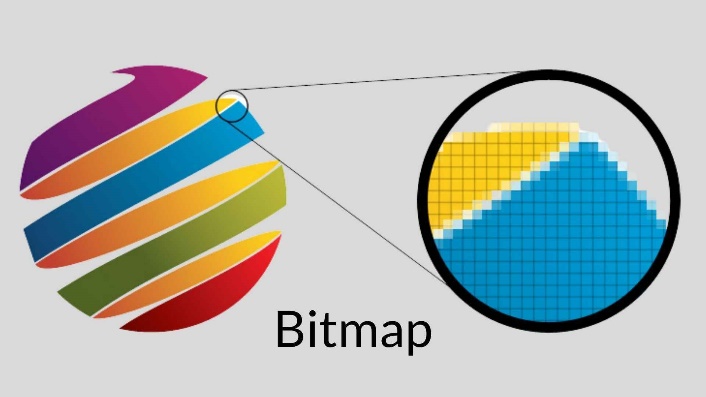
CHAPTER 14 BITMAPS AND BITBLTS

Imagine a rectangular grid overlaid on an image. Each tiny square within this grid represents a pixel, the **basic unit** of visual information.

A **bitmap,** in its simplest form, is a 2D array of bits corresponding to these pixels. Each bit value determines the pixel's color or intensity, with 1 representing "on" and 0 representing "off."



A **bitmap** is a digital image made of a grid of tiny dots called **pixels**.

Think of it as a rectangular map of data:

* **The Grid:** Every image is divided into a hidden grid.
* **The Bits:** Each square in that grid (a pixel) is represented by bits in memory.
* **The Color:** In the simplest version, a bit value of **1** means the pixel is "on" (usually white), and **0** means it is "off" (black).

**Key Concepts**

* **Bit-Block Transfer (BitBlt):** This is the process of moving a block of pixels from one place to another (like from memory to your screen).
* **Memory Efficiency:** Because bitmaps are just arrays of numbers, they are a very fast way for a computer to draw complex graphics.

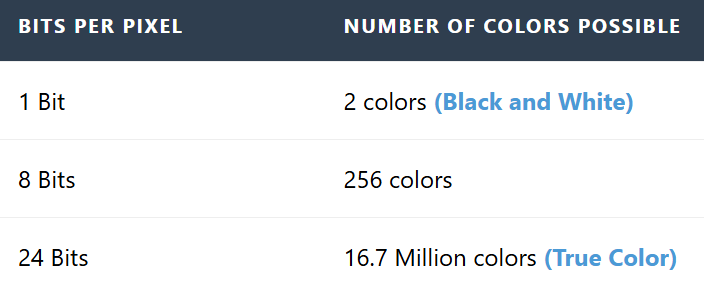
SHADES AND COLORS: BEYOND BINARY

Beyond Black and White: Colors and Shades

Most images use more than just black and white. To show different colors or shades of gray, we use **multiple bits** for every single pixel.

* **Color Depth:** The more bits you assign to a pixel, the more colors you can display.
* **The Mosaic Effect:** Just like a mosaic is made of many colored tiles, a digital image combines these bit values to create a full-color picture.





BITMAPS VS. METAFILES: TWO APPROACHES TO PICTORIAL DATA

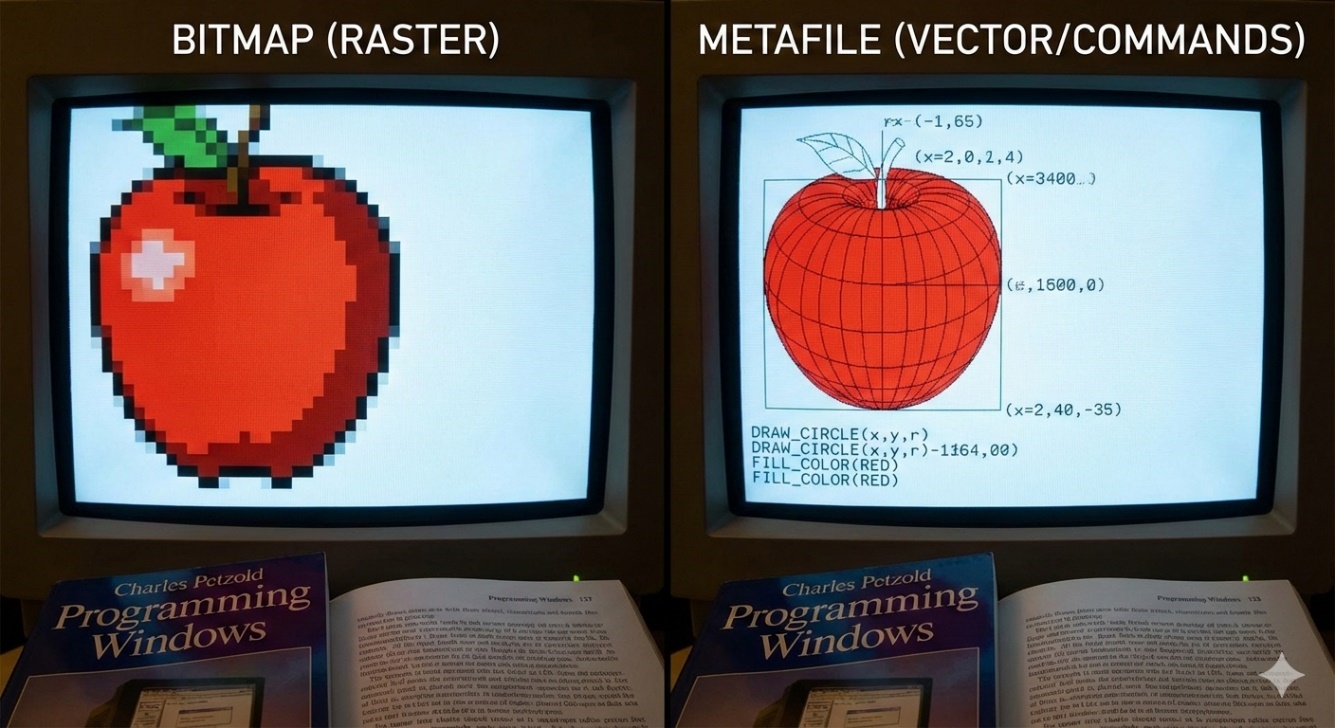
Windows uses two different ways to store and display pictures. You can think of it as the difference between a **photo** and a **drawing lesson**.

[Image comparing bitmap grid pixels vs vector metafile instructions]

I. Bitmaps (The Snapshot)

A bitmap is a direct map of the image data.

* **How it works:** It records exactly which color goes into which pixel.
* **Pros:** Very fast to display because the computer doesn't have to "think"—it just copies the pixels to the screen.
* **Cons:** Files can become very large if the image is high resolution.



II. Metafiles (The Recipe)

A metafile stores a list of commands or instructions to recreate the image (e.g., "Draw a red circle at these coordinates").

* **How it works:** It acts like a set of drawing instructions.
* **Pros:** The files are very small and can be resized without losing quality.
* **Cons:** It takes more processing power because the computer has to follow every instruction to "render" the image each time it appears.

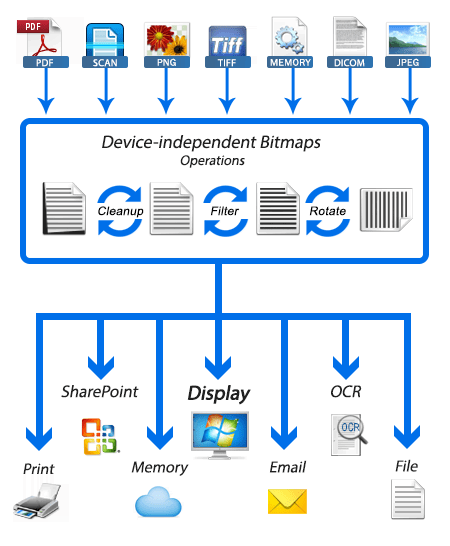
GDI Bitmaps (The "Old School" Way)

Before Windows 3.0 introduced **DIBs** (Device-Independent Bitmaps), GDI bitmaps were the standard. Even though they are older, they are still very useful today because they are fast.

I. Device-Dependent Bitmaps (DDBs)

The most important thing to know about GDI bitmaps is that they are **Device-Dependent**.

* **Tied to Hardware:** A GDI bitmap is formatted to match a specific video card or display.
* **Speed:** Because the format already matches the hardware, the computer can draw them almost instantly.
* **The Downside:** You cannot easily move a GDI bitmap from one computer to another if their screens use different color settings. It might look distorted or have the wrong colors.



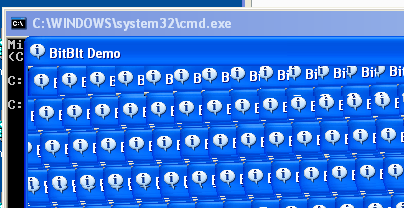
BitBlt: Moving Pixels

**BitBlt** (pronounced "bit-blit") stands for **Bit-Block Transfer**. It is the main tool Windows uses to move or combine images.

* **The Basic Job:** It copies a rectangle of pixels from one place (the **Source**) to another (the **Destination**).
* **More Than a Copy:** It doesn't just "paste." It can also **combine** pixels. For example, you can blend two images together or invert the colors while moving them.
* **Efficiency:** It is extremely fast because it moves the data in blocks rather than one pixel at a time.

**Common Uses**

* **Moving:** Sliding an image across the screen.
* **Masking:** Making certain parts of an image transparent.
* **Patterning:** Filling a shape with a specific repeating design.



This section focuses on using **GDI functions** to handle images. The main goals are:

* **Manipulation:** Creating, loading (BMP/ICO), and editing bitmaps.
* **BitBlt:** Moving and copying pixel blocks.
* **Composition:** Combining multiple images and using transparency.
* **Performance:** Learning how to handle memory efficiently using these older, faster tools.

Bitmaps: The Trade-offs

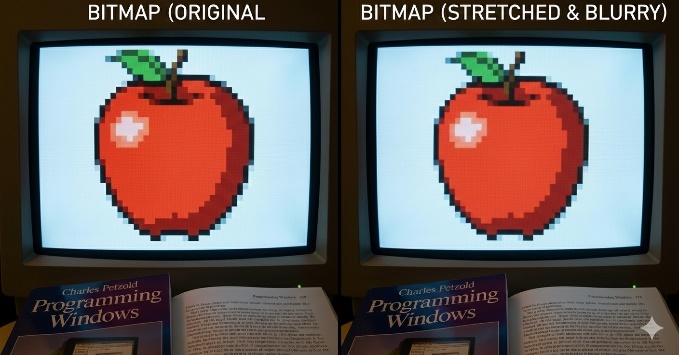
**Best for Real-World Detail:** Because bitmaps store data pixel-by-pixel, they are the only way to handle complex images like **photographs** or **video frames**.



**The "Device Dependent" Problem:** \* **Colors:** A bitmap created for a high-color screen might look broken on a basic screen.



**Scaling:** If you stretch a bitmap, it gets blurry or "blocky" because you are just making the existing pixels bigger.



**Heavy Files:** High-resolution bitmaps use a lot of RAM and storage because every single pixel requires its own data.

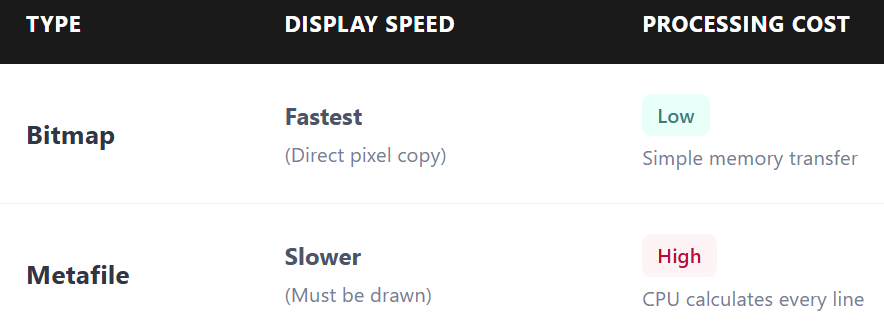


Metafiles: Strengths and Weaknesses

**Perfect Scaling:** Since it’s a list of instructions (like "draw a line"), you can stretch the image to any size and it stays perfectly sharp.

**Device Independent:** A metafile doesn't care about your screen's color settings; it just redraws itself to fit whatever device it is on.

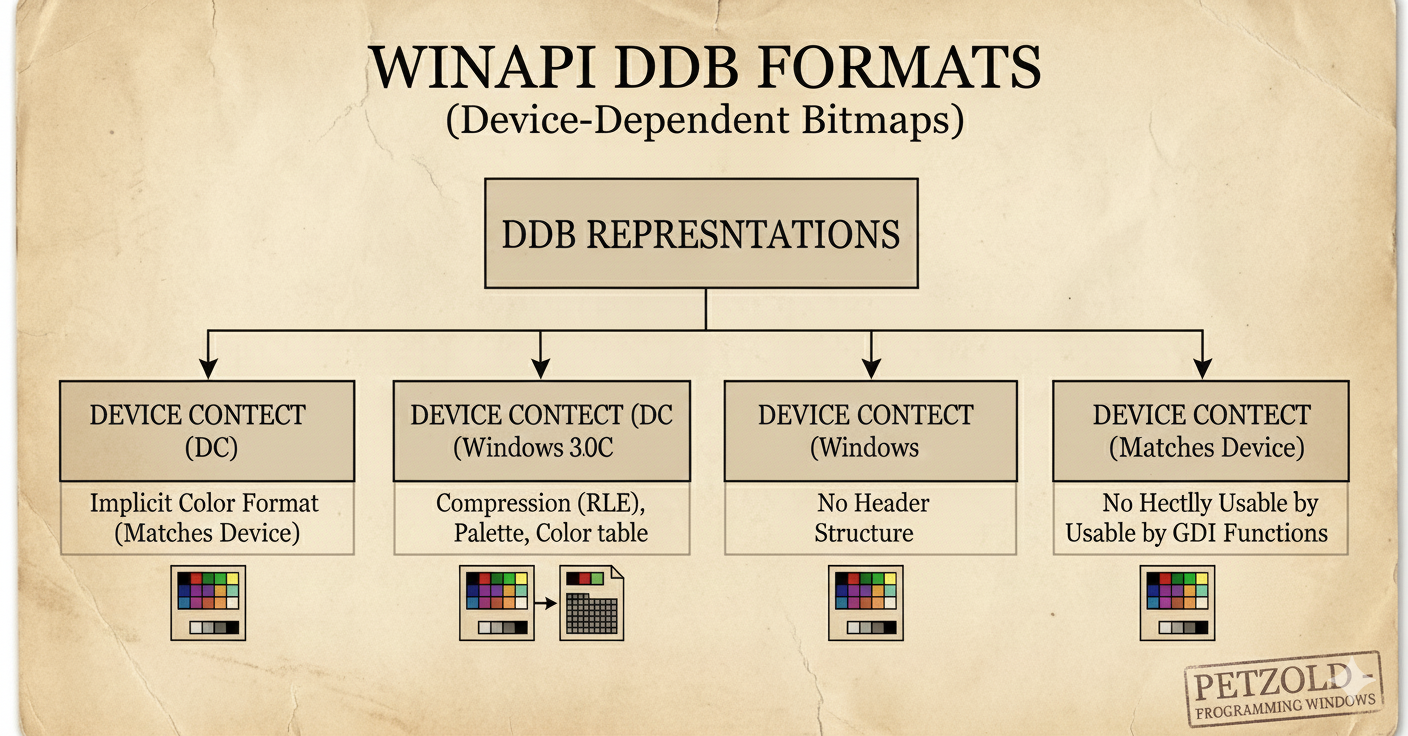
**Small Files:** Storing a few sentences of "instructions" is much smaller than storing millions of individual pixel colors.

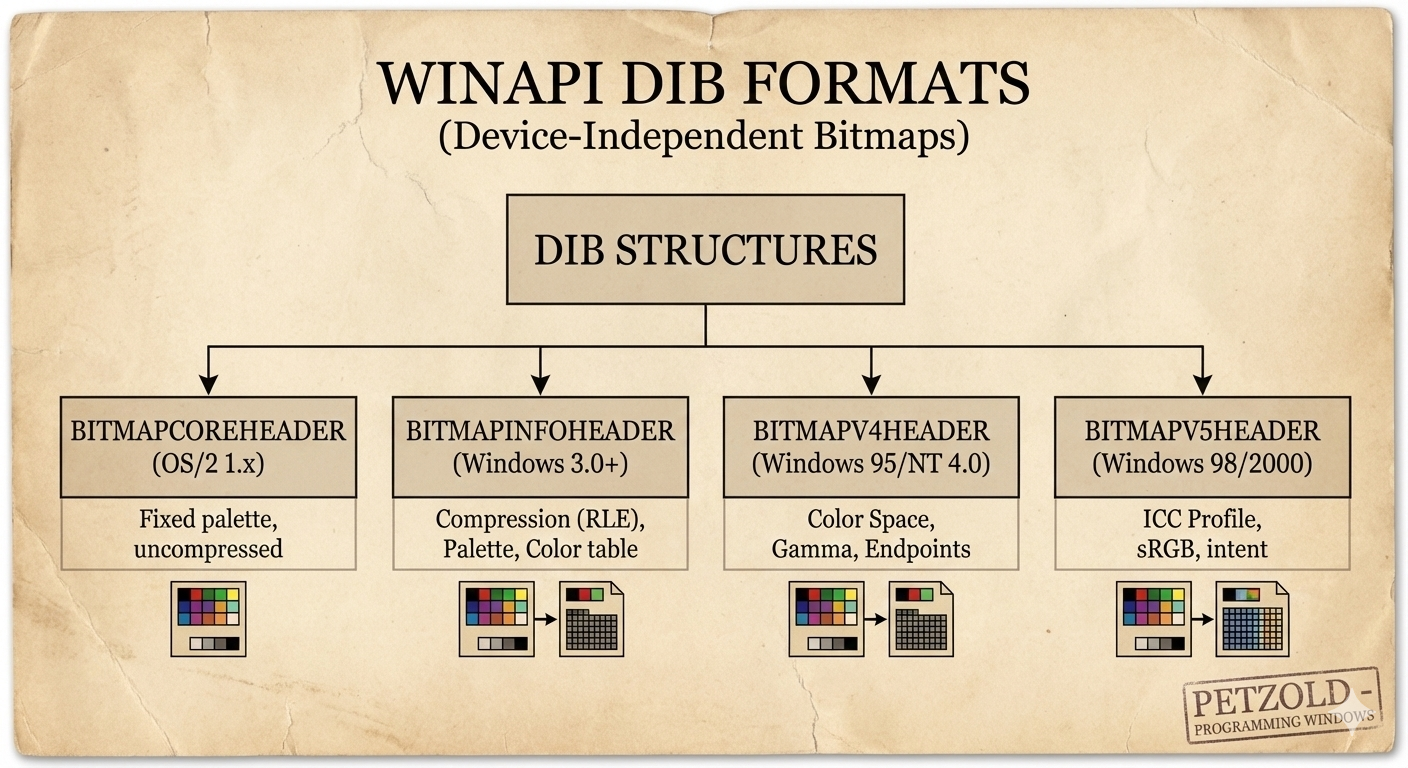


**Key Takeaway:** Use **Bitmaps** for speed and realistic detail (photos). Use **Metafiles** for diagrams, logos, and things that need to be resized frequently.

Compression & Evolution

**Bitmap Compression:** Since bitmaps are huge, we use compression (like **JPEG** or **PNG**) to make them smaller for storage and the web.



**DIBs (Device-Independent Bitmaps):** These were created to solve the "Device Dependent" problem. They allow images to look the same on any screen, regardless of the hardware. 

WINAPI BITMAPS: DIB VS. DDB

**The Core Truth:** PNG/JPG/GIF are **storage** formats. To show them on screen, Windows must unpack them into **memory** formats: either a **DIB** (The Master Copy) or a **DDB** (The Hardware Speedster).

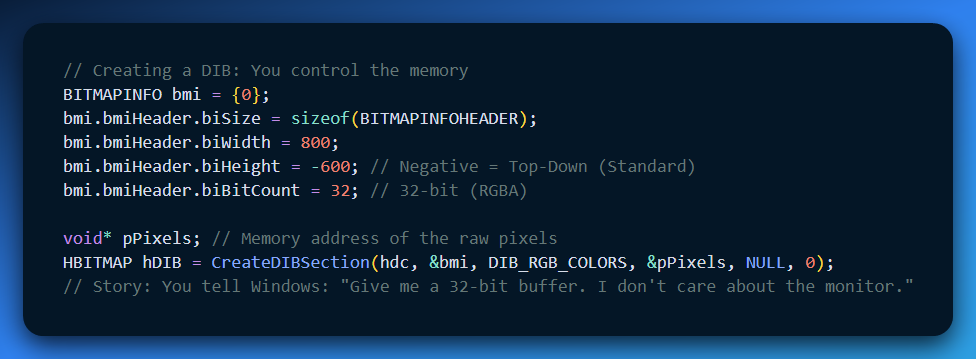
1. DIB (Device-Independent Bitmap)

A **DIB** works on any device because it contains its own format information.

A DIB is a **self-describing bitmap**. It includes a header called BITMAPINFOHEADER.

This header describes the bitmap’s **width**, **height**, and **color depth**.

The DIB header acts as a **blueprint** that tells the system how to interpret the pixel data.



A **.BMP file** is a **Device Independent Bitmap (DIB)** that also contains a **file header**.

Bitmaps are commonly used by the **clipboard** when copying and pasting images between applications.

Bitmaps are also used by **malware and packers** to manually decrypt data stored as pixels. This is possible because CreateDIBSection returns a direct pointer to the bitmap memory.

From a **reverse engineering (RE) perspective**, CreateDIBSection is an important API call. You should set a breakpoint on it.

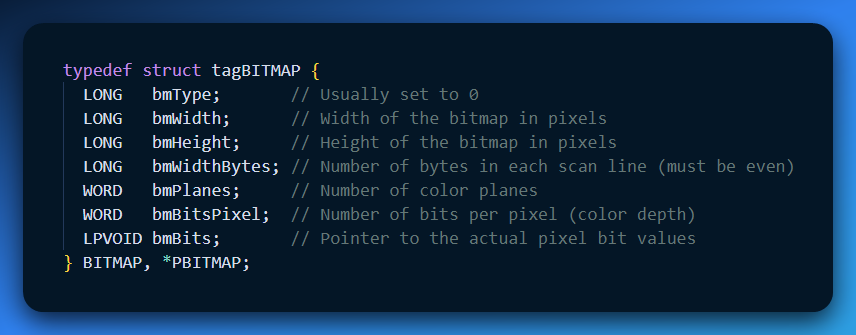
The returned pPixels pointer is the main memory location where raw data is directly read or modified.

2. DDB (Device-Dependent Bitmap)

The "Custom Suit" Format: Perfectly tailored to your hardware.

A **Device-Dependent Bitmap (DDB)** is essentially a "naked" array of pixels. Unlike other formats, it does not include a header or a color table. Instead, it relies entirely on the current settings of your display driver. If your monitor is set to 24-bit color, the DDB is automatically 24-bit.

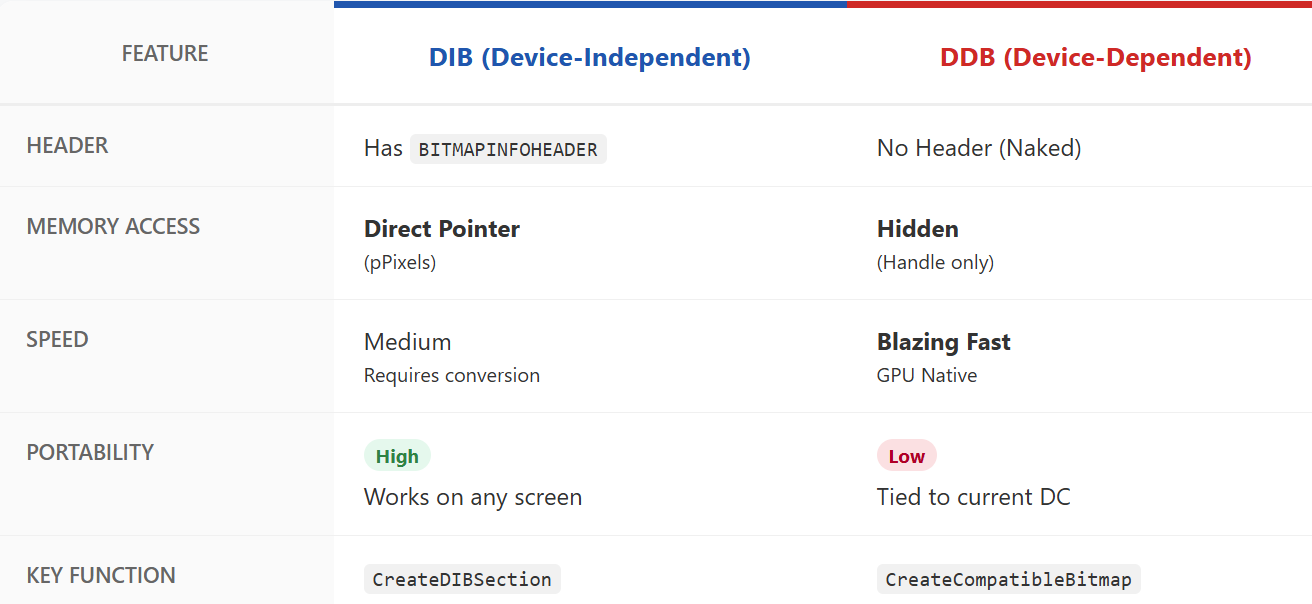
To work with a DDB in C++, you use the BITMAP structure.



**Real-World Use:**

* **UI Icons:** The "X" button or Taskbar icons. They need to be drawn instantly.
* **Video Game Sprites:** Converted to DDBs so the GPU can move them at 60+ FPS without doing "math" to convert colors.

**RE Perspective:** In a debugger, a DDB is an **opaque handle**. You can't see the pixels easily. VMProtect, Themida and other packers, might use these for UI elements to stay hardware-accelerated and hidden from simple memory scanners.



THE ORIGINS OF BITMAPS



Bitmaps are the basic building blocks of digital images. They generally come from three sources: manual creation, computer code, or hardware that captures the real world.

1. Manual Creation (The Artist's Touch)   
You can create bitmaps directly using software like Paint.

**How it works:** You place every pixel yourself using virtual brushes or pencils.

**Best for:** Detailed art where you need exact control over every dot in the image.

2. Algorithmic Generation (Code-Crafted)   
Not all images are drawn by humans. Computer code can generate bitmaps automatically.

**How it works:** Algorithms create complex patterns, textures, or fractals.

**Best for:** Images that are too complex or mathematical to draw by hand.

3. Hardware Capture (Real World to Digital)   
Hardware devices bridge the gap between physical reality and digital files. Most of these devices use **CCDs (Charge-Coupled Devices)** or **CMOS** sensors to turn light into electrical signals, which then become digital values (pixels).

**Scanners:** These use rows of CCD cells. They scan an image one line at a time, measuring light intensity and converting it into a bitmap.

**Camcorders:** These capture moving video. You can pause the video and use a "frame grabber" to save a specific moment as a still bitmap image.

**Digital Cameras:** These allow you to capture bitmaps instantly. They use internal sensors and converters (ADCs) to turn what the lens sees into a digital file stored on the camera.

Note on Storage:   
Because bitmaps can be very large, we use **compression** algorithms to reduce file size without losing too much quality.

Understanding Bitmap Dimensions

Every bitmap is a rectangular grid. To work with them in code, you need to understand how Windows measures their size and location.

1. Width and Height (The Grid)

* **Width:** The number of pixels across one horizontal row.
* **Height:** The number of pixels in one vertical column.
* **Pixel Count:** If you multiply the width by the height, you get the total number of pixels in the image.

2. The Coordinate System

Windows uses a specific "map" to find pixels within that grid:

**The Origin (0,0):** In most GDI operations, the point **(0,0)** is the **top-left corner** of the bitmap.

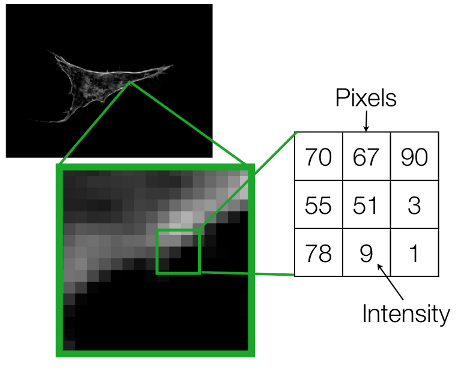
**X-Axis:** Increases as you move to the **right**.

**Y-Axis:** Increases as you move **down**.

3. Why Dimensions Matter

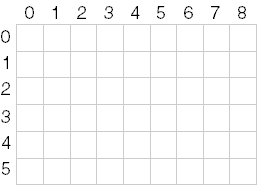
**Memory Allocation:** The computer uses the width and height to calculate exactly how much RAM is needed to store the image.

**Scaling:** If you try to draw a 100 \* 100 bitmap into a 200 \* 200 space, Windows has to stretch the pixels to fill the gap.



Shorthand Notation

To avoid cumbersome phrases, we often use a concise notation for a bitmap's dimensions. For instance, "9 by 6" describes a bitmap 9 pixels wide and 6 pixels high. Remember, the width comes first by convention.



Pixel Power

The total number of pixels in a bitmap is calculated by multiplying its width and height. In our example, 9 pixels x 6 pixels = 54 pixels. We often use cx and cy (**count x** and **count y**) to represent width and height, respectively.



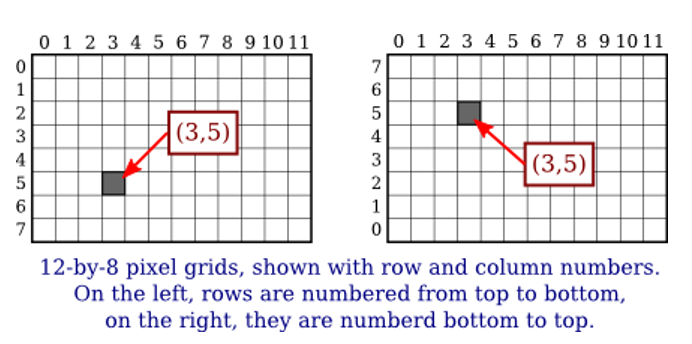
Pinpointing Pixels: The Coordinate System

To locate a specific pixel in a bitmap, Windows uses **X and Y coordinates**. This system works like a map, but it follows a few specific rules.

* **The Origin:** The top-left corner of the bitmap is always **(0, 0)**.
* **The Direction:** The X-value increases as you move **right**, and the Y-value increases as you move **down**.
* **The Zero-Based Rule:** Because counting starts at zero, the address of the very last pixel is always **one less** than the total width and height.

Example: A 9x6 Bitmap

* **Width:** 9 pixels (Indices 0 through 8)
* **Height:** 6 pixels (Indices 0 through 5)
* **Bottom-Right Corner:** The coordinate for this pixel is **(8, 5)**.



The word "resolution" is often used in two different ways, which can be confusing. To keep your notes clear, distinguish between **Screen Size** and **Detail Level**.

1. Display Resolution (Total Pixels)

This describes the size of a screen or an image in total pixels (e.g., 1920 \* 1080).

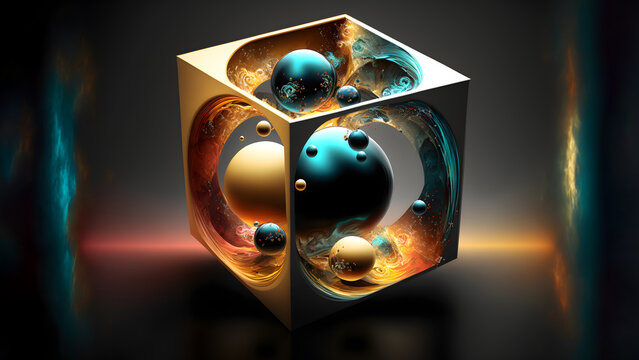
* It tells you how much **space** the image takes up on a monitor.
* In Windows GDI, this is usually what you care about when defining bitmap width and height.

2. Pixel Density (Detail Level)

This describes how tightly those pixels are packed together, measured in **DPI** (Dots Per Inch).

**Printers:** A high-quality print needs about **300 DPI**.

**Monitors:** Standard screens usually range from **72 to 96 DPI**.



3. Why the distinction matters:

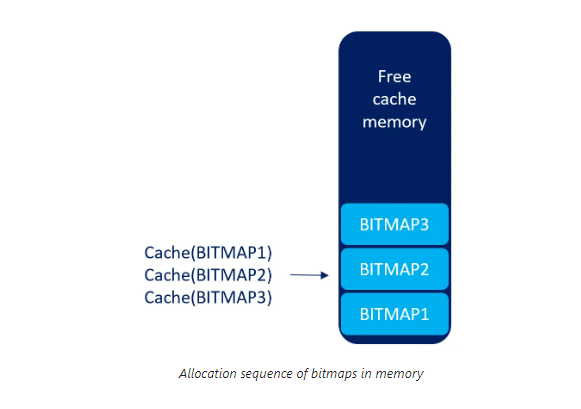
If you have a 300\*300 pixel bitmap:

* On a **72 DPI monitor**, it will look quite large (about 4 inches wide).
* On a **300 DPI printer**, it will look tiny (only 1 inch wide).

Memory Maze: Storing Bitmaps Linearly

Even though a bitmap looks like a rectangle, the computer stores it in a straight line in memory.

It saves the image **row by row**, from **top to bottom**, and each row goes **left to right**, like reading text.



Not all bitmaps are stored the same way.

**DIBs (Device-Independent Bitmaps)** can use a different storage method.

This makes them more flexible and work on different devices.

Why bitmap dimensions are important

* Width and height help you **resize images correctly**.
* Pixel coordinates let you **access specific pixels**.
* Dimensions help you **align images** with other elements on the screen.

UNVEILING THE MYSTERY OF COLOR IN BITMAPS: A DEEP DIVE INTO BIT DEPTHS AND PALETTE MAGIC

Beyond their width and height, bitmaps possess another crucial dimension – color.

This dimension, defined by the number of bits allocated to each pixel, determines the richness and complexity of the visual information they can display.

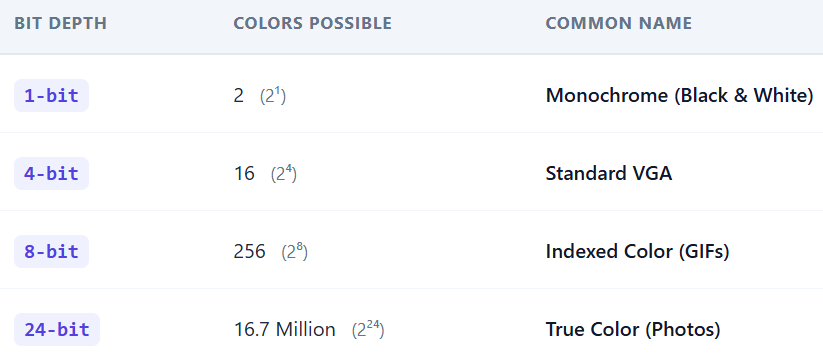
Let's explore the fascinating world of color within bitmaps:

Color Depth - Bit Depth: The Language of Color

It defines exactly how many different colors a single pixel can possibly show.

**How it works:** Each bit is a switch. The more switches you have for one pixel, the more color combinations you can create.

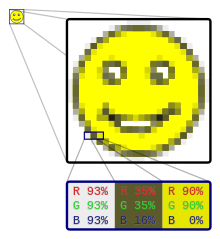
**The Math:** The number of possible colors is **2n**, where n is the number of bits.



Each pixel in a bitmap speaks the language of bits.

The number of bits assigned to it, known as the bit depth or color depth, determines how much color information it can carry.

This depth acts like a vocabulary, defining the range of colors a pixel can express.



Monochrome Masters: Bilevel and Beyond

At the simplest level, a bitmap can have just one bit per pixel, making it a "bilevel" or "monochrome" image.

This binary world allows only two states: on (typically white) or off (typically black). While seemingly limited, these monochrome masters excel in sharp lines, intricate patterns, and classic artistic expressions.



Beyond Black and White: Expanding the Palette

With more bits come more colors.

Each additional bit doubles the potential color combinations, opening doors to a richer palette.

Two bits offer four colors, **four bits** offer sixteen, and so on.

This exponential growth allows bitmaps to paint a **broader spectrum** of the world around us.

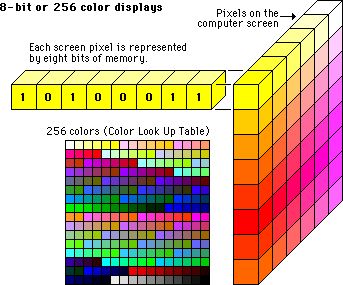


The Magic of 8-bit Palettes: A Familiar Canvas

For decades, the 8-bit world reigned supreme in digital art and early computing.

With 256 possible colors, it maintained the balance between complexity and practicality.

Artists created vibrant palettes, each pixel chosen to depict landscapes, characters, and objects in a captivating pixelated style.

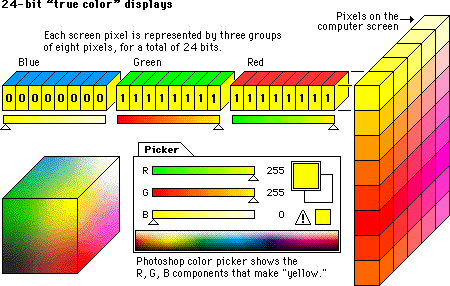


Pushing the Boundaries: 16-bit, 24-bit, and Beyond

The quest for photographic realism led to higher bit depths.

16-bit bitmaps offered a staggering 65,536 colors, while 24-bit bitmaps, the standard for modern displays, boast a whole 16.7 million colors!

This vast palette allows for near-photorealistic images, blurring the line between the digital and reality.



COLOR MAPPING: THE "CRAYON BOX" METHOD **🎨**

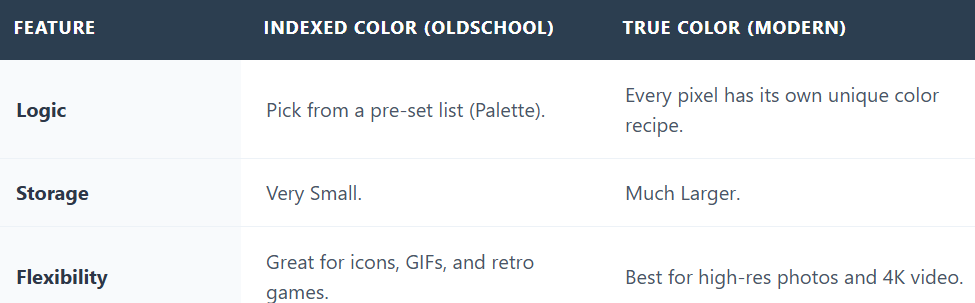
In older or optimized files, the computer doesn't store the actual color for every pixel. That would make the file huge. Instead, it uses a shortcut called **Indexed Color**.

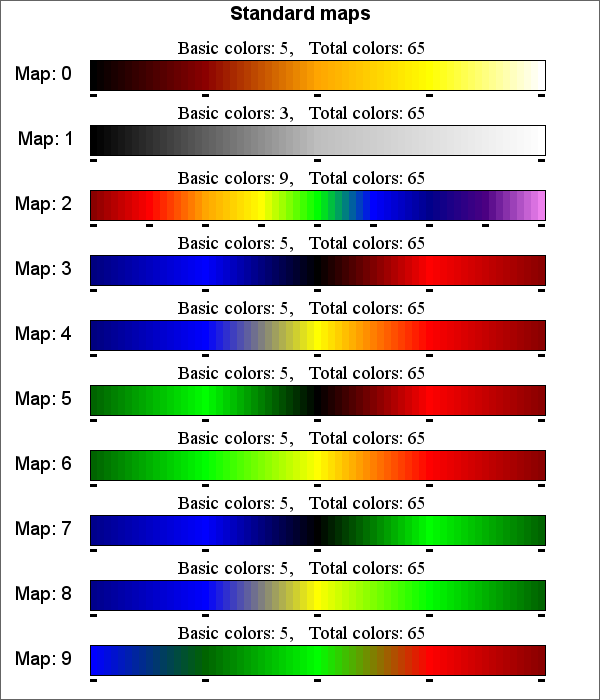
How it Works

1. **The Palette (The Crayon Box):** The file includes a small list of pre-defined colors (usually 256). Each color is assigned a simple ID number (0, 1, 2, etc.).
2. **The Bitmap (The Map):** Instead of saying "Bright Red," each pixel in your image just stores a number like **"3."**
3. **The Look-up:** When the image opens, the computer looks at the map, sees "3," checks the palette to see what color "3" is, and displays it.

Why this matters today:

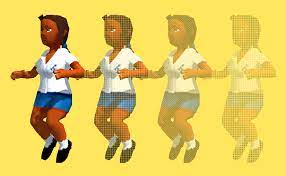
* **Performance:** It uses much less memory. A single number (the index) is smaller than a full color code.
* **Customization:** You can change the entire look of an image just by swapping the palette. If you change "Color #3" from Red to Blue, every "3" pixel in the image changes instantly. This is how "Palette Swapping" worked in old-school video games to make different enemies using the same character model.





Beyond the Basics: Dithering, Transparency, and More

In bitmap graphics, there are a lot of advanced techniques. Dithering helps create extra colors in images with limited color options, and transparency lets images blend smoothly with their background. The possibilities are endless!



Color in bitmaps is more than just a technical specification; it's a language, a tool, and a canvas for artistic expression.

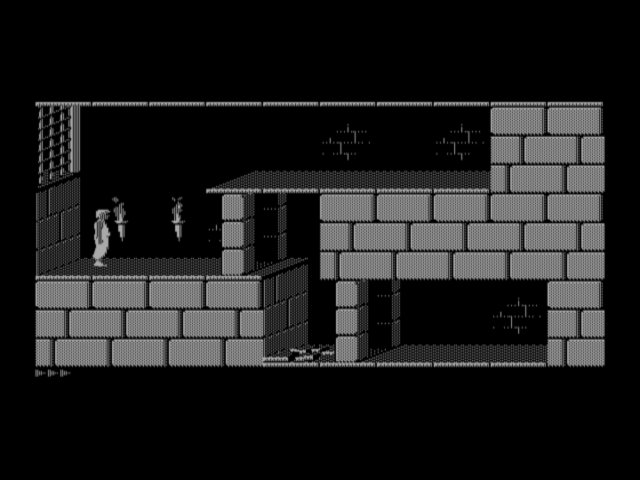
So, pick up your digital brush, dive into color, and let your creativity shine! Let’s go!!

HARDWARE HISTORY: THE EVOLUTION OF COLOR DEPTH **💾**

The quality of an image was originally limited by the **Video Card** inside the computer. As memory got cheaper, images got more colorful.

1. The Monochrome Era (1-bit)

* **The Hardware:** Hercules Graphics Card (HGC).
* **The Math:** 1 bit per pixel.
* **The Result:** Only two options: **On (White)** or **Off (Black)**.
* **Best For:** Simple text and basic lines. No shades of gray existed yet.

2. The 16-Color Leap (4-bit)

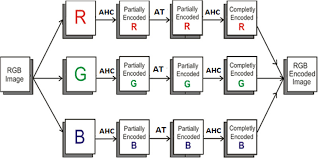
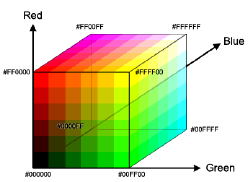
* **The Hardware:** EGA (Enhanced Graphics Adapter).
* **The Math:** 4 bits per pixel (24 = 16 possible combinations).
* **The Standard:** This became the "Windows Classic" look. It’s why old icons and cursors look like simple cartoons—they only had 16 "crayons" to work with.



3. What is IRGB Encoding?

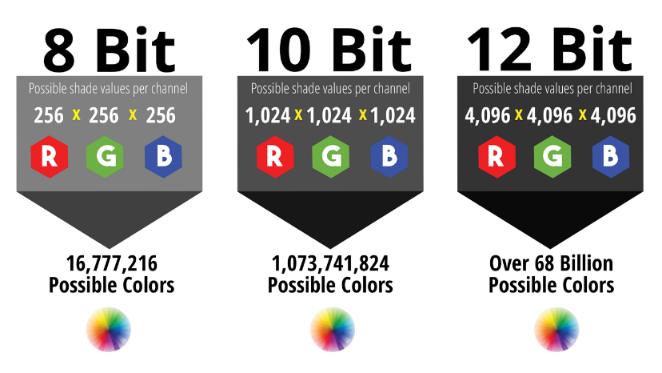
This was a shortcut used to get those 16 colors. It stands for **Intensity, Red, Green, and Blue**.

* The first 3 bits handled the actual color (R, G, B).
* The 4th bit (Intensity) acted like a **"brightness switch."** \* If Intensity was "Off," you got a dark version of the color; if it was "On," you got a bright/vivid version.

BEYOND 16 COLORS, A SPECTRUM OF POSSIBILITIES

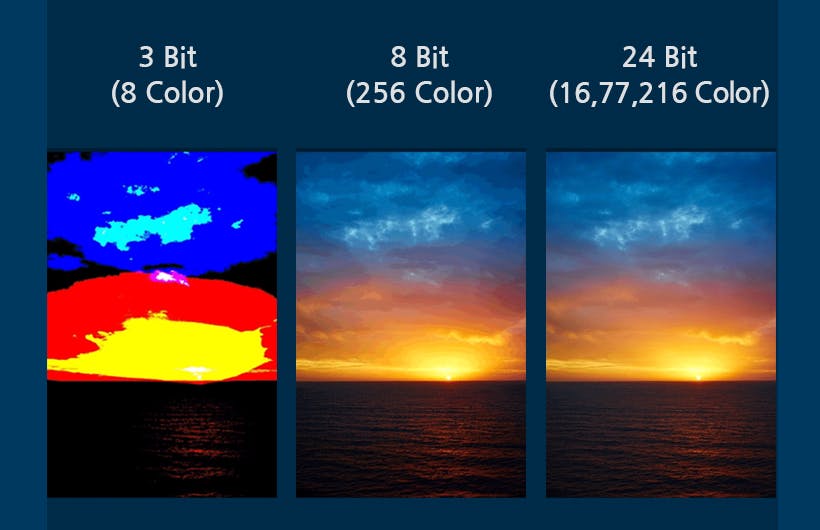
16-color bitmaps were important in the early days and are still used in some cases. But as people wanted better-looking graphics, display technology improved.



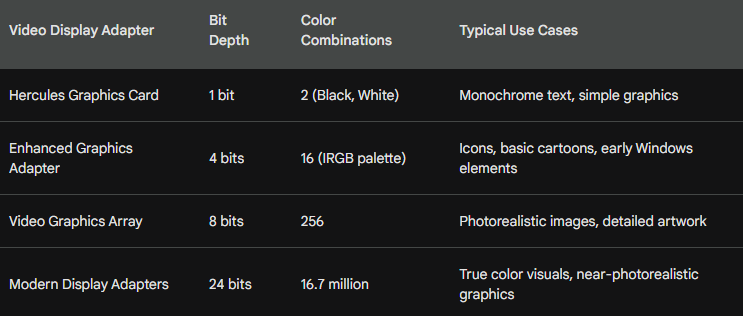
**VGA** was a big step forward—it supported 256 colors, allowing more detailed images and brighter, more colorful artwork.



Modern displays use **24-bit bitmaps**, also called ***True Color*.** They use 8 bits for red, green, and blue, which allows about 16.7 million different colors. This makes images look smooth, detailed, and realistic. Understanding bitmap history helps us choose better formats and appreciate modern graphics.



This table summarizes the color depths discussed:



Exploring Video Adapters and Bitmap Colors

Bitmaps are more than just pixels on a screen. The colors they can show depend heavily on the video adapters and hardware available at the time. As display technology evolved, so did color depth, memory use, and image quality.

In this section, we’ll look at how older hardware shaped bitmap colors and how those limitations led to the graphics we use today.

16-Color IRGB: The Early Days

Before modern displays with millions of colors, early Windows systems used a simple 16-color palette. This system came from character-based display modes and had very limited color choices, but it laid the foundation for later graphics systems.

This limited color range was called **IRGB** (Intensity, Red, Green, Blue). It came from the color limits of the IBM CGA character mode.

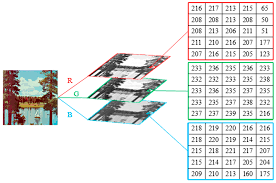
Each pixel used **4 bits**, which mapped to specific hexadecimal RGB values, as shown in the table below.



Memory Planes and Hardware Quirks

EGA graphics had a tricky memory layout. Instead of storing all four color bits together for each pixel, video memory was split into separate planes for intensity, red, green, and blue.

Luckily, Windows handled this complexity behind the scenes, so most applications didn’t need to worry about it.



From VGA to True Color: More Color, Better Graphics

The 16-color limit didn’t last long. In 1987, **VGA** introduced 8 bits per pixel, which meant images could use **256 colors**. This allowed graphics to look more detailed and realistic.



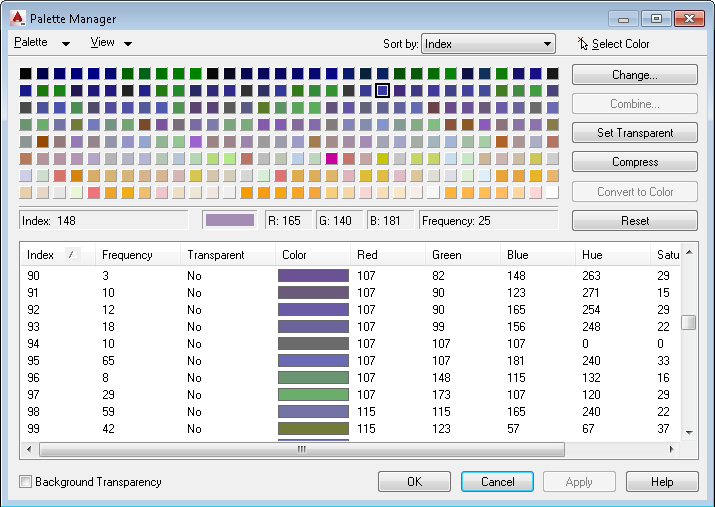
Early VGA systems had a drawback: to use 256 colors, they had to switch to a lower screen resolution, which wasn’t great for Windows. Later, **SVGA** fixed this by supporting 256 colors at **640×480 resolution**, making it the new standard.



Palette Magic: Windows Takes Control

Even though VGA supported 256 colors, Windows reserved **20 colors** for the system. Applications could use the remaining colors through the **Windows Palette Manager**.

This allowed programs to manage colors better and display images that looked closer to real life.

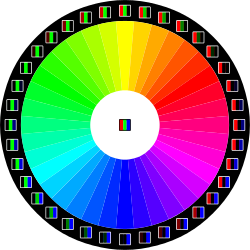


This table summarizes the reserved colors:

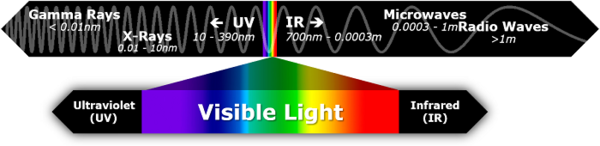


Moving Beyond Limits: High Color and True Color

To get even better visuals, video adapters moved to **16-bit (High Color)** and **24-bit (True Color)** modes. High Color could show thousands of colors, while True Color could display **millions**, making images much more realistic.

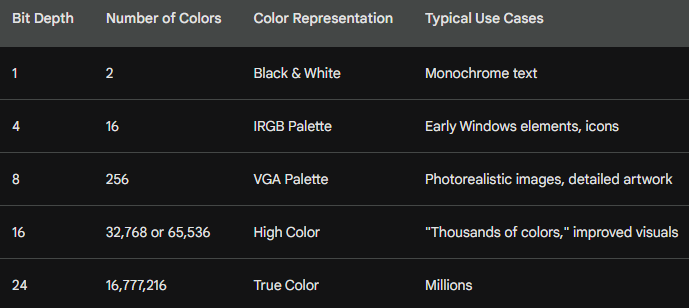


The **24-bit format** uses 3 bytes per pixel and can represent almost the full range of colors the human eye can see. This became the standard for years.



A Guide to Color Depths

Here’s a quick reference table summarizing the discussed color depths and their characteristics:



BITMAP SUPPORT IN GDI: FROM LEGACY TO MODERN MAGIC

The **Windows Graphics Device Interface (GDI)** has come a long way, from early monochrome displays to supporting millions of colors. Here’s how bitmap support evolved:

Early Days: GDI Bitmaps and Color Limits

Before Windows 3.0, **GDI bitmaps** were tightly linked to the hardware.

* Monochrome screens could only display **black and white** bitmaps.
* A 16-color VGA adapter meant bitmaps were limited to the **IRGB palette**.

Because of this, bitmaps weren’t very flexible—they couldn’t easily move between devices with different color capabilities.



DIBs: The Device-Independent Revolution

Windows 3.0 brought a major upgrade with **Device-Independent Bitmaps (DIBs)**. Unlike earlier bitmaps, DIBs weren’t tied to specific hardware.

* Each DIB includes its **own color table**, mapping pixel values to actual RGB colors.
* This made it possible to display the same bitmap on **any raster device**, no matter the hardware.
* Converting colors to match the device’s capabilities could still be tricky, but the flexibility was a game-changer.

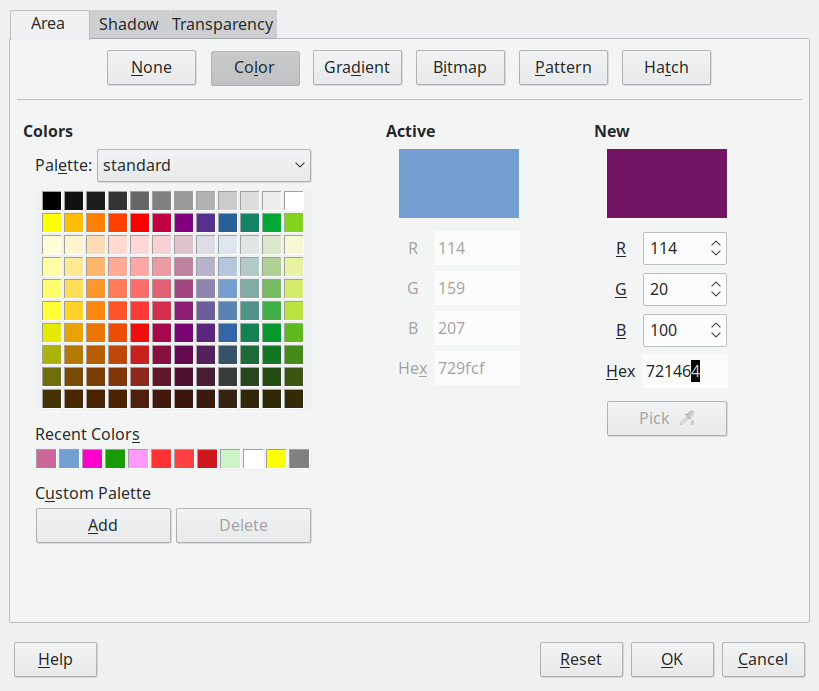


The Palette Manager: A Color Partner for DIBs

Windows 3.0 also introduced the **Palette Manager**, which worked hand-in-hand with DIBs on **256-color displays**.

* Applications could customize colors using the Palette Manager.
* This helped DIBs display accurately across different devices.

Together, DIBs and the Palette Manager gave developers **much more control over color**, paving the way for richer graphics.



Evolving DIBs, ICM, and Beyond

Microsoft kept improving DIBs in later versions of Windows.

* **Windows 95/NT 4.0** introduced **Image Color Management (ICM)**.
* ICM lets DIBs define **exact color requirements**, going beyond the limits of the display hardware.
* This ensures colors stay consistent across different devices, making visuals more accurate.

Legacy and Modern Bitmaps

Even with DIBs, older **GDI bitmaps** are still important.

* Concepts like **bit-block transfer** and how legacy bitmaps interact with GDI are essential to understand.
* Knowing both old and new methods gives you a solid foundation for working with bitmaps in Windows.

Mastering the Bitmap Landscape

To navigate GDI bitmaps:

1. Start with **basic GDI bitmaps**.
2. Move on to **DIBs** and then **ICM**.
3. Understand the historical evolution—it helps you **appreciate improvements** and **use modern tools effectively**.

With this knowledge, you can create **better visuals**, improve user experiences, and push the limits of what bitmap graphics can do in Windows.

THE BITBLT: A POWERFUL PIXEL MOVER AND THE ENGINE OF 2D GRAPHICS

In the world of bitmaps, **BitBlt** (“bit-block transfer”) is a key tool.

It doesn’t just copy pixels—it **moves, combines, and manipulates them**.

BitBlt is central to **drawing, animations and complex graphics** in Windows.

Understanding BitBlt gives you the **power to control visuals** and fully master the GDI graphics environment.

**BitBlt** (pronounced "bit-blit") stands for **Bit**wise **Bl**ock **T**ransfer.

It is the fundamental operation that makes 2D graphics fast.

1. The Origin: From Memory to Pixels

* **The "BLT" Instruction:** Originally appeared on the **DEC PDP-10** computer. It was a fast way to move large chunks of data from one spot in memory to another.
* **SmallTalk's Innovation:** Developers realized that if you treat a screen like a block of memory, you can use "BLT" to move images around instantly.
* **The Verb:** Programmers still use the term **"blitting"** to describe drawing an image onto the screen.

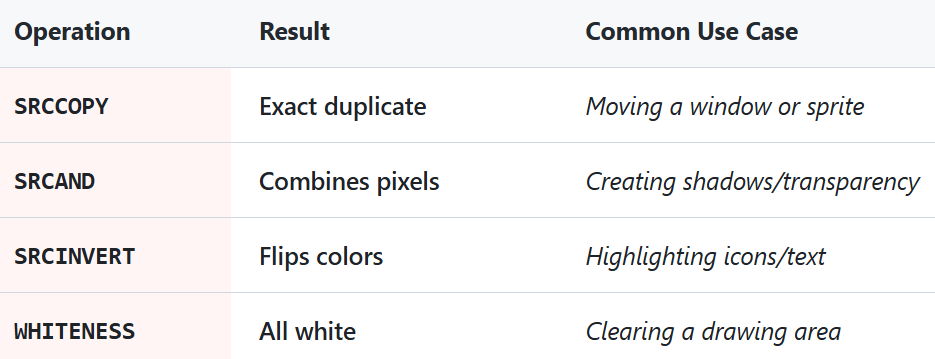
2. It’s Not Just a Copy-Paste

While it *can* copy pixels, BitBlt is actually a **Logic Engine**. It looks at three things: a **Source** (the image), a **Pattern** (like a brush), and a **Destination** (the screen). It combines them using **Boolean Logic** (AND, OR, XOR, NOT).

* **Simple Copy (SRCCOPY):** Just moves pixels from A to B.
* **Transparency & Masking:** Uses the **AND** and **OR** operations. By using a "mask" (a black and white version of your image), BitBlt can filter out a background so your character doesn't have a white box around them.
* **Color Inversion (NOT):** Instantly flips colors (e.g., turning black to white). This was a classic way to show "selected" text or icons in early Windows.
* **Blending & Patterns:** It can combine a repeating texture with an image to create shadows or transparency effects.

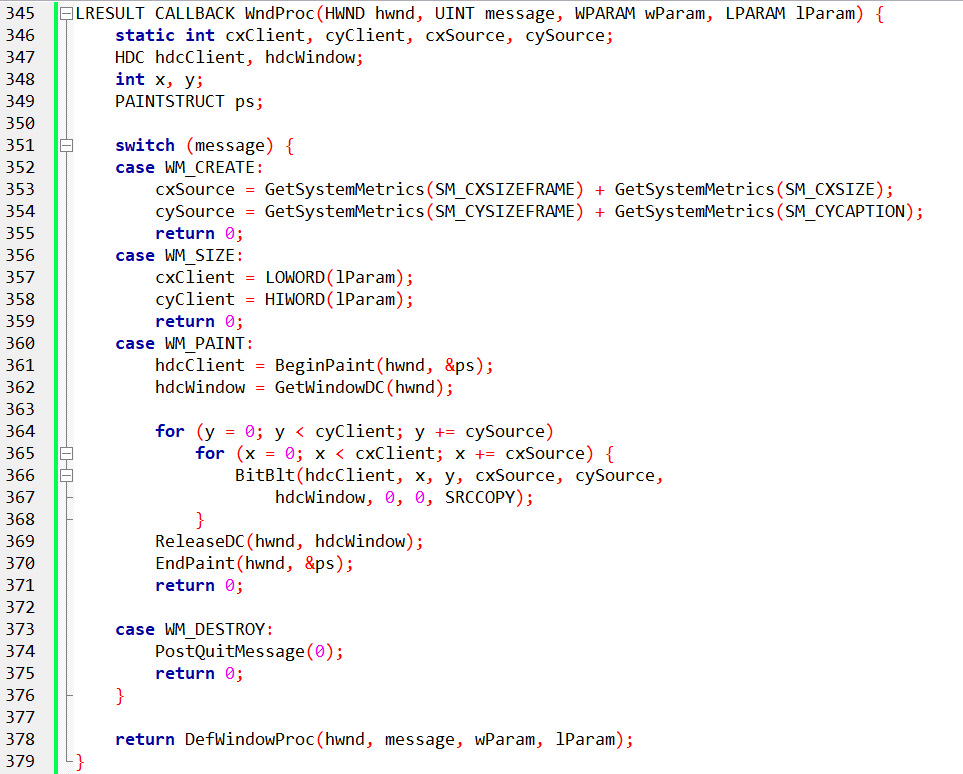
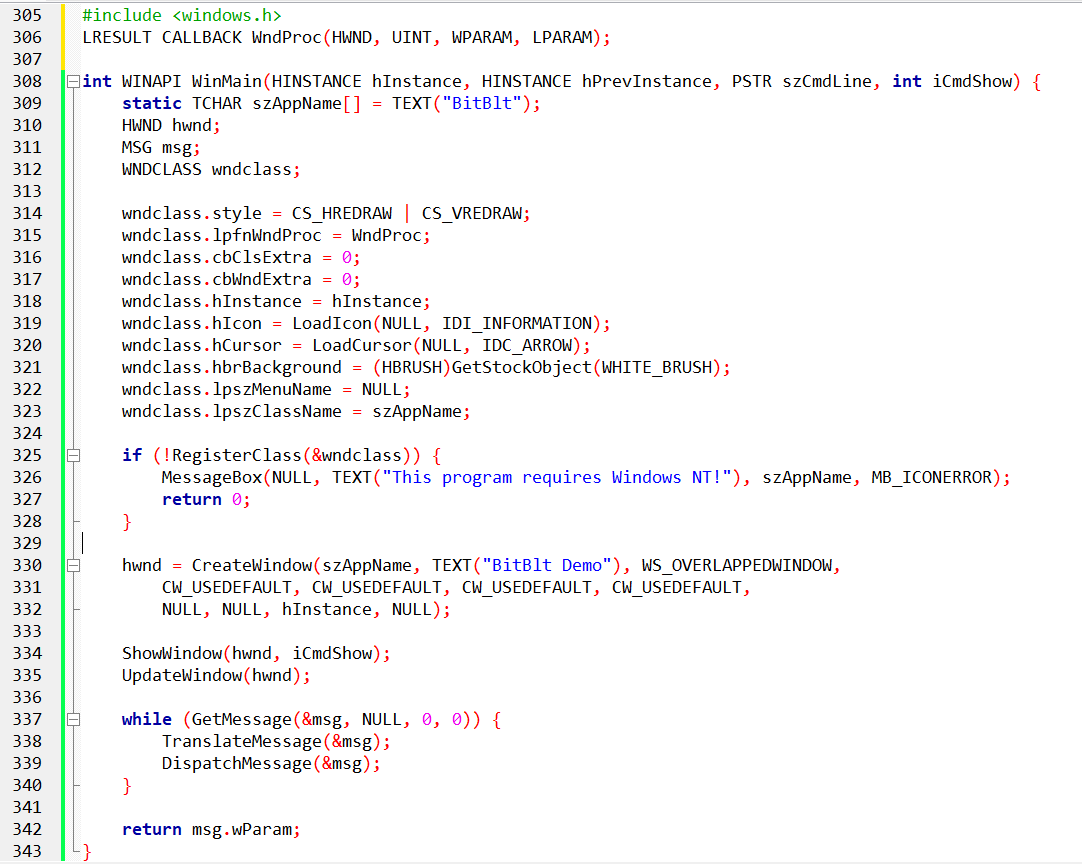
3. Why it Matters Today

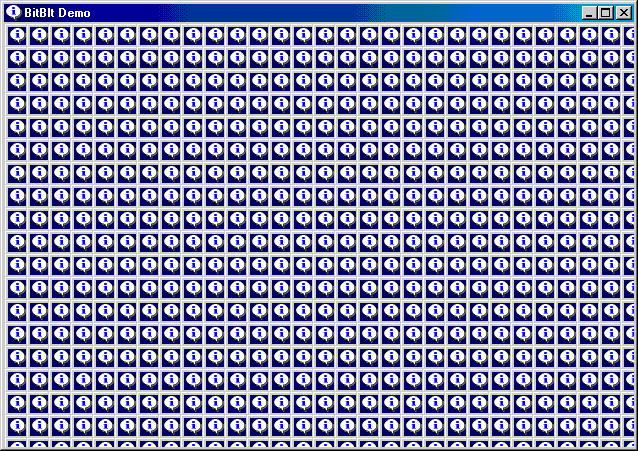
In the 2020s, while 3D games use GPUs (Graphics Processing Units), almost every **2D interface** (like the windows on your desktop, browser rendering, and mobile app menus) still *relies on the logic of BitBlt* to render quickly without draining your battery.



The BITBLT program below demonstrates a basic use case.

It copies the program's system menu icon, located at the top left corner of the window, to its client area.





The Important Ideas

BitBlt copies pixels from one device context (DC) to another.

That’s it. **Source → destination.**

1️. Source and Destination

You always have:

* **Source DC** → where pixels come from
* **Destination DC** → where pixels go

In this program:

* The source is the **window**
* The destination is the **client area**

Same screen, different coordinate origins.

2️. Coordinates Matter

* (xSrc, ySrc) → where copying starts in the source
* (xDst, yDst) → where pixels land in the destination
* Changing destination coordinates lets you copy the same image **many times**.

This is how the **tiling effect** happens.

3️. Size Controls the Pattern

* cx and cy decide how big each copied block is.
* The loop moves across the client area using these sizes.
* Result: the window content is repeated until the client area is filled.

4️. Raster Operation (ROP)

* dwROP decides **how pixels are combined**.
* This code uses: SRCCOPY → direct copy, no effects.
* Other modes (XOR, OR) can create visual tricks, but they are optional.

What This Program Is Demonstrating

BitBlt can copy **real screen pixels**, not images.

You can copy the same source many times by looping.

Visual effects come from:

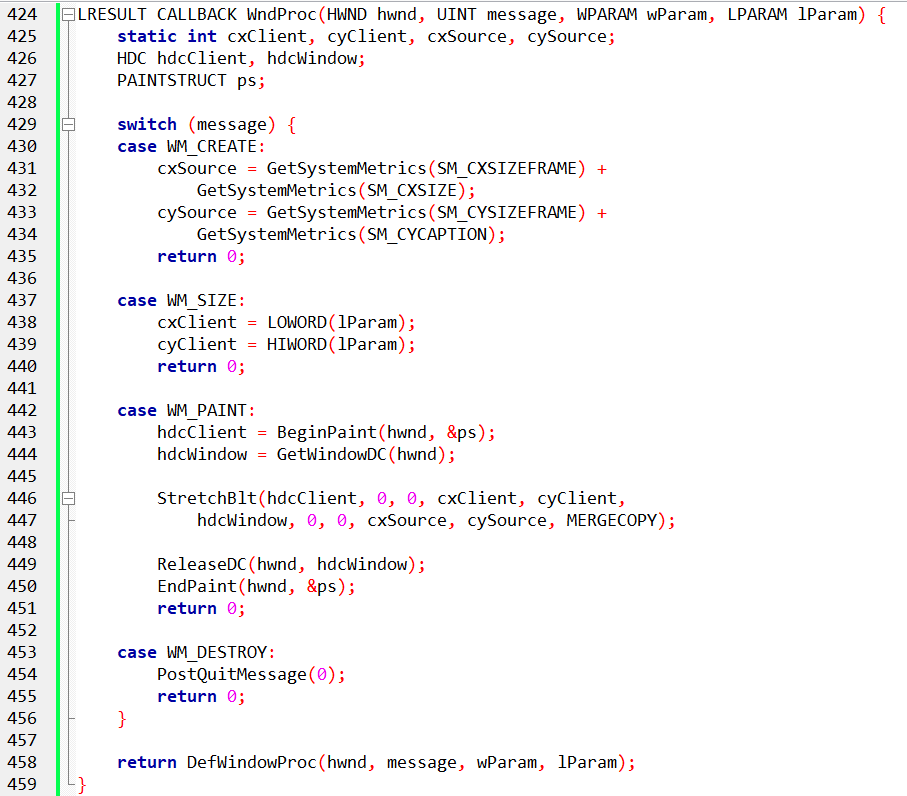
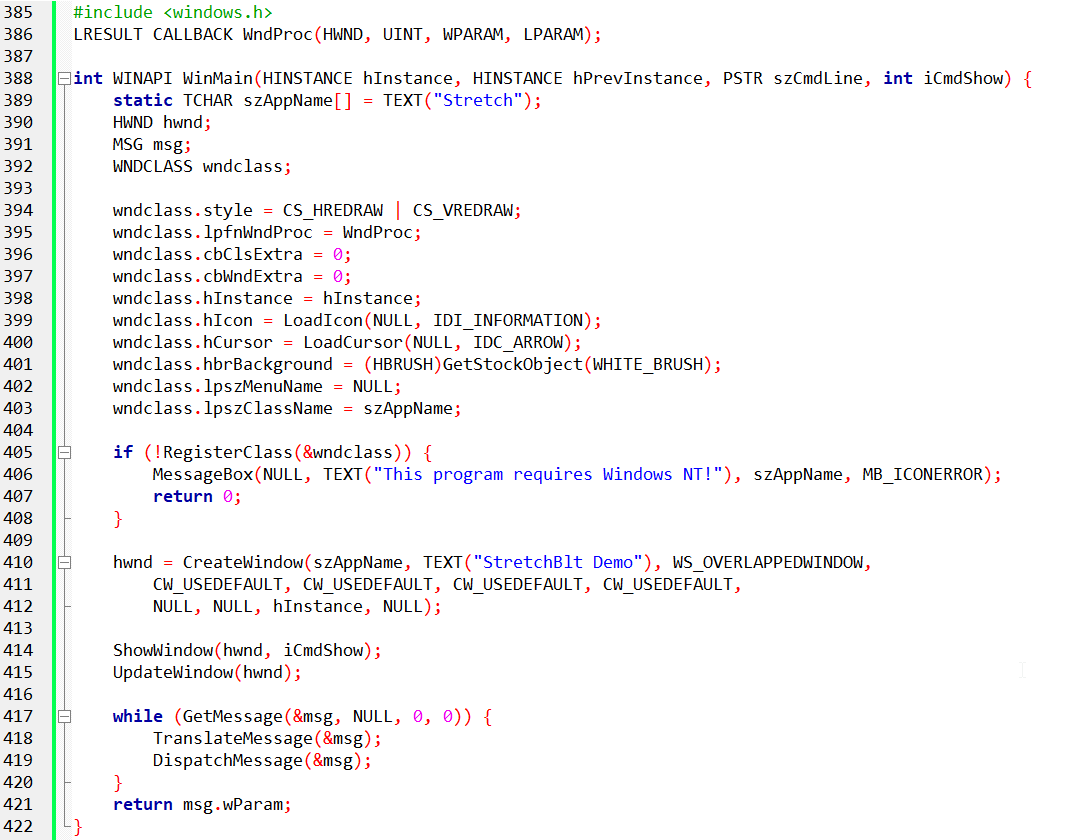
* coordinates
* size
* raster operation

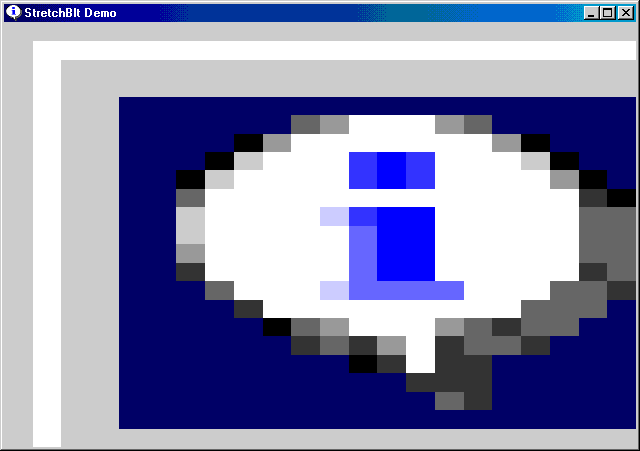
Nothing else.

BitBlt works on **video memory**, not off-screen images.

If the source goes off-screen, copying may break.

Source and destination DCs must be compatible.





StretchBlt — The Point

StretchBlt copies pixels **and** changes their size.  
It can stretch or shrink the image while copying.

That’s the only difference from BitBlt.

1. When to Use It

* Use BitBlt → same size copy
* Use StretchBlt → resized copy

2. The Only Parameters That Matter

* **Source rectangle** → original image size
* **Destination rectangle** → final image size
* **ROP** → how pixels are combined

Everything else is just coordinates.

3. What the STRETCH Program Is Showing

* The source image is the **system menu icon**.
* The destination rectangle is the **entire client area**.
* Result: the icon is stretched to fill the window.

Only **one** StretchBlt call does all the work.

4. How Stretching Works

* Bigger destination → image stretches
* Smaller destination → image shrinks
* Aspect ratio is **not protected** unless you do it yourself

Think: resizing an image without “lock aspect ratio”.

4. Raster Operation (ROP)

* MERGECOPY blends pixels while stretching.
* SRCCOPY does a plain resize.
* Other ROPs are optional effects, not required knowledge here.

5. Important Limits (Worth Remembering)

Stretching reduces image quality. Large scaling causes blur or pixelation.

Source and destination DCs must be compatible.

Mapping Modes + BitBlt — The Real Story

1️. Logical Units vs Pixels

BitBlt and StretchBlt work in **logical units**, not pixels.

Mapping modes decide how logical units turn into **real pixels**.

That’s the core idea.

2️. Conversion Happens First

Before copying pixels:

Windows converts **source** coordinates to pixels.

Windows converts **destination** coordinates to pixels.

Each DC is converted **separately**.

You don’t control this step — Windows does.

3️. Same Mapping Mode = Simple Copy

If both DCs use the **same mapping mode**:

The pixel sizes match.

Windows performs a normal BitBlt.

No scaling. No stretching.

4️. Different Mapping Modes = StretchBlt

If pixel sizes **don’t match** after conversion:

Windows automatically uses **StretchBlt behavior**.

Scaling is unavoidable in this case.

This is why mapping modes matter.

5️. Flipping and Mirroring (Important Trick)

StretchBlt allows **negative width or height**.

Negative values flip the image.

Examples:

* Negative width → mirror left ↔ right
* Negative height → flip top ↔ bottom

No extra math. Just signs.

6. What to Remember

* Mapping modes affect how sizes are interpreted.
* Conversion to pixels always happens first.
* Stretching happens when sizes don’t match.
* Negative sizes = flipping.

That’s it.

StretchBlt Stretching Modes — What You Need

Stretching means pixels must be **reused or dropped**.  
This can cause blur or artifacts.  
Stretch modes control **how pixels are handled**.

The Modes

1️⃣. BLACKONWHITE

* Favors **black pixels**
* Best for **black-on-white monochrome** images
* Bad for color images (black artifacts)

2️⃣. WHITEONBLACK

* Favors **white pixels**
* Best for **white-on-black monochrome** images
* Can wash out bright areas

3️⃣. COLORONCOLOR (Most Useful)

* Drops rows/columns, no blending
* Best general choice for **color images**
* Can look blocky with large scaling

4️⃣. HALFTONE

* Averages colors
* Uses a halftone palette
* Rarely used outside special effects

5. What to Actually Remember

* Monochrome → BLACKONWHITE or WHITEONBLACK
* Color images → COLORONCOLOR
* Fancy effects → HALFTONE

**That’s It**

Raster Operations (ROP) — The Core Idea

BitBlt and StretchBlt don’t just copy pixels.  
They combine **three inputs**:

* **Source** → what you copy
* **Destination** → what’s already there
* **Pattern** → the current brush in the destination DC

The ROP decides how these three are combined.

1. Why There Are So Many ROPs

* Each pixel can be **on (1)** or **off (0)**.
* Combining source, destination, and pattern gives **256 possible rules**.
* Windows names only the common ones.
* The rest are numeric codes.

You don’t need to memorize them.

2. The ROPs Worth Knowing

**BLACKNESS**

* Fills the destination with black.
* Source and pattern are ignored.

**SRCCOPY (Most Used)**

* Copies source directly to destination.
* No blending, no tricks.

**NOTSRCCOPY**

* Inverts the source before copying.
* White becomes black, black becomes white.

**SRCERASE**

* Removes source pixels from the destination.
* Destination pixels remain where source is empty.

**MERGECOPY**

* Combines source with the pattern brush.
* Used for simple blending effects.

3. How to Think About ROPs

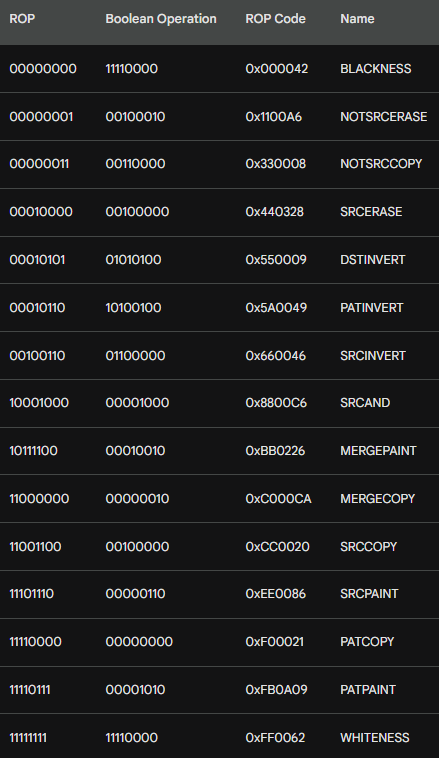
* ROPs are **bitwise rules**, not image filters.
* They are fast, simple, and very low-level.
* Most programs only ever use **SRCCOPY**.

Everything else is for effects.

4. What to Remember

* ROP = rule for combining pixels
* 3 inputs: source, destination, pattern
* 256 possible operations
* Only a few are commonly useful

*Below is just a small version of the table of 256 entries…ROP table has a lot of entries.*



**Raster Operations (ROPs)** determine how pixels from a **Source (S)**, a **Pattern (brush)**, and a **Destination (D)** are combined to produce a final output.

* **ROP Code:** A numeric identifier used by graphics functions like BitBlt or StretchBlt to specify the type of raster operation.
* **Logic:** The operation is performed using **bitwise mathematics** such as AND, OR, XOR, and NOT.
* **Monochrome Logic:** Each pixel is represented as 0 = Black and 1 = White. The ROP defines how these bits are combined.
* **Color Logic:** The same bitwise operation is applied independently to each bit of the **color components** (R, G, B) of the pixels.

Key ROP Operations



What the ROP Table Shows

* **ROP code**  
  A number Windows uses to decide the pixel rule.
* **Name**  
  A readable label (like BLACKNESS, PATCOPY).
* **Bitwise logic**  
  The actual rule using AND, OR, XOR, NOT.
* **Boolean form**  
  Same rule, written in simple C-style logic.
* **Monochrome example**  
  Shows the result using:
  + 0 = black
  + 1 = white

This makes the behavior easier to see.

ROPs Worth Knowing

**PATCOPY**

* Copies the **pattern** to the destination.
* Source is ignored.

**PATPAINT**

* Uses **pattern + destination**.
* Black source pixels force white.
* Destination turns black only in specific cases.

You don’t need the full logic — just know it mixes pattern and destination.

Color Displays (What Matters)

* Color pixels have multiple bits (R, G, B).
* ROP logic runs **per bit**.
* Final color depends on the system palette.

Details beyond this are not important for now.

Key Points to Remember

* ROPs are **bitwise rules**, not image effects.
* Pattern acts like a **stencil**.
* Source provides pixel data.
* Destination is what already exists.

That’s the mental model.

ROPs are the primary method for creating **non-rectangular images** (transparency masking) by using bitwise AND/OR operations to "filter" out specific pixel areas.

**Per-Bit Operation:** On color screens, ROPs don't "see" colors; they manipulate the binary bits representing those colors.

**The Palette:** The final visual color depends on the video board’s palette mapping. If the palette changes, the same ROP code may produce a different visual color.

PATBLT: THE SIMPLEST BRUSH FOR YOUR CANVAS

PatBlt fills a rectangle using the **current brush**.  
There is **no source image**.

1. When to Use It

* You want to **fill or invert** an area.
* You don’t need pixels from another DC.
* You only care about the **pattern + destination**.

2. Function Call



No source DC. No stretching. No copying.

What the Parameters Mean:

* **hdc** → destination DC
* **x, y** → top-left position
* **cx, cy** → size of the rectangle
* **dwROP** → raster operation to apply

That’s all.

3. What PatBlt Actually Does

Applies a **ROP rule** to a rectangle.

Uses:

* the **destination**
* the **current brush (pattern)**

Source is not involved.

Think: *apply a rule to an area*.

4. ROP Codes Limitation

PatBlt supports **only ROPs that don’t need a source**.

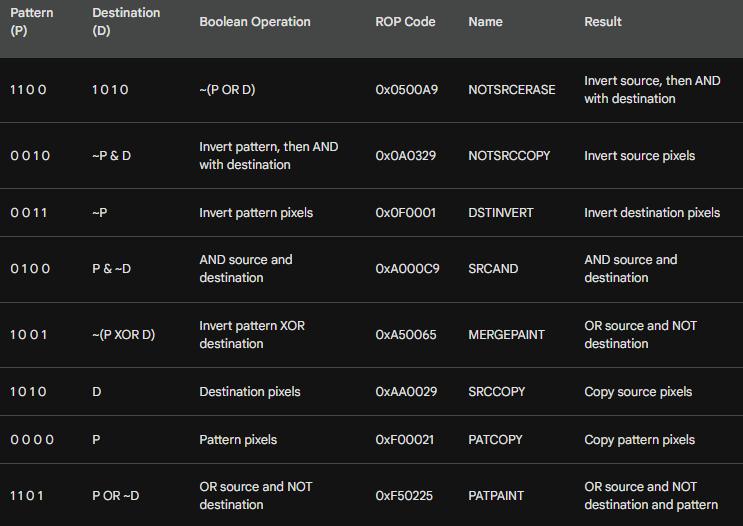
Total: **16 ROP codes**.

These operate using:

* pattern
* destination
* or constants (black / white).

5. What to Remember

* PatBlt = pattern-based fill
* No source DC
* Simple and fast
* Limited ROP set (16)



**Functionality:** Each ROP code is a unique mathematical instruction for how PatBlt modifies pixels within a rectangle.

**Creative Control:** Experimenting with different codes allows for various visual effects and "digital brush" styles.

**Deep Dive:** Use the bitwise logic table to predict exactly how a specific code will change your image.

**The Goal:** Master the relationship between bitwise operations and the resulting visual output.

PatBlt: Your Handy Tool for Rectangle Magic

PatBlt is used to apply a brush/pattern to a rectangular area. It only uses the **Pattern (P)** and **Destination (D)**; it does **not** use a source bitmap.

1. Black and White Magic

**Black Rectangle:** PatBlt (hdc, x, y, cx, cy, BLACKNESS);  
Fills the rectangle with Black (all bits to 0).   
Think of it as a digital eraser!

**White Rectangle:** PatBlt (hdc, x, y, cx, cy, WHITENESS);   
Fills the rectangle with White (all bits to 1).

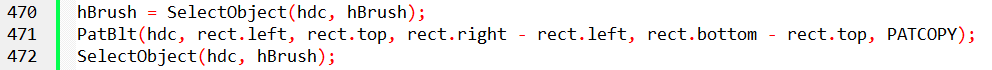
2. Inversion Power:

**Invert Rectangle:** PatBlt (hdc, x, y, cx, cy, DSTINVERT);  
This handy trick flips the colors of the specified rectangle(~D)  
Pixels that were black turn white, and vice versa.  
Colors become their negatives.

**Invert with White Brush:** PatBlt (hdc, x, y, cx, cy, PATINVERT);   
XORs the current Pattern with the Destination (P ^ D)   
Inverts using the current brush.  
With a white brush selected in the DC, behavior is similar to DSTINVERT.

3. Behind the Scenes

FillRect internally uses PatBlt. I mean,   
The common FillRect function is actually a wrapper for PatBlt.   
FillRect selects your brush into the Device Context (DC) and then calls PatBlt using the PATCOPY ROP code.



PatBlt Coordinates & Mapping Modes — Simple Version

1. Rectangle Coordinates

PatBlt draws a rectangle using:

* (x, y) → starting corner
* (cx, cy) → width and height

Rectangle = (x, y) to (x + cx, y + cy)

Signs of cx and cy affect **direction**:

* Negative cx → rectangle extends left
* Negative cy → rectangle extends down (or up in some mapping modes)

2. Mapping Modes

**MM\_LOENGLISH**:

* x increases → right
* y increases → up

Use **negative cy** to paint above the anchor point

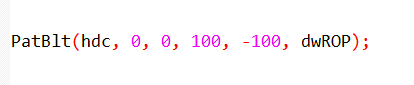
Use **positive cx** to paint right of the anchor point

3. Key Points

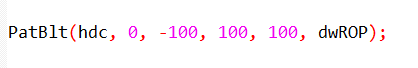
* (x, y) = anchor corner
* cx, cy = width and height (signs control direction)
* Mapping mode changes the coordinate orientation

Painting Your Square Inch

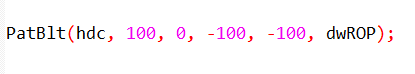
Here are four valid ways to paint a one-inch square at the upper left corner of the client area using PatBlt:



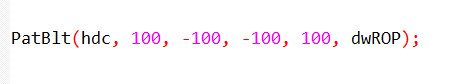
This sets the upper left corner at (0, 0), uses a width of 100, and a negative height of -100 to paint above the point.



This again uses (0, -100) as the upper left corner, with a width of 100 and a positive height to paint below the point.



This shifts the rectangle to the right by setting (100, 0) as the upper left corner, then uses negative values for both width and height to paint a square inch within the bounding box.



This combines the rightward shift with the previous approach, placing the upper left corner at (100, -100) and using negative width and positive height to paint the square inch.

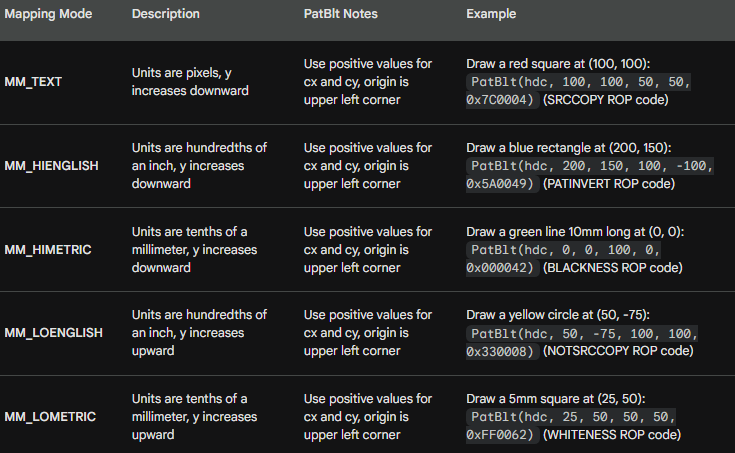
Remember!

* Always use the mapping mode's coordinate rules to determine the signs of cx and cy for accurate rectangle positioning.
* Experiment with different values to see how PatBlt interacts with your chosen mapping mode.

Mapping Modes: Your Guide to Precise Patblt Painting

Understanding mapping modes is the only way you will get to work well with PatBlt, which is your GDI brush, with accuracy.

Here's a quick guide to some common modes and their impact on PatBlt parameters:



Remember

This is just a glimpse into the world of mapping modes.

Windows offers many more options with unique properties.

Always refer to the official documentation.

Experimenting with different modes and parameters is the best way to get creative!

GDI BITMAP OBJECTS — DDB VS DIB (Repeat)

1. Device-Dependent Bitmap (DDB)

* Tied to a specific display device.
* Fast for operations on that device (screen, DC).
* Efficient for **direct drawing**.

2. Device-Independent Bitmap (DIB)

* Stores image data **independent of the device**.
* Can be used on any device or platform.
* Slower than DDB for direct device operations.

3. Converting Between DDB and DIB

* DIB → DDB: Optimized for a specific device.
* DDB → DIB: Makes the image platform-independent.
* Conversion allows flexibility **without losing data**.

4. When to Use

* **DDB** → high performance, device-specific tasks.
* **DIB** → flexible, cross-platform use or image processing.

5. Key Points

* Both exist for different purposes.
* DDB = speed, DIB = flexibility.
* Knowing when to use each is essential for Windows graphics programming.

*Three primary functions handle DDB creation:*

CreateBitmap

* Specify width (cx), height (cy), color planes (cPlanes), bits per pixel (cBitsPixel).
* Optional: provide pixel data with bits or leave NULL for empty bitmap.



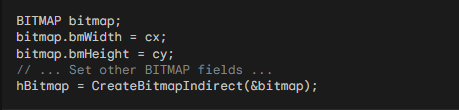
CreateCompatibleBitmap

* Automatically matches a device context (hdc).
* Simple and safe for drawing on a specific screen or DC.



CreateBitmapIndirect

* Use a pre-filled BITMAP structure.
* Gives precise control over bitmap properties, including initial pixel data (bmBits).



Parameter Breakdown:

* **cx, cy:** Width and height of the DDB in pixels.
* **cPlanes:** Number of color planes (typically 1 for monochrome, 4 for CMYK).
* **cBitsPixel:** Number of bits per pixel (1 for monochrome, 8 for typical RGB).
* **bits:** Pointer to an array of initial pixel data (optional).
* **hdc:** Handle to the device context for compatible bitmap creation.
* **bitmap:** A pre-filled BITMAP struct for CreateBitmapIndirect

Destroying DDBs

* Use DeleteObject(hBitmap) when done.
* Prevents **memory leaks** and keeps your app stable.



**Memory usage:** frees space when DDBs are no longer needed

**Performance:** avoids unnecessary memory allocation

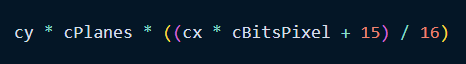
**Stability:** prevents crashes or slowdowns

Memory Allocation and Padding

Windows allocates memory for the DDB based on its dimensions, but with some padding:

Each row of pixels has an even number of bytes (padding with zeros if needed).

The total allocated memory is:



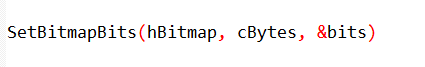
Practical Tips

* Use **CreateCompatibleBitmap** most of the time.
* Use **CreateBitmap** for custom bitmaps.
* Use **CreateBitmapIndirect** or GetObject for detailed control.
* Always **delete DDBs** with DeleteObject after use.
* Experiment with parameters to understand memory and performance effects.

This next section dives deep into **bitmap bits**, and their **manipulation**, plus we talk about **device dependence.**

Setting and Getting Pixel Bits

DDBs can be created with initial pixel data or remain uninitialized. Two functions handle bit manipulation:



This function allows you to replace existing pixel data or initialize a new DDB with your desired bit pattern. cBytes specifies the number of bytes to copy, and bits is a pointer to the source buffer containing the pixel data.



This function retrieves a copy of the existing pixel data from a DDB into the provided buffer pointed to by bits.

Understanding Bit Arrangement:

Pixel bits are arranged starting from the top row, adhering to a crucial rule: each row must have an even number of bytes.

Monochrome DDBs (1 plane, 1 bit per pixel) have simple bit values: 1 represents the "on" state and 0 the "off" state.

Non-monochrome DDBs (multiple planes or bits per pixel) are more complex. Their pixel values directly influence the displayed color, but without referencing a fixed color table.

Device Dependence and Palette Lookups:

The displayed color of a non-monochrome DDB pixel relies on the device's specific palette lookup table (PLT).

The pixel value acts as an index into this table to determine the actual RGB color displayed on the screen. This dependency on the device's PLT makes non-monochrome DDBs highly device-specific, limiting their portability and predictability.

Practical Takeaways:

Monochrome DDBs: Setting pixel bits directly using SetBitmapBits is straightforward and effective.

Non-monochrome DDBs: Avoid relying on interpreting pixel values directly. Use them as an abstract representation of color for device-specific drawing operations.

Newer Alternatives: SetDIBits and GetDIBits offer greater flexibility and device independence for color DDB manipulation. We'll explore them in the next chapter.

Additional Functions:

SetBitmapDimensionEx/GetBitmapDimensionEx: These functions allow setting and retrieving a bitmap's dimensions in 0.1 mm units. This information serves as a tag for associating metrical data with a DDB, but it has no direct impact on drawing or display.

* Understand the concept of device dependence when dealing with non-monochrome DDBs.
* Leverage SetBitmapBits and GetBitmapBits cautiously, mainly for monochrome DDBs.
* Consider SetDIBits and GetDIBits for more versatile color DDB manipulation in future chapters.

MEMORY DEVICE CONTEXT

The memory device context (MDC) is a crucial element in GDI, serving as a virtual canvas for drawing operations before displaying them on the actual screen. Understanding its role and interaction with DDBs is essential for mastering GDI techniques.

What is an MDC?

Unlike a regular device context (DC) associated with a physical device like a screen or printer, an MDC exists solely in memory.

It acts as a "compatible" counterpart to a real DC, inheriting its properties but offering a flexible drawing space decoupled from the physical output device.

Creating an MDC:

*Creating an MDC involves two primary methods:*

Using CreateCompatibleDC: This function takes a handle to a real DC (e.g., the video display DC) and creates an MDC compatible with its parameters. This ensures your drawings in the MDC will translate smoothly to the target device.



Using NULL with CreateCompatibleDC: This shortcut creates an MDC compatible with the video display DC, making it ideal for general drawing tasks without specifying a specific real device.

Selecting a DDB:

An MDC's display surface is initially tiny (1x1 pixel), limiting its usefulness. To unlock its potential, you need to select a GDI bitmap object (DDB) into it using SelectObject:



This essentially assigns the DDB as the MDC's drawing canvas, allowing you to draw on the bitmap's surface. However, remember that:

Only compatible DDBs can be selected. The DDB's color planes and bits per pixel must match the MDC's compatibility settings.

Bizarre DDBs (e.g., 5 planes, 3 bpp) won't work due to incompatibility issues.

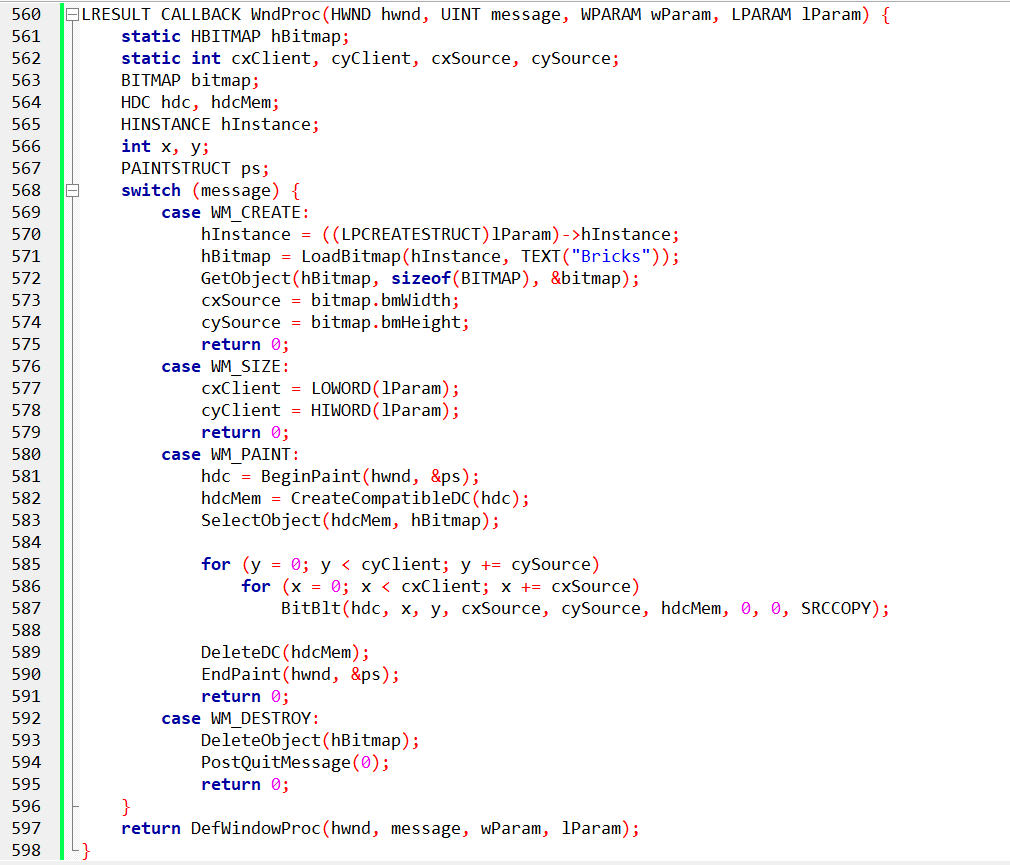
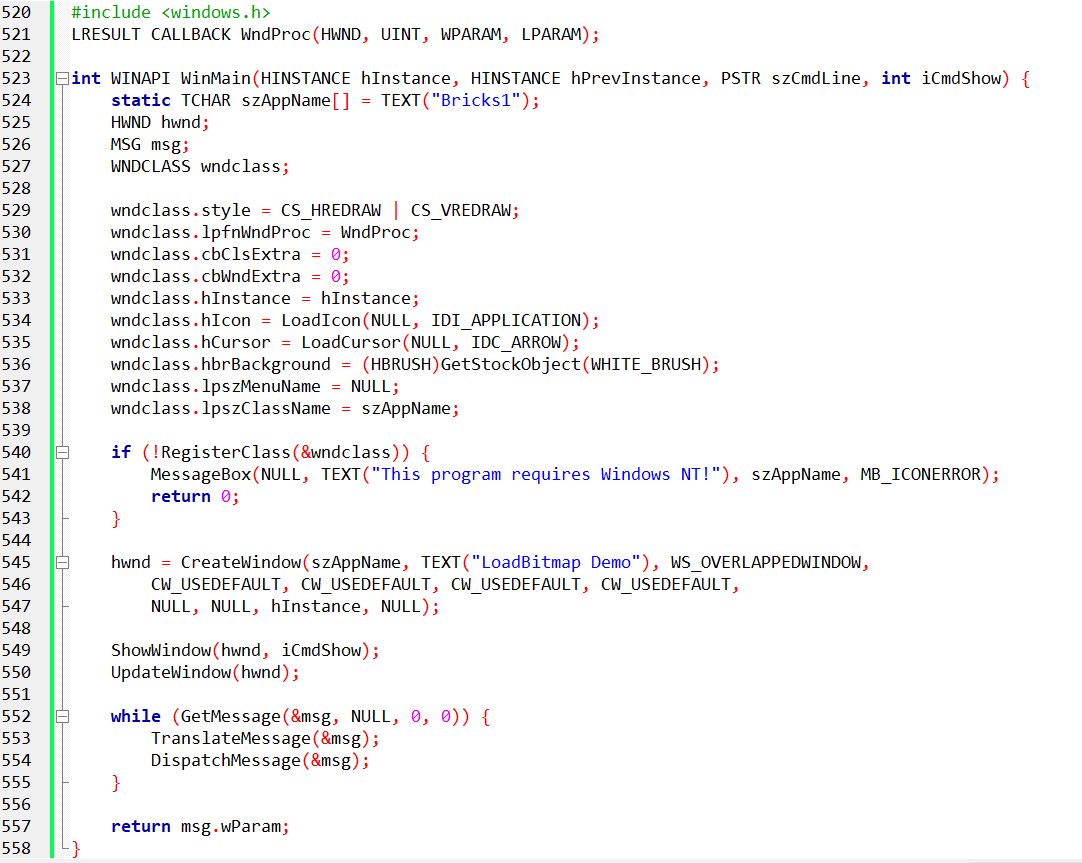
MDC's Power and Potential:

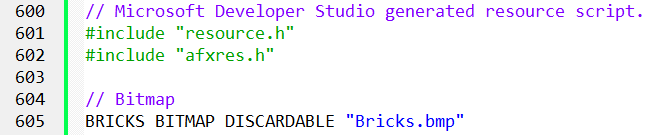
Once a DDB is selected, the MDC becomes a powerful tool for off-screen drawing:

* You can use GDI drawing functions to manipulate the bitmap directly, creating complex visuals without affecting the screen until you're ready.
* You can leverage BitBlt to transfer drawn content from the MDC to the real DC, effectively displaying your off-screen artwork on the actual screen.
* Conversely, you can use BitBlt to capture a portion of the screen into a bitmap using the MDC as a temporary storage container.

Key Takeaways:

* MDCs provide a flexible drawing space independent of the physical output device.
* Create MDCs compatible with specific devices or the video display using CreateCompatibleDC.
* Select compatible DDBs into the MDC to create a drawing canvas.
* Leverage MDCs for off-screen drawing and manipulation before displaying the final results.
* Master BitBlt to transfer content between MDCs and real DCs for versatile drawing and screen capture.







BRICKS.BMP IN BRICKS1.C

The BRICKS.BMP file plays a crucial role in the BRICKS1.C program, serving as the source of the repeating brick pattern displayed on the window. Let's delve deeper into its purpose and how it interacts with the code:

Resource Definition:

BRICKS BITMAP DISCARDABLE "Bricks.bmp": This line in the BRICKS1.RC file defines a resource called BRICKS of type BITMAP.

DISCARDABLE: This keyword indicates that the resource can be unloaded from memory when not needed, potentially improving memory management.

"Bricks.bmp": This path points to the actual bitmap file location, which should be within the project directory or a specific resource folder.

Loading and Using the Bitmap:

LoadBitmap: In the WM\_CREATE case of the WndProc function, the line hBitmap = LoadBitmap(hInstance, TEXT("Bricks")) retrieves the BRICKS resource using its name. This function returns a handle for the loaded bitmap object.

GetObject: To obtain information about the bitmap, the program uses GetObject(hBitmap, sizeof(BITMAP), &bitmap) in the same case. This fills the bitmap structure with details like width, height, and color planes.

BitBlt: In the WM\_PAINT case, the program uses BitBlt to repeatedly copy the BRICKS bitmap onto the window surface. The loop iterates through the entire client area, tiling the bitmap horizontally and vertically to create the brick wall effect.

Key Points:

* BRICKS.BMP provides the source data for the brick pattern.
* LoadBitmap loads the bitmap resource into memory and returns a handle for accessing it.
* GetObject gathers information about the bitmap's dimensions and format.
* BitBlt efficiently copies the bitmap onto the window at various positions, creating the repeating pattern.

Additional Notes:

* The resource definition allows discarding the bitmap when not in use, potentially optimizing memory usage.
* The program ensures proper cleanup by deleting the bitmap handle in the WM\_DESTROY case.



MONOCHROME BITMAP CREATION

BRICKS2.C introduces a fascinating alternative to resource-based bitmap creation: directly defining monochrome bitmaps within your program. This section dives deep into this technique, explaining its advantages and the process involved.

The Power of Direct Monochrome Bitmap Creation:

For small, monochrome images, resource creation can be unnecessary overhead. BRICKS2.C showcases the power of directly defining the bit pattern within the code, offering several benefits:

* Simplified workflow: Eliminates the need for separate resource files and loading procedures.
* Greater control: You have direct access to every bit, allowing precise manipulation of the pixel pattern.
* Enhanced flexibility: Changes to the image can be easily implemented by modifying the bit sequence within your code.

Understanding the Monochrome Format:

As demonstrated in the example, monochrome bitmaps are essentially binary grids where:

* 0 represents black pixels.
* 1 represents white pixels.

By reading these bits from left to right and grouping them into 8-bit sequences, you create the byte data for the bitmap. Padding with zeros ensures an even number of bytes if the width isn't a multiple of 16.

Constructing the BITMAP Structure and Byte Array:

The example provides the code for setting up the BITMAP structure and the BYTE array containing the bitmap data:

* BITMAP structure: This defines key parameters like width, height, and byte width.
* BYTE array: This stores the actual bit sequence in byte format.

Creating the Bitmap:

Three approaches are presented for creating the bitmap object:

* CreateBitmapIndirect with bmBits: This method directly assigns the byte array to the bmBits field of the BITMAP structure and then creates the bitmap.
* CreateBitmapIndirect followed by SetBitmapBits: This separates the creation and data assignment steps, allowing flexibility for modifying the bit data later.
* CreateBitmap with all parameters: This one-line approach combines defining the BITMAP structure and assigning the byte data in a single step.

BRICKS2.C IN ACTION:

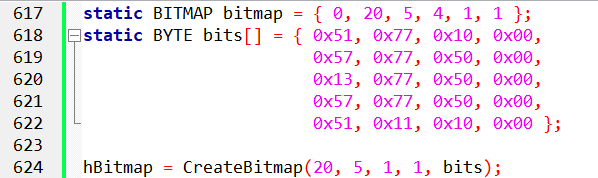
The BRICKS2 program leverages direct monochrome bitmap creation to display a repeating brick pattern without relying on a resource file. This highlights the potential for creating simple visuals directly within your code.

Key Takeaways:

* Direct monochrome bitmap creation offers a flexible and efficient alternative to resource-based approaches.
* Understanding the binary format and constructing the necessary structures empowers you to define your own bit patterns.
* BRICKS2.C demonstrates the practical application of this technique, showcasing its potential for creating custom visuals.

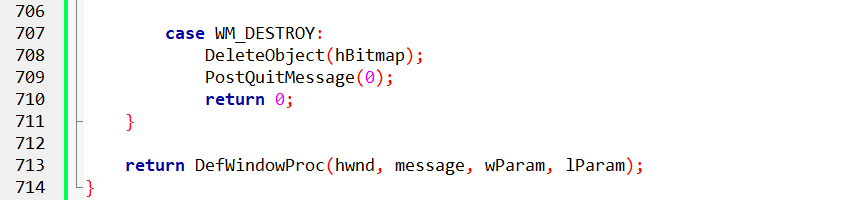
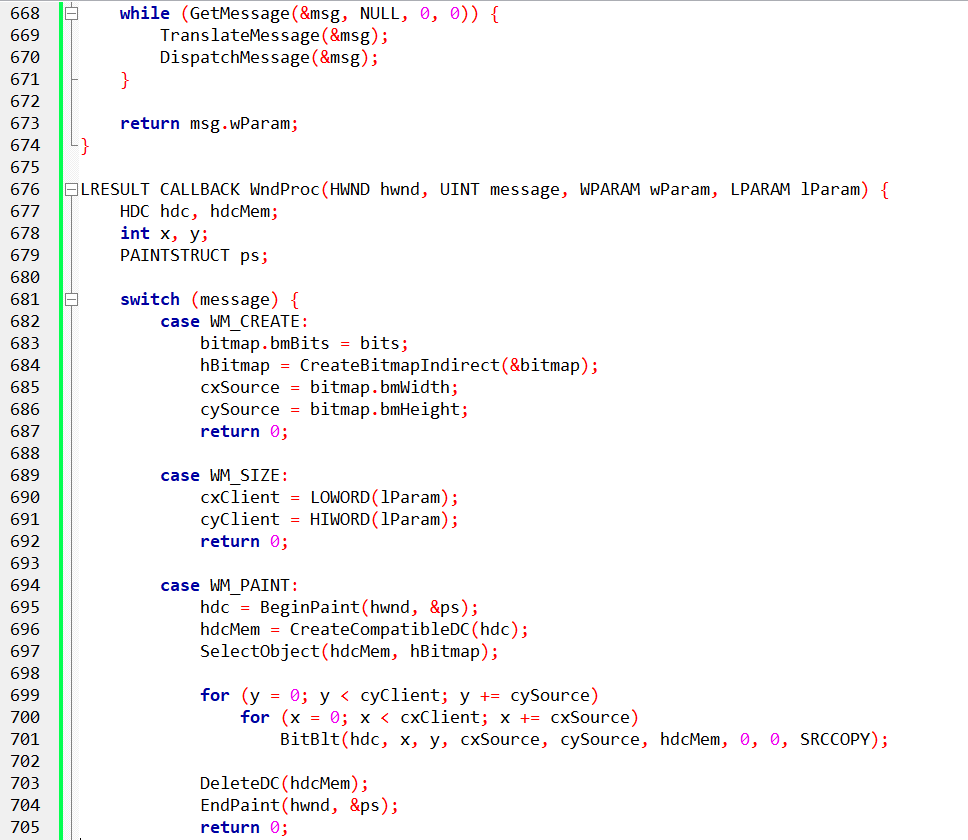
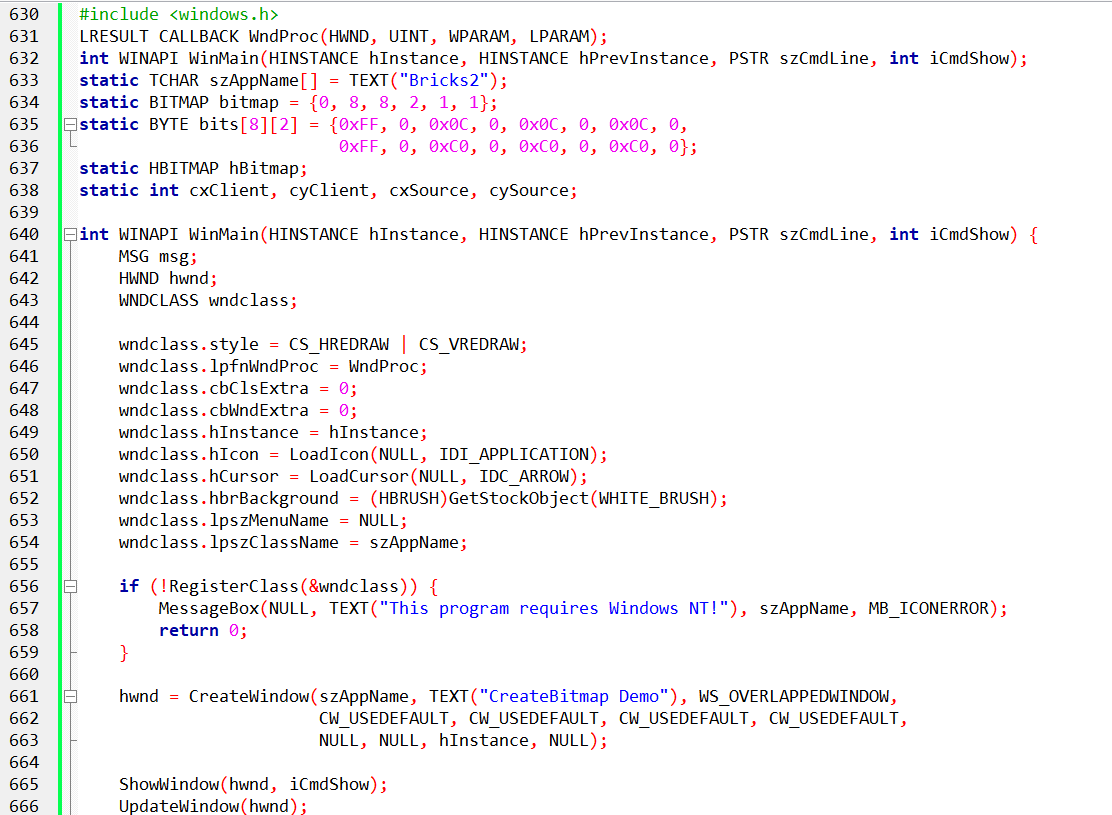
Remember:

* This approach is most effective for small, monochrome images.
* Experimentation and exploration are key to mastering this technique and pushing its creative boundaries.
* Feel free to ask any further questions you may have about monochrome bitmaps, their creation process, or specific aspects of BRICKS2.C! I'm here to support your journey into the world of efficient and direct bitmap manipulation.



This code sets up a BITMAP structure with the specified dimensions and initializes a BYTE array with the corresponding monochrome bits.

The CreateBitmap function is then used to create the monochrome bitmap directly with the provided dimensions and bit data.



BRICKS2.C: MASTERING MONOCHROME BITMAPS IN GDI

BRICKS2.C delves into the fascinating world of directly creating monochrome bitmaps within your program, offering an alternative to resource-based approaches. Let's dive deep into its code and explain the key aspects:

Defining the Monochrome Bitmap:

static BITMAP bitmap: This structure defines key parameters like width (8 pixels), height (8 pixels), byte width (2 bytes), and color planes (1 for monochrome).

static BYTE bits[8][2]: This two-dimensional array stores the actual bit pattern for the bitmap. Each byte represents 8 pixels, with 1 being white and 0 being black. The two-dimensional structure reflects the 8x8 grid of the bitmap.

Creating the GDI Bitmap Object:

WM\_CREATE initializes bitmap.bmBits to point to the bits array, essentially linking the bit data to the structure.

hBitmap = CreateBitmapIndirect(&bitmap): This function creates a GDI bitmap object based on the information provided in the bitmap structure.

Tiling the Bitmap on the Window:

WM\_PAINT iterates through the client area using nested loops.

For each iteration, BitBlt copies the entire bitmap from the memory DC (holding the "Bricks" pattern) onto the client DC at specific coordinates.

This effectively tiles the brick pattern across the window, creating the desired visual effect.

Clean Up and Handling Different Messages:

WM\_DESTROY deletes the hBitmap object to release resources.

Other messages like WM\_SIZE update internal variables based on the new client area dimensions.

Beyond Monochrome, Limitations and Next Steps:

The example highlights the simplicity of this approach for monochrome images.

However, attempting this with a color bitmap under different video modes will fail due to device dependencies.

The next chapter introduces Device-Independent Bitmaps (DIBs), which offer a more flexible and robust solution for handling color bitmaps in GDI.

Key Takeaways:

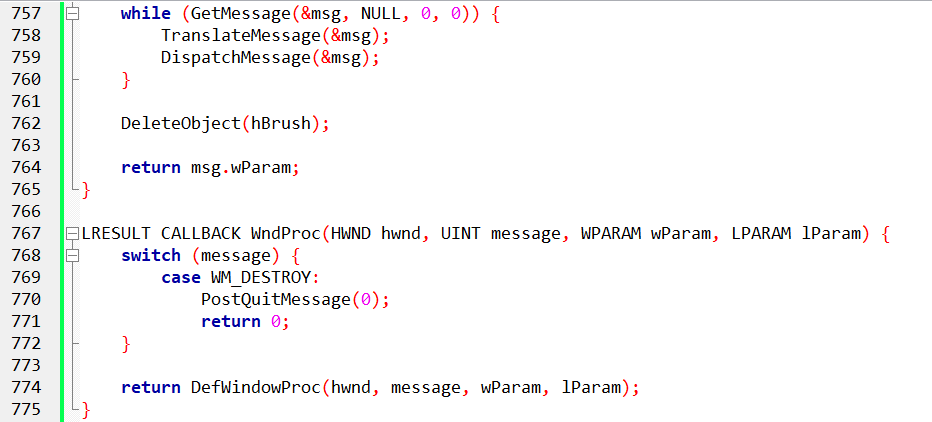
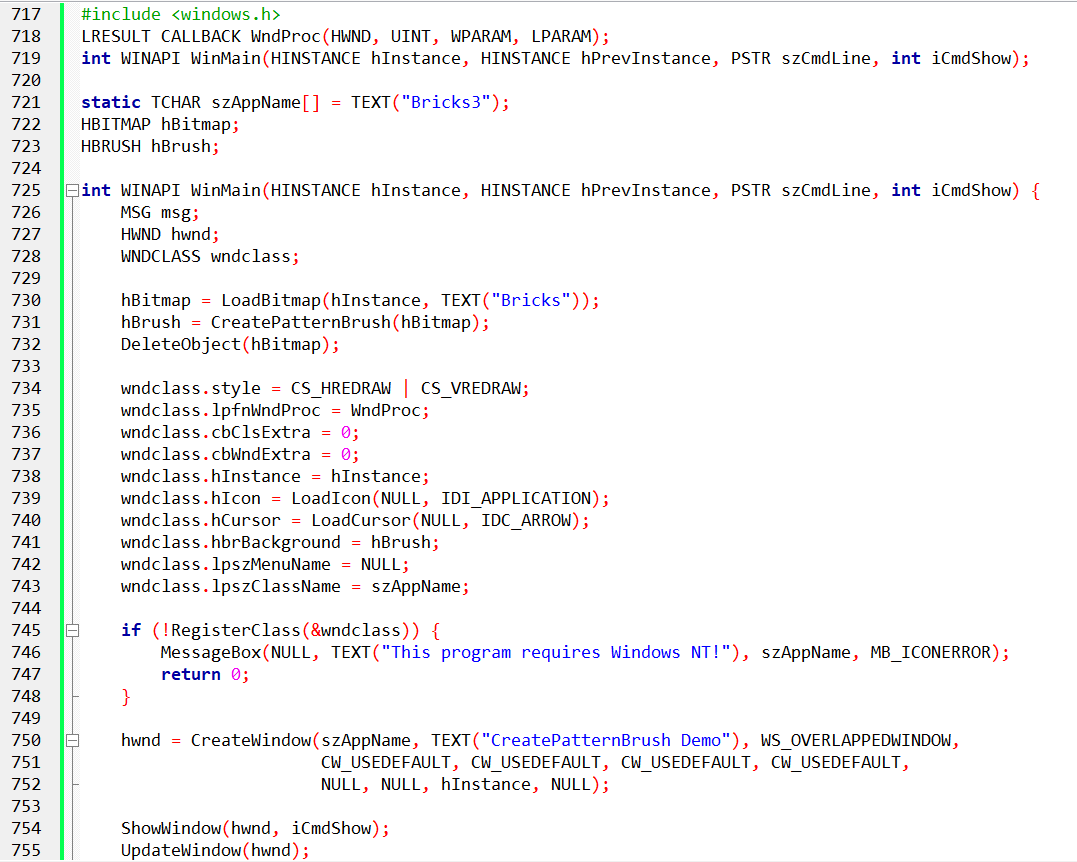
BRICKS2.C demonstrates direct creation of monochrome bitmaps within your code.

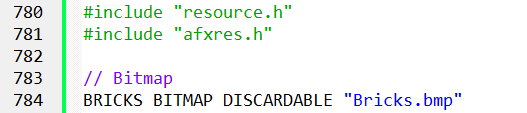
Understanding the bit pattern format and GDI functions like CreateBitmapIndirect empowers you to define custom visuals.

This approach is ideal for simple monochrome images, but limitations arise with color and device compatibility.

Remember:

* Experimentation is key to mastering this technique and exploring its potential.
* The next chapter delves into DIBs, offering a powerful tool for working with color bitmaps in a device-independent manner.





BRICKS3.C: UNVEILING THE POWER OF PATTERN BRUSHES

BRICKS3.C, while seemingly devoid of code at first glance, holds a hidden gem: the magic of pattern brushes. Let's delve deep into its essence and understand how it works:

Creating the Pattern Brush:

The program starts by loading the familiar "Bricks" bitmap using LoadBitmap.

The crucial line lies in creating a pattern brush using CreatePatternBrush with the loaded bitmap as its handle. This function essentially extracts the pattern from the bitmap and turns it into a reusable brush object.

Importantly, the program then deletes the original bitmap, as the pattern information is now captured within the brush.

Setting the Brush as Background:

The WNDCLASS structure for the window is defined, and its hbrBackground field is assigned the newly created pattern brush. This instructs the window to use the "Bricks" pattern as its default background.

The Magic Unfolds:

When the window is created, it automatically inherits the specified background brush. This paints the entire client area with the repeating brick pattern, eliminating the need for explicit drawing commands.

The window procedure handles the WM\_DESTROY message by deleting the pattern brush before exiting.

Understanding the Resource File:

The included excerpt from BRICKS3.RC defines a resource called "Bricks" of type BITMAP and specifies the file location as "Bricks.bmp". This ensures the bitmap image is readily available for loading within the program.

Key Takeaways:

* BRICKS3 demonstrates the power of pattern brushes, allowing you to utilize bitmaps as reusable background textures.
* This approach simplifies window drawing by eliminating the need for custom code to paint patterns.
* The resource file ensures convenient access to the bitmap image.

Remember:

* Pattern brushes provide a flexible and efficient way to create visually appealing backgrounds.
* Experimenting with different bitmap patterns can unlock creative possibilities for your applications.

DIVING DEEPER INTO BRICKS3

BRICKS3, while seemingly simple, holds hidden gems that unlock the power of brushes and bitmaps in GDI. Let's delve deeper:

BRICKS3: Reusing the Bricks Pattern as a Brush:

Instead of explicitly drawing the brick pattern, BRICKS3 uses the "Bricks.bmp" image as a source for creating a pattern brush with CreatePatternBrush.

This brush captures the repeating pattern from the bitmap and paints it as the window's background automatically.

The window procedure remains minimal, as the brush handles the repetitive drawing.

Understanding GDI Brushes:

Brushes are small bitmaps, typically 8x8 pixels, used for filling areas with textures or patterns.

You can create brushes from bitmaps using CreatePatternBrush or by setting the BS\_PATTERN style in a LOGBRUSH structure used with CreateBrushIndirect.

Windows 98 only utilizes the upper-left corner of the bitmap for brushes exceeding 8x8 pixels, while Windows NT uses the entire image.

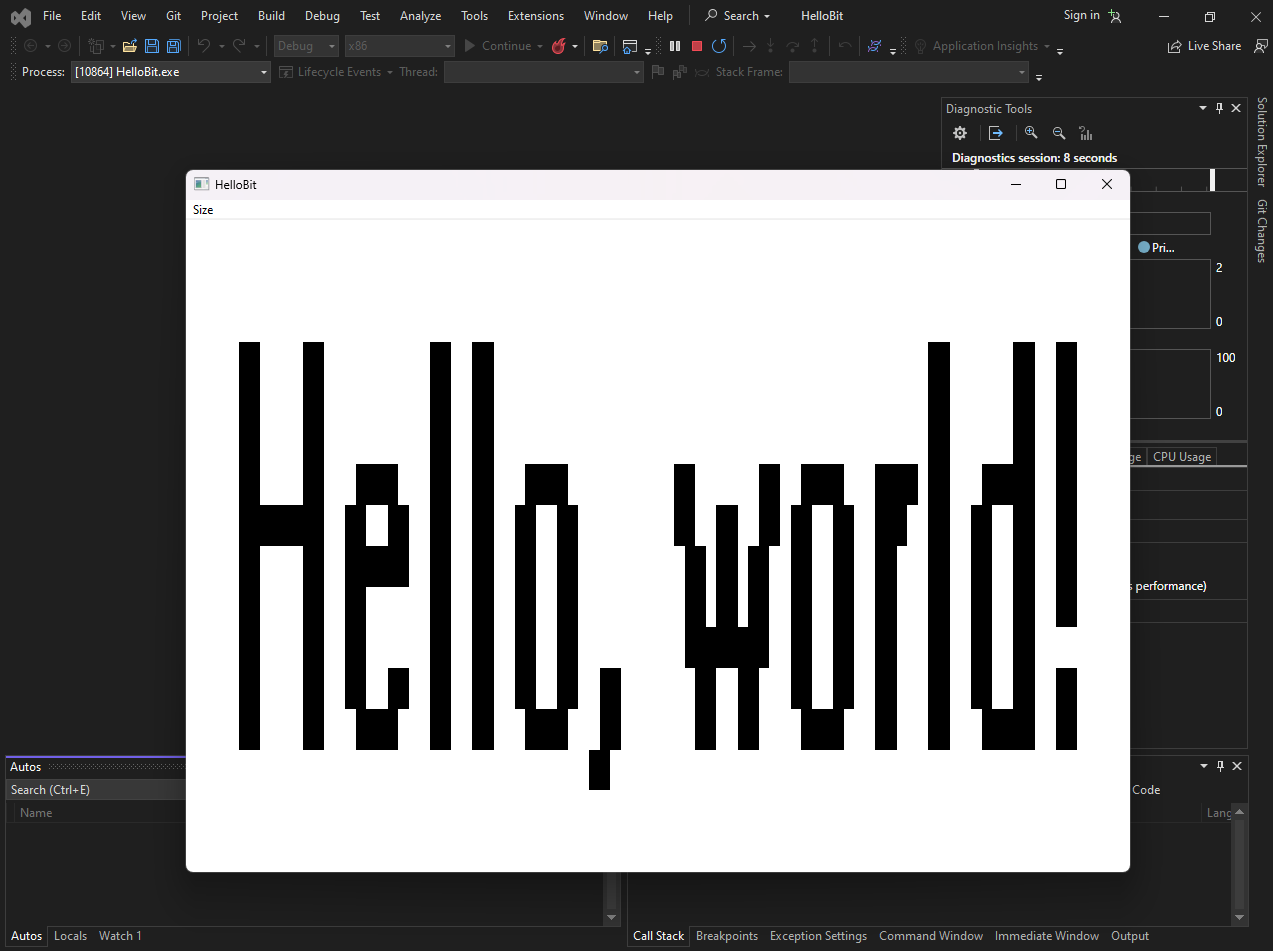
Memory Device Contexts and Bitmaps:

We've seen bitmaps used as sources for drawing on windows. But you can also use them as drawing surfaces!

By selecting a bitmap into a memory device context (MDC), you gain an off-screen canvas for drawing.

You can then use various GDI functions like LineTo, Ellipse, or even text rendering on the bitmap within the MDC.

The HELLOBIT program showcases this by displaying "Hello, world!" on a bitmap and then using BitBlt or StretchBlt to transfer it onto the client area.



Key Takeaways:

* Brushes offer a powerful way to reuse bitmaps as background textures for windows and other GDI objects.
* Memory device contexts paired with bitmaps provide a flexible canvas for off-screen drawing and manipulation.
* Understanding the differences in bitmap usage between Windows 98 and NT is crucial for optimal results.

Remember:

Experimentation is key to mastering brushes and bitmaps. Try different patterns, drawing techniques, and bitmap sizes to explore their potential.

Always delete GDI objects like brushes and bitmaps when finished to avoid resource leaks.

HELLOBIT: MASTERING BITMAPS AND DRAWING IN GDI

HELLOBIT, at first glance, might appear simple. But beneath its surface lies a fascinating interplay of bitmaps, memory device contexts (MDCs), and drawing techniques. Let's delve deeper:

Creating the Text Bitmap:

Instead of directly drawing text on the window, HELLOBIT creates a dedicated bitmap to hold the "Hello, world!" message.

GetTextExtentPoint32 determines the text size, which defines the bitmap dimensions for compatibility with the video display.

An MDC, also compatible with the display, is created to serve as the off-screen canvas.

TextOut efficiently renders the text onto the bitmap within the MDC, offering precise control over its appearance.

Memory Device Context Magic:

The created MDC and bitmap become key players throughout the program.

The MDC acts as an intermediary, allowing drawing operations on the bitmap without affecting the window directly.

This separation offers flexibility and avoids unnecessary redrawing of the entire window.

Displaying the Text:

Two menu options control how the text bitmap is displayed on the window:

Big: StretchBlt scales the bitmap to fill the entire client area, stretching the text proportionally.

Small: BitBlt copies the bitmap repeatedly across the client area, creating a tiled effect.

Both approaches showcase the versatility of bitmaps and GDI drawing functions.

Important Takeaways:

HELLOBIT demonstrates the power of creating bitmaps for specific drawing tasks.

MDCs provide a convenient off-screen canvas for manipulating and preparing visuals before displaying them on the window.

Understanding different drawing functions like StretchBlt and BitBlt empowers you to manipulate and display bitmaps in various ways.

Additional Points:

While stretching text can lead to pixelation and jagged edges, it serves as an example of bitmap scaling.

HELLOBIT cleans up properly by deleting the MDC and bitmap resources in the WM\_DESTROY message.

Remember:

Experiment with different drawing techniques and bitmap manipulation methods to unlock further creative possibilities.

Combining bitmaps with other GDI objects and functions can lead to stunning and dynamic visuals in your applications.

UNVEILING THE TECHNIQUE OF SHADOW BITMAPS: SKETCH TAKES THE STAGE

SKETCH, at first glance, might appear like a simple paint program. But beneath its surface lies a fascinating technique: the power of shadow bitmaps. Let's delve deeper into its secrets:

Building the Shadow Bitmap Canvas:

SKETCH, unlike previous programs, doesn't directly draw on the window. Instead, it creates a hidden "shadow bitmap" that acts as an off-screen canvas.

This bitmap is cleverly sized to accommodate the largest possible display resolution, ensuring compatibility across different systems.

Dedicated memory device contexts (MDCs) are created for both the window and the bitmap, enabling independent drawing operations.

Mouse Magic: Drawing and Erasing:

The program leverages mouse buttons to control drawing and erasing on the shadow bitmap.

Pressing the left mouse button selects a black pen, while the right button switches to a white pen, mimicking an eraser.

As you move the mouse, lines are drawn in the corresponding color onto both the shadow bitmap in the MDC and the actual window.

This real-time update creates the illusion of drawing directly on the window, while maintaining a separate bitmap record.

The Limits of Simplicity:

SKETCH showcases the efficiency of shadow bitmaps, but it also exposes its limitations.

Clearing the entire drawing requires restarting the program, as there's no dedicated "clear" function implemented.

This highlights the trade-off between simplicity and functionality in this basic implementation.

A Homage to History:

The SKETCH program, with its black and white drawing capabilities, pays homage to the early Apple Macintosh advertisements.

It reminds us of the evolution of paint programs and the journey from simple drawing tools to the advanced features we enjoy today.

Key Takeaways:

Shadow bitmaps offer a powerful technique for drawing on windows by utilizing off-screen memory device contexts.

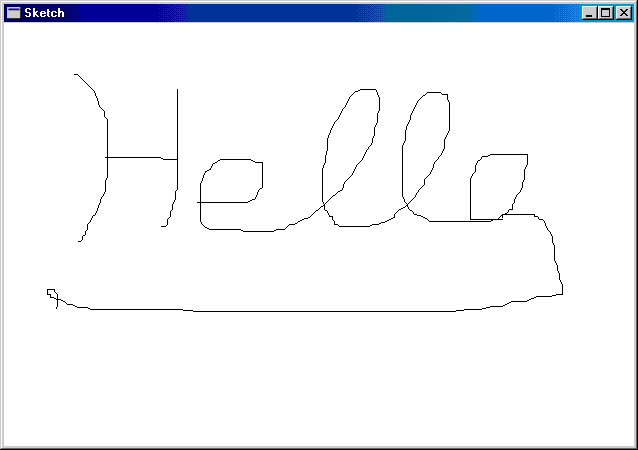
SKETCH demonstrates the effectiveness of this approach while showcasing its limitations and potential for further development.

Understanding the trade-offs between simplicity and functionality allows us to appreciate the advancements in paint programs.

Remember:

Experimenting with different drawing techniques and bitmap manipulation methods can lead to exciting possibilities in your applications.

Combining shadow bitmaps with other GDI functions and objects can unlock even greater visual capabilities.



SHADOW BITMAP SIZE IN SKETCH: A BALANCING ACT BETWEEN PERFORMANCE AND FUTURE-PROOFING

SKETCH's shadow bitmap size plays a crucial role in its functionality and efficiency. Let's dive deeper into this balancing act:

Maximizing the Canvas:

Ideally, the shadow bitmap should be large enough to accommodate the entire client area of a maximized window.

This ensures the program can handle any potential window size changes without limitations.

GetSystemMetrics vs. Brute Force:

SKETCH initially attempts to calculate the maximum window size using GetSystemMetrics information. This provides a basic estimate, but doesn't account for future display changes.

The program then employs a "brute force" approach using EnumDisplaySettings to iterate through all available video display modes.

This ensures the shadow bitmap can handle even larger resolutions than the current one, potentially exceeding the current display size by four times or more.

Memory Considerations and Error Handling:

Creating such a large bitmap can be memory-intensive, potentially requiring several megabytes.

SKETCH checks for existing bitmaps to avoid redundant allocation and throws an error if one isn't available, preventing crashes due to memory limitations.

Mouse Capture and Drawing Logic:

SKETCH captures the mouse to track its movements within and outside the window during drawing.

This allows lines to be drawn seamlessly even if the mouse exits the window boundaries while holding the button down.

Implementing more complex drawing logic might necessitate separating the drawing code into two functions, one for each device context (window and bitmap) to maintain code clarity and efficiency.

Window Size Experiment and Shadow Bitmap Persistence:

When drawing with a smaller SKETCH window and then maximizing it, the shadow bitmap surprisingly includes the drawing done outside the original window due to mouse capture.

This highlights the power and potential pitfalls of shadow bitmaps. While it provides a persistent drawing surface, it can also capture unintended actions outside the visible window area.

Key Takeaways:

Choosing the appropriate shadow bitmap size involves balancing future-proofing against memory limitations.

SKETCH's approach offers a comprehensive solution, but alternative methods like dynamic resizing or partial updates could be explored for efficiency.

Understanding the implications of mouse capture and bitmap persistence is crucial for designing intuitive and predictable drawing experiences.

Remember:

Experiment with different shadow bitmap sizing strategies and drawing logic to find the best balance for your specific needs.

Consider factors like memory usage, performance, and user expectations when making decisions about bitmap size and interaction.

BEYOND TEXT: UNLEASHING THE POWER OF BITMAPS IN MENUS

While traditional menus rely solely on text, GRAFMENU pushes the boundaries by incorporating bitmaps, opening doors to a more visual and informative menu experience. Let's delve deeper into its innovative approach:

From Icons to Expressive Elements:

GRAFMENU discards the typical folder, paperclip, and trash can icons, replacing them with expressive bitmaps.

These bitmaps transcend mere decoration; they visually represent menu options, enhancing user understanding and navigation.

Font Powerhouse:

The "FONT" menu showcases the potential of bitmaps in displaying different fonts and sizes.

Each option in the popup menu utilizes a bitmap with the actual font applied, providing a clear preview of what the user selects.

Beyond Icons: Hatch Patterns, Colors, and More:

Imagine menu items with line width options visualized through bitmaps with varying line thicknesses.

Hatch patterns for fill styles could be represented directly, offering users an immediate visual reference.

Colors can come alive with bitmaps showcasing different palettes or custom color combinations.

The Power of Memory Device Contexts:

GRAFMENU demonstrates how memory device contexts (MDCs) empower bitmap creation within the program.

These MDCs act as off-screen canvases, allowing you to design and manipulate bitmaps without affecting the actual menu appearance.

Visual Hierarchy and Storytelling:

The "Help" bitmap in the system menu adds a subtle touch of humor and visual storytelling, subtly hinting at the potential frustration of new users.

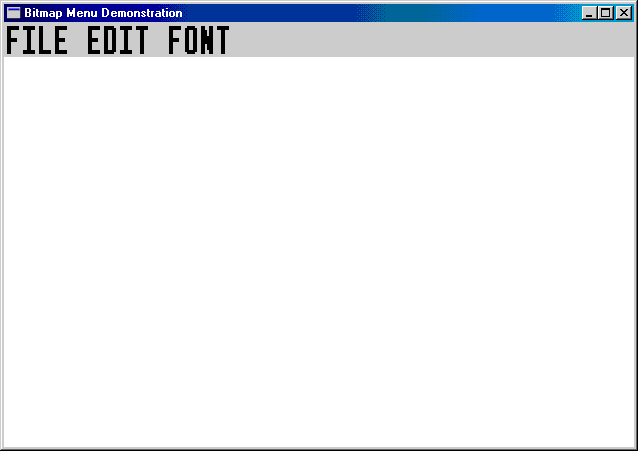
This demonstrates how bitmaps can go beyond mere functionality and inject personality into the user interface.

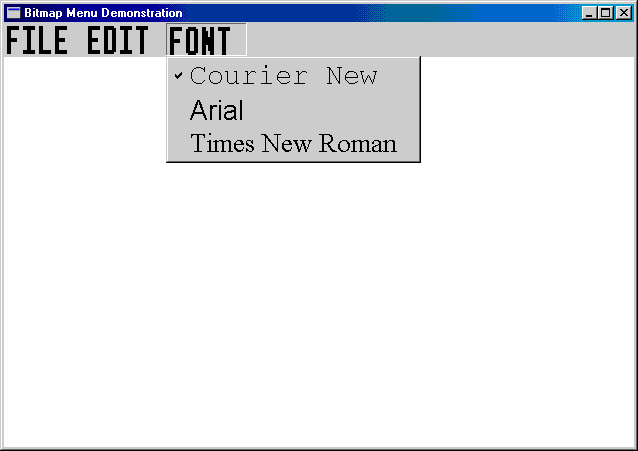
Key Takeaways:

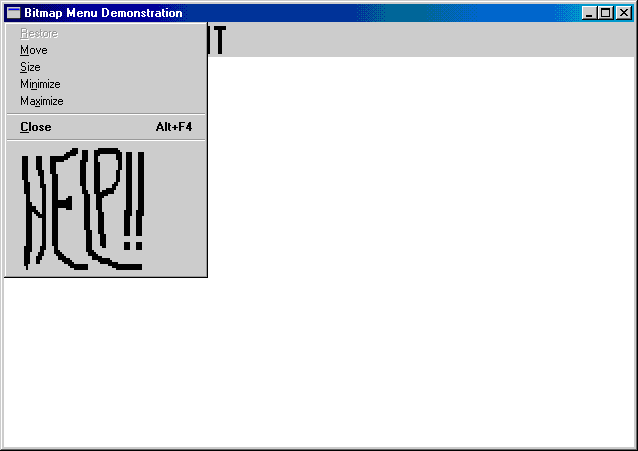
Bitmaps offer a powerful way to enrich menu items beyond simple text, enhancing user understanding and engagement.

MDCs provide a flexible platform for creating and manipulating bitmaps within your program without affecting the actual menu display.

Experimenting with different bitmap designs, including fonts, line styles, colors, and even humor, can unlock new possibilities for intuitive and expressive menus.







Remember:

While bitmaps offer great flexibility, ensure their design is clear, consistent, and aligns with your program's overall aesthetic.

Accessibility considerations are crucial; ensure bitmap-based menus are accessible for users with visual impairments.

GRAFMENU: UNPACKING THE PROGRAM'S FUNCTIONALITY

GRAFMENU is a program that demonstrates the use of bitmaps in menus, going beyond the typical text-based options. Here's an in-depth explanation of its code and functionalities:

Menu Structure:

The program utilizes two resource files: GRAFMENU.RC defines the menu structure and RESOURCE.H contains resource IDs for menu items and bitmaps.

Two separate menus are defined: MENUFILE for file-related actions and MENUEDIT for editing functionalities.

Bitmap Magic:

GRAFMENU leverages bitmaps to visually represent menu options instead of text. This adds a layer of visual clarity and intuitiveness.

Four bitmaps are used:

* BitmapFile for the "File" menu.
* BitmapEdit for the "Edit" menu.
* BitmapFont for the "Font" submenu label.
* BitmapHelp for the "Help" menu item in the system menu.

Each bitmap is loaded from a resource file and stretched to fit the display resolution, ensuring consistent visual quality across different systems.

Font Powerhouse:

The "Font" submenu showcases the power of bitmaps in representing different fonts.

Three bitmaps are dynamically generated using memory device contexts (MDCs):

One for each font type (Courier New, Arial, Times New Roman) with the actual font name drawn on it.

This allows users to visually see the font before selecting it, enhancing user experience and clarity.

System Menu Integration:

GRAFMENU adds a custom "Help" item to the system menu using a bitmap.

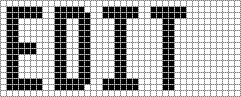
This adds a touch of personality and visual interest to the program.

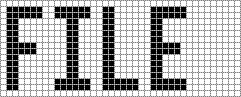
Functionality and Event Handling:

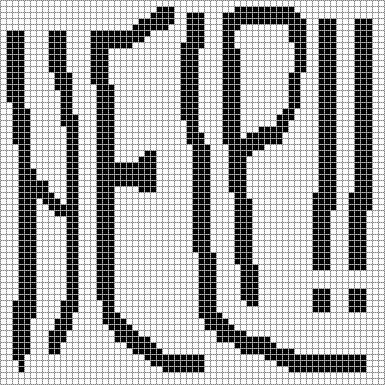
The program defines various menu item IDs and handles user interactions through the WndProc function.

Selecting menu items triggers specific actions like displaying the "Help" message box or checking/unchecking font options.

When the program closes, all bitmaps used in the menus are properly deleted to avoid memory leaks.







Key Takeaways:

GRAFMENU demonstrates a creative and innovative approach to menu design using bitmaps.

It highlights the potential of bitmaps to enhance user experience by providing visual cues and improving clarity.

The program's code provides valuable insights into bitmap manipulation, resource handling, and menu interaction techniques.

Further Exploration:

Experiment with different bitmap designs and styles to create unique and visually appealing menus.

Consider implementing dynamic bitmap generation based on menu item states or user preferences.

Explore incorporating sound effects or animations for an even more engaging user experience.

Remember, GRAFMENU is just one example of how bitmaps can be used to enhance menu functionality. Be creative, experiment, and push the boundaries of menu design!

DEEP DIVE INTO GRAFMENU'S BITMAP MAGIC: UNPACKING THE CODE

GRAFMENU leverages bitmaps to create visually captivating and informative menus, deviating from the typical text-based approach. Here's a detailed breakdown of the code and its functionalities:

1. Menu Creation and Structure:

CreateMyMenu: This function assembles the main menu by:

* Creating an empty menu with CreateMenu.
* Loading the "MenuFile" popup for File options using LoadMenu.
* Stretching and loading the "BitmapFile" bitmap for the File menu label with StretchBitmap and LoadBitmap.
* Appending the File menu bitmap and popup to the main menu with AppendMenu.

Similarly, the Edit menu is created with its popup and bitmap.

2. Dynamic Font Bitmaps:

GetBitmapFont: This function generates a bitmap displaying a specific font name.

* It takes an integer parameter (0, 1, or 2) corresponding to the desired font: "Courier New", "Arial", or "Times New Roman".
* It uses CreateIC and GetTextMetrics to obtain the screen's device context and system font size.
* A memory device context compatible with the screen is created with CreateCompatibleDC.
* A logical font twice the system font size with the chosen facename is created using CreateFont.
* The SelectObject function selects this font into the memory device context, saving the system font handle.
* GetTextExtentPoint32 retrieves the text dimensions, and CreateBitmap creates a bitmap based on those dimensions.
* The text is written to the bitmap using TextOut.
* Finally, the system font is reselected, the temporary font deleted, and both device contexts released.

3. Scaling Bitmaps for Different Resolutions:

StretchBitmap: This function scales a bitmap to fit the current display resolution.

* It retrieves the screen's device context and system font metrics.
* Two memory device contexts are created compatible with the screen.
* The provided bitmap handle is obtained, and its dimensions are retrieved with GetObject.
* A new BITMAP structure with adjusted dimensions based on the system font size is created.
* A new bitmap with the adjusted dimensions is created using CreateBitmapIndirect.
* Both bitmaps are selected into their respective memory device contexts.
* StretchBlt is used to copy and stretch the original bitmap onto the new one.
* Finally, the temporary bitmap and device contexts are cleaned up.

4. Integrating Bitmaps into Menus:

CreateMyMenu uses StretchBitmap and GetBitmapFont to create bitmaps for the Font submenu options.

* These font bitmaps and the "BitmapFont" label are appended to the Edit menu popup using AppendMenu.
* The WndProc function handles menu interactions, responding to selections and checking/unchecking font options.

Key Takeaways:

GRAFMENU demonstrates:

* Creating bitmaps from scratch and loading them from resources.
* Scaling bitmaps to fit different display resolutions.
* Integrating bitmaps into menus for visual clarity and user experience enhancement.

Further Exploration:

* Experiment with generating different bitmap styles and designs for menu items.
* Implement dynamic bitmap generation based on menu item states or user preferences.
* Combine bitmaps with sound effects or animations for a more interactive menu experience.

Remember, GRAFMENU is just one example of using bitmaps for innovative menu design. Be creative, explore the possibilities, and push the boundaries of user interface design with the power of bitmaps!

Bonus:

The provided codebox can be incorporated into the explanation to illustrate specific functions like GetBitmapFont or StretchBitmap. This would further enhance the understanding of how each piece of code contributes to the overall functionality of GRAFMENU.

GRAFMENU'S MENU IN DEPTH

GRAFMENU's menu design goes beyond the usual text-based options, employing captivating bitmaps for both visual appeal and clarity. Let's delve deeper into how this is achieved:

Building the Edit Menu:

Loading Popup Menu: Similar to the File menu, LoadMenu retrieves the pre-defined "MenuEdit" popup from the resource file.

Stretching Edit Label: The "BitmapEdit" bitmap is loaded and then scaled using StretchBitmap to fit the current display resolution.

Adding to Main Menu: AppendMenu combines the stretched bitmap and the edit popup into a single menu item for the main menu.

Constructing the Font Submenu:

Creating Empty Popup: A blank popup menu is created with CreateMenu to hold the font options.

Generating Font Bitmaps: A loop iterates through three font options (Courier New, Arial, Times New Roman). For each, GetBitmapFont generates a bitmap displaying the corresponding font name.

Adding Font Options: Each font bitmap is appended to the popup menu as a separate item using AppendMenu. The provided IDM\_FONT\_COUR + i identifies each option uniquely.

Completing the Window Menu:

Stretching Font Label: Similar to the Edit menu, the "BitmapFont" label is stretched to the appropriate size using StretchBitmap.

Adding Font Submenu: The stretched font label and the font popup are combined into another menu item for the main menu with AppendMenu.

Setting Menu: Finally, SetMenu assigns the completed window menu to the program window.

Enhancing the System Menu:

GetSystemMenu retrieves the existing system menu handle.

The "BitmapHelp" image is loaded and scaled using StretchBitmap for optimal display.

A separator and the stretched help bitmap are appended to the system menu using AppendMenu, providing easy access to help.

Cleaning Up and Miscellaneous Notes:

GRAFMENU dedicates a function to properly disposing of all bitmaps before program termination.

Some additional points to consider:

* In top-level menus, Windows automatically adjusts the menu bar height to accommodate the tallest bitmap.
* Checkmarks can be used with bitmapped popup items, but they appear in the standard size. Customized checkmarks and SetMenuItemBitmaps offer alternative solutions.
* "Owner-draw" menus provide another approach for displaying non-text content or custom fonts.
* Keyboard navigation for bitmapped menus requires handling the WM\_MENUCHAR message to interpret user key presses and associate them with specific menu items.

Conclusion:

GRAFMENU demonstrates the potential of bitmaps in menu design, offering visual clarity and enhancing user experience. By understanding the code breakdown and the underlying principles, you can gain valuable insights into creating your own innovative and captivating menu interfaces.

DEEP DIVE INTO BITMASK: MASKING BITMAPS FOR ELLIPTICAL SHAPES

BITMASK showcases an ingenious technique for displaying non-rectangular bitmap images, specifically how to turn a rectangular image into an ellipse. Let's dive deeper into the code and understand the magic behind this visual transformation:

1. Setting the Stage:

The WndProc function handles all window messages.

Global variables store handles for the original image (hBitmapImag), mask bitmap (hBitmapMask), and WinMain's instance (hInstance).

cxClient and cyClient hold the client area dimensions, while cxBitmap and cyBitmap store the image dimensions obtained from GetObject.

2. Image Preparation:

WM\_CREATE loads the image using LoadBitmap and retrieves its size with GetObject.

A compatible memory DC (hdcMemImag) is created and the image is selected into it.

3. Mask Creation and Transformation:

A monochrome mask bitmap with the same dimensions as the image is created with CreateBitmap.

Another compatible memory DC (hdcMemMask) is created and the mask is selected into it.

The mask is first filled black using SelectObject and Rectangle, then a white ellipse is drawn using SelectObject and Ellipse.

4. Masking the Image:

The BitBlt function performs the magic. It uses the SRCAND raster operation, which combines the source (image) and destination (mask) bitmaps pixel-by-pixel.

White pixels in the mask allow the corresponding image pixels to show, while black pixels hide them, creating the desired elliptical shape.

5. Painting the Elliptical Image:

WM\_PAINT updates the display.

Compatible memory DCs are created for both the image and mask bitmaps.

The image is centered in the client area using calculations involving cxClient, cyClient, cxBitmap, and cyBitmap.

Two BitBlt operations are performed:

The first uses the mask with the 0x220326 raster operation to erase unwanted areas and create the elliptical outline.

The second uses the SRCPAINT operation to paint the masked image within the ellipse.

Finally, the memory DCs are cleaned up.

6. Cleanup and Completion:

WM\_DESTROY releases the image and mask bitmaps before exiting the program.

Key Takeaways:

BITMASK demonstrates how a mask bitmap and the BitBlt function with specific raster operations can create non-rectangular image displays.

This technique allows you to display bitmaps in various shapes, not just rectangles, enhancing visual interest and design flexibility.

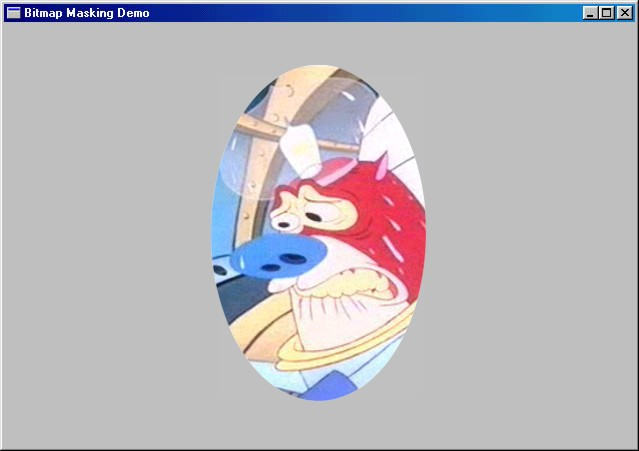
Understanding the code breakdown and the principles behind masking empowers you to implement similar techniques in your own applications.

Further Exploration:

Experiment with different mask shapes and bitmap content to create unique visual effects.

Explore other raster operations and their combinations for more advanced masking possibilities.

Combine masking with other graphical techniques like transparency and alpha blending for even richer visual experiences.



DIVING DEEPER INTO BITMASK'S NOTES:

1. Versatile Mask Potential:

BITMASK isn't limited to Matthew's picture; it can display any image using the same masking technique.

The MATTHEW.BMP file simply acts as a placeholder for any bitmap you want to transform.

2. Gray Background Verification:

The light gray background serves as a visual confirmation that the masking is working correctly.

If the image remained entirely white after masking, it would be difficult to distinguish between proper masking and simply painting the image white.

3. Bitwise Magic in Masking:

The SRCAND operation in WM\_CREATE performs a bitwise AND between the mask (1s for white, 0s for black) and the image (1s for set pixels, 0s for unset pixels).

This operation retains image pixels where the mask is white (1) and sets them to black where the mask is black (0), effectively creating the elliptical shape.

4. Elliptical Window Outline:

The first WM\_PAINT BitBlt uses a custom raster operation (D & ~S) to achieve the black ellipse outline.

This operation inverts the mask (black ellipse becomes white, white background becomes black) and then performs an AND with the window surface.

This AND operation sets the window pixels to black wherever the inverted mask is black, creating the desired elliptical outline.

5. Image Display within the Ellipse:

The second WM\_PAINT BitBlt uses SRCPAINT to overlay the masked image onto the window.

SRCPAINT performs a bitwise OR, leaving the window background untouched and copying the image pixels within the ellipse onto the window surface.

Additional Notes:

Creating complex masks, like ones that completely erase the original background, may require manual manipulation in a paint program.

Experimenting with different mask designs and raster operations unlocks further possibilities for creative image manipulation within the BITMASK framework.

Remember:

This explanation avoids technical jargon and focuses on the underlying concepts and logic of the masking process.

The goal is to provide a clear understanding of how BITMASK achieves its visual transformation without getting bogged down in the specifics of the code.

DEEP DIVE INTO BOUNCE: BOUNCING BALL ANIMATION EXPLAINED

BOUNCE showcases a basic animation technique using a bitmap and a timer to bring a ball to life on the screen. Let's delve deeper into the code and understand how it works:

1. Setting the Stage:

The program defines constants and global variables like hBitmap for the bitmap handle and xCenter, yCenter for the ball's position.

WM\_CREATE retrieves device pixel aspect ratios for scaling the ball and sets a timer to trigger animation updates every 50 milliseconds.

2. Ball Construction and Scaling:

WM\_SIZE recalculates the client area dimensions and adjusts the ball size accordingly.

It scales the ball diameter to 1/16th of the smaller dimension (height or width) and adds a margin for smooth movement.

A memory DC compatible with the video display is created for drawing the ball bitmap.

3. Drawing the Ball with a Hatch and Margin:

A compatible bitmap with the desired size (cxTotal, cyTotal) is created in memory.

The entire bitmap is filled white using Rectangle to erase any previous ball remnants.

A diagonally hatched brush is selected and used to draw the ball as an ellipse within the bitmap, leaving margins around its edges.

4. Animation Loop and Ball Movement:

The timer triggers WM\_TIMER at regular intervals to update the ball's position.

The program uses GetDC and CreateCompatibleDC to access memory DCs for drawing and blitting.

A BitBlt operation with SRCCOPY copies the entire ball bitmap onto the window at the new center coordinates (xCenter, yCenter).

The ball's center coordinates are incremented by cxMove and cyMove to simulate movement.

Boundary checks ensure the ball bounces off the window edges by reversing its direction when it reaches the limits.

5. Cleaning Up and Ending the Animation:

WM\_DESTROY releases the bitmap handle and kills the timer before exiting the program.

Points to Ponder:

This implementation is a basic example; advanced animation techniques involve other ROP codes for blending and effects.

The Windows palette and AnimatePalette function offer additional animation possibilities.

For more sophisticated animation, exploring DirectX or other animation libraries is recommended.

Key Takeaways:

BOUNCE demonstrates how a simple bitmap and timer combination can create basic animation.

Understanding the code logic behind drawing, movement, and collision detection provides a foundation for exploring more complex animation techniques.

By experimenting with different ROP codes, palette manipulation, and advanced libraries, you can create more visually engaging and dynamic animations in your Windows applications.

DEEP DIVE INTO SCRAMBLE'S NOTES: SCRAMBLED SECRETS EXPLAINED

1. Rude But Revealing:

SCRAMBLE may seem mischievous, but it showcases advanced techniques for manipulating screen content using memory device contexts (DCs).

2. Locking the Screen:

LockWindowUpdate prevents other programs from updating the screen while SCRAMBLE works its magic, ensuring smooth scrambling.

GetDCEx with DCX\_LOCKWINDOWUPDATE acquires a device context for the entire screen, giving SCRAMBLE direct access.

3. Dimension Downscaling:

Dividing the full screen size by 10 creates smaller rectangles for swapping, preventing the entire screen from flickering at once.

This also reduces the computational burden, making the scrambling smoother.

4. Memory DC as Temporary Storage:

A compatible memory DC acts as a temporary holding space for the swapped rectangles.

This avoids rewriting the entire screen directly, making the process more efficient.

5. The Choreographed Swap:

Three BitBlt operations orchestrate the rectangle exchange:

* Copy to Memory: The first BitBlt copies the first rectangle to the memory DC.
* Overlay Replacement: The second BitBlt overwrites the first rectangle area with the second rectangle from the screen.
* Memory to Screen: The third BitBlt copies the swapped rectangle from the memory DC back onto the second rectangle area.

6. Unscramble and Unlock:

SCRAMBLE remembers the swapping order and reverses the process, restoring the original screen before unlocking it.

This ensures you get your unscrambled screen back after the visual chaos!

7. Beyond Swapping:

Memory DCs can also copy specific portions of bitmaps. The code snippet demonstrates extracting a bitmap's upper left quadrant.

This technique opens doors for more creative bitmap manipulations and image composition within your Windows applications.



Remember:

SCRAMBLE is a playful example, but it highlights powerful techniques for manipulating screen content and bitmaps using memory DCs.

Understanding the core principles behind these operations empowers you to explore more sophisticated visual effects and image processing tasks in your Windows development projects.

DEEP DIVE INTO BLOWUP: CAPTURING AND MAGNIFYING SCREEN REGIONS

BLOWUP, unlike the mischievous SCRAMBLE, empowers you to capture and magnify specific areas of your screen. Let's dissect its functionalities:

1. Capturing the Rectangle:

BLOWUP uses LockWindowUpdate to prevent screen flickering during capture.

GetDCEx with DCX\_CACHE and DCX\_LOCKWINDOWUPDATE acquires a device context (DC) for the entire screen, allowing direct access for drawing.

2. User-Driven Selection:

Left-clicking enables mouse capture, transforming your cursor into a crosshair for precise selection.

Right-clicking initiates the capture process and displays the selected rectangle with an inverted color effect.

3. Dynamic Block Update:

While dragging the mouse, InvertBlock redraws the selected area with the inverted color effect, providing real-time feedback.

4. Image Capture and Stretch:

Releasing the left mouse button triggers image capture. BLOWUP uses BitBlt to copy the selected area from the screen DC to a newly created bitmap.

GetClientRect retrieves the client area dimensions of the BLOWUP window.

StretchBlt scales the captured bitmap to fit the client area, enabling magnification or reduction depending on the selection size.

5. Menu and Clipboard Integration:

BLOWUP provides a menu for editing the captured image.

CopyBitmap creates a duplicate of the captured bitmap for clipboard operations.

OpenClipboard, EmptyClipboard, and SetClipboardData enable copying the image to the clipboard for sharing with other applications.

BLOWUP also allows pasting images from the clipboard, replacing its existing captured image.

6. Memory Management and Cleanup:

BLOWUP meticulously deletes temporary bitmaps and DCs to prevent memory leaks.

DeleteObject frees resources associated with bitmaps and DCs when they're no longer needed.

PostQuitMessage sends a termination message to close the program gracefully.

Key Takeaways:

BLOWUP demonstrates advanced techniques for capturing screen regions and manipulating bitmaps.

Understanding its code logic empowers you to build applications that interact with screen content and perform image manipulations.

BLOWUP's menu and clipboard integration showcase its versatility for practical use cases like image sharing and editing.

Further Exploration:

Experiment with different StretchBlt modes to achieve various scaling effects.

Explore other bitmap manipulation functions like MaskBlt and TransparentBlt for advanced image compositing.

Combine BLOWUP's capture capabilities with image processing libraries for tasks like object detection or image analysis.