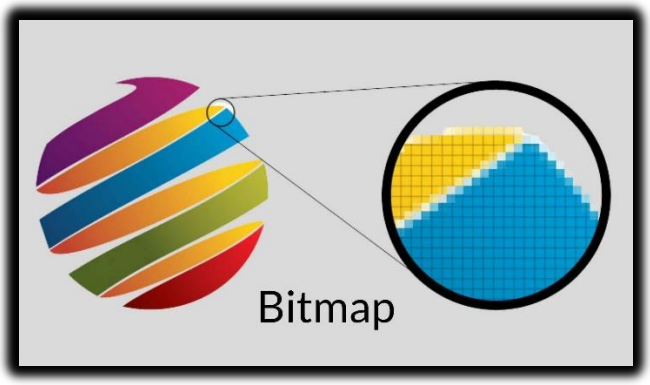
CHAPTER 14 BITMAPS AND BITBLTS

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.

BITS PER PIXEL	NUMBER OF COLORS POSSIBLE
1 Bit	2 colors **(Black and White)**
8 Bits	256 colors
24 Bits	16.7 Million colors **(True Color)**

BITMAPS VS. METAFILES: TWO APPROACHES TO PICTORIAL DATA

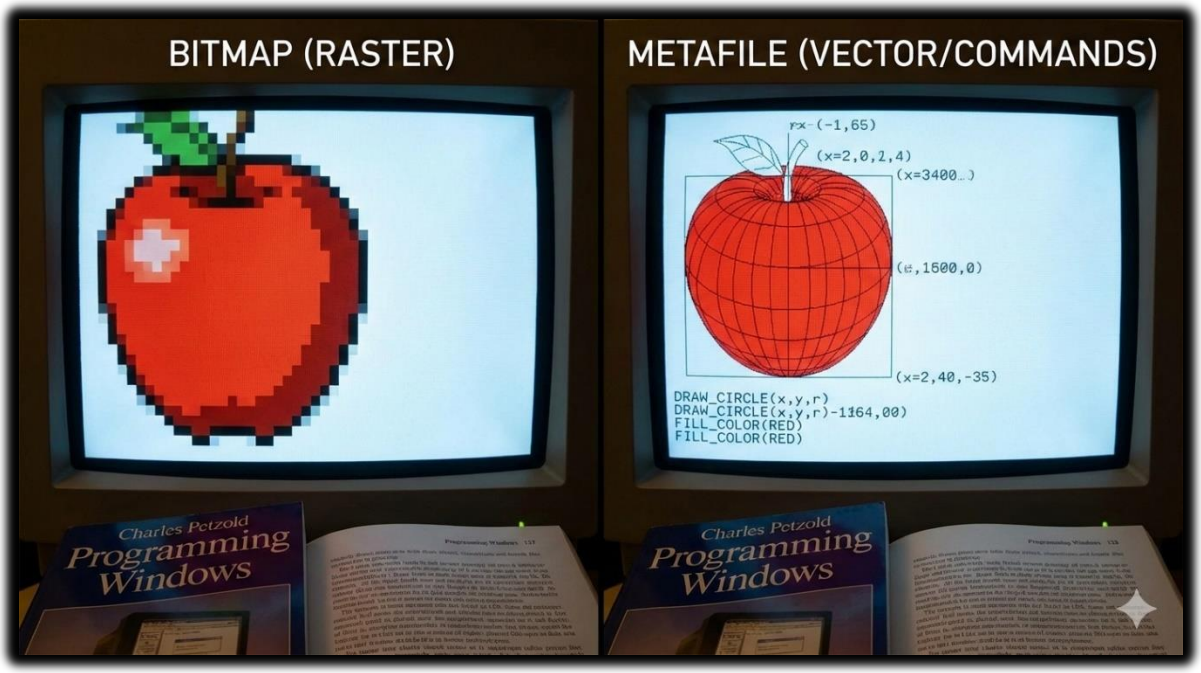
Windows uses two different ways to store and display pictures. You can think of it as the difference between a **photo** and a **drawing lesson**.

[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.

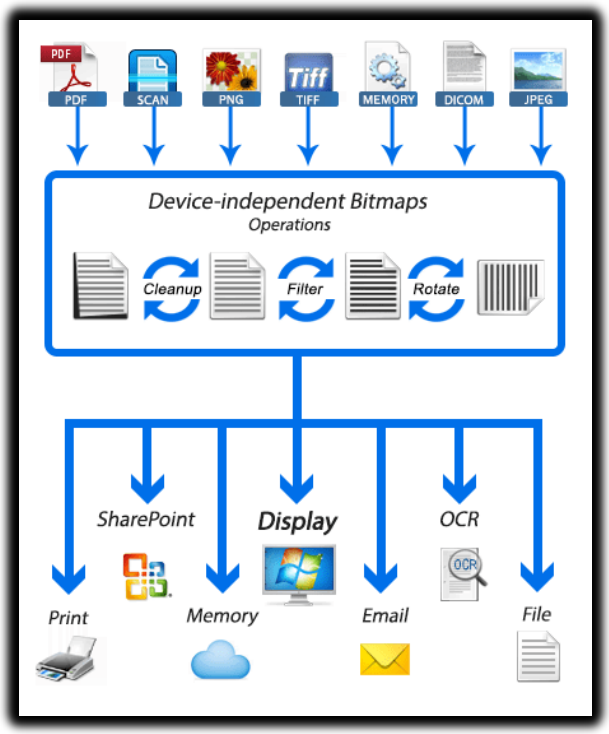
GDI Bitmaps (The "Old School" Way)

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:

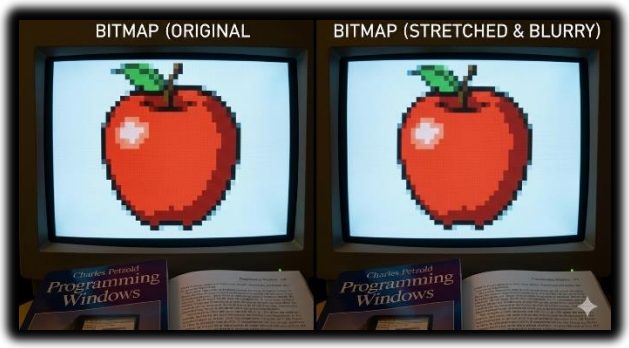
- **Manipulation:** Creating, loading (BMP/ICO), and editing bitmaps.
- **BitBlt:** Moving and copying pixel blocks.
- **Composition:** Combining multiple images and using transparency.
- **Performance:** Learning how to handle memory efficiently using these older, faster tools.

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.

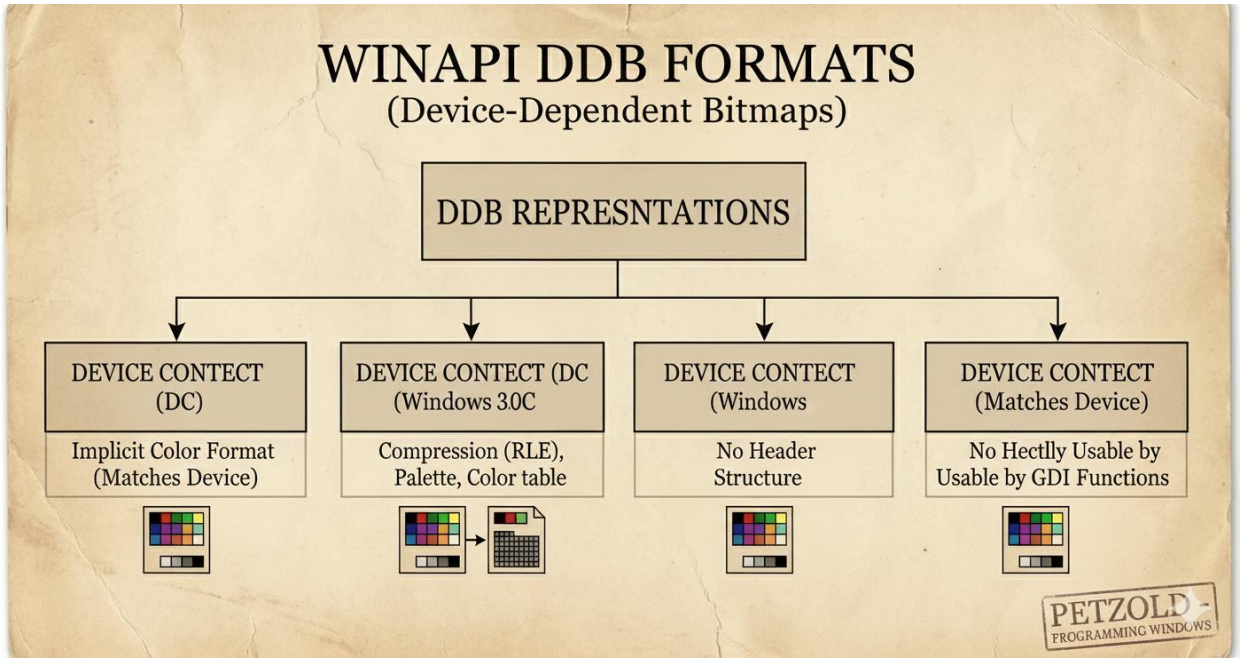
Small Files: Storing a few sentences of "instructions" is much smaller than storing millions of individual pixel colors.

TYPE	DISPLAY SPEED	PROCESSING COST
Bitmap	**Fastest** (Direct pixel copy)	Low Simple memory transfer
Metafile	**Slower** (Must be drawn)	High CPU calculates every line

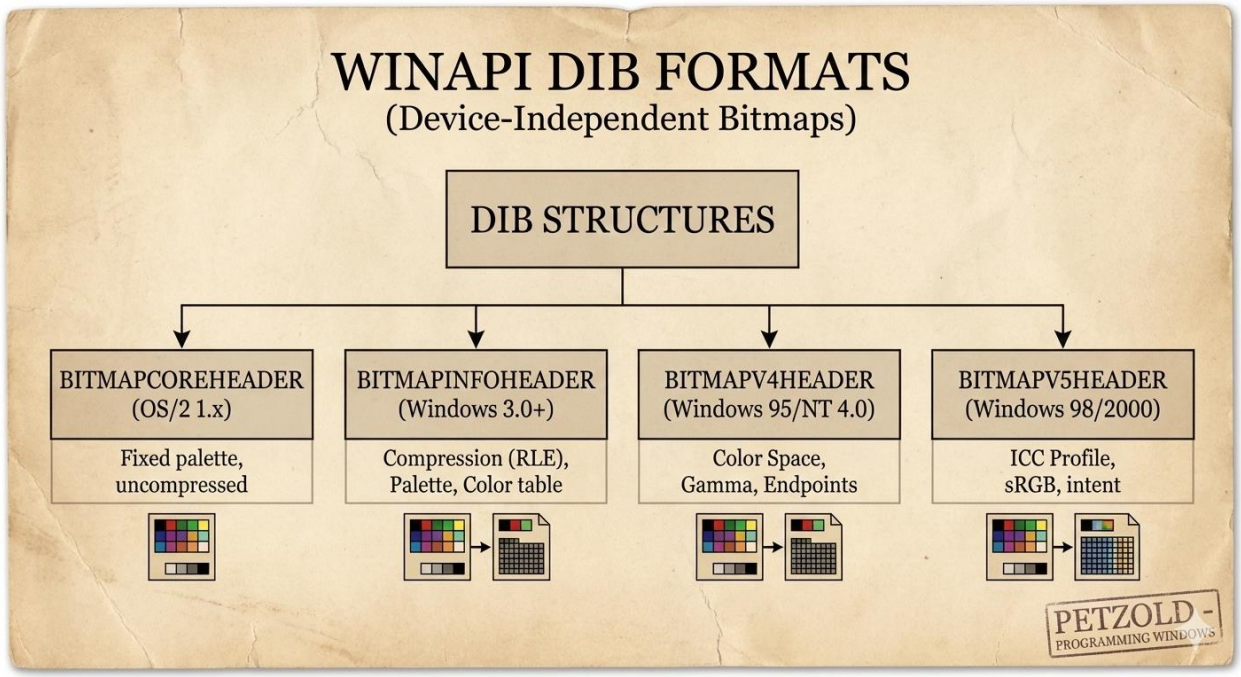
Key Takeaway: Use **Bitmaps** for speed and realistic detail (photos). Use **Metafiles** for diagrams, logos, and things that need to be resized frequently.

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.

```
// Creating a DIB: You control the memory
BITMAPINFO bmi = {0};
bmi.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
bmi.bmiHeader.biWidth = 800;
bmi.bmiHeader.biHeight = -600; // Negative = Top-Down (Standard)
bmi.bmiHeader.biBitCount = 32; // 32-bit (RGBA)

void* pPixels; // Memory address of the raw pixels
HBITMAP hDIB = CreateDIBSection(hdc, &bmi, DIB_RGB_COLORS, &pPixels, NULL, 0);
// Story: You tell Windows: "Give me a 32-bit buffer. I don't care about the monitor."
```

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.

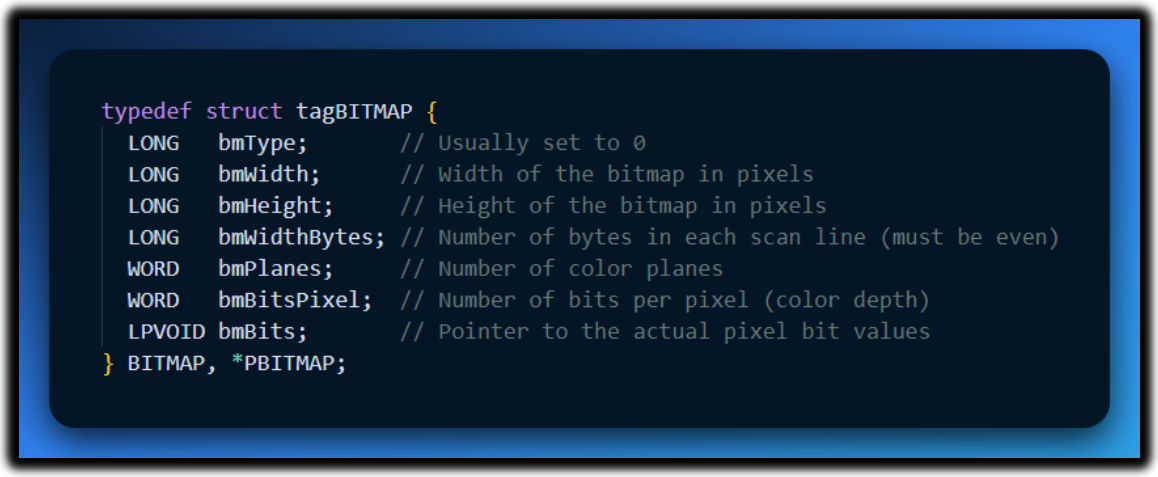
The returned pPixels pointer is the main memory location where raw data is directly read or modified.

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.



```
typedef struct tagBITMAP {
  LONG   bmType;       // Usually set to 0
  LONG   bmWidth;      // Width of the bitmap in pixels
  LONG   bmHeight;     // Height of the bitmap in pixels
  LONG   bmWidthBytes; // Number of bytes in each scan line (must be even)
  WORD   bmPlanes;     // Number of color planes
  WORD   bmBitsPixel;  // Number of bits per pixel (color depth)
  LPVOID bmBits;       // Pointer to the actual pixel bit values
} BITMAP, *PBITMAP;
```

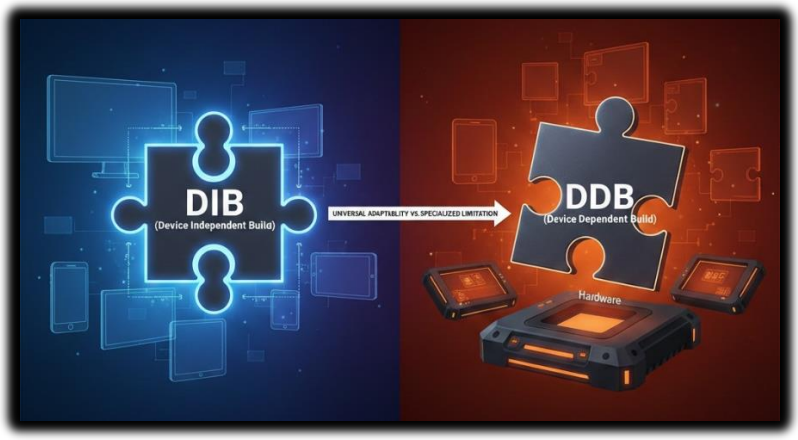
Real-World Use:

- **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.

FEATURE	DIB (Device-Independent)	DDB (Device-Dependent)
HEADER	Has `BITMAPINFOHEADER`	No Header (Naked)
MEMORY ACCESS	**Direct Pointer** (pPixels)	**Hidden** (Handle only)
SPEED	Medium Requires conversion	**Blazing Fast** GPU Native
PORTABILITY	High Works on any screen	Low Tied to current DC
KEY FUNCTION	`CreateDIBSection`	`CreateCompatibleBitmap`

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.

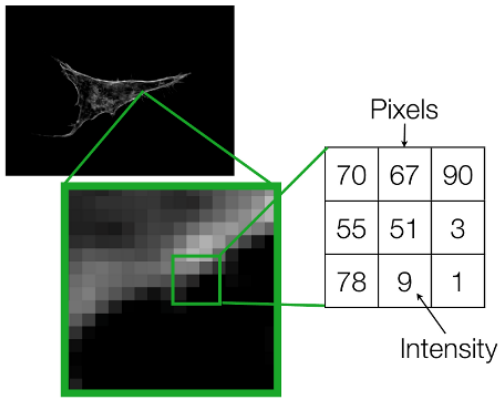
X-Axis: Increases as you move to the **right**.

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

3. Why Dimensions Matter

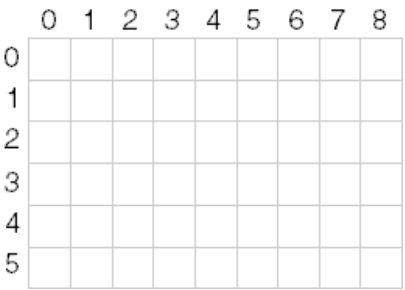
Memory Allocation: The computer uses the width and height to calculate exactly how much RAM is needed to store the image.

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.

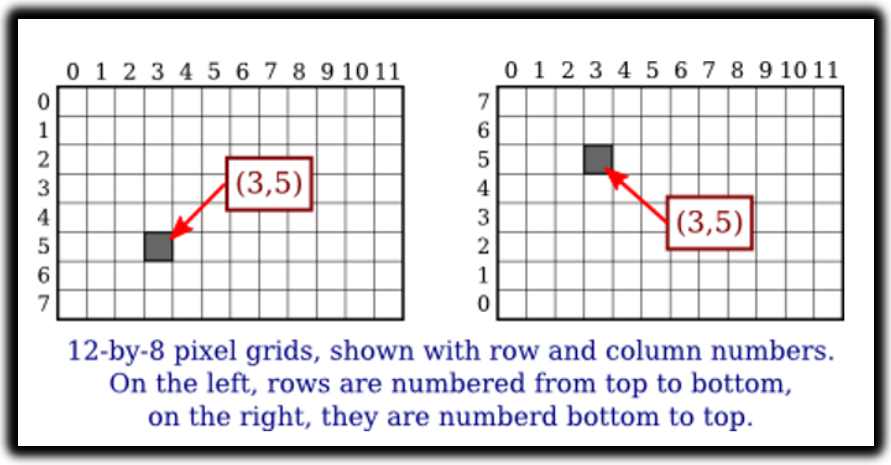
Pinpointing Pixels: The Coordinate System

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)**.



12-by-8 pixel grids, shown with row and column numbers.
On the left, rows are numbered from top to bottom,
on the right, they are numberd bottom to top.

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).

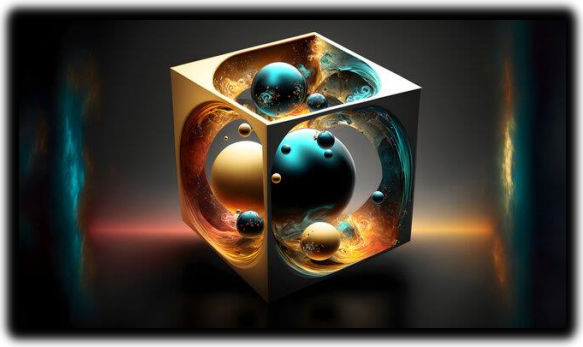
- It tells you how much **space** the image takes up on a monitor.
- In Windows GDI, this is usually what you care about when defining bitmap width and height.

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:

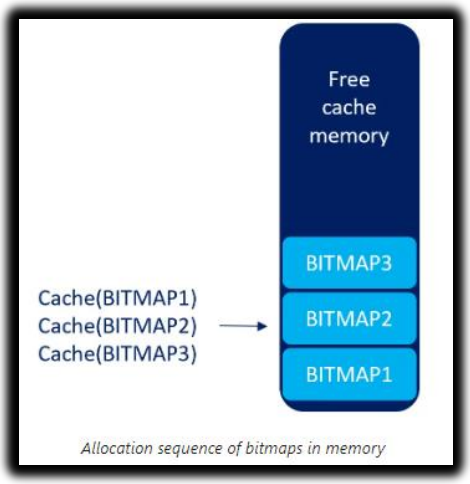
If you have a 300*300 pixel bitmap:

- 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.

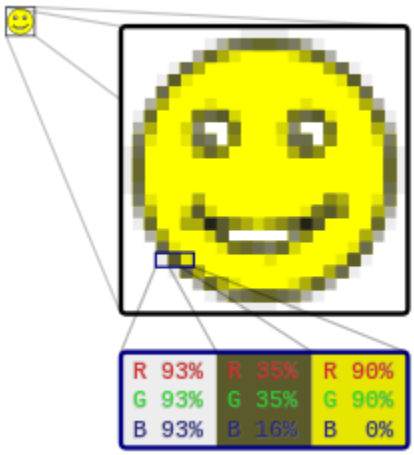
The Math: The number of possible colors is $\mathbf{2^n}$, where n is the number of bits.

BIT DEPTH	COLORS POSSIBLE	COMMON NAME
1-bit	2 (2^1)	Monochrome (Black & White)
4-bit	16 (2^4)	Standard VGA
8-bit	256 (2^8)	Indexed Color (GIFs)
24-bit	16.7 Million (2^{24})	True Color (Photos)

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.

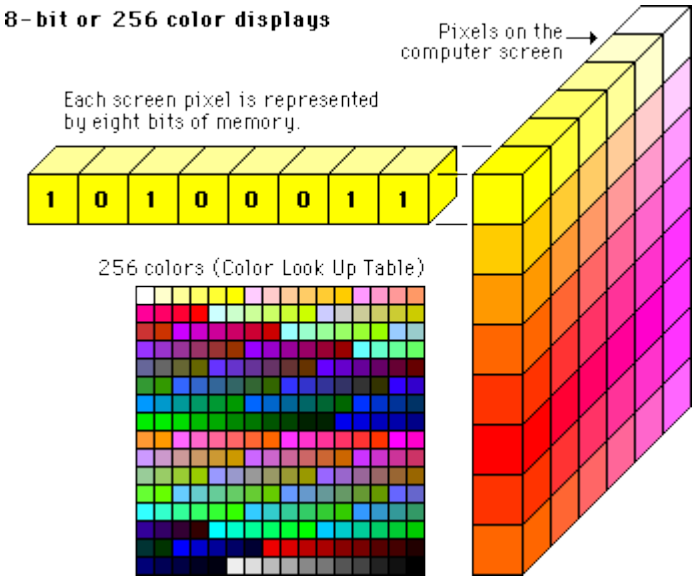
This exponential growth allows bitmaps to paint a **broader spectrum** of the world around us.

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.

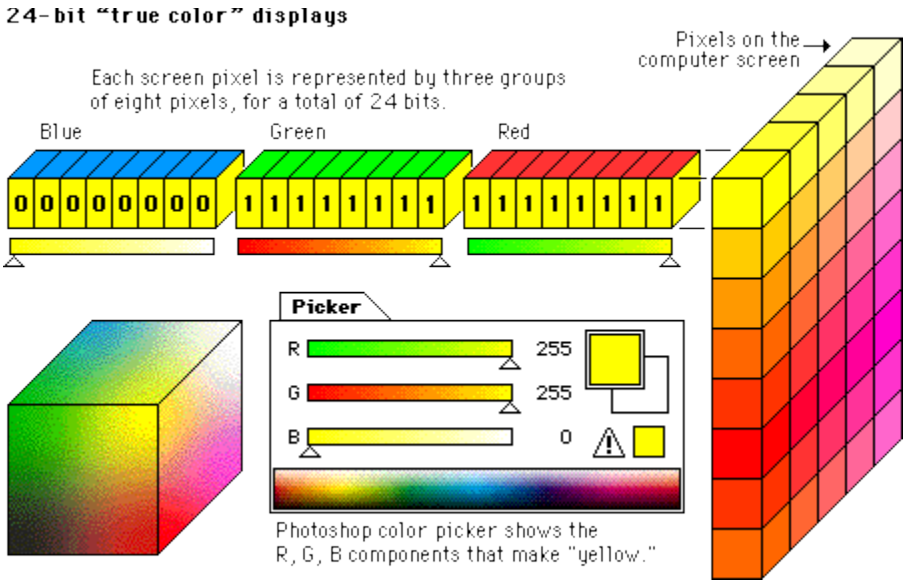


Pushing the Boundaries: 16-bit, 24-bit, and Beyond

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**.

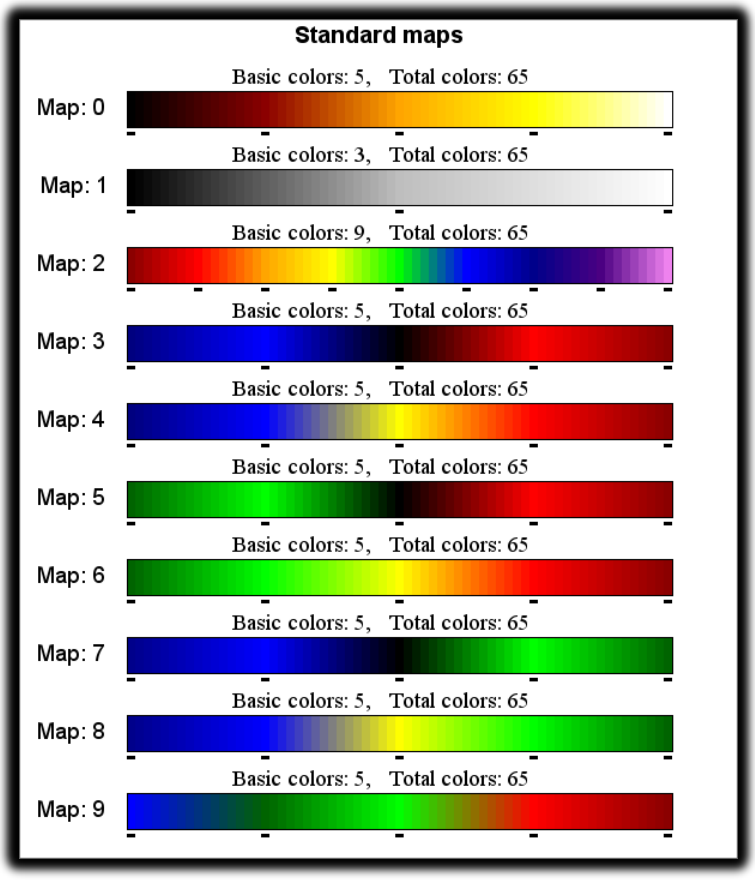
How it Works

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:

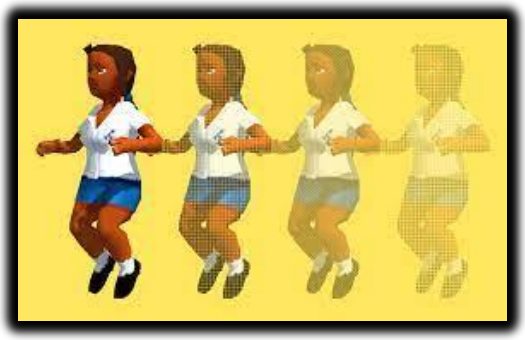
- **Performance:** It uses much less memory. A single number (the index) is smaller than a full color code.
- **Customization:** You can change the entire look of an image just by swapping the palette. If you change "Color #3" from Red to Blue, every "3" pixel in the image changes instantly. This is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

FEATURE	INDEXED COLOR (OLDSCHOOL)	TRUE COLOR (MODERN)
Logic	Pick from a pre-set list (Palette).	Every pixel has its own unique color recipe.
Storage	Very Small.	Much Larger.
Flexibility	Great for icons, GIFs, and retro games.	Best for high-res photos and 4K video.



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.

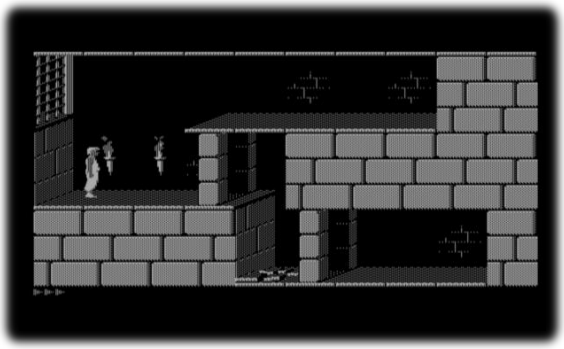
So, pick up your digital brush, dive into color, and let your creativity shine! Let's go!!

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.

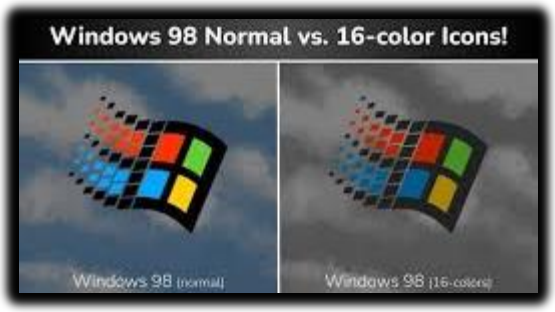
1. The Monochrome Era (1-bit)

- **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)

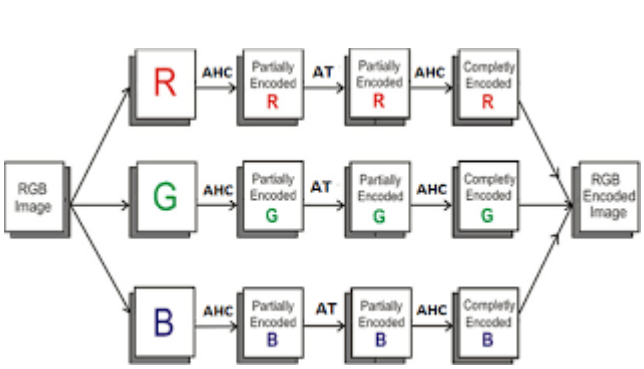
- **The Hardware:** EGA (Enhanced Graphics Adapter).
- **The Math:** 4 bits per pixel (2^4 = 16 possible combinations).
- **The Standard:** This became the "Windows Classic" look. It's why old icons and cursors look like simple cartoons—they only had 16 "crayons" to work with.

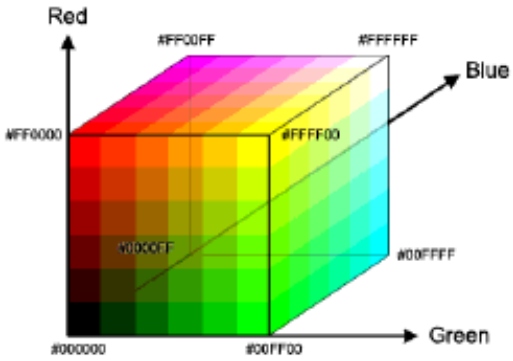


3. What is IRGB Encoding?

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

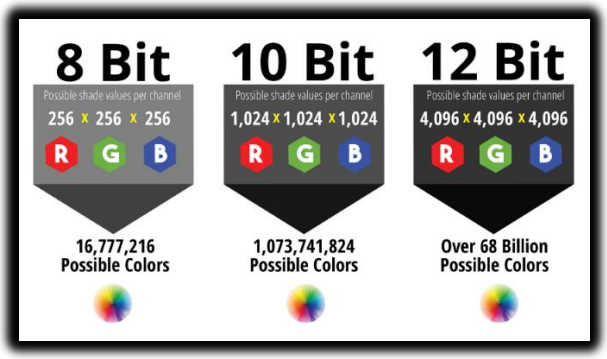
- The first 3 bits handled the actual color (R, G, B).
- The 4th bit (Intensity) acted like a **"brightness switch."** * If Intensity was "Off," you got a dark version of the color; if it was "On," you got a bright/vivid version.





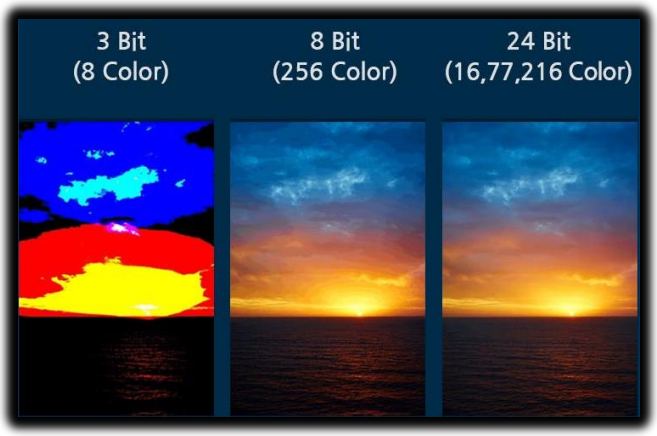
BEYOND 16 COLORS, A SPECTRUM OF POSSIBILITIES

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:

Video Display Adapter	Bit Depth	Color Combinations	Typical Use Cases
Hercules Graphics Card	1 bit	2 (Black, White)	Monochrome text, simple graphics
Enhanced Graphics Adapter	4 bits	16 (IRGB palette)	Icons, basic cartoons, early Windows elements
Video Graphics Array	8 bits	256	Photorealistic images, detailed artwork
Modern Display Adapters	24 bits	16.7 million	True color visuals, near-photorealistic graphics

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.

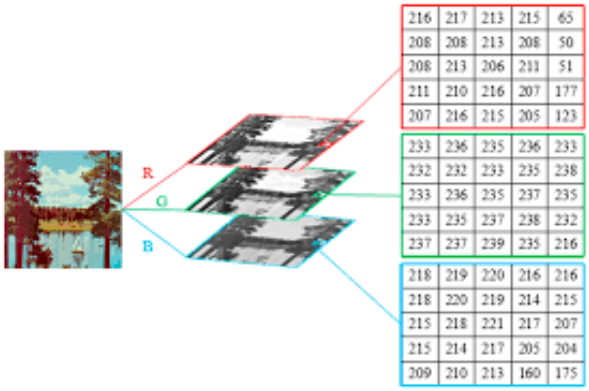
Each pixel used **4 bits**, which mapped to specific hexadecimal RGB values, as shown in the table below.

IRGB	RGB Color	Color Name
0000	00-00-00	Black
0001	00-00-80	Dark Blue
0010	00-80-00	Dark Green
...	...	...
1100	FF-00-00	Red
1101	FF-00-FF	Magenta
1110	FF-FF-00	Yellow
1111	FF-FF-FF	White

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

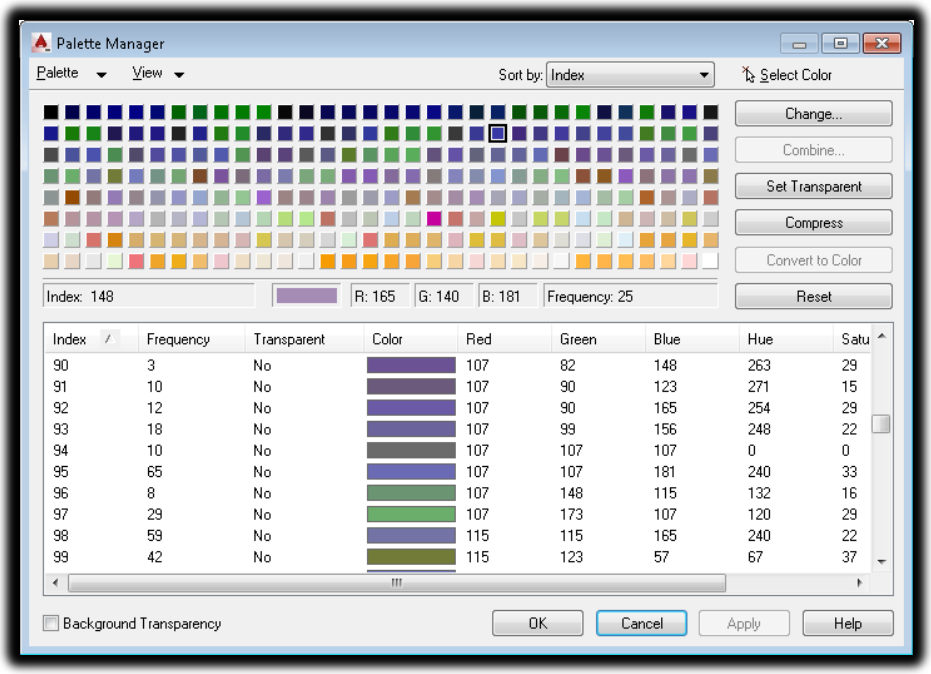
The 16-color limit didn't last long. In 1987, **VGA** introduced 8 bits per pixel, which meant images could use **256 colors**. This allowed graphics to look more detailed and realistic.

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.

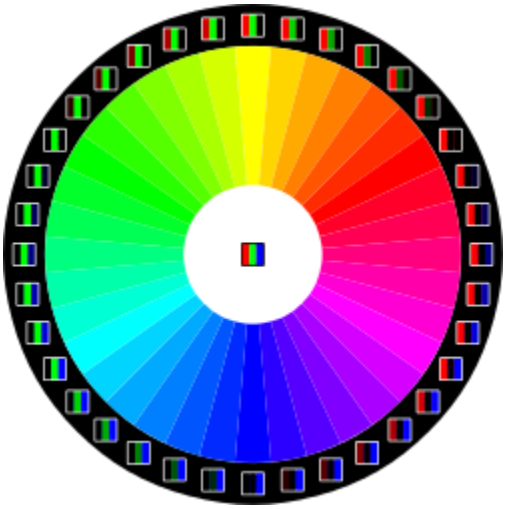


This table summarizes the reserved colors:

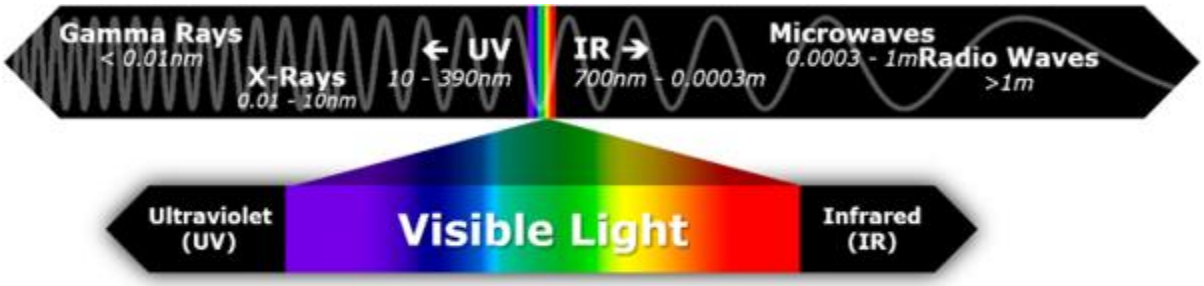
Color Value	RGB Color	Color Name
00000000	00-00-00	Black
...	...	...
11111000	80-80-80	Dark Gray
11111001	FF-00-00	Red
11111010	00-FF-00	Green
...	...	...
11111111	FF-FF-FF	White

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:

Bit Depth	Number of Colors	Color Representation	Typical Use Cases
1	2	Black & White	Monochrome text
4	16	IRGB Palette	Early Windows elements, icons
8	256	VGA Palette	Photorealistic images, detailed artwork
16	32,768 or 65,536	High Color	"Thousands of colors," improved visuals
24	16,777,216	True Color	Millions

BITMAP SUPPORT IN GDI: FROM LEGACY TO MODERN MAGIC

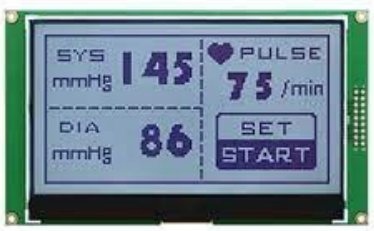
The **Windows Graphics Device Interface (GDI)** has come a long way, from early monochrome displays to supporting millions of colors. Here's how bitmap support evolved:

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.

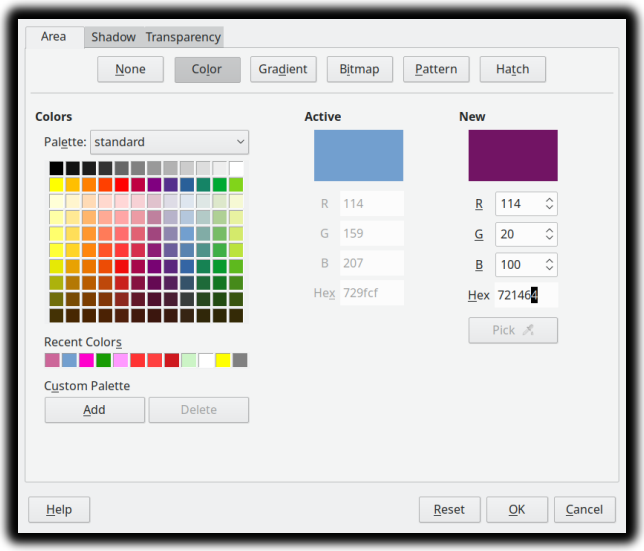
- Each DIB includes its **own color table**, mapping pixel values to actual RGB colors.
- This made it possible to display the same bitmap on **any raster device**, no matter the hardware.
- Converting colors to match the device's capabilities could still be tricky, but the flexibility was a game-changer.

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

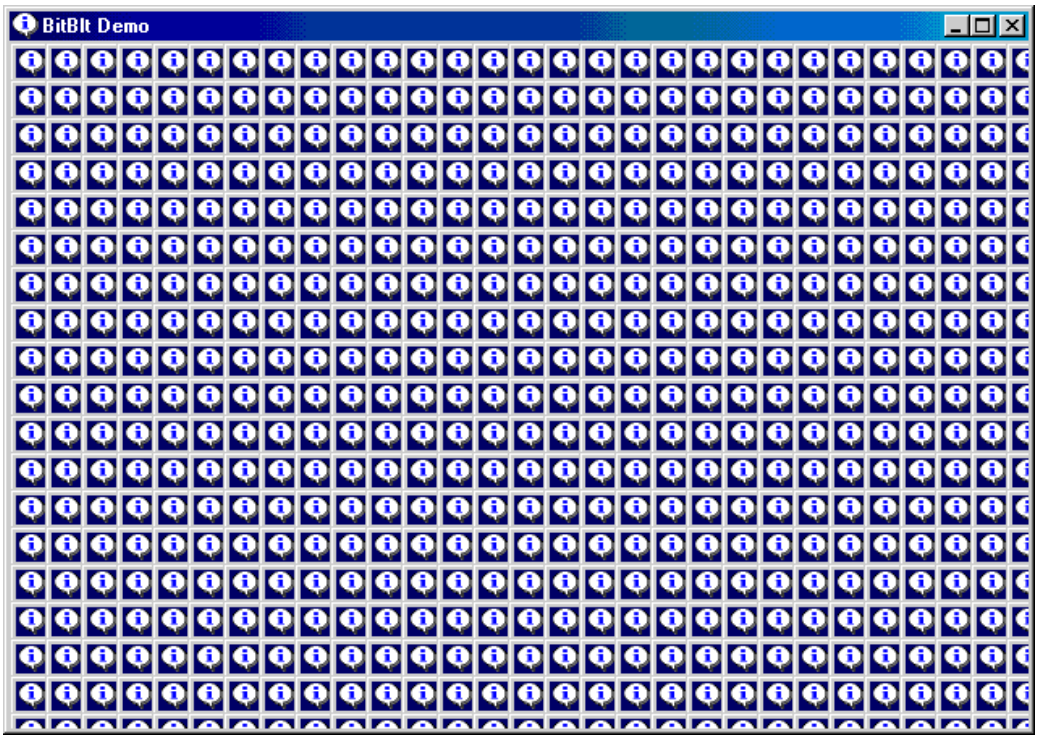
In the 2020s, while 3D games use GPUs (Graphics Processing Units), almost every **2D interface** (like the windows on your desktop, browser rendering, and mobile app menus) still *relies on the logic of BitBlt* to render quickly without draining your battery.

Operation	Result	Common Use Case
`SRCCOPY`	Exact duplicate	*Moving a window or sprite*
`SRCAND`	Combines pixels	*Creating shadows/transparency*
`SRCINVERT`	Flips colors	*Highlighting icons/text*
`WHITENESS`	All white	*Clearing a drawing area*

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.

```c
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
    static TCHAR szAppName[] = TEXT("BitBlt");
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;

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

    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
        return 0;
    }

    hwnd = CreateWindow(szAppName, TEXT("BitBlt Demo"), WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL);

    ShowWindow(hwnd, iCmdShow);
    UpdateWindow(hwnd);

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

    return msg.wParam;
}
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
     static int cxClient, cyClient, cxSource, cySource;
     HDC hdcClient, hdcWindow;
     int x, y;
     PAINTSTRUCT ps;

     switch (message) {
     case WM_CREATE:
         cxSource = GetSystemMetrics(SM_CXSIZEFRAME) + GetSystemMetrics(SM_CXSIZE);
         cySource = GetSystemMetrics(SM_CYSIZEFRAME) + GetSystemMetrics(SM_CYCAPTION);
         return 0;
     case WM_SIZE:
         cxClient = LOWORD(lParam);
         cyClient = HIWORD(lParam);
         return 0;
     case WM_PAINT:
         hdcClient = BeginPaint(hwnd, &ps);
         hdcWindow = GetWindowDC(hwnd);

         for (y = 0; y < cyClient; y += cySource)
             for (x = 0; x < cxClient; x += cxSource) {
                 BitBlt(hdcClient, x, y, cxSource, cySource,
                     hdcWindow, 0, 0, SRCCOPY);
             }
         ReleaseDC(hwnd, hdcWindow);
         EndPaint(hwnd, &ps);
         return 0;

     case WM_DESTROY:
         PostQuitMessage(0);
         return 0;
     }

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



The Important Ideas

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

That's it. **Source → destination.**

1. 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.

2. 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.

3. 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.

4. 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

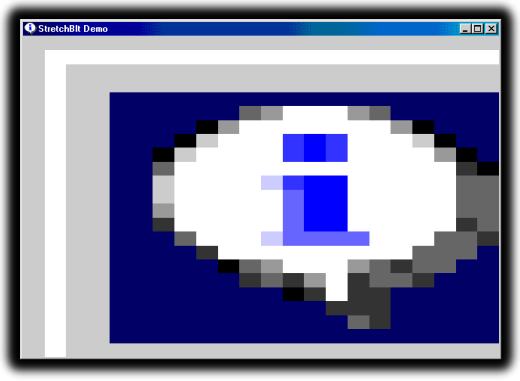
Nothing else.

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.

```c
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
    static TCHAR szAppName[] = TEXT("Stretch");
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;

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

    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
        return 0;
    }

    hwnd = CreateWindow(szAppName, TEXT("StretchBlt Demo"), WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL);

    ShowWindow(hwnd, iCmdShow);
    UpdateWindow(hwnd);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static int cxClient, cyClient, cxSource, cySource;
    HDC hdcClient, hdcWindow;
    PAINTSTRUCT ps;

    switch (message) {
    case WM_CREATE:
        cxSource = GetSystemMetrics(SM_CXSIZEFRAME) +
            GetSystemMetrics(SM_CXSIZE);
        cySource = GetSystemMetrics(SM_CYSIZEFRAME) +
            GetSystemMetrics(SM_CYCAPTION);
        return 0;

    case WM_SIZE:
        cxClient = LOWORD(lParam);
        cyClient = HIWORD(lParam);
        return 0;

    case WM_PAINT:
        hdcClient = BeginPaint(hwnd, &ps);
        hdcWindow = GetWindowDC(hwnd);

        StretchBlt(hdcClient, 0, 0, cxClient, cyClient,
            hdcWindow, 0, 0, cxSource, cySource, MERGECOPY);

        ReleaseDC(hwnd, hdcWindow);
        EndPaint(hwnd, &ps);
        return 0;

    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }

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



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

1. 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.

2. 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.

3. 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.

4. 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.

5. 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

1. BLACKONWHITE

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

2. WHITEONBLACK

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

3. COLORONCOLOR (Most Useful)

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

4. 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.

ROP	Boolean Operation	ROP Code	Name
00000000	11110000	0x000042	BLACKNESS
00000001	00100010	0x1100A6	NOTSRCERASE
00000011	00110000	0x330008	NOTSRCCOPY
00010000	00100000	0x440328	SRCERASE
00010101	01010100	0x550009	DSTINVERT
00010110	10100100	0x5A0049	PATINVERT
00100110	01100000	0x660046	SRCINVERT
10001000	00001000	0x8800C6	SRCAND
10111100	00010010	0xBB0226	MERGEPAINT
11000000	00000010	0xC000CA	MERGECOPY
11001100	00100000	0xCC0020	SRCCOPY
11101110	00000110	0xEE0086	SRCPAINT
11110000	00000000	0xF00021	PATCOPY
11110111	00001010	0xFB0A09	PATPAINT
11111111	11110000	0xFF0062	WHITENESS

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

NAME	LOGIC	RESULT / DESCRIPTION
BLACKNESS	0	All pixels become black.
WHITENESS	1	All pixels become white.
PATCOPY	P	Copies the pattern directly to the destination.
SRCCOPY	S	Copies the source directly to the destination.
DSTINVERT	~D	Inverts the destination pixels (creates a negative image).
PATPAINT	DPSnoo	Complex mix of pattern, destination, and source OR'd together.

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

```
PatBlt(hdc, x, y, cx, cy, dwROP)
```

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)

Pattern (P)	Destination (D)	Boolean Operation	ROP Code	Name	Result
1100	1010	~(P OR D)	0x0500A9	NOTSRCERASE	Invert source, then AND with destination
0010	~P & D	Invert pattern, then AND with destination	0x0A0329	NOTSRCCOPY	Invert source pixels
0011	~P	Invert pattern pixels	0x0F0001	DSTINVERT	Invert destination pixels
0100	P & ~D	AND source and destination	0xA000C9	SRCAND	AND source and destination
1001	~(P XOR D)	Invert pattern XOR destination	0xA50065	MERGEPAINT	OR source and NOT destination
1010	D	Destination pixels	0xAA0029	SRCCOPY	Copy source pixels
0000	P	Pattern pixels	0xF00021	PATCOPY	Copy pattern pixels
1101	P OR ~D	OR source and NOT destination	0xF50225	PATPAINT	OR source and NOT destination and pattern

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.

```
470   hBrush = SelectObject(hdc, hBrush);
471   PatBlt(hdc, rect.left, rect.top, rect.right - rect.left, rect.bottom - rect.top, PATCOPY);
472   SelectObject(hdc, hBrush);
```

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:

```
PatBlt(hdc, 0, 0, 100, -100, dwROP);
```

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.

```
PatBlt(hdc, 0, -100, 100, 100, dwROP);
```

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

```
PatBlt(hdc, 100, 0, -100, -100, dwROP);
```

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.

```
PatBlt(hdc, 100, -100, -100, 100, dwROP);
```

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:

Mapping Mode	Description	PatBlt Notes	Example
MM_TEXT	Units are pixels, y increases downward	Use positive values for cx and cy, origin is upper left corner	Draw a red square at (100, 100): `PatBlt(hdc, 100, 100, 50, 50, 0x7C0004)` (SRCCOPY ROP code)
MM_HIENGLISH	Units are hundredths of an inch, y increases downward	Use positive values for cx and cy, origin is upper left corner	Draw a blue rectangle at (200, 150): `PatBlt(hdc, 200, 150, 100, -100, 0x5A0049)` (PATINVERT ROP code)
MM_HIMETRIC	Units are tenths of a millimeter, y increases downward	Use positive values for cx and cy, origin is upper left corner	Draw a green line 10mm long at (0, 0): `PatBlt(hdc, 0, 0, 100, 0, 0x000042)` (BLACKNESS ROP code)
MM_LOENGLISH	Units are hundredths of an inch, y increases upward	Use positive values for cx and cy, origin is upper left corner	Draw a yellow circle at (50, -75): `PatBlt(hdc, 50, -75, 100, 100, 0x330008)` (NOTSRCCOPY ROP code)
MM_LOMETRIC	Units are tenths of a millimeter, y increases upward	Use positive values for cx and cy, origin is upper left corner	Draw a 5mm square at (25, 50): `PatBlt(hdc, 25, 50, 50, 50, 0xFF0062)` (WHITENESS ROP code)

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.

```
hBitmap = CreateBitmap(cx, cy, cPlanes, cBitsPixel, bits);
```

CreateCompatibleBitmap

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

```
hBitmap = CreateCompatibleBitmap(hdc, cx, cy);
```

CreateBitmapIndirect

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

```
BITMAP bitmap;
bitmap.bmWidth = cx;
bitmap.bmHeight = cy;
// ... Set other BITMAP fields ...
hBitmap = CreateBitmapIndirect(&bitmap);
```

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.

```
DeleteObject(hBitmap);
```

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:

```
cy * cPlanes * ((cx * cBitsPixel + 15) / 16)
```

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:

```
SetBitmapBits(hBitmap, cBytes, &bits)
```

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.

```
GetBitmapBits(hBitmap, cBytes, &bits)
```

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

DDB Memory & Pixel Layout

Scanline Alignment: Bits are arranged starting from the **top row**. Every row (scanline) **must** be padded to an even number of bytes (16-bit alignment).

I. Monochrome DDBs (1 bit per pixel):

- 0 = Black (Off)
- 1 = White (On)
- Direct manipulation via SetBitmapBits is safe and predictable here.

II. Color DDBs (Multiple bits per pixel):

- Pixel values are **indexes**, not actual colors.
- The value points to a location in the hardware's **Palette Lookup Table (PLT)**.
- **Warning:** Because the PLT varies between computers, the same pixel value (e.g., 0x05) might be Red on one machine and Blue on another.

Key DDB Functions

I. Data Manipulation

- **SetBitmapBits / GetBitmapBits:** Best used **only for monochrome**. Avoid these for color bitmaps because of device dependence.
- **SetDIBits / GetDIBits:** The modern standard. These convert device-independent data (DIB) into the device's current format (DDB) automatically. Use these for all color operations.

II. Metadata Tags

SetBitmapDimensionEx / GetBitmapDimensionEx: * Stores the physical size of the bitmap in **0.1 mm units**.

Crucial: This is just a "tag." Windows does not use this data for drawing or scaling; it is strictly for your own application's record-keeping.

Summary of Best Practices

1. **Don't assume color values:** Never hard-code a hex value for a color DDB; use GDI functions to translate colors.
2. **Align your rows:** If manually building a bitmap buffer, ensure every row ends on an even-byte boundary.
3. **Prefer DIB functions:** Use SetDIBits to ensure your colors look the same across different hardware.

MEMORY DEVICE CONTEXT (MDC)

What is an MDC?

A **Memory DC** is a **virtual device context** in system memory.

It acts like a normal DC (has colors, fonts, mapping modes) but **doesn't draw on the screen directly**.

Default MDC = 1x1 pixel. You need a bitmap to make it useful.

Why Use an MDC?

Allows **off-screen drawing**.

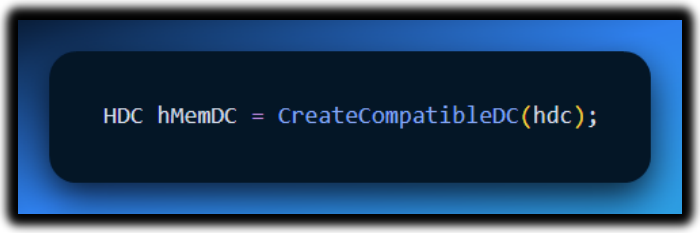
You can prepare graphics without flickering the screen.

Useful for:

- Complex graphics
- Bitmaps manipulation
- Capturing or copying screen areas

How to Create an MDC

1. Compatible with a specific DC



Matches color depth and resolution of the real DC (hdc).

2. Compatible with the screen

```
HDC hMemDC = CreateCompatibleDC(NULL);
```

Quick way to create a general-purpose MDC.

3. Attaching a Bitmap (DDB)

- MDC alone is tiny.
- Use a **DDB** as the drawing surface:

```
HBITMAP hBmp = CreateCompatibleBitmap(hdc, width, height);
SelectObject(hMemDC, hBmp);
```

Requirements:

- Color planes and bits per pixel must **match the MDC**.
- Incompatible bitmaps will fail.

4. Using the MDC

- Draw on the **bitmap** using normal GDI functions (Rectangle, TextOut, etc.).
- Transfer content to the real DC with **BitBlt**:

```
BitBlt(hdcDest, 0, 0, width, height, hMemDC, 0, 0, SRCCOPY);
```

Capture part of the screen into a bitmap using MDC + BitBlt.

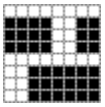
Key Points

- MDC = off-screen drawing canvas.
- Always select a **compatible DDB** before drawing.
- Use **CreateCompatibleDC** to match a real DC or the screen.
- BitBlt = move content between MDC and real DC.
- Useful for **double buffering**, graphics manipulation, and screen capture.
- *Don't forget, open your TTS software and read this 5 times* - 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.

```c
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
    static TCHAR szAppName[] = TEXT("Bricks1");
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;

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

    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
        return 0;
    }

    hwnd = CreateWindow(szAppName, TEXT("LoadBitmap Demo"), WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL);

    ShowWindow(hwnd, iCmdShow);
    UpdateWindow(hwnd);

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

    return msg.wParam;
}
```

```c
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static HBITMAP hBitmap;
    static int cxClient, cyClient, cxSource, cySource;
    BITMAP bitmap;
    HDC hdc, hdcMem;
    HINSTANCE hInstance;
    int x, y;
    PAINTSTRUCT ps;
    switch (message) {
        case WM_CREATE:
            hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
            hBitmap = LoadBitmap(hInstance, TEXT("Bricks"));
            GetObject(hBitmap, sizeof(BITMAP), &bitmap);
            cxSource = bitmap.bmWidth;
            cySource = bitmap.bmHeight;
            return 0;
        case WM_SIZE:
            cxClient = LOWORD(lParam);
            cyClient = HIWORD(lParam);
            return 0;
        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            hdcMem = CreateCompatibleDC(hdc);
            SelectObject(hdcMem, hBitmap);

            for (y = 0; y < cyClient; y += cySource)
                for (x = 0; x < cxClient; x += cxSource)
                    BitBlt(hdc, x, y, cxSource, cySource, hdcMem, 0, 0, SRCCOPY);

            DeleteDC(hdcMem);
            EndPaint(hwnd, &ps);
            return 0;
        case WM_DESTROY:
            DeleteObject(hBitmap);
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc(hwnd, message, wParam, lParam);
}
```

```
600  // Microsoft Developer Studio generated resource script.
601  #include "resource.h"
602  #include "afxres.h"
603
604  // Bitmap
605  BRICKS BITMAP DISCARDABLE "Bricks.bmp"
```



BRICKS.BMP in BRICKS1.C — Simple Notes

Purpose: BRICKS.BMP is the **source image** for the brick pattern in the window.

The program **tiles** this bitmap across the client area.

I. Resource Definition

```
BRICKS BITMAP DISCARDABLE "Bricks.bmp"
```

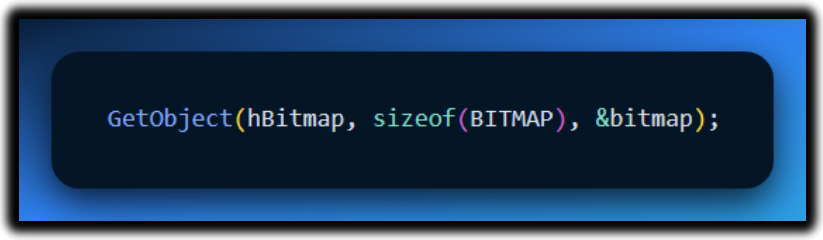
- **BRICKS** → resource name
- **BITMAP** → resource type
- **DISCARDABLE** → can be unloaded from memory when not needed
- "Bricks.bmp" → path to the bitmap file

II. Loading the Bitmap

LoadBitmap - Loads the resource into memory and returns a **handle**.

```
hBitmap = LoadBitmap(hInstance, TEXT("Bricks"));
```

GetObject - Fills a BITMAP structure with width, height, and color info.



III. Drawing the Bricks

BitBlt in WM_PAINT copies the bitmap to the window:

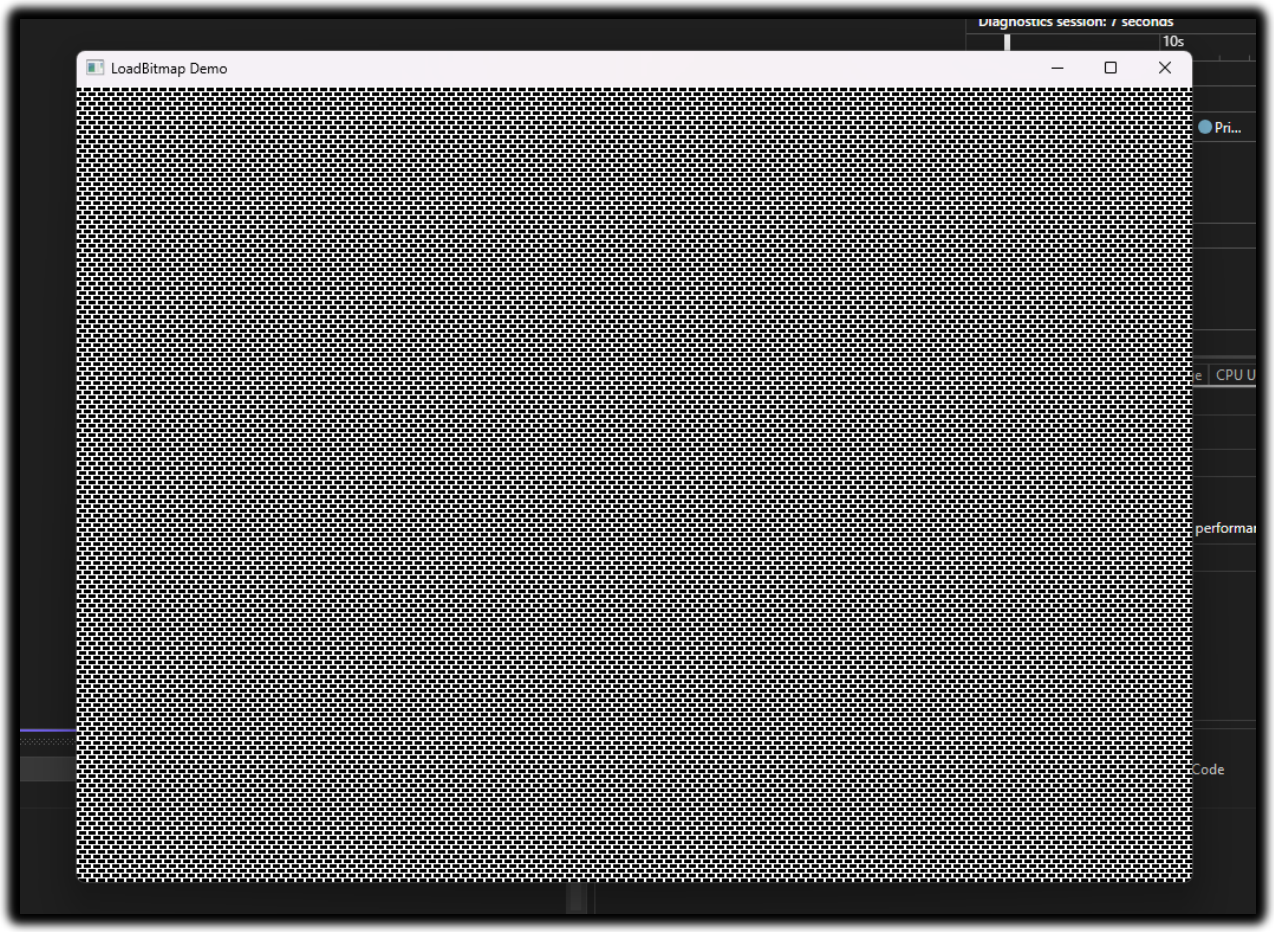
- Loops through the client area.
- Tiles horizontally and vertically.
- Creates the repeating brick wall effect.

IV. Cleanup

Delete the bitmap handle in WM_DESTROY to **free memory**.

V. Key Takeaways

- BRICKS.BMP = source for the brick pattern.
- LoadBitmap → load resource.
- GetObject → get size and format.
- BitBlt → draw/ tile bitmap.
- DeleteObject → clean up memory.



Resize me, and see the wonders😈

Monochrome Bitmap Creation

BRICKS2.C demonstrates an alternative to resource-based bitmaps: defining **monochrome bitmaps directly in code**. This approach is especially useful for small images where creating resource files can feel like unnecessary overhead.

I. Why Create Bitmaps Directly?

Direct bitmap creation offers several advantages:

- **Simpler workflow:** No need for separate resource files or loading steps.
- **Precise control:** You can manipulate individual bits for exact pixel patterns.
- **Flexible changes:** Updating the image is as easy as modifying the bit sequence in your code.

II. Understanding the Monochrome Format

Monochrome bitmaps are essentially **binary grids**:

- 0 → black pixel
- 1 → white pixel

Bits are read left to right and grouped into **8-bit sequences** to form the byte data. If the width isn't a multiple of 16, padding with zeros ensures full bytes.

III. Constructing the Bitmap

Two main elements are needed:

- **BITMAP structure:** Defines width, height, and byte width.
- **BYTE array:** Holds the actual bitmap data in byte form.

IV. Creating the Bitmap Object

There are three common methods:

1. **CreateBitmapIndirect with bmBits:** Assigns the byte array directly to the BITMAP structure.
2. **CreateBitmapIndirect + SetBitmapBits:** Separates creation and data assignment, useful if you plan to modify bits later.
3. **CreateBitmap with all parameters:** Combines structure definition and byte data assignment in a single step.

V. BRICKS2.C in Action

BRICKS2 uses this technique to draw a repeating brick pattern **without a resource file**, showing how direct bitmap creation can simplify small, custom visuals.

VI. Key Takeaways

- Direct monochrome bitmap creation is **flexible, efficient, and great for small images**.
- Understanding the binary format and constructing BITMAP structures allows you to **define custom patterns**.
- BRICKS2.C illustrates **practical applications** of this approach.

Tip: Experimenting with bit patterns is the best way to master this technique and discover creative possibilities.

```
617   static BITMAP bitmap = { 0, 20, 5, 4, 1, 1 };
618   static BYTE bits[] = { 0x51, 0x77, 0x10, 0x00,
619                          0x57, 0x77, 0x50, 0x00,
620                          0x13, 0x77, 0x50, 0x00,
621                          0x57, 0x77, 0x50, 0x00,
622                          0x51, 0x11, 0x10, 0x00 };
623
624   hBitmap = CreateBitmap(20, 5, 1, 1, bits);
```

This code defines a **BITMAP structure** with the desired dimensions and fills a **BYTE array** with the corresponding monochrome pixel data.

Then, it uses CreateBitmap to generate the bitmap directly from this data.

```c
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow);
static TCHAR szAppName[] = TEXT("Bricks2");
static BITMAP bitmap = {0, 8, 8, 2, 1, 1};
static BYTE bits[8][2] = {0xFF, 0, 0x0C, 0, 0x0C, 0, 0x0C, 0,
                          0xFF, 0, 0xC0, 0, 0xC0, 0, 0xC0, 0};
static HBITMAP hBitmap;
static int cxClient, cyClient, cxSource, cySource;

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

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

    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
        return 0;
    }

    hwnd = CreateWindow(szAppName, TEXT("CreateBitmap Demo"), WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL);

    ShowWindow(hwnd, iCmdShow);
    UpdateWindow(hwnd);

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

    return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    HDC hdc, hdcMem;
    int x, y;
    PAINTSTRUCT ps;

    switch (message) {
        case WM_CREATE:
            bitmap.bmBits = bits;
            hBitmap = CreateBitmapIndirect(&bitmap);
            cxSource = bitmap.bmWidth;
            cySource = bitmap.bmHeight;
            return 0;

        case WM_SIZE:
            cxClient = LOWORD(lParam);
            cyClient = HIWORD(lParam);
            return 0;

        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            hdcMem = CreateCompatibleDC(hdc);
            SelectObject(hdcMem, hBitmap);

            for (y = 0; y < cyClient; y += cySource)
                for (x = 0; x < cxClient; x += cxSource)
                    BitBlt(hdc, x, y, cxSource, cySource, hdcMem, 0, 0, SRCCOPY);

            DeleteDC(hdcMem);
            EndPaint(hwnd, &ps);
            return 0;

        case WM_DESTROY:
            DeleteObject(hBitmap);
            PostQuitMessage(0);
            return 0;
    }

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

Bricks2.c: Mastering Monochrome Bitmaps in GDI

Let's strip back the fluff and look at **BRICKS2.C** through a purely functional lens.

Essentially, this program is a lesson in **manual memory mapping** for GDI. Instead of loading an .bmp file, you are telling the computer exactly which bits to turn on and off in a tiny 8x8 grid.

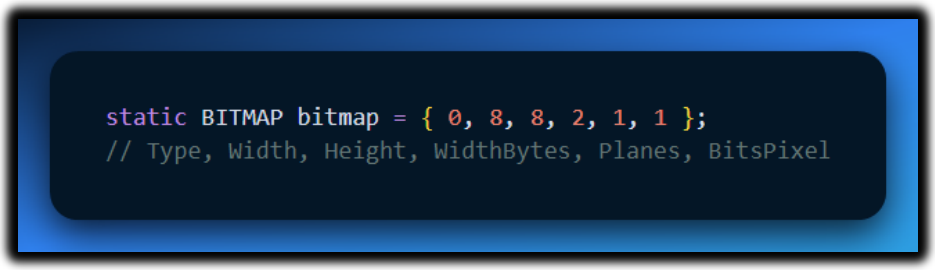
1. The Raw Data: bits[8][2]

This is the most critical part of the code. Because it's a monochrome bitmap, each bit represents a pixel.

- **Width:** 8 pixels (requires 1 byte).
- **Padding:** GDI requires bitmap rows to be multiples of **WORDs** (2 bytes). This is why the array is [8][2], even though we only "need" one byte per row.
- **The Pattern:** If your byte is 0xFF (binary 11111111), you get a solid line. If it's 0x01 (00000001), you get a single dot.

2. The Link: CreateBitmapIndirect

While CreateBitmap takes parameters individually, CreateBitmapIndirect takes a pointer to a BITMAP structure. It's cleaner for static data.



```
static BITMAP bitmap = { 0, 8, 8, 2, 1, 1 };
// Type, Width, Height, WidthBytes, Planes, BitsPixel
```

In WM_CREATE, you simply point bitmap.bmBits to your bits array. You've now turned a raw array into a GDI object handle (hBitmap).

3. The Performance: BitBlt

Instead of drawing lines or points manually (which is slow), BitBlt (Bit Block Transfer) performs a high-speed memory copy from your **Memory Device Context** to the **Window Device Context**.

4. The Tiling Logic

The nested loops in WM_PAINT simply calculate the offsets:

1. Grab the width/height of the window (cxClient, cyClient).
2. Run two for loops incrementing by 8 (the size of your brick).
3. BitBlt the pattern at every (x, y) coordinate.

5. Why this is "Old School"

This technique is the foundation of **Pattern Brushes**.

In modern Win32 programming, you wouldn't usually BitBlt in a loop to fill a background; you'd create a brush from the bitmap (CreatePatternBrush) and let Windows handle the tiling automatically during a background erase.

6. The Limitation

As your text noted, this is **Device-Dependent**.

This code works here because monochrome bitmaps are universal
(1 bit per pixel is always 1 bit per pixel).

If you tried this with color using a raw array, it would break the moment you moved from an 8-bit color display to a 32-bit color display because the "bits per pixel" would no longer match your array.

```c
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow);

static TCHAR szAppName[] = TEXT("Bricks3");
HBITMAP hBitmap;
HBRUSH hBrush;

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

    hBitmap = LoadBitmap(hInstance, TEXT("Bricks"));
    hBrush = CreatePatternBrush(hBitmap);
    DeleteObject(hBitmap);

    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance;
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = hBrush;
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = szAppName;

    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
        return 0;
    }

    hwnd = CreateWindow(szAppName, TEXT("CreatePatternBrush Demo"), WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL);

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

    DeleteObject(hBrush);

    return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    switch (message) {
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }

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

```c
#include "resource.h"
#include "afxres.h"

// Bitmap
BRICKS BITMAP DISCARDABLE "Bricks.bmp"
```

BRICKS3.C: UNVEILING THE POWER OF PATTERN BRUSHES

BRICKS3.C is about efficiency. If **BRICKS2** was "doing it the hard way" by manually looping BitBlt, **BRICKS3** is letting the OS do the heavy lifting by turning a bitmap into a **Brush**.

---

1. The Strategy: Bitmap ⏭ Brush

Instead of treating the bitmap as a "picture" to be copied, you treat it as a "paint."

- **LoadBitmap**: Pulls the 8x8 "Bricks" grid from the resources.
- **CreatePatternBrush**: Converts that bitmap into a logical GDI Brush ($hBrush$).
- **The Hand-off**: Once the Brush is created, GDI has cached the pattern. You can (and should) DeleteObject(hBitmap) immediately to free up memory.

2. The Window Class Integration

This is the "hidden gem." You don't even need code in WM_PAINT.

```
// Load Bitmap - Loads Bricks.bmp from resources.
HBITMAP hBmp = LoadBitmap(hInstance, "Bricks");

// Create Pattern Brush - Converts the bitmap into a reusable brush.
// The original bitmap can be deleted after brush creation.
HBRUSH hBrush = CreatePatternBrush(hBmp);
DeleteObject(hBmp);

// Set Brush as Window Background
// The window automatically uses the brush to fill its client area.
WNDCLASS wc = {0};
wc.hbrBackground = hBrush;

// Delete the brush in WM_DESTROY to avoid resource leaks
DeleteObject(hBrush);
PostQuitMessage(0);

// Resource File (BRICKS3.RC)
Bricks BITMAP "Bricks.bmp"
```

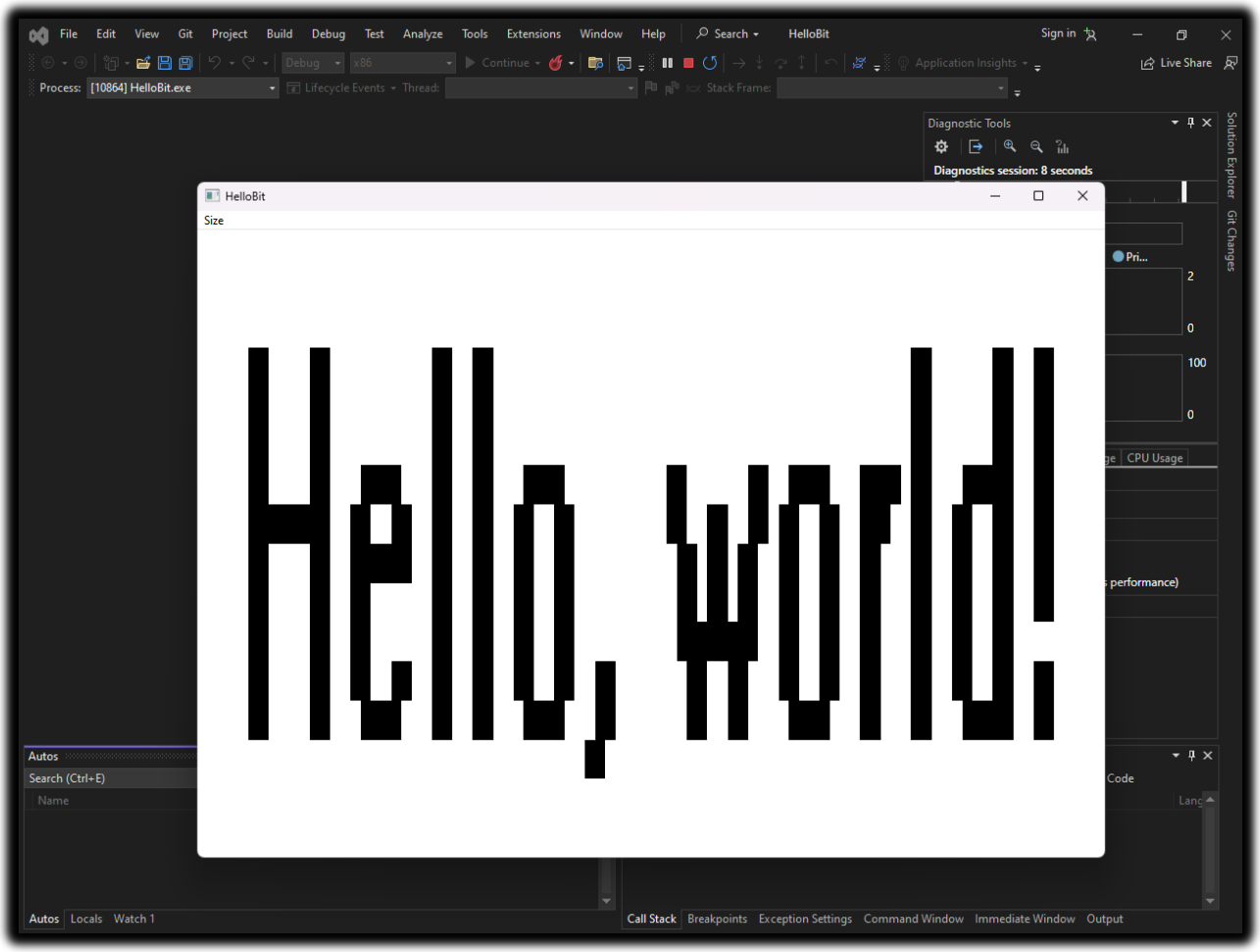
The resource file makes the bitmap available for loading by the program.

Concepts

- **Pattern Brush:** Uses a bitmap to fill areas repeatedly.
- **Brush vs. Bitmap:** Brushes are for filling areas; bitmaps can be drawn or used as brushes.
- **Memory Device Contexts (MDC):** Off-screen drawing using bitmaps, later transferred to the window using BitBlt or StretchBlt.
- Reusing a bitmap as a brush simplifies window painting.
- Always delete GDI objects (brushes, bitmaps) to avoid leaks.

Brush behavior differs by OS:

- Windows 98: uses only upper-left 8x8 pixels if bitmap is larger.
- Windows NT: uses the entire bitmap.



HELLOBIT – Bitmaps & Drawing in GDI

HELLOBIT demonstrates creating **bitmaps for off-screen drawing** using a **Memory Device Context (MDC)**, then displaying them in different ways on a window.

I. Create Text Bitmap

```
SIZE textSize;
GetTextExtentPoint32(hdc, "Hello, world!", 13, &textSize);

HBITMAP hBmp = CreateCompatibleBitmap(hdc, textSize.cx, textSize.cy);
HDC hMemDC = CreateCompatibleDC(hdc);
SelectObject(hMemDC, hBmp);

TextOut(hMemDC, 0, 0, "Hello, world!", 13);
```

- Calculates text size for bitmap dimensions.
- Creates an MDC and bitmap for off-screen drawing.
- Draws text onto the bitmap using TextOut.

II. Display Bitmap

Big (stretch to fit window):

```
StretchBlt(hdc, 0, 0, width, height, hMemDC, 0, 0, bmpWidth, bmpHeight, SRCCOPY);
```

Small (tile across window):

```
BitBlt(hdc, x, y, bmpWidth, bmpHeight, hMemDC, 0, 0, SRCCOPY);
```

Demonstrates scaling and tiling of bitmaps.

III. Cleanup

Properly frees GDI resources in WM_DESTROY.

IV. Concepts

Memory Device Context (MDC): Off-screen canvas for drawing on bitmaps before displaying.

Bitmaps for Drawing: Allows manipulation of visuals without directly affecting the window.

BitBlt: copies bitmap as-is (good for tiling).

StretchBlt: scales bitmap to fit target area (can cause pixelation).

Using MDCs prevents flickering and unnecessary redrawing.

Bitmaps give precise control over text and graphics before painting the window.

Scaling bitmaps may degrade quality; tiling preserves clarity.

Always cleanup GDI objects to avoid resource leaks.

SKETCH — Using Shadow Bitmaps

I. Shadow Bitmap Concept

- SKETCH **doesn't draw directly on the window**.
- Creates an **off-screen bitmap** (shadow bitmap) in memory.
- A **memory DC (MDC)** is used for drawing on this bitmap.

II. Drawing & Erasing

Mouse buttons control pen color:

- Left = black pen
- Right = white pen (eraser)

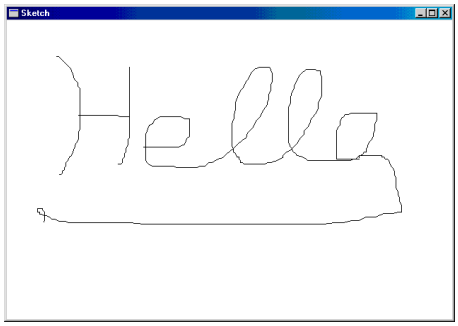
Lines are drawn on **both the shadow bitmap and the window** → creates real-time drawing effect.

III. Notes / Limitations

- Clearing the drawing requires restarting → no built-in clear function.
- Simple but shows **power of off-screen drawing**.

IV. Key Takeaways

- Shadow bitmaps = off-screen canvas for smooth drawing.
- Drawing updates in memory first, then **BitBlt** to the window.
- Useful for reducing flicker and keeping a persistent copy.
- Can combine with other GDI functions for more advanced graphics.



Shadow Bitmap Size in SKETCH

I. Choosing the Size

Bitmap should **cover the largest possible client area** → allows window resizing without breaking the drawing.

Can use:

- GetSystemMetrics → basic max size estimate.
- EnumDisplaySettings → brute-force approach to handle larger/future resolutions (sometimes 4× bigger than current display).

II. Memory Considerations

- Large bitmaps = high memory usage (several MBs).
- Always **check if bitmap already exists** before creating a new one.
- Throw an error if memory allocation fails → prevents crashes.

III. Mouse Capture & Drawing

- Captures mouse **inside and outside the window** for seamless drawing.
- Drawing may persist outside original window if mouse moves beyond boundaries.
- Optional: Separate drawing code for window DC vs shadow bitmap DC for clarity.

IV. Key Takeaways

- Shadow bitmap size = **balance between memory usage and future-proofing**.
- Dynamic resizing or partial updates can improve efficiency.
- Mouse capture + persistent bitmap = convenient but can cause unintended drawings outside visible area.

Using Bitmaps in Menus (GRAFMENU)

I. Why Bitmaps in Menus

Traditional menus = text only.

GRAFMENU uses **bitmaps to make menus visual** → helps users understand options faster.

II. Examples of Bitmap Use

- **Icons replaced with expressive images** → more meaningful than generic folder/paperclip icons.
- **Font menu** → each font option shows a bitmap preview with the actual font applied.
- **Line widths, hatch patterns, colors** → visualized with bitmaps for immediate reference.

III. How It Works

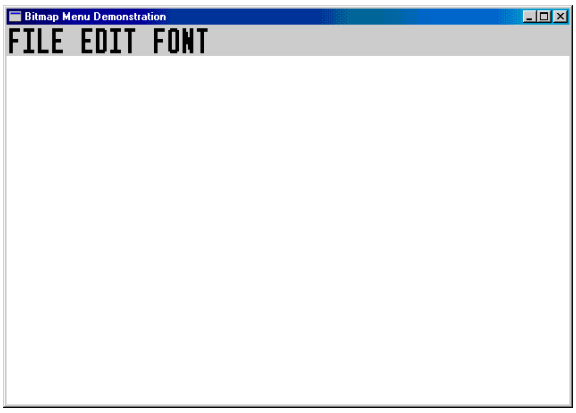
- **Memory Device Contexts (MDCs)** act as off-screen canvases.
- Bitmaps are created and manipulated in MDCs without affecting the visible menu until ready.

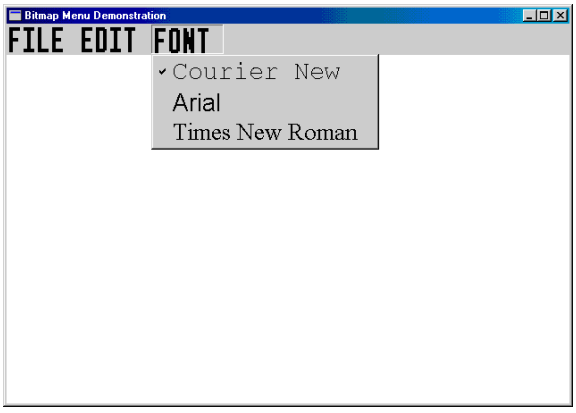
IV. Extra Touches

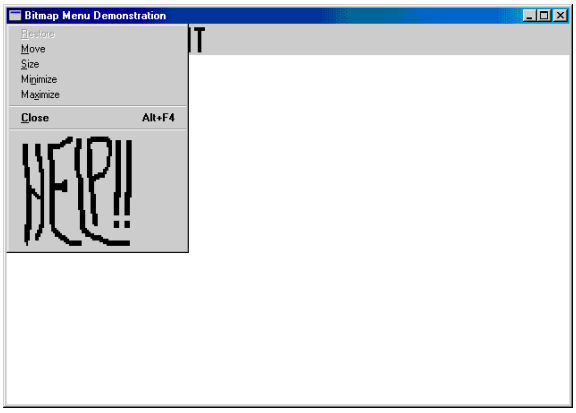
- Bitmaps can add **personality or humor** (e.g., Help menu showing a funny icon).
- Beyond function → bitmaps improve clarity, aesthetics, and user engagement.

V. Key Takeaways

- Bitmaps make menus **intuitive, visual, and expressive**.
- MDCs = safe space to create and modify bitmaps.
- Experiment with **fonts, line styles, colors, or small visual jokes** to make menus more engaging.







Remember

Bitmaps offer great flexibility, but always ensure their design is clear.

Consistency and alignment with your program's overall look is something to look after.

Accessibility considerations are important too.

Ensure bitmap-based menus are accessible for users with visual impairments, if possible.

GRAFMENU

This program demonstrates a specific Windows capability: replacing standard text menu items (like "File") with **Bitmaps handles (HBITMAP)**. This allows for icons, custom fonts, or entirely graphical interfaces in the menu bar.

I. The Core Mechanism (AppendMenu)

In standard Windows programming, you add a menu item like this:

```
AppendMenu(hMenu, MF_STRING, IDM_FILE, "File");
```

In GRAFMENU, we change the flag and pass a bitmap handle instead of a string:

```
AppendMenu(hMenu, MF_BITMAP, IDM_FILE, (LPSTR)hBitmapFile);
```

Windows automatically renders the image. If the image is taller than the menu bar, Windows automatically expands the menu bar height to fit it.

II. Technique A: The "Stretcher" (Scaling Static Bitmaps)

The program loads bitmaps for "File" and "Edit" from the resource file (.RC). However, raw bitmaps rarely match the user's system font size. If the bitmap is too small, it looks silly; if too big, it looks clunky.

The Solution: StretchBitmap() This helper function dynamically resizes the bitmap at runtime to match the system's text dimensions.

The Logic:

1. **Measure:** Get the system font height (GetTextMetrics).
2. **Create Canvas:** Create a Memory Device Context (DC) compatible with the screen.
3. **Draw:** Use StretchBlt (Stretch Block Transfer) to copy the original bitmap into a new, correctly sized bitmap.
4. **Result:** A handle to a new bitmap that fits the menu bar perfectly.

III. Technique B: The "Generator" (Dynamic Font Previews)

The "Font" submenu is the coolest part of this program. It doesn't load images from disk; it **draws them in memory**. It shows "Arial" *written in Arial* and "Courier" *written in Courier*.

The Recipe (GetBitmapFont):

1. **Create a Blank Bitmap:** Initialize a white rectangle in memory.
2. **Create the Font:** Use CreateFont to generate a logical font object (e.g., Times New Roman).
3. **Select & Draw:** Select that font into the Memory DC and use TextOut to write the name onto the invisible bitmap.
4. **Finalize:** The function returns the HBITMAP, which is then handed to AppendMenu.

IV. The System Menu Hack

The "System Menu" is the menu that pops up when you click the program icon (top-left) or right-click the title bar.

GRAFMENU inserts a "Help" bitmap into this standard Windows menu.

How to touch the untouchable:

```
// 1. Get the handle to the system menu (FALSE = give me the handle, don't reset it)
HMENU hSysMenu = GetSystemMenu(hwnd, FALSE);

// 2. Add your custom bitmap item
AppendMenu(hSysMenu, MF_BITMAP, IDM_HELP, (LPSTR)hBitmapHelp);
```

V. The "Gotchas" (Technical Constraints)

Using bitmaps in menus comes with three major trade-offs you must handle:

Keyboard Accelerators Break: Standard menus use Alt+F for File. Bitmaps have no text, so Windows doesn't know which key triggers them.

- ✓ *Fix:* You must process the WM_MENUCHAR message to manually tell Windows, "If the user presses 'F', open the first bitmap."

Checkmarks are Ugly: If you check a bitmap item (CheckMenuItem), Windows draws the standard checkmark next to it. If your bitmap is large, the tiny checkmark looks out of place.

- ✓ *Fix:* Use SetMenuItemBitmaps to provide your own custom "Checked" and "Unchecked" images.

Memory Leaks: Unlike text strings, bitmaps consume GDI resources. You **must** delete these bitmaps (DeleteObject) when the program closes (WM_DESTROY), or you will leak memory every time the app runs.

VI. Quick Review

Question 1: Which API function flag tells Windows that you are passing a picture handle instead of text? *(Answer: MF_BITMAP inside AppendMenu.)*

Question 2: Why can't we just load the bitmaps from the Resource file and use them directly? *(Answer: Because we don't know the user's screen resolution or font size. We must use StretchBlt to scale them so they look proportional to the menu bar.)*

Question 3: What happens if you try to use Alt-keys (like Alt+F) on a bitmap menu without extra code? *(Answer: Nothing happens. The computer beeps. You must implement WM_MENUCHAR to handle keyboard input for graphical menus.)*

MASKING BITMAPS FOR ELLIPTICAL SHAPES

BITMASK shows how to display a bitmap in a **non-rectangular shape**—in this case, an ellipse—using a mask bitmap.

I. How It Works (Code Reference)

1. **Load the image**
    - ✓ WM_CREATE → LoadBitmap loads the rectangular image.
    - ✓ Get its size with GetObject.
    - ✓ Select it into a compatible memory DC.
2. **Create the mask**
    - ✓ CreateBitmap → monochrome mask same size as the image.
    - ✓ Draw a **white ellipse** on black background using Rectangle + Ellipse in a memory DC.
3. **Masking the image**
    - ✓ Use BitBlt with **SRCAND** / SRCPAINT raster operations:
        - White pixels in the mask show the image.
        - Black pixels hide it.
    - ✓ This produces the elliptical crop effect.

4. **Paint to the window**
    - ✓ WM_PAINT → center the masked image using client and bitmap dimensions.
    - ✓ Two BitBlt calls: one applies the mask, the other paints the visible image.
5. **Cleanup**
    - ✓ WM_DESTROY → delete both image and mask bitmaps.

II. Key Takeaways

- Masks + BitBlt allow **non-rectangular bitmaps**.
- Raster operations (SRCAND, SRCPAINT) are essential for masking.
- This technique can be adapted for **circles, stars, or any custom shapes**.

III. Where to Look in the Code

- WM_CREATE → image + mask creation
- WM_PAINT → BitBlt masking logic
- WM_DESTROY → cleanup

No need to memorize every step—just understand **mask → BitBlt → shape** pattern.

BITMASK: Key Notes

Masking works for any bitmap, not just MATTHEW.BMP.

Gray background is just for visual checking; it's not essential to understand the technique.

Masking logic:

- SRCAND → keeps image pixels where the mask is white, hides pixels where mask is black.
- SRCPAINT → paints the masked image onto the window without altering the background.

Elliptical effect comes from the mask shape—modify the mask to change the shape.

Done. Simple. We can skip the rest—read the code, it already shows.

BOUNCING BALL ANIMATION EXPLAINED

Purpose: Animate a bouncing ball using a bitmap and a timer.

Core Logic:

1. **Setup:**
   - ✓ WM_CREATE → sets up timer (50ms) and scaling info.
   - ✓ Global vars track ball position (xCenter, yCenter) and movement deltas (cxMove, cyMove).
2. **Ball Bitmap:**
   - ✓ WM_SIZE → calculates client area, adjusts ball size.
   - ✓ Creates memory DC and compatible bitmap.
   - ✓ Fills bitmap white, then draws ball with a diagonally hatched ellipse, leaving a margin.
3. **Animation:**
   - ✓ WM_TIMER → updates ball position.
   - ✓ BitBlt copies bitmap to the window at new coordinates.
   - ✓ Reverses direction when hitting window edges for bouncing effect.
4. **Cleanup:**
   - ✓ WM_DESTROY → deletes bitmap, kills timer.

Extra Notes:

- This is **basic animation**.
- Advanced effects can use different ROP codes, palette manipulation, or DirectX for smoother graphics.

Where to look in the code:

- WM_CREATE → initialization
- WM_SIZE → bitmap and memory DC setup
- WM_TIMER → movement & collision
- WM_DESTROY → cleanup

Done. You can see all implementation details by reading those message handlers in the code. No need to over-explain.

SCRAMBLE'S NOTES: SCRAMBLED SECRETS EXPLAINED

Purpose: Scramble portions of the screen using memory DCs for fun/visual effects.

Core Logic:

1. **Freeze the Screen:**
   - ✓ LockWindowUpdate prevents other apps from drawing while scrambling.
   - ✓ GetDCEx with DCX_LOCKWINDOWUPDATE gives full-screen access.
2. **Divide & Conquer:**
   - ✓ Screen divided into smaller rectangles (width/10, height/10) to reduce flicker and CPU load.
3. **Use Memory DC:**
   - ✓ Temporary memory DC holds rectangles while swapping.
   - ✓ Avoids rewriting the screen directly → smoother effect.
4. **Rectangle Swap (3-step BitBlt):**
   - ✓ Copy first rectangle → memory DC.
   - ✓ Copy second rectangle → first rectangle's spot on screen.
   - ✓ Copy memory DC → second rectangle's spot on screen.
5. **Undo & Cleanup:**
   - ✓ SCRAMBLE tracks swaps to restore original screen.
   - ✓ Unlock window with LockWindowUpdate(FALSE).
6. **Extra Tip:**
   - ✓ Memory DCs can also extract bitmap sections, e.g., upper-left quadrant → opens up creative bitmap manipulations.

Where to read in code:

- WM_CREATE → setup
- Timer or loop → rectangle swapping
- Cleanup → unlock and release DC

Takeaway:
SCRAMBLE is basically a playful demo, but it **teaches screen manipulation, memory DC usage, and BitBlt tricks**. All the heavy explanation? You see it clearly in the code itself.

BLOWUP: SCREEN CAPTURE & MAGNIFICATION (LEAN NOTES)

Purpose: Capture a screen region, magnify it, and allow basic menu/clipboard operations.

Core Steps:

1. **Freeze Screen for Capture:**
   - ✓ LockWindowUpdate prevents flickering.
   - ✓ GetDCEx with DCX_CACHE | DCX_LOCKWINDOWUPDATE gets the full-screen DC.
2. **User Selection:**
   - ✓ Left-click → capture start (crosshair cursor).
   - ✓ Right-click → triggers capture of selected rectangle.
   - ✓ Mouse dragging uses InvertBlock to give real-time feedback.
3. **Image Capture & Stretch:**
   - ✓ BitBlt copies selected area from screen DC to a bitmap.
   - ✓ GetClientRect → BLOWUP window size.
   - ✓ StretchBlt scales bitmap to fit the window.
4. **Menu & Clipboard Integration:**
   - ✓ Menu allows editing and copying.
   - ✓ CopyBitmap duplicates bitmap.
   - ✓ OpenClipboard, EmptyClipboard, SetClipboardData → copy to clipboard.
   - ✓ Can also paste clipboard images into the window.
5. **Cleanup:**
   - ✓ Delete temporary bitmaps and DCs with DeleteObject.
   - ✓ PostQuitMessage closes program gracefully.

Where to see in code:

- WM_LBUTTONDOWN / WM_MOUSEMOVE / WM_LBUTTONUP → selection & capture logic.
- WM_PAINT → drawing/stretched bitmap.
- Menu handlers → clipboard & editing.
- WM_DESTROY → cleanup.

Key Takeaways:

- BLOWUP demonstrates **screen capture + magnification** with BitBlt/StretchBlt.
- Menu & clipboard integration make it practical for image sharing/editing.
- Core principle: capture → store in bitmap → display with scaling.

Next Steps for Experimentation:

- Try different StretchBlt modes.
- Combine with MaskBlt or TransparentBlt for masking effects.
- Integrate image processing libraries for detection, analysis, or effects.

Lesson from chapter 14:
I realized you don't need too much detailed notes.
You need to understand what's going on.
Even if its not fully understood.
There's Microsoft documentation, Petzold's book, other WinAPI books,
and your own practice will make things make sense…