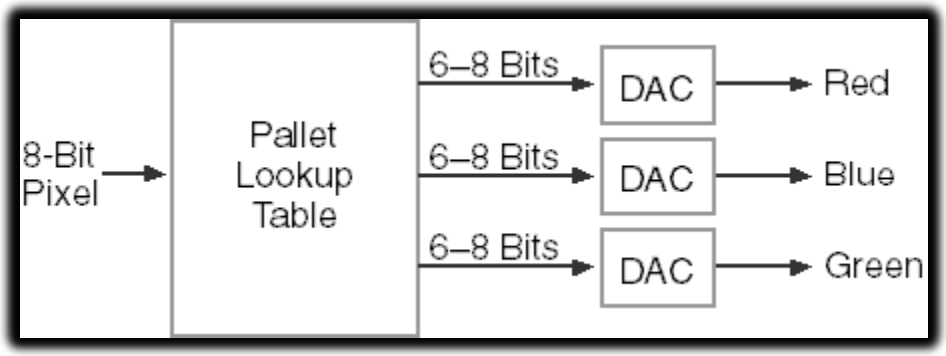
CHAPTER 16: THE PALETTE MANAGER - REVEALING THE WORLD OF 256 COLORS

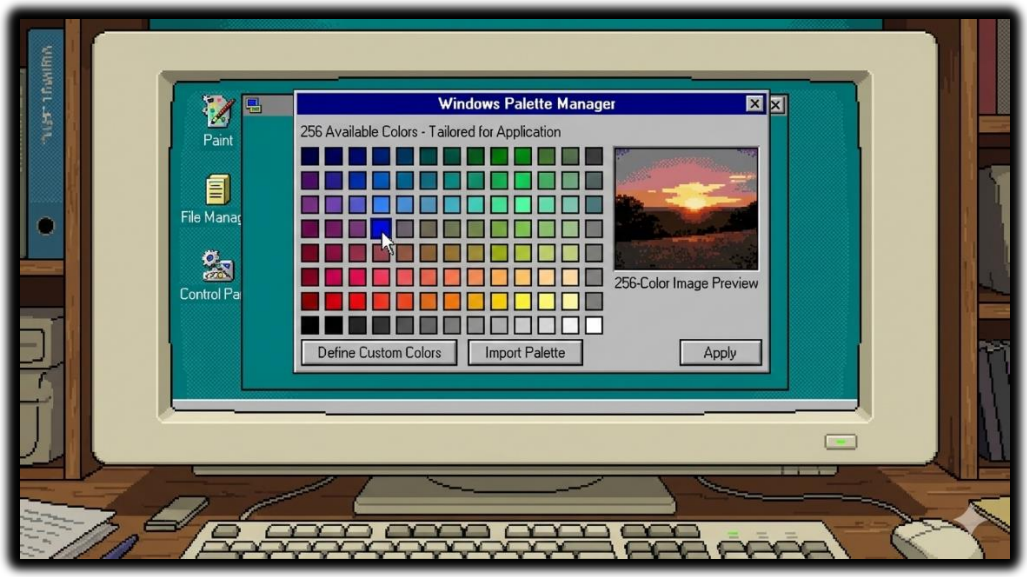
This chapter explores the Windows Palette Manager—a tool that exists largely because of hardware constraints.

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

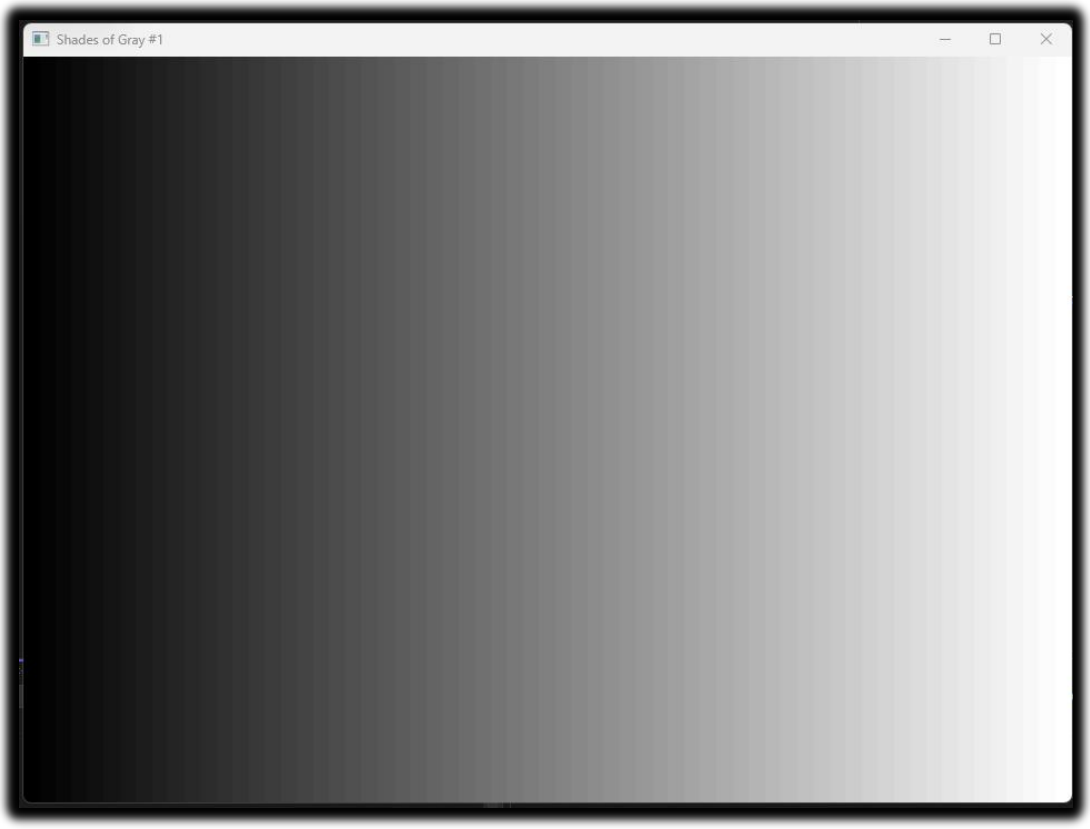
- **System palette** = global 256-color table
- **20 colors are locked** for Windows UI
- **236 colors max** for applications
- **Logical palettes** let apps request colors
- **Active window wins**
- Palette issues only exist in **8-bit modes**

🧩 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

```c
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
    static TCHAR szAppName[] = TEXT("Grays1");
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;
    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance;
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = szAppName;
    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
        return 0;
    }
    hwnd = CreateWindow(szAppName, TEXT("Shades of Gray #1"), WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL);
    ShowWindow(hwnd, iCmdShow);
    UpdateWindow(hwnd);
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static int cxClient, cyClient;
    HBRUSH hBrush;
    HDC hdc;
    int i;
    PAINTSTRUCT ps;
    RECT rect;
    switch (message) {
    case WM_SIZE:
        cxClient = LOWORD(lParam);
        cyClient = HIWORD(lParam);
        return 0;
    case WM_PAINT:
        hdc = BeginPaint(hwnd, &ps);

        // Draw the fountain of grays
        for (i = 0; i < 65; i++) {
            rect.left = i * cxClient / 65;
            rect.top = 0;
            rect.right = (i + 1) * cxClient / 65;
            rect.bottom = cyClient;

            hBrush = CreateSolidBrush(RGB(min(255, 4 * i), min(255, 4 * i), min(255, 4 * i)));
            FillRect(hdc, &rect, hBrush);
            DeleteObject(hBrush);
        }
        EndPaint(hwnd, &ps);
        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }

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



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

```c
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
    static TCHAR szAppName[] = TEXT("Grays2");
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;
    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance;
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = szAppName;

    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
        return 0;
    }
    hwnd = CreateWindow(szAppName, TEXT("Shades of Gray #2"), WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL);

    ShowWindow(hwnd, iCmdShow);
    UpdateWindow(hwnd);
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
```

```c
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static HPALETTE hPalette;
    static int cxClient, cyClient;
    HBRUSH hBrush;
    HDC hdc;
    int i;
    LOGPALETTE* plp;
    PAINTSTRUCT ps;
    RECT rect;
    switch (message) {
    case WM_CREATE:
        // Set up a LOGPALETTE structure and create a palette
        plp = malloc(sizeof(LOGPALETTE) + 64 * sizeof(PALETTEENTRY));
        plp->palVersion = 0x0300;
        plp->palNumEntries = 65;
        for (i = 0; i < 65; i++) {
            plp->palPalEntry[i].peRed = (BYTE)min(255, 4 * i);
            plp->palPalEntry[i].peGreen = (BYTE)min(255, 4 * i);
            plp->palPalEntry[i].peBlue = (BYTE)min(255, 4 * i);
            plp->palPalEntry[i].peFlags = 0;
        }
        hPalette = CreatePalette(plp);
        free(plp);
        return 0;
    case WM_SIZE:
        cxClient = LOWORD(lParam);
        cyClient = HIWORD(lParam);
        return 0;
    case WM_PAINT:
        hdc = BeginPaint(hwnd, &ps);
        // Select and realize the palette in the device context
        SelectPalette(hdc, hPalette, FALSE);
        RealizePalette(hdc);
```

```c
        // Draw the fountain of grays
        for (i = 0; i < 65; i++) {
            rect.left = i * cxClient / 64;
            rect.top = 0;
            rect.right = (i + 1) * cxClient / 64;
            rect.bottom = cyClient;
            hBrush = CreateSolidBrush(PALETTERGB(min(255, 4 * i), min(255, 4 * i), min(255, 4 * i)));
            FillRect(hdc, &rect, hBrush);
            DeleteObject(hBrush);
        }
        EndPaint(hwnd, &ps);
        return 0;

    case WM_QUERYNEWPALETTE:
        if (!hPalette)
            return FALSE;

        hdc = GetDC(hwnd);
        SelectPalette(hdc, hPalette, FALSE);
        RealizePalette(hdc);
        InvalidateRect(hwnd, NULL, TRUE);
        ReleaseDC(hwnd, hdc);
        return TRUE;

    case WM_PALETTECHANGED:
        if (!hPalette || (HWND)wParam == hwnd)
            break;

        hdc = GetDC(hwnd);
        SelectPalette(hdc, hPalette, FALSE);
        RealizePalette(hdc);
        UpdateColors(hdc);
        ReleaseDC(hwnd, hdc);
        break;
```

```c
    case WM_DESTROY:
        DeleteObject(hPalette);
        PostQuitMessage(0);
        return 0;
    }

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

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.

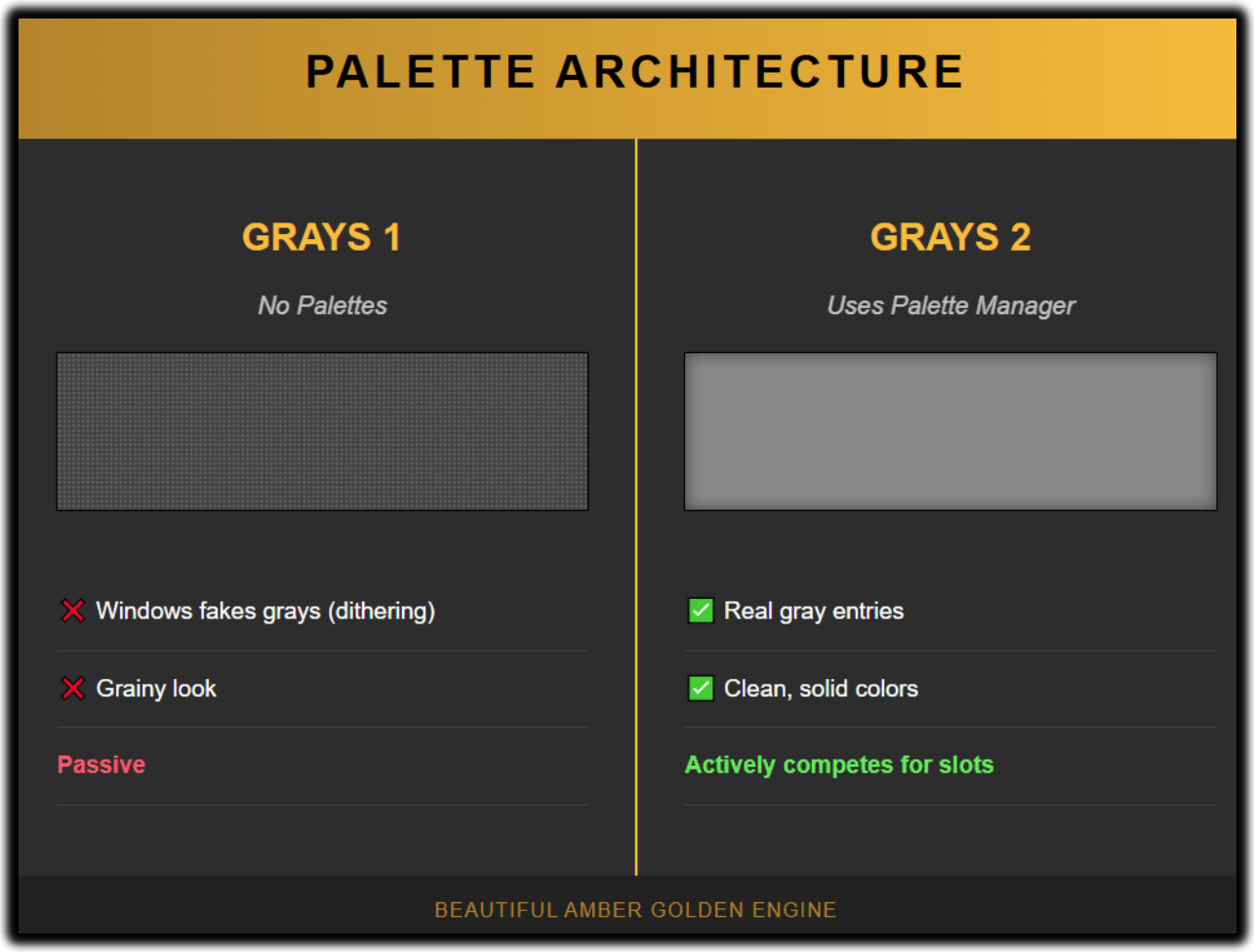
Everything else in the program supports that goal.

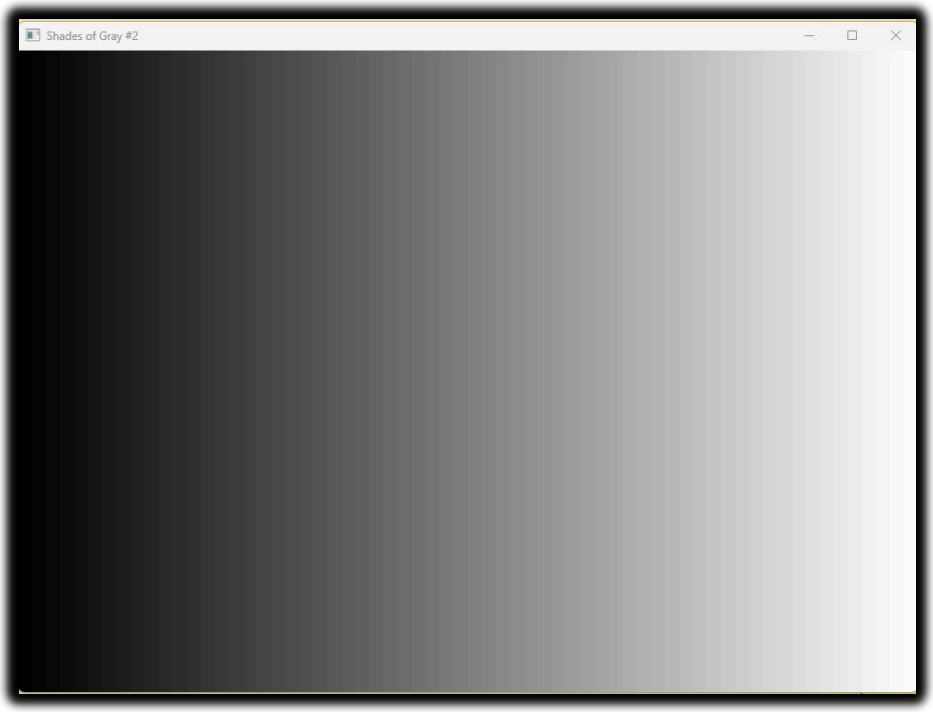
II. What GRAYS2 Does

- 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. PALETTERGB — The One Subtle Rule

When a logical palette is selected:

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

"Use *this* palette entry, not the closest guess"

So GRAYS2 uses **PALETTERGB** 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
- PALETTERGB ensures your colors come from *your* palette

LOGICAL PALETTE DEFINITION

```
// LOGPALETTE structure definition
typedef struct {
    WORD palVersion;
    WORD palNumEntries;
    PALETTEENTRY palPalEntry[1];
} LOGPALETTE, *PLOGPALETTE;
// PALETTEENTRY structure definition
typedef struct {
    BYTE peRed;
    BYTE peGreen;
    BYTE peBlue;
    BYTE peFlags;
} PALETTEENTRY, *PPALETTEENTRY;

// Creating a logical palette in GRAYS2
LOGPALETTE* plp = malloc(sizeof(LOGPALETTE) + 64 * sizeof(PALETTEENTRY));
plp->palVersion = 0x0300;
plp->palNumEntries = 65;
for (int i = 0; i < 65; i++) {
    plp->palPalEntry[i].peRed = (BYTE)min(255, 4 * i);
    plp->palPalEntry[i].peGreen = (BYTE)min(255, 4 * i);
    plp->palPalEntry[i].peBlue = (BYTE)min(255, 4 * i);
    plp->palPalEntry[i].peFlags = 0;
}
hPalette = CreatePalette(plp);
free(plp);
// Selecting and realizing the logical palette in WM_PAINT
case WM_PAINT:
    hdc = BeginPaint(hwnd, &ps);
    // Select and realize the palette in the device context
    SelectPalette(hdc, hPalette, FALSE);
    RealizePalette(hdc);
    // Draw the fountain of grays
    for (i = 0; i < 65; i++) {
        // Using PALETTERGB to specify color from logical palette
        hBrush = CreateSolidBrush(PALETTERGB(min(255, 4 * i), min(255, 4 * i), min(255, 4 * i)));
        FillRect(hdc, &rect, hBrush);
        DeleteObject(hBrush);
    }
    EndPaint(hwnd, &ps);
    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

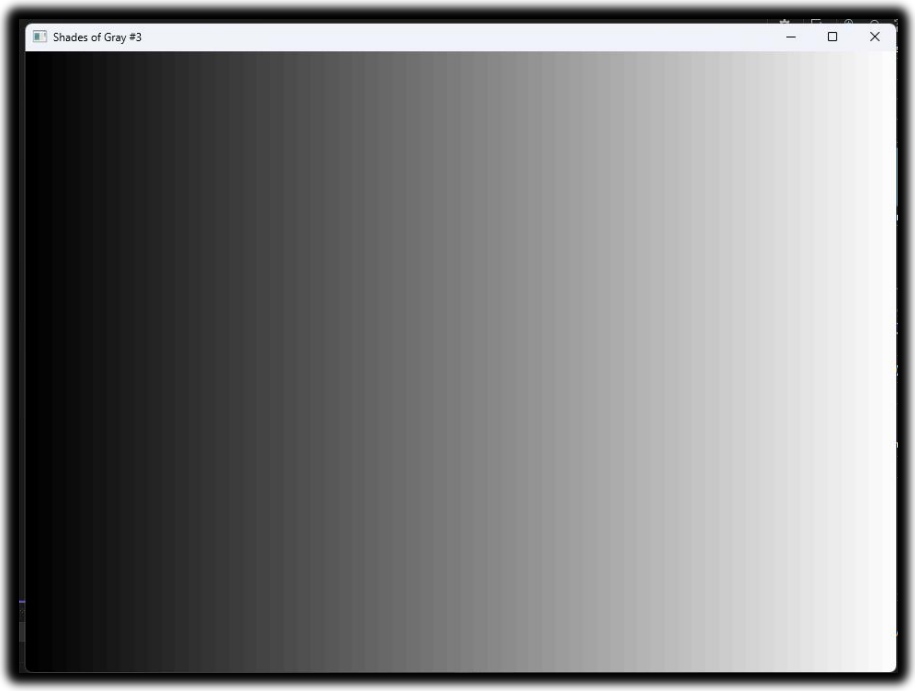
That's it.
Everything else is just plumbing.

GRAYS3 PROGRAM

```
// LOGPALETTE structure definition
typedef struct {
    WORD palVersion;
    WORD palNumEntries;
    PALETTEENTRY palPalEntry[1];
} LOGPALETTE, *PLOGPALETTE;

// PALETTEENTRY structure definition
typedef struct {
    BYTE peRed;
    BYTE peGreen;
    BYTE peBlue;
    BYTE peFlags;
} PALETTEENTRY, *PPALETTEENTRY;

// Creating a logical palette in GRAYS3
LOGPALETTE* plp = malloc(sizeof(LOGPALETTE) + 64 * sizeof(PALETTEENTRY));
plp->palVersion = 0x0300;
plp->palNumEntries = 65;

for (int i = 0; i < 65; i++) {
    plp->palPalEntry[i].peRed = (BYTE)min(255, 4 * i);
    plp->palPalEntry[i].peGreen = (BYTE)min(255, 4 * i);
    plp->palPalEntry[i].peBlue = (BYTE)min(255, 4 * i);
    plp->palPalEntry[i].peFlags = 0;
}

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

// WM_PAINT message processing in GRAYS3
case WM_PAINT:
    hdc = BeginPaint(hwnd, &ps);

    // Select and realize the palette in the device context
    SelectPalette(hdc, hPalette, FALSE);
    RealizePalette(hdc);

    // Draw the fountain of grays using PALETTEINDEX
    for (i = 0; i < 65; i++) {
        rect.left = i * cxClient / 64;
        rect.top = 0;
        rect.right = (i + 1) * cxClient / 64;
        rect.bottom = cyClient;

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

    EndPaint(hwnd, &ps);
    return 0;

// Handling WM_QUERYNEWPALETTE and WM_PALETTECHANGED
case WM_QUERYNEWPALETTE:
    if (!hPalette)
        return FALSE;

    hdc = GetDC(hwnd);
    SelectPalette(hdc, hPalette, FALSE);
    RealizePalette(hdc);
    InvalidateRect(hwnd, NULL, FALSE);
    ReleaseDC(hwnd, hdc);
    return TRUE;

case WM_PALETTECHANGED:
    if (!hPalette || (HWND)wParam == hwnd)
        break;

    hdc = GetDC(hwnd);
    SelectPalette(hdc, hPalette, FALSE);
    RealizePalette(hdc);
    UpdateColors(hdc);
    ReleaseDC(hwnd, hdc);
    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:

```
00011010
11100101
```

Windows applies the ROP like this:

```
newBits = oldBits XOR mask
```

Only *after that* does Windows interpret those bits:

```
pixel bits → palette index → RGB color
```

💥 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

```c
// SYSPAL1 - Displays the contents of the system palette

#include <windows.h>

// Function to check if the display supports a 256-color palette
BOOL CheckDisplay(HWND hwnd);

// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);

// Application entry point
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;

    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance;
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = TEXT("SysPal1");

    // Register the window class
    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), TEXT("SysPal1"), MB_ICONERROR);
        return 0;
    }

    // Create the main window
    hwnd = CreateWindow(TEXT("SysPal1"), TEXT("System Palette #1"), WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL);
    if (!hwnd)
        return 0;

    // Show and update the window
    ShowWindow(hwnd, iCmdShow);
    UpdateWindow(hwnd);

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

    return msg.wParam;
}

// Function to check if the display supports a 256-color palette
BOOL CheckDisplay(HWND hwnd) {
    HDC hdc;
    int iPalSize;

    hdc = GetDC(hwnd);
    iPalSize = GetDeviceCaps(hdc, SIZEPALETTE);
    ReleaseDC(hwnd, hdc);

    // Display an error message if not 256 colors
    if (iPalSize != 256) {
        MessageBox(hwnd, TEXT("This program requires that the video display mode have a 256-color palette."),
            TEXT("SysPal1"), MB_ICONERROR);
        return FALSE;
    }

    return TRUE;
}

```

```
// Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static int cxClient, cyClient;
    static SIZE sizeChar;
    HDC hdc;
    PAINTSTRUCT ps;
    PALETTEENTRY pe[256];
    TCHAR szBuffer[16];

    switch (message) {
    case WM_CREATE:
        // Initialize and check display mode
        if (!CheckDisplay(hwnd))
            return -1;

        hdc = GetDC(hwnd);
        SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
        GetTextExtentPoint32(hdc, TEXT("FF-FF-FF"), 10, &sizeChar);
        ReleaseDC(hwnd, hdc);
        return 0;

    case WM_SIZE:
        // Update client area dimensions on size change
        cxClient = LOWORD(lParam);
        cyClient = HIWORD(lParam);
        return 0;

    case WM_PAINT:
        // Paint the system palette entries
        hdc = BeginPaint(hwnd, &ps);
        SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
        GetSystemPaletteEntries(hdc, 0, 256, pe);

        for (int i = 0, x = 0, y = 0; i < 256; i++) {
            wsprintf(szBuffer, TEXT("%02X-%02X-%02X"), pe[i].peRed, pe[i].peGreen, pe[i].peBlue);
            TextOut(hdc, x, y, szBuffer, lstrlen(szBuffer));

             if ((x += sizeChar.cx) + sizeChar.cx > cxClient) {
                 x = 0;

                 if ((y += sizeChar.cy) > cyClient)
                     break;
             }
         }

         EndPaint(hwnd, &ps);
         return 0;

     case WM_PALETTECHANGED:
         // Invalidate client area on palette change
         InvalidateRect(hwnd, NULL, FALSE);
         return 0;

     case WM_DESTROY:
         // Clean up and exit
         PostQuitMessage(0);
         return 0;
    }

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

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.

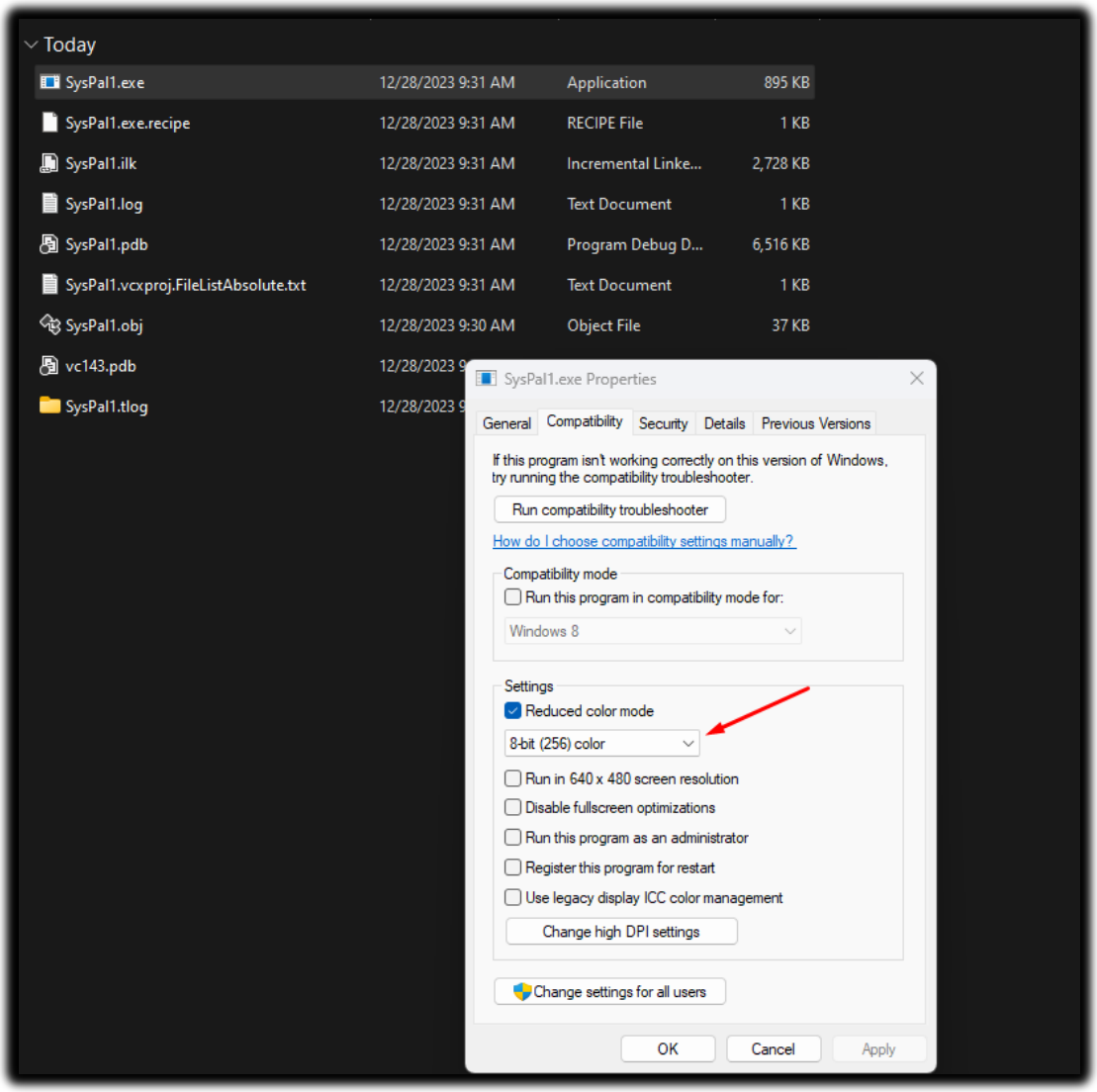
This makes the program especially useful for developers working with:

- 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

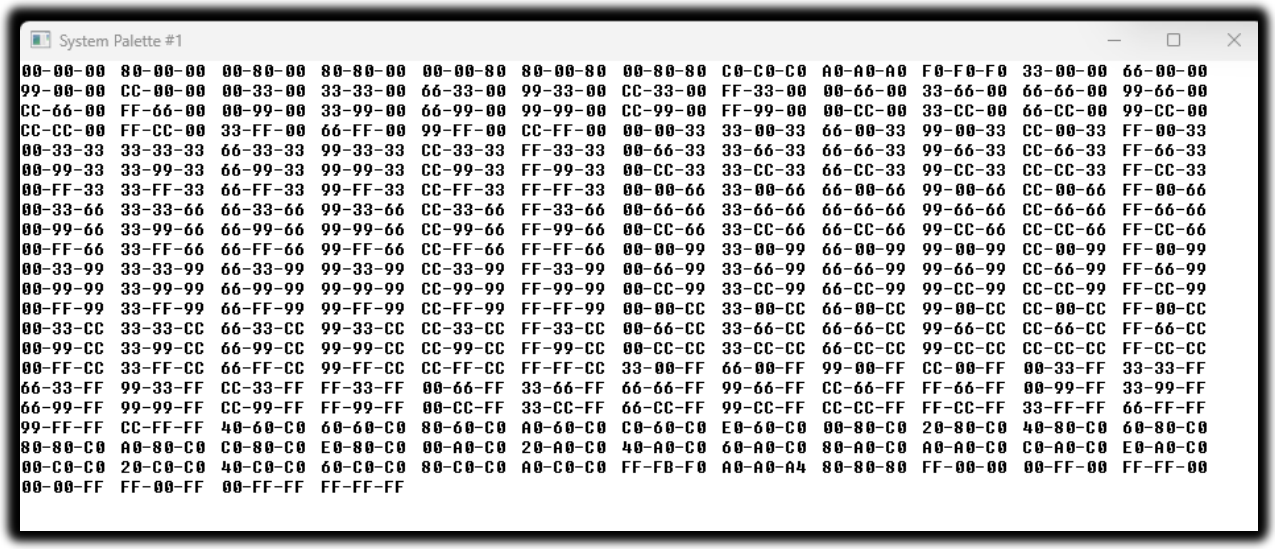
The program will not display meaningful output the first time if the system is not already in 256-color mode.


Today
SysPal1.exe 12/28/2023 9:31 AM Application 895 KB
SysPal1.exe.recipe 12/28/2023 9:31 AM RECIPE File 1 KB
SysPal1.ilk 12/28/2023 9:31 AM Incremental Linke... 2,728 KB
SysPal1.log 12/28/2023 9:31 AM Text Document 1 KB
SysPal1.pdb 12/28/2023 9:31 AM Program Debug D... 6,516 KB
SysPal1.vcxproj.FileListAbsolute.txt 12/28/2023 9:31 AM Text Document 1 KB
SysPal1.obj 12/28/2023 9:30 AM Object File 37 KB
vc143.pdb
SysPal1.tlog
SysPal1.exe Properties
General
Compatibility
Security
Details
Previous Versions
If this program isn't working correctly on this version of Windows, try running the compatibility troubleshooter.
Run compatibility troubleshooter
How do I choose compatibility settings manually?
Compatibility mode
Run this program in compatibility mode for:
Windows 8
Settings
Reduced color mode
8-bit (256) color
Run in 640 x 480 screen resolution
Disable fullscreen optimizations
Run this program as an administrator
Register this program for restart
Use legacy display ICC color management
Change high DPI settings
Change settings for all users
OK
Cancel
Apply

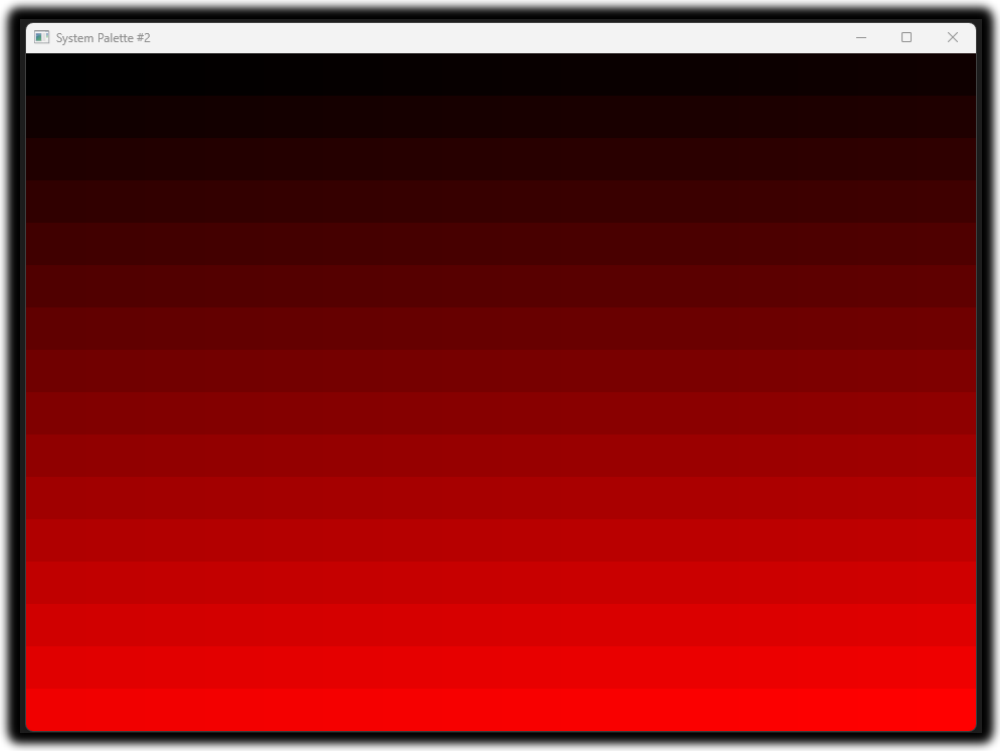
Output

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.



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.

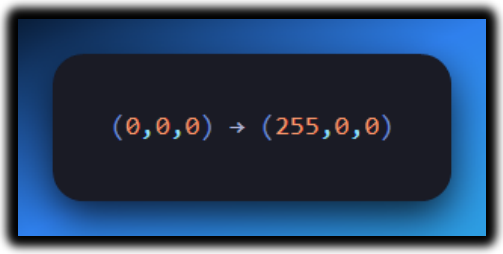
III. Why the Output Might Be Red & Black

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.

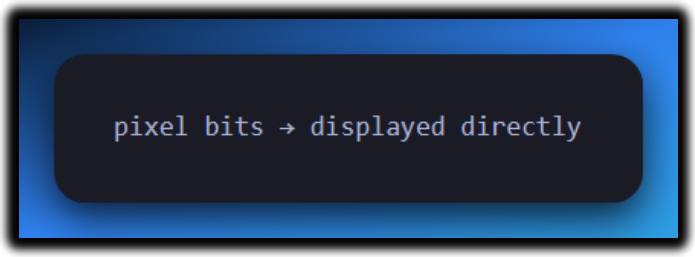
VI. Why BitBlt / StretchBlt Are Mentioned

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 🔑

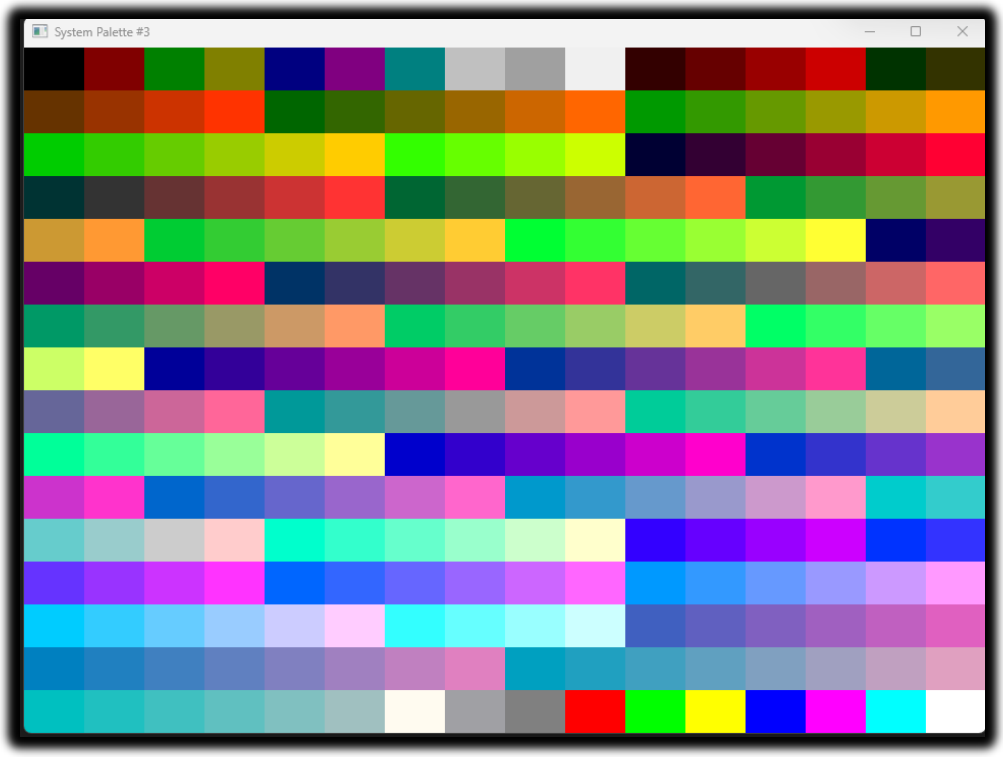
SYSPAL2 is not a demo app.
It's a **palette truth machine**.

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 × 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.

Its 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

```c
/*--------------------------------------
   FADER.C -- Palette Animation Demo
              (c) Charles Petzold, 1998
  --------------------------------------*/

#include <windows.h>

#define ID_TIMER 1

TCHAR szAppName [] = TEXT ("Fader") ;
TCHAR szTitle   [] = TEXT ("Fader: Palette Animation Demo") ;

static LOGPALETTE lp ;

HPALETTE CreateRoutine (HWND hwnd)
{
     HPALETTE hPalette ;

     lp.palVersion             = 0x0300 ;
     lp.palNumEntries          = 1 ;
     lp.palPalEntry[0].peRed   = 255 ;
     lp.palPalEntry[0].peGreen = 255 ;
     lp.palPalEntry[0].peBlue  = 255 ;
     lp.palPalEntry[0].peFlags = PC_RESERVED ;

     hPalette = CreatePalette (&lp) ;

     SetTimer (hwnd, ID_TIMER, 50, NULL) ;
     return hPalette ;
}

void PaintRoutine (HDC hdc, int cxClient, int cyClient)
{
     static TCHAR szText [] = TEXT (" Fade In and Out ") ;
     int          x, y ;
     SIZE         sizeText ;

     SetTextColor (hdc, PALETTEINDEX (0)) ;
     GetTextExtentPoint32 (hdc, szText, lstrlen (szText), &sizeText) ;

     for (x = 0 ; x < cxClient ; x += sizeText.cx)
     for (y = 0 ; y < cyClient ; y += sizeText.cy)
     {
          TextOut (hdc, x, y, szText, lstrlen (szText)) ;
     }
     return ;
}

void TimerRoutine (HDC hdc, HPALETTE hPalette)
{
     static BOOL bFadeIn = TRUE ;

     if (bFadeIn)
     {
          lp.palPalEntry[0].peRed   -= 4 ;
          lp.palPalEntry[0].peGreen -= 4 ;

          if (lp.palPalEntry[0].peRed == 3)
               bFadeIn = FALSE ;
     }
     else
     {
          lp.palPalEntry[0].peRed   += 4 ;
          lp.palPalEntry[0].peGreen += 4 ;

          if (lp.palPalEntry[0].peRed == 255)
               bFadeIn = TRUE ;
     }

     AnimatePalette (hPalette, 0, 1, lp.palPalEntry) ;
     return ;
}

void DestroyRoutine (HWND hwnd, HPALETTE hPalette)
{
     KillTimer (hwnd, ID_TIMER) ;
     DeleteObject (hPalette) ;
     return ;
}
```

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

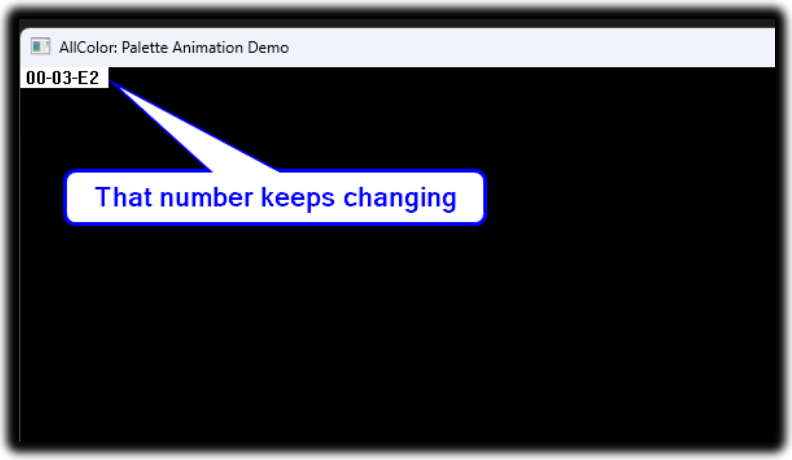
---

✅ 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

```c
/*------------------------------------------
   ALLCOLOR.C -- Palette Animation Demo
                 (c) Charles Petzold, 1998
   ------------------------------------------*/

#include <windows.h>
#define ID_TIMER    1

TCHAR szAppName [] = TEXT ("AllColor") ;
TCHAR szTitle   [] = TEXT ("AllColor: Palette Animation Demo") ;
static int          iIncr ;
static PALETTEENTRY pe ;

HPALETTE CreateRoutine (HWND hwnd)
{
     HDC         hdc ;
     HPALETTE    hPalette ;
     LOGPALETTE lp ;
     // Determine the color resolution and set iIncr
     hdc = GetDC (hwnd) ;
     iIncr = 1 << (8 - GetDeviceCaps (hdc, COLORRES) / 3) ;
     ReleaseDC (hwnd, hdc) ;
     // Create the logical palette
     lp.palVersion             = 0x0300 ;
     lp.palNumEntries          = 1 ;
     lp.palPalEntry[0].peRed   = 0 ;
     lp.palPalEntry[0].peGreen = 0 ;
     lp.palPalEntry[0].peBlue  = 0 ;
     lp.palPalEntry[0].peFlags = PC_RESERVED ;

     hPalette = CreatePalette (&lp) ;
     // Save global for less typing
     pe = lp.palPalEntry[0] ;
     SetTimer (hwnd, ID_TIMER, 10, NULL) ;
     return hPalette ;
}

void DisplayRGB (HDC hdc, PALETTEENTRY * ppe)
{
     TCHAR szBuffer [16] ;
     wsprintf (szBuffer, TEXT (" %02X-%02X-%02X "), ppe->peRed, ppe->peGreen, ppe->peBlue) ;
     TextOut (hdc, 0, 0, szBuffer, lstrlen (szBuffer)) ;
}

void PaintRoutine (HDC hdc, int cxClient, int cyClient)
{
     HBRUSH   hBrush ;
     RECT     rect ;
     //Draw Palette Index 0 on entire window
     hBrush = CreateSolidBrush (PALETTEINDEX (0)) ;
     SetRect (&rect, 0, 0, cxClient, cyClient) ;
     FillRect (hdc, &rect, hBrush) ;
     DeleteObject (SelectObject (hdc, GetStockObject (WHITE_BRUSH))) ;
     // Display the RGB value
     DisplayRGB (hdc, &pe) ;
     return ;
}

void TimerRoutine (HDC hdc, HPALETTE hPalette)
{
     static BOOL  bRedUp = TRUE, bGreenUp = TRUE, bBlueUp = TRUE ;
     // Define new color value
     pe.peBlue += (bBlueUp ? iIncr : -iIncr) ;
     if (pe.peBlue == (BYTE) (bBlueUp ? 0 : 256 - iIncr))
     {
          pe.peBlue = (bBlueUp ? 256 - iIncr : 0) ;
          bBlueUp ^= TRUE ;
          pe.peGreen += (bGreenUp ? iIncr : -iIncr) ;
          if (pe.peGreen == (BYTE) (bGreenUp ? 0 : 256 - iIncr))
          {
               pe.peGreen = (bGreenUp ? 256 - iIncr : 0) ;
               bGreenUp ^= TRUE ;
               pe.peRed += (bRedUp ? iIncr : -iIncr) ;
               if (pe.peRed == (BYTE) (bRedUp ? 0 : 256 - iIncr))
               {
                    pe.peRed = (bRedUp ? 256 - iIncr : 0) ;
                    bRedUp ^= TRUE ;
               }
          }
     }
     // Animate the palette
     AnimatePalette (hPalette, 0, 1, &pe) ;
     DisplayRGB (hdc, &pe) ;
     return ;
}

void DestroyRoutine (HWND hwnd, HPALETTE hPalette)
{
     KillTimer (hwnd, ID_TIMER) ;
     DeleteObject (hPalette) ;
     return ;
}
```



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

---

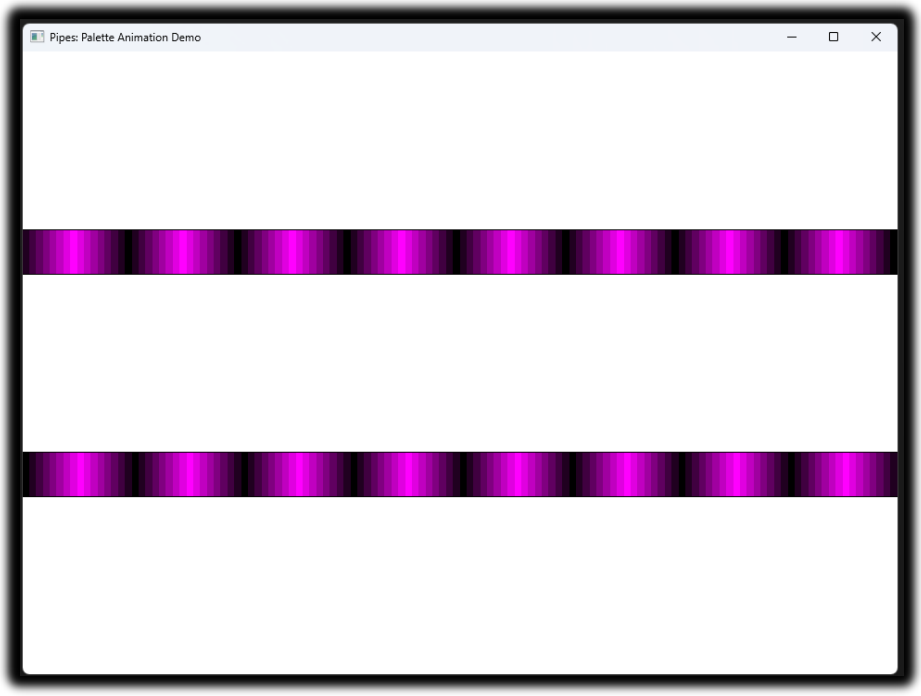
Notes for Developers

- 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.

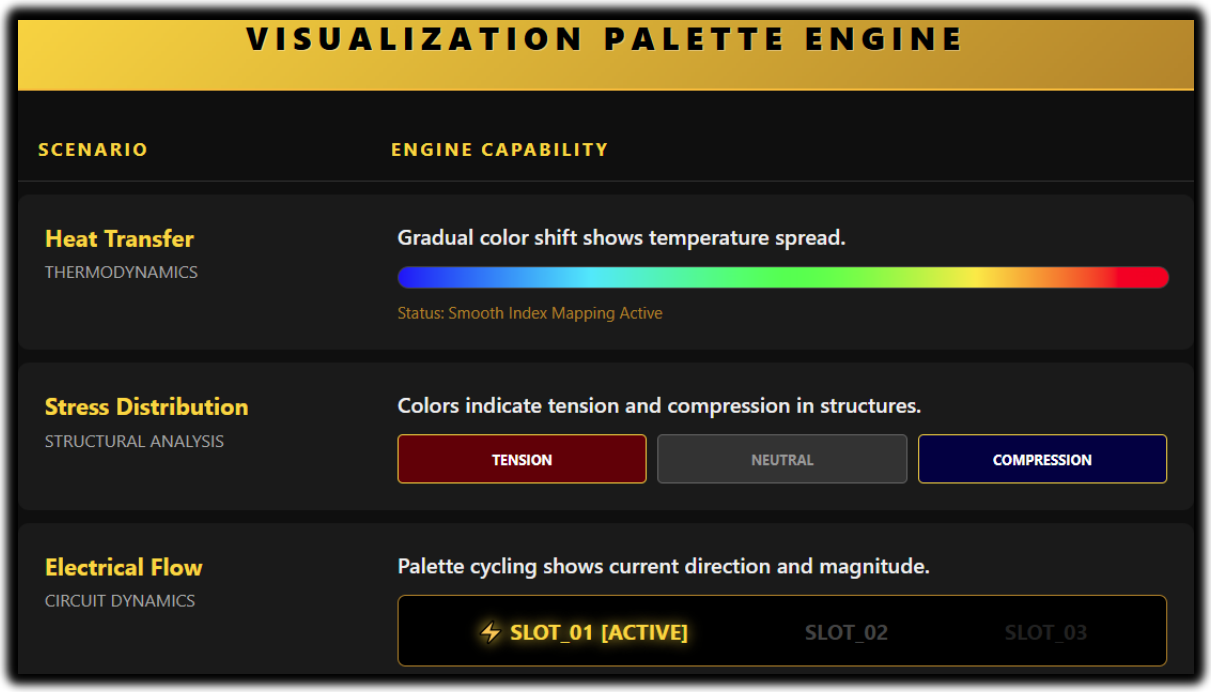
VI. Clean Exit: DestroyRoutine

- 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:



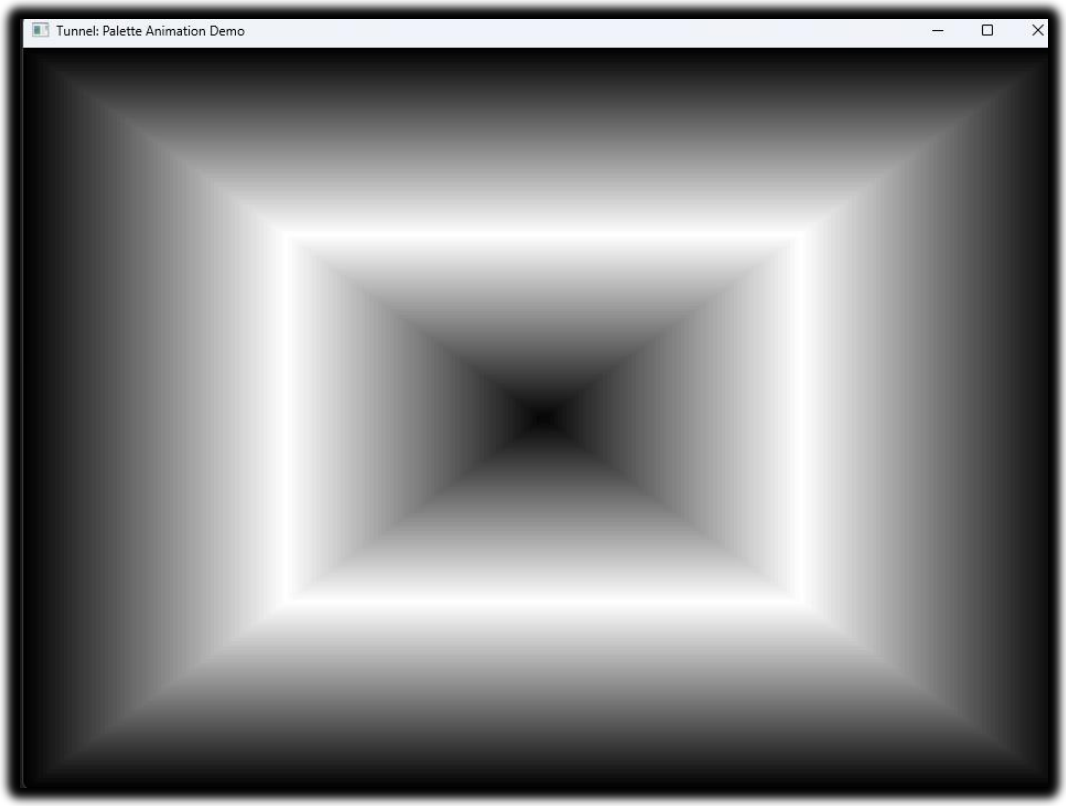
✅ Summary

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 (≤ 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.

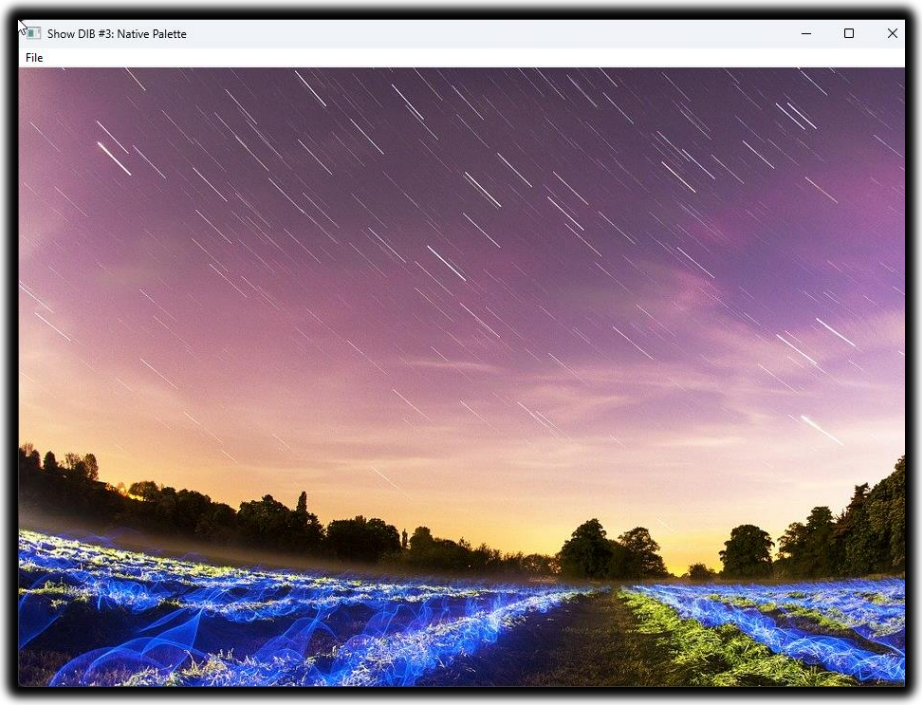
V. Summary

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. DibSetPixelColor

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**.

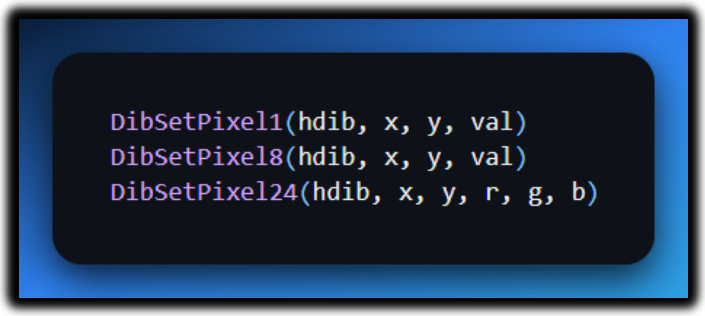
a) DibPixelPtr macros – get pointer to a pixel

```
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.



✅ **Benefits:**

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

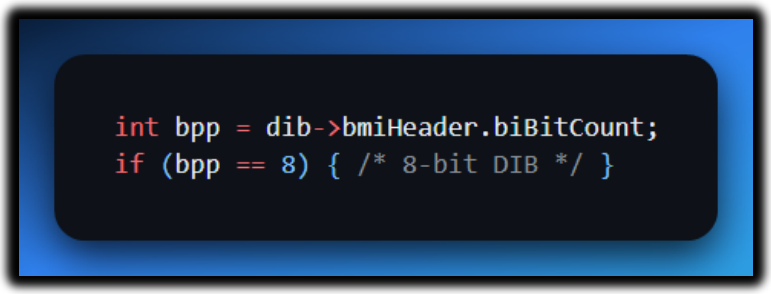
⚠️ **Notes:**

- 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:



Practical Example:

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

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

```
// Assuming the DIB has been loaded and the HDIB handle is available

// Get the pixel value at coordinates (x, y) in an 8-bit DIB
unsigned char pixelValue = DibGetPixel8(hdib, x, y);

// Set the pixel at coordinates (x, y) in a 24-bit DIB to the RGB value (r, g, b)
DibSetPixel24(hdib, x, y, r, g, b);
```

DIB / BITMAP HANDLING IN C++

I. Load an Image (Deserialize)

Reads a BMP file from disk into a program as an HBITMAP.

```
HBITMAP DeserializeBitmap(const char* filename) {
    return (HBITMAP)LoadImage(NULL, filename, IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE);
}
```

Key point: Only supports BMP here. Returns a handle to the bitmap.

II. Save an Image (Serialize)

Writes an HBITMAP back to disk.

```
void SerializeBitmap(HBITMAP hBitmap, const char* filename) {
    BITMAP bmp;
    GetObject(hBitmap, sizeof(BITMAP), &bmp);

    HDC hdc = GetDC(NULL);
    HDC hdcMem = CreateCompatibleDC(hdc);
    SelectObject(hdcMem, hBitmap);

    SaveDIBAsBMP(hdcMem, filename);

    ReleaseDC(NULL, hdc);
    DeleteDC(hdcMem);
}
```

Key point: Converts in-memory bitmap to file. SaveDIBAsBMP handles BMP serialization.

III. Working with Pixels (HDIB)

HDIB = handle to a Device-Independent Bitmap (DIB). Lets you access or modify individual pixels.

```
HDIB hdib = CreateDIB(...);        // Create a DIB handle
DibSetPixel8(hdib, x, y, p);       // Set pixel at (x, y) to value 'p'
unsigned char pixelValue = DibGetPixel8(hdib, x, y); // Get pixel value
DestroyDIB(hdib);                   // Release when done
```

- DibSetPixel8 / DibGetPixel8 → 8-bit pixel operations (256 colors max).
- Use **DibPixelPtr macros** for direct memory access: faster than using the getter/setter functions.
- HDIB abstracts the DIB structure, so you don't manipulate raw memory manually.

IV. Quick Analogy

- **Deserialize:** Take a photo out of storage.
- **HDIB:** Use a tiny brush to edit pixels.
- **Serialize:** Save it back.

V. Why use HDIB handles?

- Encapsulates DIB structure.
- Easier to pass between functions.
- Lifetime management of bitmaps is explicit.
- Specific to DIBs, unlike file handles or window handles.

✅ **Tip:** Use direct pixel pointers for large-scale pixel operations; it's much faster than function calls for each pixel.

DIBBLE PROGRAM

Dibble is a Windows sample program for **loading, displaying, editing, and converting Device Independent Bitmaps (DIBs)**.

It demonstrates core DIB operations: display, scroll, flip, rotate, palette handling, clipboard, and color-depth conversion.

WndProc – Handles window messages, user commands, and display updates.

Static Handles:

- hdib – Current DIB
- hPalette – Current palette
- hBitmap – Device-dependent bitmap for fast display

Modules:

- DIBHELP – Core DIB operations (load, save, copy, pixel access)
- DIBPAL – Palette creation
- DIBCONV – Color depth conversions

File Operations

- Open/Save DIB files using standard Windows dialogs
- Deletes existing DIB, palette, and bitmap before loading new file
- Sends messages: WM_USER_DELETEDIB, WM_USER_CREATEPAL

Display Options

- Actual size, centered, stretched, isotropically stretched
- Uses DisplayDib function (via BitBlt/StretchBlt)
- Scroll bars visible only for actual-size display
- fHalftonePalette flag ensures smooth rendering for stretched images

Clipboard Support

- Cut, copy, paste DIBs
- Uses packed DIB format for clipboard operations

Image Transformations

- **Flip** – horizontal flip using DibFlipHorizontal
- **Rotate** – 90° right rotation using DibRotateRight
- Uses DibGetPixel / DibSetPixel macros for pixel-level operations
- Handles multiple bit depths (1, 4, 8, 16, 24, 32-bit)

Palette Management

Supports multiple palette types:

- DIB color table, halftone, all-purpose
- Grayscale (various levels), uniform colors, optimized palettes

User commands trigger deletion/creation of palette via messages:

- WM_USER_DELETEPAL, WM_USER_CREATEPAL

Palette changes recreate hBitmap for fast rendering

Printing

- Prints DIBs via standard printer DC
- DisplayDib used for consistent rendering

🎟️ Miscellaneous features

Menu & Accelerators:

- Menus: File, Edit, Show, Palette, Gray Shades, Uniform Colors, Optimized, Convert, Help
- Accelerators: Ctrl+O, Ctrl+S, Ctrl+C, Ctrl+V, Ctrl+X, Delete

Resource IDs (IDM_*) map menu items to commands

Dynamic Handle Recreation:

- hBitmap must be recreated whenever a new DIB or palette is created

Scrolling:

- Only enabled for actual-size DIBs
- Scroll positions passed to DisplayDib during WM_PAINT

🎞️ Implementation Highlights

- Uses memory DCs for off-screen bitmap manipulation
- DisplayDib supports multiple display modes with halftone/color-on-color distinction
- Flip/rotate operations demonstrate pixel-level transformations
- Clipboard handling shows packing/unpacking DIBs
- Modular design: WndProc orchestrates message handling, palette, and DIB management

✅ Summary:

Dibble is an educational program showing how to manipulate DIBs in Windows, including display, scrolling, palette handling, file/clipboard operations, and basic image transformations. It uses a modular design with messages and static handles to manage resources efficiently.

DIBPAL.C

This file is essentially a **Palette Factory**. Its job is to look at an image (or a math formula) and spit out a HPALETTE (Windows Palette Handle) so we can draw 256-color images without them looking like garbage.

Part 1: The Dumb Generators (Basic Math)

These functions don't look at the image pixels deeply. They just follow a recipe.

PALETTE GENERATION STRATEGIES

Function	Strategy	Best For...
`DibPalDibTable`	**The Copy-Cat.** It reads the Color Table in the DIB file header and copies exact colors into a Windows Palette.	Images with a perfect 8-bit color table already attached.
`DibPalVga`	**The Retro.** Creates the standard 16 colors (Bright/Dark Red, Cyan, etc.) used by old DOS/Win 3.1 machines.	Simulating old-school interfaces or very simple icons.
`DibPalUniformGrays`	**50 Shades of Grey.** Ignores color; creates a gradient from Black (0,0,0) to White (255,255,255).	Black & White photos (Medical imaging, X-Rays).
`DibPalUniformColors`	**The Cube.** Slices the RGB 3D cube evenly (e.g., 6 shades per Red, Green, and Blue).	Scientific data where you need a mathematically predictable spread.
`DibPalAllPurpose`	**The Safety Net.** A "Best Hits" mix: 216 standard colors + 31 smooth grays + system colors.	General purpose use when image content is unknown.

Part 2: The "Smart" Generators (Algorithms)

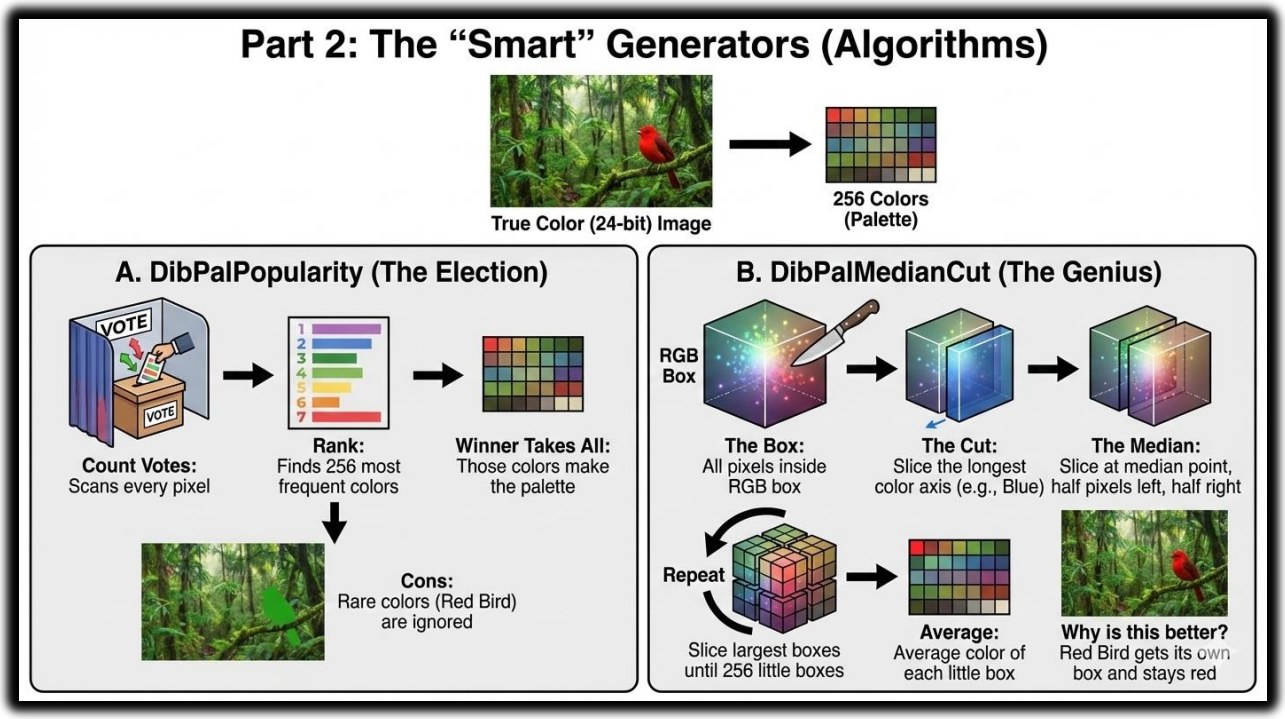
These functions analyze the actual pixels in the image to create a custom palette. This is used when you load a True Color (24-bit) image and need to squash it down to 256 colors.

I. DibPalPopularity (The Election)

The Logic:

1. **Count Votes:** It scans every pixel in the image.
2. **Rank:** It finds the 256 colors that appear most often.
3. **Winner Takes All:** Those 256 colors make the palette. Rare colors are ignored completely.

- **Pros:** Very fast.
- **Cons:** If you have a picture of a forest (10,000 greens) with a tiny red bird (50 reds), the bird might turn green because "Red" didn't get enough votes to make the palette.



II. DibPalMedianCut (The Genius)

This is the industry standard for color reduction. It uses the **CutBox** and **FindAverageColor** functions.

The Logic (The Cake Slicing Analogy): Imagine all the colors in your image are distinct points floating inside a 3D box (Red, Green, Blue axes).

1. **The Box:** Put all 1 million pixels inside one giant RGB box.
2. **The Cut:** Find which side of the box is longest (widest color range). Let's say the image has lots of Blues (Dark Blue to Light Blue). The "Blue" axis is long.

3. **The Median:** Find the median point where half the pixels are on the left, and half are on the right. Slice the box there.
4. **Repeat:** Now you have 2 boxes. Repeat the process recursively. Keep slicing the biggest boxes until you have 256 little boxes.
5. **Average:** Take the *average* color of all pixels inside each little box. That becomes one palette entry.

- **Why is this better?** Unlike the "Popularity" method, this ensures that even if there are only a few Red pixels (the bird), if they are far away from the Greens in 3D space, they get their own little box. The bird stays red.

Part 3: Deep Dive into CutBox (The Recursive Engine)

You asked for the purpose of the **Recursive** nature of CutBox.

The Problem: If we just sliced the color cube into a grid (like DibPalUniformColors), we waste space. We might have 50 palette slots for "Purple," but the image has zero purple pixels.

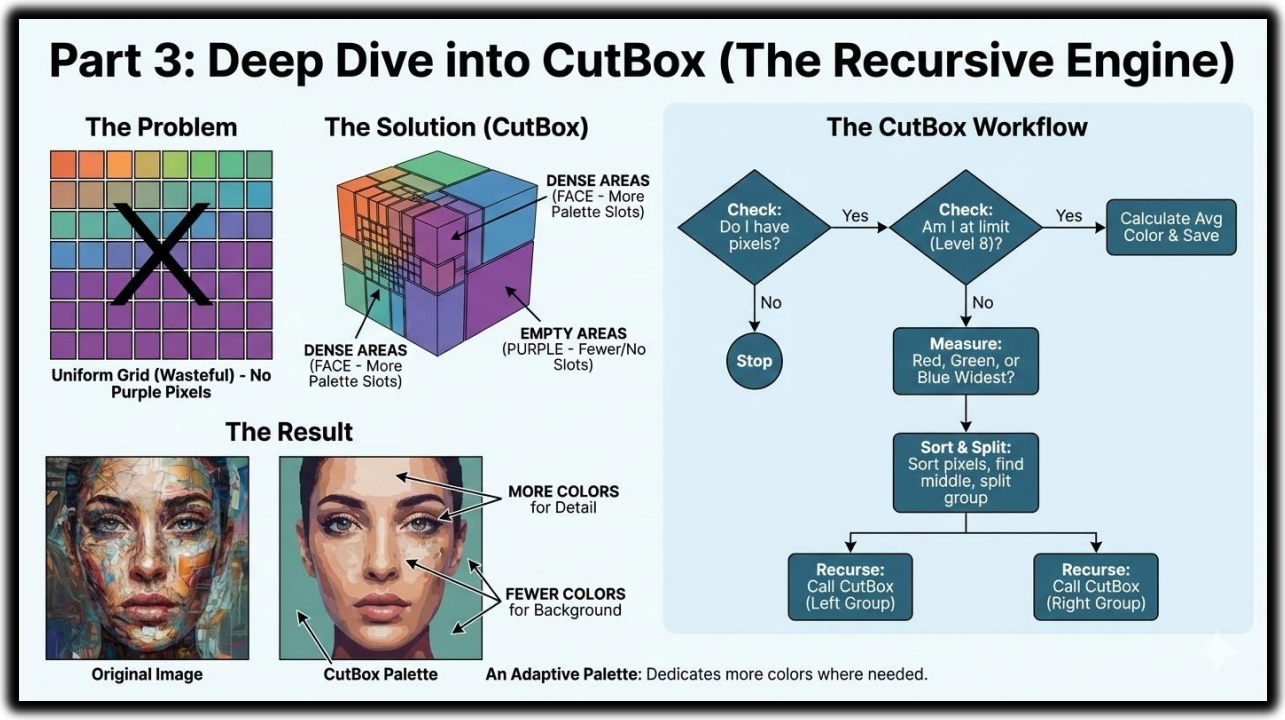
The Solution (CutBox): Recursion allows the algorithm to be **Adaptive**.

- **Dense Areas:** If an area has tons of different colors (like a human face), CutBox keeps splitting that box over and over, assigning many palette slots to skin tones.
- **Empty Areas:** If an area has no pixels (like Purple), it never gets split.

The CutBox Workflow:

1. **Check:** Do I have pixels? No? Stop.
2. **Check:** Am I at the limit (Level 8)? Yes? Calculate average color and Save.
3. **Measure:** Is the Red, Green, or Blue range widest?
4. **Sort & Split:** Sort pixels by that color, find the middle pixel, and split the group in two.
5. **Recurse:** Call CutBox for the Left group. Call CutBox for the Right group.

The Result: A palette that dedicates more colors to the parts of the image that need detail, and fewer colors to flat backgrounds.



4. Finishing the Median Cut (The "Clean Up")

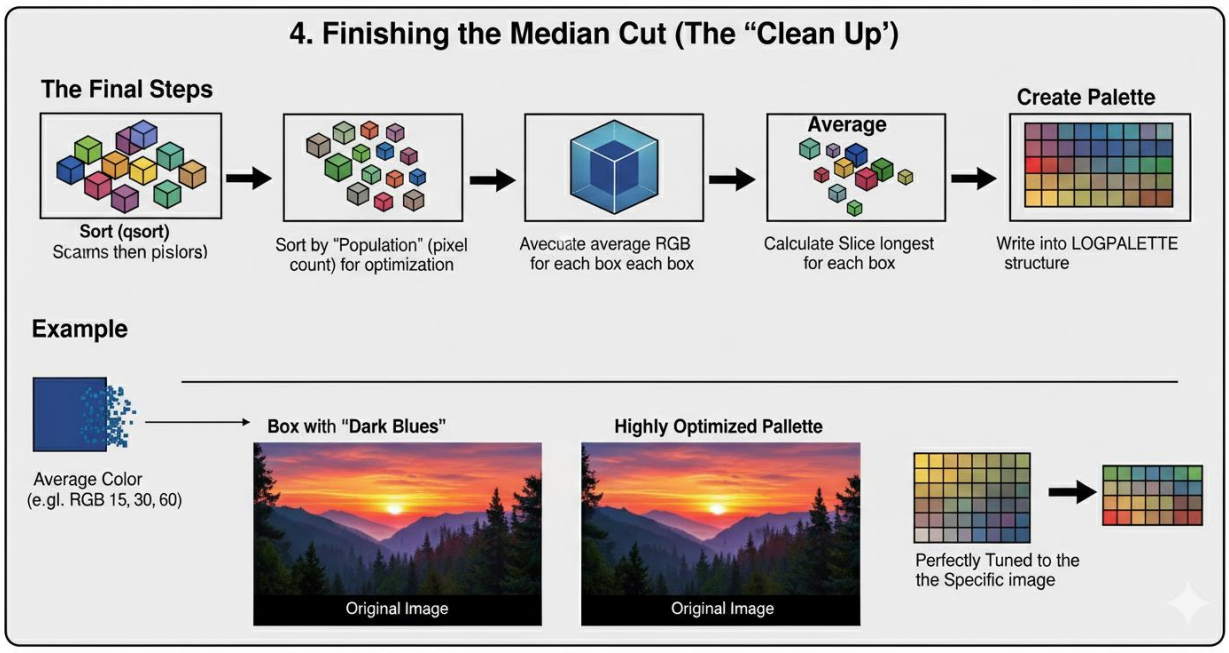
Once CutBox has finished slicing the 3D color cube, you are left with up to 256 small boxes.

Each box contains a cluster of similar colors (e.g., one box holds all the "Dark Navy Blues").

The Final Steps:

1. **Sort (qsort):** The algorithm sorts these boxes by "Population" (how many pixels are inside).
    - ✓ *Why?* To put the most common colors at the start of the palette. This is useful for optimization.
2. **Average (FindAverageColor):** It calculates the average RGB of every pixel inside the box.
    - ✓ *Example:* If a box has 50 "Dark Blues" and 50 "slightly lighter Dark Blues," the palette entry becomes the perfect average of them.
3. **Create Palette:** It writes these 256 averages into the LOGPALETTE structure.

The Result: A highly optimized palette where the colors are perfectly tuned to the specific image.



---

5. The "Popularity" Algorithm (The Voter)

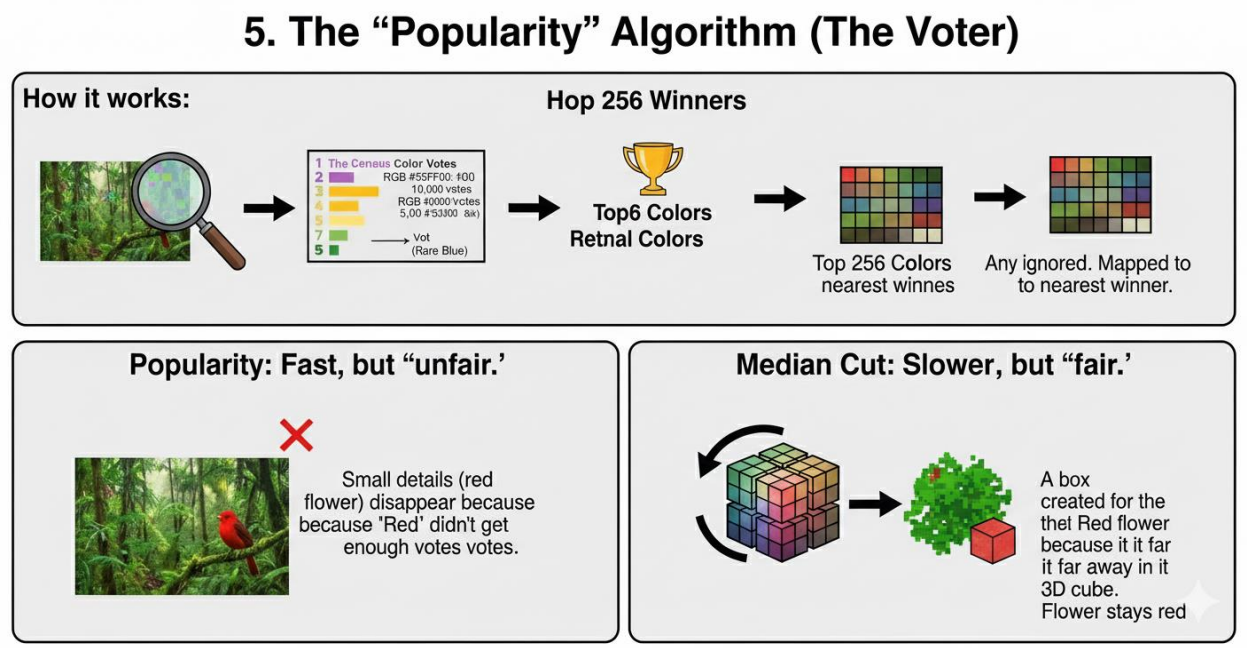
Your notes mention DibPalPopularity as an alternative to Median Cut. This is the **"First Past the Post"** election system for colors.

How it works:

1. **The Census:** It scans every single pixel in the image and keeps a running count.
    - "RGB #55FF00: 10,000 votes"
    - "RGB #000000: 5,000 votes"
    - "RGB #123456: 1 vote"
2. **The Winners:** It takes the top 256 colors with the most votes.
3. **The Losers:** Any color that didn't make the Top 256 is ignored. It gets mapped to the nearest winner.

Median Cut vs. Popularity:

- **Popularity:** Fast, but "unfair." Small details (like a red flower in a green field) disappear because "Red" didn't get enough votes to beat the 256 shades of Green.
- **Median Cut:** Slower, but "fair." It creates a box for the Red flower because it is far away from Green in the 3D cube, ensuring the flower stays red.



---

6. The "Octree" Algorithm (The Tree Structure)

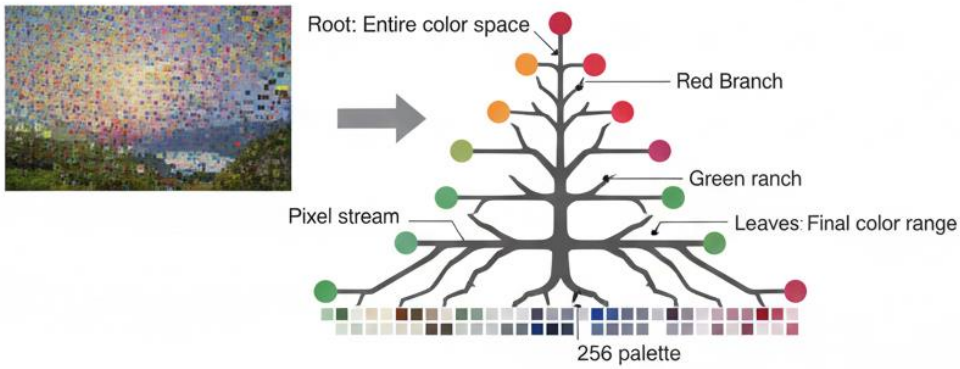
Your notes briefly mention **Octree Quantization**. This is the modern rival to Median Cut. Instead of slicing a "Box," it builds a "Tree."

- **Root:** The entire color space.
- **Branches:** Each level zooms in on a specific color range (Red branch, Green branch).
- **Leaves:** The final colors.

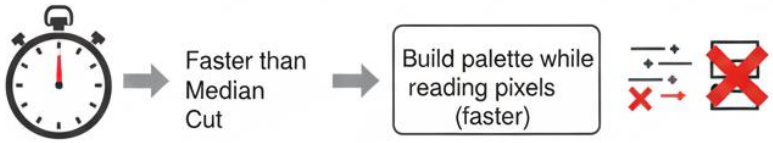
Why use it? It is faster than Median Cut because you build the palette *while* reading the pixels, rather than reading them all first and then sorting.

6. “Octree” Algorithm (The Tree Structure)

How it works:



Why use it?



Result

Original Highs'Color

Octree Quantization - Optimized Pallette

7. Final Cheat Sheet: Choosing Your Weapon

You now have a toolkit of functions in DIBPAL.C. Here is when to use each one:

PALETTE OPTIMIZATION STRATEGIES

Function	Strategy	Best Use Case
`DibPalDibTable`	**Copy/Paste**	Standard 8-bit BMP loads. Direct use of internal colors.
`DibPalAllPurpose`	**The Safety Net**	Displaying multiple different images simultaneously.
`DibPalUniformGrays`	**Gradient**	Medical imaging, X-Rays, or vintage B&W photography.
`DibPalVga`	**Retro**	1990s interface simulation (16 classic bright colors).
`DibPalUniformColors`	**Math Grid**	Scientific charts (Predictable 6x6x6 RGB distribution).
`DibPalPopularity`	**Vote Count**	Fast conversion where tiny color details are secondary.
`DibPalMedianCut`	**Smart Slicing**	**High-quality photo conversion to 256 colors. The gold standard.**

Conclusion: Handling bitmaps in Windows is a battle between **Storage** (DIBs are great for disks), **Speed** (DDBs are great for screens), and **Colors** (Palettes are necessary for 256-color screens). This DIBPAL.C library is your bridge between those three worlds.

DIBCONV.C PROGRAM

```
HDIB DibConvert(HDIB hdibSrc, int iBitCountDst)
{
    HDIB hdibDst;
    HPALETTE hPalette;
    int i, x, y, cx, cy, iBitCountSrc, cColors;
    PALETTEENTRY pe;
    RGBQUAD rgb;
    WORD wNumEntries;

    cx = DibWidth(hdibSrc);
    cy = DibHeight(hdibSrc);
    iBitCountSrc = DibBitCount(hdibSrc);

    if (iBitCountSrc == iBitCountDst)
        return NULL;

    // DIB with color table to DIB with larger color table:
    if ((iBitCountSrc < iBitCountDst) && (iBitCountDst <= 8))
    {
        cColors = DibNumColors(hdibSrc);
        hdibDst = DibCreate(cx, cy, iBitCountDst, cColors);

        for (i = 0; i < cColors; i++)
        {
            DibGetColor(hdibSrc, i, &rgb);
            DibSetColor(hdibDst, i, &rgb);
        }

        for (x = 0; x < cx; x++)
            for (y = 0; y < cy; y++)
            {
                DibSetPixel(hdibDst, x, y, DibGetPixel(hdibSrc, x, y));
            }
    }
    // Any DIB to DIB with no color table
    else if (iBitCountDst >= 16)
    {
        hdibDst = DibCreate(cx, cy, iBitCountDst, 0);

        for (x = 0; x < cx; x++)
            for (y = 0; y < cy; y++)
            {
                DibGetPixelColor(hdibSrc, x, y, &rgb);
                DibSetPixelColor(hdibDst, x, y, &rgb);
            }
    }
    // DIB with no color table to 8-bit DIB
    else if (iBitCountSrc >= 16 && iBitCountDst == 8)
    {
        hPalette = DibPalMedianCut(hdibSrc, 6);
        GetObject(hPalette, sizeof(WORD), &wNumEntries);
        hdibDst = DibCreate(cx, cy, 8, wNumEntries);

        for (i = 0; i < (int)wNumEntries; i++)
        {
            GetPaletteEntries(hPalette, i, 1, &pe);
            rgb.rgbRed = pe.peRed;
            rgb.rgbGreen = pe.peGreen;
            rgb.rgbBlue = pe.peBlue;
            rgb.rgbReserved = 0;
            DibSetColor(hdibDst, i, &rgb);
        }

        for (x = 0; x < cx; x++)
            for (y = 0; y < cy; y++)
            {
                DibGetPixelColor(hdibSrc, x, y, &rgb);
                DibSetPixel(hdibDst, x, y, GetNearestPaletteIndex(hPalette, RGB(rgb.rgbRed, rgb.rgbGreen, rgb.rgbBlue)));
            }

        DeleteObject(hPalette);
    }
    // Any DIB to monochrome DIB
    else if (iBitCountDst == 1)
    {
        hdibDst = DibCreate(cx, cy, 1, 0);
        hPalette = DibPalUniformGrays(2);

        for (i = 0; i < 2; i++)
        {
            GetPaletteEntries(hPalette, i, 1, &pe);
            rgb.rgbRed = pe.peRed;
            rgb.rgbGreen = pe.peGreen;
            rgb.rgbBlue = pe.peBlue;
            rgb.rgbReserved = 0;
            DibSetColor(hdibDst, i, &rgb);
        }

        for (x = 0; x < cx; x++)
            for (y = 0; y < cy; y++)
            {
                DibGetPixelColor(hdibSrc, x, y, &rgb);
                DibSetPixel(hdibDst, x, y, GetNearestPaletteIndex(hPalette, RGB(rgb.rgbRed, rgb.rgbGreen, rgb.rgbBlue)));
            }

        DeleteObject(hPalette);
    }
    // All non-monochrome DIBs to 4-bit DIB
    else if (iBitCountSrc >= 8 && iBitCountDst == 4)
    {
        hdibDst = DibCreate(cx, cy, 4, 0);
        hPalette = DibPalVga();

        for (i = 0; i < 16; i++)
        {
            GetPaletteEntries(hPalette, i, 1, &pe);
            rgb.rgbRed = pe.peRed;
            rgb.rgbGreen = pe.peGreen;
            rgb.rgbBlue = pe.peBlue;
            rgb.rgbReserved = 0;
            DibSetColor(hdibDst, i, &rgb);
        }

        for (x = 0; x < cx; x++)
            for (y = 0; y < cy; y++)
            {
                DibGetPixelColor(hdibSrc, x, y, &rgb);
                DibSetPixel(hdibDst, x, y, GetNearestPaletteIndex(hPalette, RGB(rgb.rgbRed, rgb.rgbGreen, rgb.rgbBlue)));
            }

        DeleteObject(hPalette);
    }
    // Should not be necessary
    else
        hdibDst = NULL;

    return hdibDst;
}
```

To see zoom and the code will be very clear or go to page 770 of the book.

```
/* DIBCONV.H header file for DIBCONV.C */

HDIB DibConvert(HDIB hdibSrc, int iBitCountDst);
```

The DibConvert function is essentially the DIB's Swiss Army knife—it handles changing bit depths, color tables, and even converting full-color images into monochrome or paletted versions.

Its role is to make sure that no matter what format your source DIB is in, you can get it into a format suitable for display, printing, or further processing.

It's used whenever you need to:

1. **Match a new color depth** – e.g., converting a 4-bit image to 8-bit, or an 8-bit image to 24-bit.
2. **Create a monochrome version** – for printing or limited-color displays.
3. **Adapt a bitmap to a device or palette** – e.g., for halftone displays or VGA palettes.
4. **Ensure compatibility with other modules** – e.g., the DisplayDib function in Dibble expects the bitmap to match the display's color depth or palette.

Basically, if you want your DIB to "play nicely" with a different bit depth or palette, DibConvert steps in.

Congratulations! You have reached the summit of the Bitmap Mountain. This final function is the "Universal Translator" that ties everything we have learned together.

The Goal: The Universal Adapter

In the real world, you might load a 24-bit JPEG but need to display it on an old 8-bit screen. Or you might have a 4-bit icon and want to save it as a 24-bit BMP.

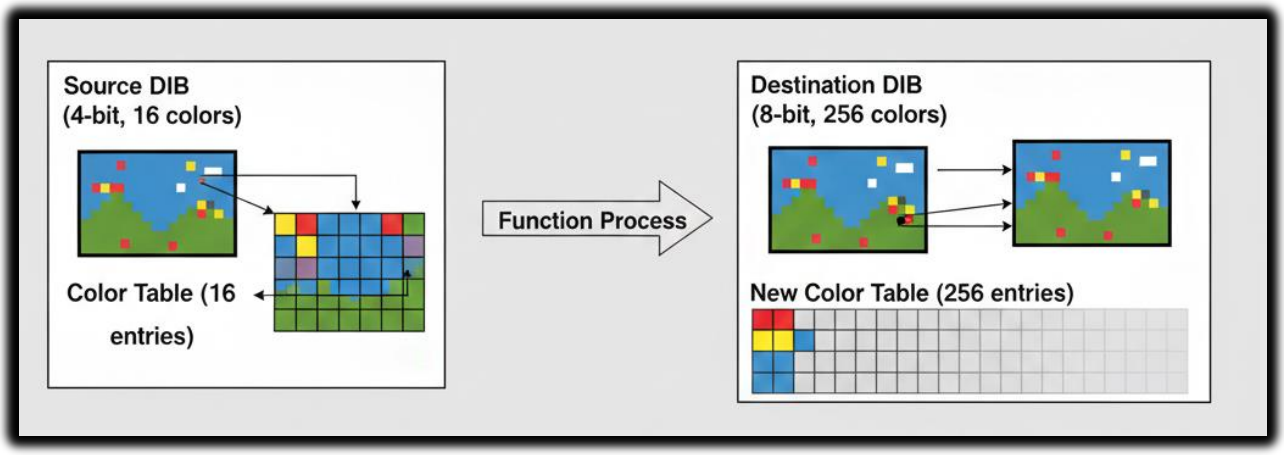
DibConvert is a massive switchboard.

- **Input:** A Source DIB + A Target Bit Count (e.g., 1, 4, 8, 24).
- **Output:** A brand new DIB in the requested format.

It doesn't just copy memory; it has to mathematically transform color data.

I. Scenario 1: DIB with Color Table → DIB with a Larger Color Table

- **When it happens:** The source DIB has fewer bits per pixel than the destination (but both ≤ 8 bits).
- **What the function does:**
    1. Creates a new DIB with the desired higher bit count.
    2. Copies the color table from the source DIB into the new DIB.
    3. Iterates over each pixel, reading the original color index and setting it in the new DIB.
- **Why it's important:** This ensures that low-bit images can take advantage of a larger palette without losing their original color mapping. For example, an old 4-bit DIB can now use 8-bit indexing while retaining the original colors.



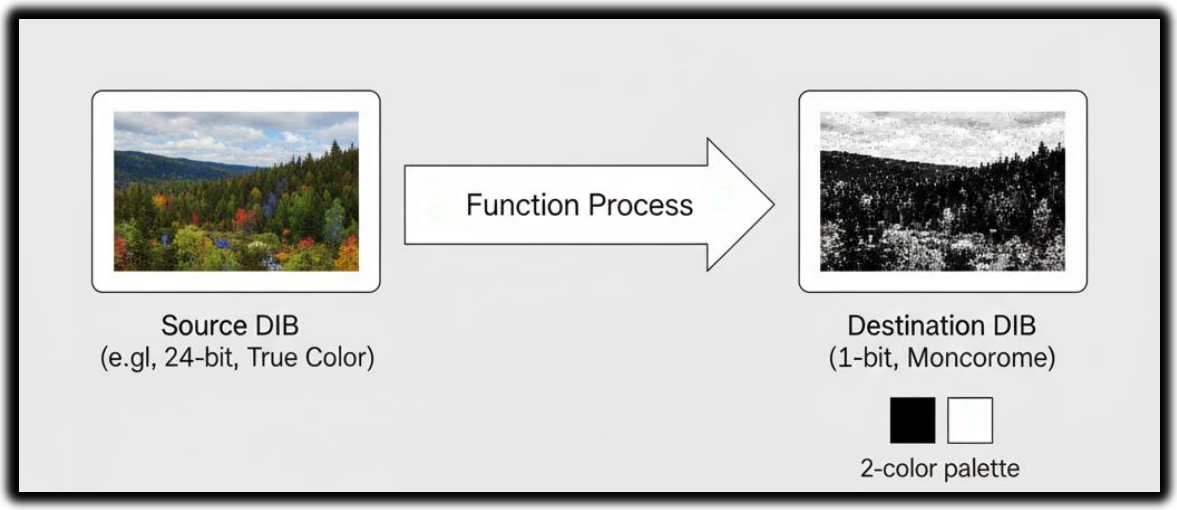
II. Scenario 2: Any DIB → Monochrome DIB

When it happens: The destination DIB has a bit count of 1.

What the function does:

- Creates a new 1-bit DIB.
- Generates a 2-color grayscale palette using DibPalUniformGrays.
- Iterates over each pixel in the source DIB:
    1. Retrieves its color.
    2. Converts the color to a corresponding grayscale value.
    3. Sets the appropriate black or white pixel in the destination DIB.

Why it's important: This is used when you need a simplified, high-contrast version of an image, such as for printing on black-and-white printers or for certain graphics operations where only two colors are allowed.



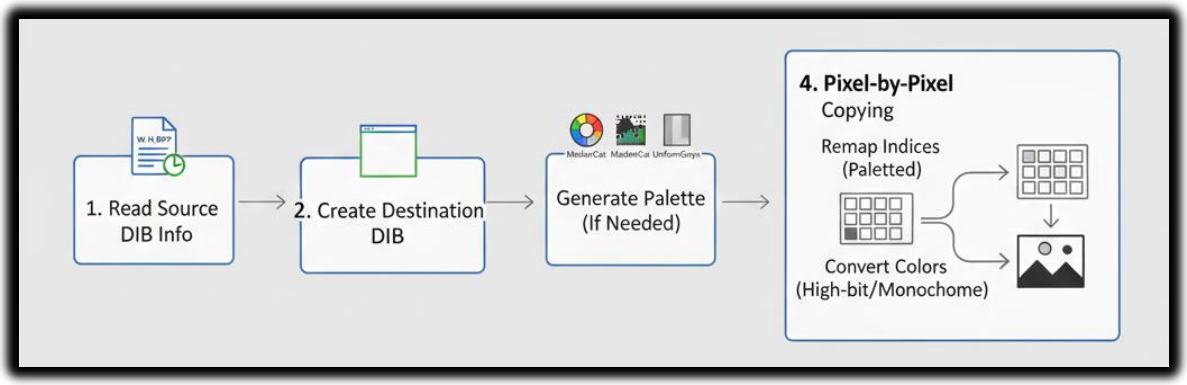
III. Other Notable Scenarios

1. **Any DIB → DIB without a color table (16+ bits)**
    - The function simply maps each pixel's true color directly into the destination DIB. No palette needed.
2. **DIB without color table → 8-bit DIB**
    - Uses DibPalMedianCut to generate a palette from the source image's colors, reducing the image into 256 colors.
3. **Non-monochrome DIB → 4-bit DIB**
    - Uses a 16-color VGA palette (DibPalVga) and remaps each pixel accordingly.
4. **No conversion needed**
    - If the source and destination bit counts match, the function returns NULL, because no changes are necessary.

IV. How It Works Internally

- The function always starts by reading **width, height, and bit count** of the source DIB.
- It then **creates a destination DIB** suitable for the conversion.
- If the new DIB needs a **palette**, it generates one using helper functions like DibPalVga, DibPalMedianCut, or DibPalUniformGrays.
- Finally, it performs **pixel-by-pixel copying**, either remapping indices (for paletted DIBs) or converting actual colors (for high-bit or monochrome DIBs).

This careful, step-by-step approach ensures that the conversion preserves as much visual fidelity as possible within the limits of the new bit depth.



V. TLDR – Why DibConvert is Awesome

- Makes **bit-depth changes seamless**.
- Handles **palette creation automatically**.
- Can convert **to monochrome or paletted images**.
- Supports both **low-bit and high-bit DIBs**.
- Ensures **compatibility with display and printing routines** in Dibble.

In short, DibConvert is the behind-the-scenes hero that makes all other bitmap operations possible—without it, flipping, rotating, displaying, or printing DIBs would be a nightmare.

And off we go to the final chapters of the book. This was the largest topic, dealing with bitmaps, but it was really fun…