyRoutine stops the timer and frees memory (palette and LOGPALETTE).

Always release GDI objects in WinAPI to avoid **leaks**—simple but critical.

---

## Summary

TUNNEL is a **palette-based animation demo**:

- Uses **gray shades and palette shifts** to simulate motion.
- **Timer-driven updates** create a continuous movement effect.
- **Static rectangles + dynamic colors = illusion of traveling through a tunnel.**

Compared to PIPES: the concept is the same—animate colors, not shapes. This keeps CPU/GPU usage low while producing a visually impressive effect.

---

## PACKED DIBS & PALETTE POWER: DISPLAYING REAL-WORLD IMAGES

Packed DIBs let you **store image data + color info together**—essential for displaying real-world images in **8-bit video modes**. Petzold provides custom functions (packedDIB.c/.h) to make this manageable.

### I. Extracting Palette Information

Packed DIB functions help access color info in the image:

FUNCTION CALL	ENGINE PURPOSE
PackedDibGetColorsUsed	How many colors are actually used ( $\leq$ total entries)
PackedDibGetNumColors	Total entries in the color table
PackedDibGetColorTablePtr	Memory address of the color table
PackedDibGetColorTableEntry	Get a specific color by index

This is the **first step** before creating a palette.

## II. Building a Logical Palette

- PackedDibCreatePalette takes the **color table** from the DIB.
- Generates a **Windows logical palette** object.
- This palette is used for proper color mapping when drawing the image on screen.

Think of it as translating the image's native colors into something Windows can display accurately.

## III. Displaying the Image

- Use the logical palette with **GDI functions** to render the image.
- Even in **8-bit modes**, you can display images with their intended colors.

Palette + rectangles = real-world image rendering without full 24-bit support.

## IV. Practical Notes & Caveats

- **Function Ordering:** Bottom-up; later functions depend on earlier results.
- **Efficiency:** Functions like PackedDibGetPixel are **slow** due to nested calls.
- **Limitations:** Only designed for palette extraction and basic handling.
- **OS/2 DIBs:** Some DIBs need special handling (check header size).
- **No Palette for 16/24/32-bit DIBs:** Only indexed images get a logical palette.

Petzold hints at **better methods** later in the chapter for efficiency and flexibility.

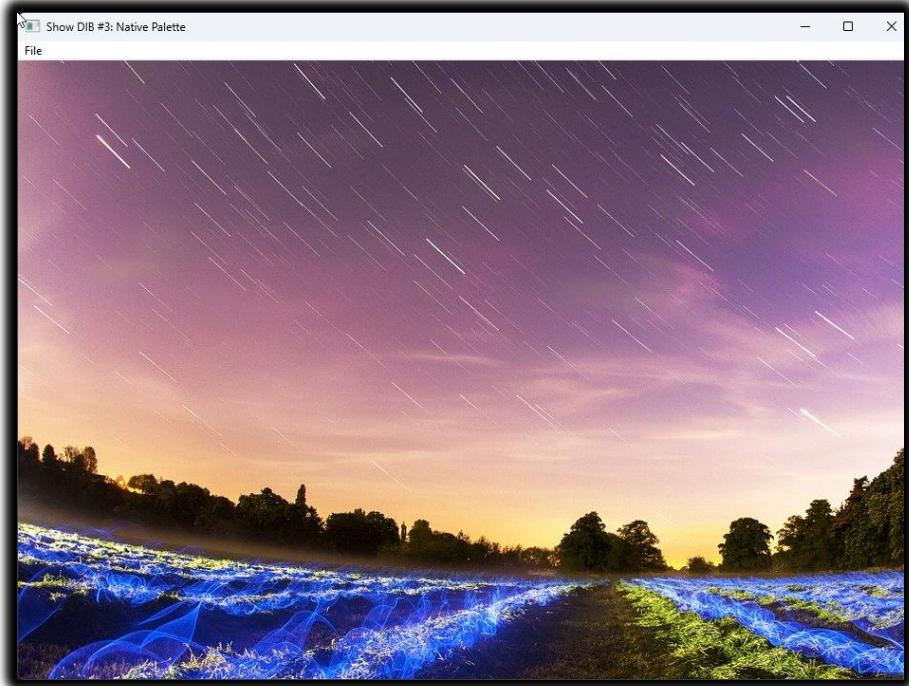
## V. Summary

Packed DIBs + logical palettes allow you to:

- Load and display **real-world images** in 8-bit modes
- Extract color info with packedDIB functions
- Render images **accurately using Windows GDI**

This setup is **foundational** for upcoming programs like showdib3, which combines packedDIB.c/h + your .bmp files + afresx.h.

# SHOWDIB3 PROGRAM: The Native Palette Display



## I. The Goal: True Color on Limited Hardware

Most programs simply copy DIB pixels directly to the screen. On a 256-color display, this produces poor results because Windows forces the image to use the system default palette, which is dominated by bright primary colors and lacks subtle shades.

SHOWDIB3 takes a smarter approach. Instead of relying on the system palette, it reads the exact colors required by the image from the DIB color table, builds a custom Windows palette, and instructs the system to use those colors when drawing the image. The result is a display that looks as close as possible to the original, even on limited hardware.

---

## II. The Mechanism: Packed DIBs

The program relies on a helper function called `PackedDibLoad`, which works with packed DIBs.

In a normal DIB, the header, color table, and pixel data may be stored in separate memory locations. A packed DIB places all of these components into a single, contiguous block of memory.

This layout simplifies the code, making it easier to load, pass around, and process the image as a single object.

---

### III. The Workflow: How It All Fits Together

#### Step A: Loading the Image (WM\_COMMAND → Open)

The entire image file is read into memory as a packed DIB.

Next, the program creates a palette using PackedDibCreatePalette. This function examines the DIB's color table and builds an HPALETTE that matches those colors exactly.

At this point, the program has both the image data and a custom palette ready for display.

#### Step B: Painting the Image (WM\_PAINT)

This is where the palette-aware rendering happens.

First, the program selects the custom palette into the device context. Then it calls RealizePalette, which causes Windows to remap the hardware palette so the requested colors are available.

Finally, SetDIBitsToDevice draws the image to the screen. Because the palette was realized first, the image appears with correct color relationships instead of being distorted by the system defaults.

#### Step C: Playing Nicely with Other Applications

Since the program temporarily takes control of the system palette, it must cooperate with other running applications.

When SHOWDIB3 becomes the active window, it realizes its palette so the image displays correctly. When it loses focus, it releases control so other applications can restore their own colors.

This behavior is handled through messages such as WM\_QUERYNEWPALETTE and related palette notifications.

---

## IV. The Edge Case: High-Color Images on 8-Bit Displays

Problems arise when a high-color image, such as a 24-bit JPEG with millions of colors, is loaded on an 8-bit display. These images do not contain a color table, so there is no predefined palette to extract.

SHOWDIB3 does not fully solve this case. Instead, it falls back to using standard system colors, which often results in a grainy or inaccurate appearance.

A proper solution would require dynamic color analysis—scanning every pixel, determining the most frequently used colors, and constructing a new palette from that data. This is a complex process and is intentionally left for later discussion.

---

## V. Summary Checklist

- Load the DIB into memory
  - Extract colors and create an HPALETTE
  - Paint the image by selecting the palette, realizing it, and drawing the DIB
  - Respond to focus changes by updating palette ownership
- 

# SHOWDIB4, SHOWDIB5, AND SHOWDIB6 — ONE PROBLEM, THREE STRATEGIES

After SHOWDIB3 demonstrates image-specific palettes, the next programs explore **alternative palette strategies**. Each one answers a different question about how 8-bit color can be managed more effectively.

## SHOWDIB4: One Palette for Everything

SHOWDIB4 abandons the idea of creating a new palette for each image. Instead, it builds **one all-purpose palette** at startup and uses it for every DIB displayed.

This palette is created during WM\_CREATE by the CreateAllPurposePalette function and remains in use for the lifetime of the program.

Because the palette never changes, SHOWDIB4 avoids the repeated creation, realization, and destruction of per-image palettes.

## I. Key Characteristics

Uses a single logical palette for all images

Palette contains 247 entries

Designed to avoid conflicts with the 20 reserved system colors

Includes:

- 31 gray shades
- 216 RGB combinations

Some entries intentionally duplicate reserved colors or existing grays. When the peFlags field is set to zero, Windows automatically avoids inserting duplicates into the system palette.

## II. Why This Matters

Using one palette simplifies palette management and produces **consistent color results** across different images. This approach is not optimal for every image, but it is predictable and easy to manage.

SHOWDIB4 is primarily a **demonstration program**, allowing you to compare how a fixed palette performs against the image-specific palette used in SHOWDIB3.

## SHOWDIB5: Let Windows Do the Work

SHOWDIB5 takes a different approach by relying on **Windows' built-in halftone palette** instead of a custom one.

During WM\_CREATE, the program calls CreateHalftonePalette and uses that palette for all rendering. Unlike earlier versions, SHOWDIB5 also changes how images are drawn.

## I. What's Different

- Uses the Windows halftone palette
- Displays images with StretchDIBits instead of SetDIBitsToDevice
- Sets the stretch mode to HALFTONE
- Aligns the brush origin with SetBrushOrgEx

## II. Why This Combination Works

When HALFTONE stretching is used together with the halftone palette, Windows applies a **dithering pattern** derived from the palette colors. This significantly improves color approximation on 8-bit displays.

The improvement is especially noticeable in **flesh tones and subtle gradients**, where earlier approaches tend to fail.

## III. Trade-Off

The better visual quality comes at a cost: **slower rendering**. Halftoning requires additional processing, making this approach less suitable for performance-critical scenarios.

## SHOWDIB6: Speed Through Palette Indices

SHOWDIB6 focuses on **performance**, not palette quality.

Instead of treating the DIB color table as RGB values, SHOWDIB6 uses **palette indices** by setting the DIB\_PAL\_COLORS flag when displaying the bitmap.

## I. What This Means

With DIB\_PAL\_COLORS, the DIB color table contains **indices into the currently selected logical palette**, not RGB triples. Windows can therefore map pixels directly to device colors without performing a nearest-color search.

## II. How It Works

- Load an 8-bit DIB
- Create a logical palette from the DIB's color table
- Replace RGB entries in the DIB color table with WORD indices
- Select the palette into the device context
- Display using SetDIBitsToDevice with DIB\_PAL\_COLORS

### III. Advantages

- Faster rendering in 8-bit modes
- No color matching or nearest-color calculations
- Ideal when the palette exactly matches the DIB

### IV. Limitations

- Only practical when the palette is derived from the DIB itself
- Not compatible with all-purpose palettes
- DIBs must be converted back to RGB before saving or copying to the clipboard

This technique is best used **only for display**, not for storage or interchange.

## Big-Picture Comparison

Program	Palette Strategy	Strength	Trade-Off
SHOWDIB4	One custom palette	Simplicity and consistency	Not image-optimal
SHOWDIB5	Windows halftone + HALFTONE stretch	Best visual quality	Slower rendering
SHOWDIB6	Palette indices (DIB_PAL_COLORS)	Maximum performance	Limited flexibility

## Takeaway

These programs are not incremental upgrades — they are **alternative answers to the same problem**:

- SHOWDIB4 simplifies palette management
- SHOWDIB5 prioritizes visual quality
- SHOWDIB6 prioritizes speed

Together, they illustrate the trade-offs involved in palette-based graphics and show why there is no single “best” solution for 8-bit color rendering.

## SHOWDIB7 AND SHOWDIB8 — SAME GOAL, DIFFERENT PATHS

SHOWDIB7 and SHOWDIB8 do essentially the same job:  
load a bitmap, manage a palette correctly, and display the image.

What changes is **how the bitmap is stored and drawn**.

### SHOWDIB7: Convert First, Then Draw

SHOWDIB7 takes the loaded DIB and converts it into a **Device-Dependent Bitmap (DDB)**.

#### I. What it does differently

- Loads a DIB from disk
- Creates a logical palette from the DIB's color table
- Selects and realizes the palette
- Converts the DIB into a DDB using CreateDIBitmap
- Displays the image using BitBlt

#### II. Why this matters

A DDB is already in a format the device understands, so drawing it with BitBlt is fast and simple. Once the DDB is created, the original DIB data can be freed.

This approach follows **traditional GDI usage** and works well when you just need to display the bitmap.

## SHOWDIB8: Skip Conversion, Use a DIB Section

SHOWDIB8 keeps the bitmap **device-independent** and uses a **DIB section** instead.

### I. What it does differently

- Loads a packed DIB
- Creates a DIB section using CreateDIBSection
- Copies the pixel data into the section
- Creates a palette from the DIB
- Displays the image using BitBlt

### II. Why this matters

A DIB section gives **direct access to pixel memory**. The program can read or modify pixels without extra copying or conversions.

This is ideal for:

- Image processing
- Frequent pixel updates
- Better performance when touching bitmap data

## The Real Difference (That's All You Need to Know)

GDI RENDERING STRATEGIES		
Program	Bitmap Type	Why Use It
SHOWDIB7	DDB (Device Dependent)	Simple, classic GDI drawing
SHOWDIB8	DIB Section	Direct pixel access, better control

Both still:

- Use palettes correctly
  - Select and realize palettes before drawing
  - Display using BitBlt in WM\_PAINT
- 

## Important Palette Notes

In SHOWDIB7, the palette **must be selected and realized before** calling CreateDIBitmap, because the DDB is created using device colors.

In SHOWDIB8, the palette is **not needed during creation** of the DIB section. The bitmap remains device-independent, and palette handling happens later when drawing.

That's the key behavioral difference.

- SHOWDIB7 converts the bitmap to match the device
- SHOWDIB8 keeps the bitmap independent and faster to manipulate
- The painting code is almost identical
- The strategy shift happens during **File Open**, not WM\_PAINT

Nothing magical.

No new palette rules.

Just different tools for different needs.

---

## BUILDING A BETTER DIB LIBRARY IN C ↘

Packed DIBs are simple, but slow when reading or writing individual pixels. DIB sections fix that:

- **Faster:** Access pixels directly—no extra copies.
- **Flexible:** Still device-independent, like packed DIBs.
- **Optimized for Windows NT:** Runs smoother on modern systems.

In short: DIB sections give you speed **and** control without the packed DIB overhead.

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

## I. Defining the HDIB Handle:

```
typedef void * HDIB;
```

This declaration defines an opaque handle called HDIB. It's just a reference to a DIB section object. The actual layout and internal details are intentionally hidden—you use the handle, and Windows takes care of the rest.

## II. 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 stores everything needed to work with the DIB section: the device context, bitmap handle, image size, color depth, bytes per row, and—most importantly—a pointer (pBits) to the raw pixel data. Having direct access to this memory makes reading and modifying individual pixels fast and straightforward, using functions like DibGetPixel and DibSetPixel.

### III. Library Functions

The library provides a small set of helper functions for working with DIB sections. Each function does one clear job and hides the messy details.

- **DibCreate(cx, cy, bitsPerPixel)**  
Creates a new DIB section with the given width, height, and color depth. It allocates the memory, sets everything up, and returns an HDIB handle.
- **DibGetPixel(hdib, x, y)**  
Reads the pixel at position (x, y) by accessing the bitmap memory directly through pBits.
- **DibSetPixel(hdib, x, y, color)**  
Writes a color value directly into the bitmap at (x, y). No drawing calls, just raw memory access.
- **DibDestroy(hdib)**  
Cleans up everything associated with the DIB section. Call this when you're done.

### IV. Why This Works Well

- **Fast pixel access**  
Pixels are read and written directly in memory. No copying, no extra conversions.
- **Less bookkeeping**  
The library handles allocation and cleanup, so your code stays simple.
- **Full control**  
You can change or extend the internal structure later without touching the rest of your code.

### V. Bottom Line

Wrapping DIB sections in a small helper library gives you fast, direct pixel access without cluttering your application code. You get the performance benefits of DIB sections with clean, easy-to-use functions.

- Fields related to file mapping can be ignored here—they aren't used by this library.
- Keeping row pointers easy to reach makes pixel access simpler and faster.
- A small signature check helps catch invalid handles before they cause crashes.

That's it.

Create the DIB, touch the pixels, destroy it when finished.

## 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;
```

### Key Fields in DIBSTRUCT

- **ppRow** – An array of pointers to each row of pixels. Makes it easy to access pixel rows directly.
- **iSignature** – A small check value to make sure the structure is valid. Helps prevent errors.
- **hBitmap** – The handle to the DIB section returned by CreateDIBSection. Needed for GDI operations.
- **pBits** – Pointer to the raw bitmap data. Lets you read and write pixels directly.
- **DIBSECTION** – Contains information about the DIB, such as width, height, color masks, and compression.
- **iRShift / iLShift** – Arrays used to extract RGB components from 16-bit and 32-bit pixels when BI\_BITFIELDS compression is used.

---

### Functions in DIBHELP.C

These functions make it easier to work with DIB sections:

- **DibIsValid(HDIB hdib)** – Checks if a DIB handle is valid before using it.
- **DibBitmapHandle(hdib)** – Returns the bitmap handle.
- **DibWidth / DibHeight / DibBitCount** – Get basic info: width, height, and color depth.
- **DibRowLength** – Returns the number of bytes per row (including padding).
- **DibNumColors / DibMask / DibRShift / DibLShift** – Access color info like the number of colors, masks, and shift values.

- **DibCompression / DibIsAddressable** – Check the compression type and whether pixels can be accessed directly.
  - **Size Functions** – DibInfoHeaderSize, DibMaskSize, DibColorSize, DibInfoSize, DibBitsSize, DibTotalSize help calculate memory size for saving or copying.
  - **Pointer Functions** – DibInfoHeaderPtr, DibMaskPtr, DibBitsPtr give direct access to the header, masks, and pixel data.
  - **DibGetColor / DibSetColor** – Read or modify individual entries in the DIB's color table.
- 

## Summary

- **Validation first** – Always check DibIsValid before using a DIB handle.
- **Easy access** – Most functions let you quickly get sizes, pointers, or pixel data.
- **Color handling** – Use DibGetColor / DibSetColor for palette management.
- **Compression aware** – Some DIBs may be compressed, which affects direct pixel access.

In short, the **DIBSTRUCT** stores all important DIB info, and **DIBHELP.C** functions give you fast, safe ways to read, write, and manipulate it.

---

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

## DIBHELP.C – PART 2: PIXEL FUNCTIONS

### I. DibPixelPtr

Returns a pointer to the pixel at (x, y) in a DIB.

- Checks if the DIB is addressable.
- Checks that (x, y) is inside the bitmap.
- Uses ppRow from DIBSTRUCT to locate the pixel.

```
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;

    DIBSTRUCT* pdib = (DIBSTRUCT*)hdib;
    int bytesPerPixel = DibBitCount(hdib) / 8;
    return pdib->ppRow[y] + x * bytesPerPixel;
}
```

### II. DibGetPixel

Gets the pixel value at (x, y).

- Uses DibPixelPtr.
- Handles different bit depths (1, 4, 8, 16, 24, 32).

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

    int bpp = DibBitCount(hdib);
    switch (bpp) {
        case 8: return *pPixel;
        case 16: return *((WORD*)pPixel);
        case 24: return pPixel[0] | (pPixel[1]<<8) | (pPixel[2]<<16);
        case 32: return *((DWORD*)pPixel);
        // 1-bit and 4-bit require mask and shift
    }
    return 0;
}
```

### III. DibSetPixel

Sets a pixel value at (x, y).

```
void DibSetPixel(HDIB hdib, int x, int y, DWORD color) {
    BYTE* pPixel = DibPixelPtr(hdib, x, y);
    if (!pPixel) return;

    int bpp = DibBitCount(hdib);
    switch (bpp) {
        case 8: *pPixel = (BYTE)color; break;
        case 16: *((WORD*)pPixel) = (WORD)color; break;
        case 24:
            pPixel[0] = (BYTE)(color & 0xFF);
            pPixel[1] = (BYTE)((color >> 8) & 0xFF);
            pPixel[2] = (BYTE)((color >> 16) & 0xFF);
            break;
        case 32: *((DWORD*)pPixel) = color; break;
        // 1-bit and 4-bit require mask and shift
    }
}
```

### IV. DibGetPixelColor

Returns the pixel color at (x, y) as RGBQUAD.

- Converts palette indices for 1-, 4-, 8-bit DIBs to RGB.
- For 16-, 24-, 32-bit, applies masks and shifts.

```
RGBQUAD DibGetPixelColor(HDIB hdib, int x, int y) {
    DWORD val = DibGetPixel(hdib, x, y);
    RGBQUAD color = {0};

    int bpp = DibBitCount(hdib);
    if (bpp <= 8) {
        color = DibGetColor(hdib, val); // lookup in color table
    } else if (bpp == 16 || bpp == 32) {
        DIBSTRUCT* pdib = (DIBSTRUCT*)hdib;
        color.rgbRed = (BYTE)((val & pdib->rmask) >> pdib->iRShift[0]);
        color.rgbGreen = (BYTE)((val & pdib->gmask) >> pdib->iRShift[1]);
        color.rgbBlue = (BYTE)((val & pdib->bmask) >> pdib->iRShift[2]);
    } else if (bpp == 24) {
        color.rgbRed = (BYTE)((val >> 16) & 0xFF);
        color.rgbGreen = (BYTE)((val >> 8) & 0xFF);
        color.rgbBlue = (BYTE)(val & 0xFF);
    }
    return color;
}
```

## V. DibSetColor

Sets a pixel from an RGBQUAD.

- Works for 16-, 24-, 32-bit DIBs.
- Applies masks and shifts if needed.

```
void DibSetPixelColor(HDIB hdib, int x, int y, RGBQUAD color) {
    int bpp = DibBitCount(hdib);
    DWORD val = 0;
    DIBSTRUCT* pdib = (DIBSTRUCT*)hdib;

    if (bpp == 16 || bpp == 32) {
        val = ((color.rgbRed << pdib->iLShift[0]) & pdib->rmask) |
              ((color.rgbGreen << pdib->iLShift[1]) & pdib->gmask) |
              ((color.rgbBlue << pdib->iLShift[2]) & pdib->bmask);
    } else if (bpp == 24) {
        val = (color.rgbRed << 16) | (color.rgbGreen << 8) | color.rgbBlue;
    }
    DibSetPixel(hdib, x, y, val);
}
```

### ✓ Summary

- **DibPixelPtr** – pointer to a pixel.
- **DibGetPixel / DibSetPixel** – get/set raw pixel values.
- **DibGetPixelColor / DibSetPixelColor** – get/set pixels as RGB.
- Handles **all common bit depths** (1, 4, 8, 16, 24, 32).
- Safe: checks bounds and DIB addressability.

---

## DIBHELP.C – Part 3: Advanced DIB Functions (Simplified)

### I. Mask Shift Functions

MaskToRShift / MaskToLShift

- Calculate right/left shift values from color masks.
- Needed for 16- and 32-bit DIBs, especially **BI\_BITFIELDS**.

### II. DibCreateFromInfo

- Central function for creating a DIB section.
- Calls CreateDIBSection and allocates the DIBSTRUCT.
- Initializes **row pointers (ppRow)** for top-down or bottom-up DIBs.
- Handles color masks and bitfields.

All other creation/copy functions route through this one.

### III. DibDelete

- Frees memory for a DIB section.
- Validates the DIB before deleting to prevent errors.

### IV. DibCreate

- Creates an HDIB from width, height, bit count, and optional color table size.
- Internally calls DibCreateFromInfo to handle the heavy lifting.

### V. DibCopyToInfo

- Builds a BITMAPINFO from an existing DIB.
- Copies **color masks, color table**, and other info.
- Standardized way to extract DIB info for further use.

### VI. DibCopy

- Creates a new DIB from an existing one.
- Can optionally swap width and height.
- Uses DibCopyToInfo to prepare the BITMAPINFO.

## VII. DibCopyToPackedDib / DibCopyFromPackedDib

- **DibCopyToPackedDib:** Convert DIB → packed DIB (for files or clipboard).
- **DibCopyFromPackedDib:** Rebuild DIB from packed DIB.
- Work as a **tandem** for transferring DIBs.

## VIII. DibFileLoad / DibFileSave

- DibFileLoad: Read a DIB from a .DIB file.
- DibFileSave: Save a DIB to a file.
- Handles headers and pixel data automatically.

## IX. DibCopyToDdb

- Copies a DIB to a **device-dependent bitmap (DDB)** for fast screen display.
- Requires a palette and window handle for proper rendering.
- Uses CreateDIBitmap after selecting/realizing the palette.

## Summary

- **Shift functions:** extract color info for masked DIBs.
- **DibCreateFromInfo:** backbone of all DIB creation.
- **DibDelete / DibCreate:** manage memory and creation from parameters.
- **DibCopy / DibCopyToInfo:** duplicate and extract DIB data.
- **Packed DIB functions:** read/write to file or clipboard.
- **DibCopyToDdb:** optimized rendering to screen.

Clean, safe, and structured for **any application working with DIBs**.

---

## DIBHELP.H – Macros and HDIB Handle (Simplified)

### I. HDIB Handle

- Defined as `typedef void* HDIB;`
- Abstracts the internal DIB structure.
- Lets the implementation change without breaking application code.
- Can be cast to `DIBSTRUCT*` internally for pixel access.

**Why it matters:** Keeps the interface flexible, compatible with different DIB formats, and safe for application code.

### II. Pixel Access Macros

DIBHELP.H provides **macros** for fast pixel operations, avoiding function call overhead. They are **bit-depth specific**.

#### a) DibPixelPtr macros – get pointer to a pixel

```
DibPixelPtr1(hdib, x, y)    // 1-bit
DibPixelPtr4(hdib, x, y)    // 4-bit
DibPixelPtr8(hdib, x, y)    // 8-bit
DibPixelPtr16(hdib, x, y)   // 16-bit
DibPixelPtr24(hdib, x, y)   // 24-bit
DibPixelPtr32(hdib, x, y)   // 32-bit
```

#### b) DibGetPixel macros – read pixel value quickly.

```
DibGetPixel1(hdib, x, y)
DibGetPixel8(hdib, x, y)
DibGetPixel24(hdib, x, y)
```

c) **DibSetPixel macros** – write pixel value quickly.

```
DibSetPixel1(hdc, x, y, val)
DibSetPixel8(hdc, x, y, val)
DibSetPixel24(hdc, x, y, r, g, b)
```

 **Benefits:**

- No function call overhead → faster operations.
- Direct memory access → optimized performance.
- Smaller, cleaner code → better cache and CPU utilization.

 **Notes:**

- Use the **correct bit count macro** for the DIB.
- Ensure (x, y) is within the DIB bounds.
- Limited error checking—these macros assume coordinates and bit depth are correct.

### III. Determining Bit Count

Check biBitCount in the BITMAPINFOHEADER of your DIB.

Example:

```
int bpp = dib->bmiHeader.biBitCount;
if (bpp == 8) { /* 8-bit DIB */ }
```

Practical Example:

```
1 HDIB hdib = DibCreate(100, 100, 24, 0); // create a 24-bit DIB
2 DibSetPixel24(hdib, 10, 10, 255, 0, 0); // set pixel (10,10) to red
3 RGBQUAD color = DibGetPixelColor(hdib, 10, 10); // read pixel color
```

Here's another 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.

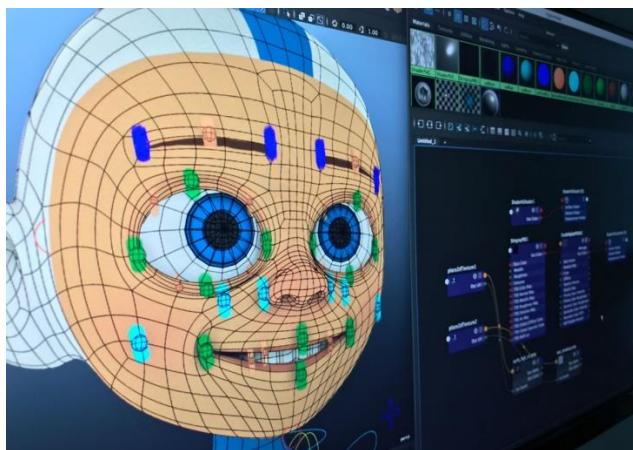
The implementation utilizes nested loops to iterate over the specified variations along the red, green, and blue axes. Within these loops, RGB values are calculated based on the current iteration and the total number of variations along each axis. The calculated RGB values are then assigned to the corresponding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

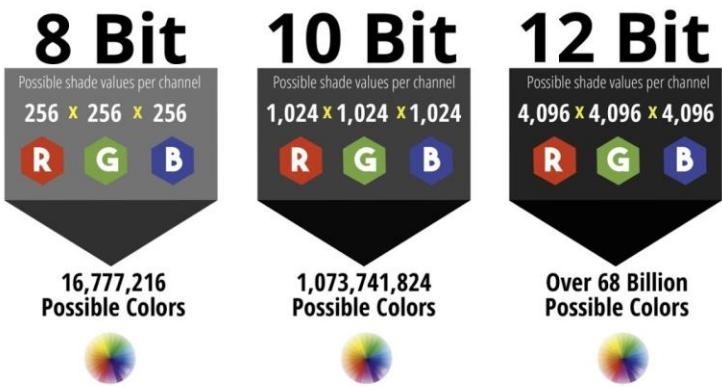


When [using the resulting palette](#) in a computer graphics application, you typically associate each color index with a pixel value or a specific graphical element. The palette can be used to determine the color of each element based on its index.

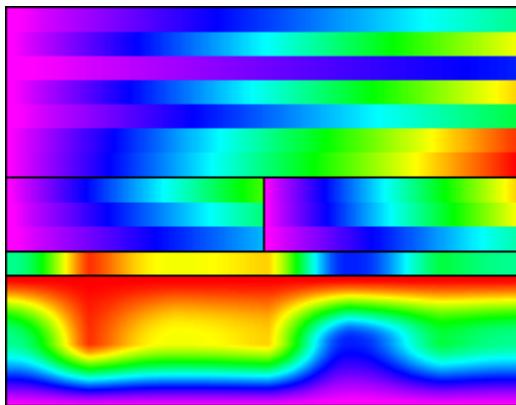
For example, if you're [rendering an image using indexed color mode](#), you would assign a color index to each pixel of the image, and then use the palette to look up the RGB value associated with that index to determine the pixel's color.

**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.



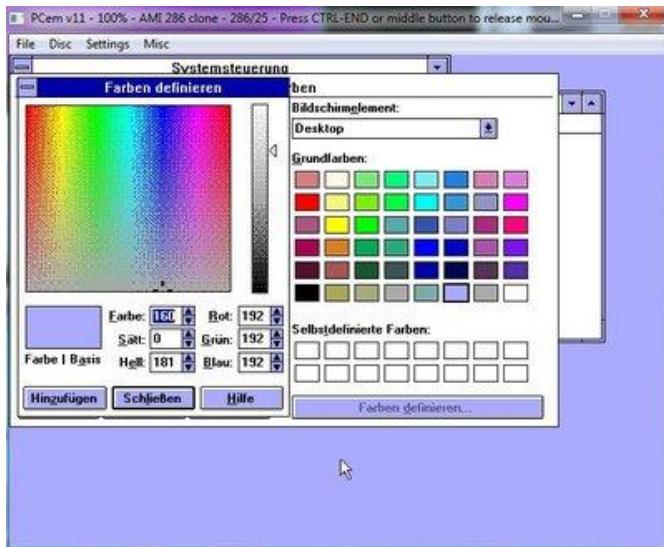
```
itsfoss@itsfoss-B250M-DS3H:~$ sudo apt update
Hit:1 https://brave-browser-apt-release.s3.brave.com stable InRelease
Hit:2 http://ppa.launchpad.net/sandromani/gimagerreader/ubuntu focal InRelease
Hit:3 http://archive.ubuntu.com/ubuntu focal InRelease
Hit:4 http://archive.canonical.com/ubuntu focal 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.

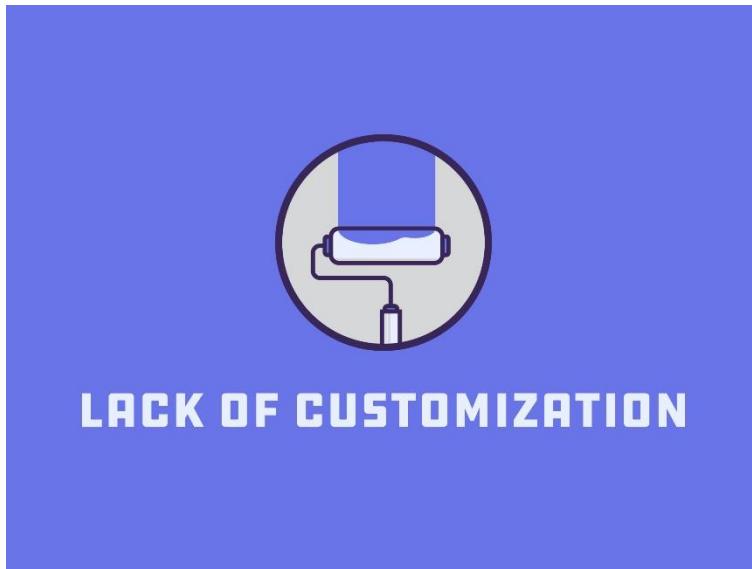


**Incompatibility with High-Color or True-Color Modes:** The standard 16 VGA colors palette is primarily designed for 8-bit color modes. If a graphics application is running in a high-color or true-color mode, the use of the standard VGA palette may not provide optimal color representation.



## 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

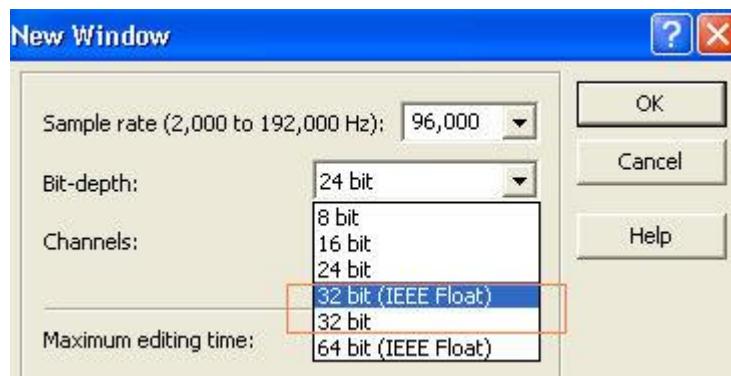
**Gray Shades:** Different numbers of gray shades can be used in a palette to achieve different effects. Using two shades of gray (black and white) creates high contrast. Adding more shades of gray improves the image quality, reducing contouring artifacts. The number of shades depends on the display 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...*