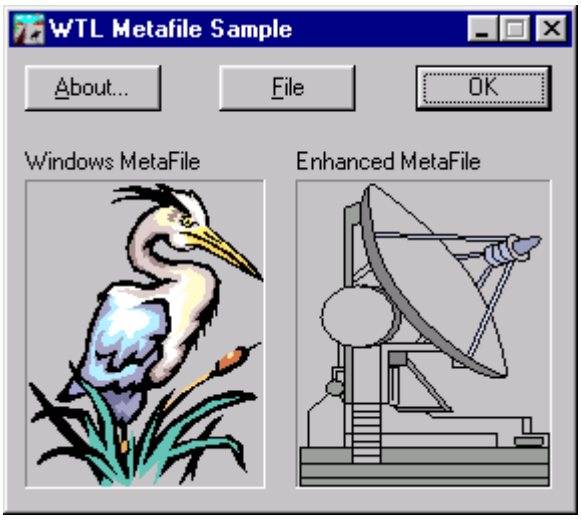
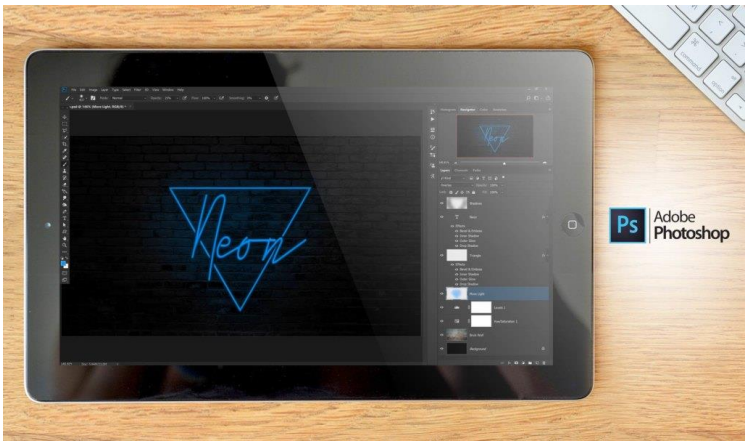
CHAPTER 18: METAFILES

Metafiles vs. Bitmaps: Metafiles are to vector graphics as bitmaps are to raster graphics. Metafiles are constructed by humans, while bitmaps generally originate from real-world images. Metafiles consist of a series of binary records that correspond to graphics function calls, while bitmaps are a collection of pixels.

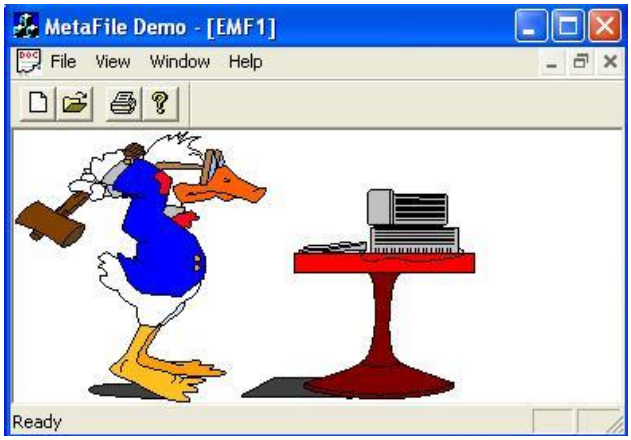


Creating and Editing: "Paint" programs create bitmaps, while "draw" programs create metafiles. In a well-designed drawing program, you can easily grab and move individual graphical objects because they are stored as separate records. In a paint program, you are generally restricted to moving or removing rectangular chunks of the bitmap.



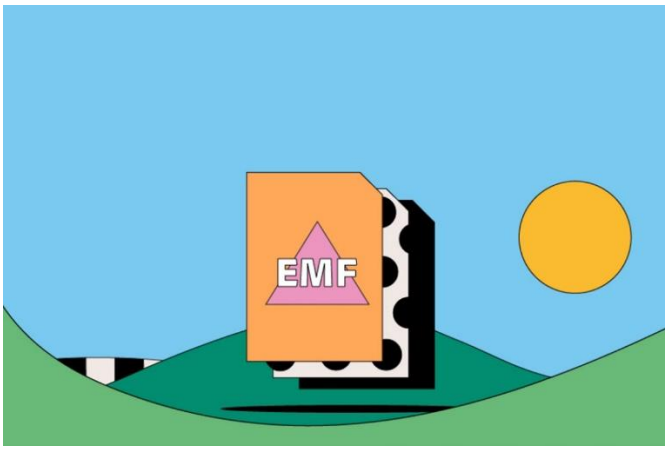
Scaling: Because metafiles describe an image in terms of graphical drawing commands, the metafile image can be scaled without loss of resolution. Bitmaps, on the other hand, cannot be scaled without losing resolution.

Converting: A metafile can be converted to a bitmap, but with some loss of information. Converting bitmaps to metafiles is much more difficult and usually requires a lot of processing power.



Uses: Metafiles are most often used for sharing pictures among programs through the clipboard, although they can also exist on disk as clip art. Because metafiles take up much less space and are more device independent than bitmaps, they are generally the preferred format for sharing and storing images.

Windows Metafile Formats: Microsoft Windows supports two metafile formats: the original metafile format, which has been supported since Windows 1.0, and the enhanced metafile format, which was developed for the 32-bit versions of Windows. The enhanced metafile format has several improvements over the old metafile format and should be used whenever possible.



UNDERSTANDING THE OLD METAFILE FORMAT: A COMPREHENSIVE GUIDE

Metafiles: Blueprints for Vector Graphics

In the realm of Windows graphics, metafiles offer a unique approach to image creation and manipulation. Unlike bitmaps, which store pixel-by-pixel representations of images, metafiles act as blueprints, storing a series of drawing commands that can be replayed to recreate the image on various devices. This mechanism offers several advantages, including:

- Device Independence: Metafiles can be rendered consistently across different displays and printers, ensuring graphical fidelity.
- Compactness: Metafiles often require less storage space compared to bitmaps, especially for images with simple shapes and lines.
- Scalability: Metafiles can be scaled without loss of quality, as the drawing commands are recalculated to maintain visual integrity.

Creating and Using Memory Metafiles

Here's a step-by-step guide to working with memory metafiles in Windows:

Metafile Device Context (MDC) Creation:

Initiate the process by calling CreateMetaFile(NULL). This function allocates memory for a metafile and returns a handle to an MDC, which serves as a virtual canvas for your drawing commands.

```
HDC hdcMeta = CreateMetaFile(NULL);
```

Drawing on the MDC:

Utilize standard GDI drawing functions like LineTo, Rectangle, TextOut, and others to construct your image on the MDC. However, these calls don't directly render on a physical device; instead, they are meticulously recorded as instructions within the metafile.

```
// Use GDI drawing functions on hdcMeta
// Example:
Rectangle(hdcMeta, 10, 10, 100, 100);
```

Closing the MDC:

Once you've completed your drawing, call CloseMetaFile to finalize the metafile and receive a handle to the encapsulated graphical data.

```
HMETAFILE hmf = CloseMetaFile(hdcMeta);
```

Playing the Metafile: Rendering the Image

To display the image stored within a metafile, follow these steps:

- Obtain a Real Device Context: Acquire a device context (DC) representing the physical device where you intend to render the image, such as a window's DC.
- Invoke PlayMetaFile: Pass the metafile handle and the real DC to the PlayMetaFile function. This initiates the execution of the recorded GDI commands, resulting in the image being meticulously drawn on the specified device.

```
case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hwnd, &ps);

        // Play the metafile on the device context
        PlayMetaFile(hdc, hmf);

        EndPaint(hwnd, &ps);
    }
    break;
```

Saving as Disk Metafile:

The CreateMetaFile function takes a single argument, which can be either NULL or a filename. If it's NULL, the metafile is stored in memory. If a filename is provided (commonly with the .WMF extension), the metafile is saved as a disk file.

```
HDC hdcMeta = CreateMetaFile(L"example.wmf");
```

Key Points to Remember:

Windows gracefully handles file operations for disk-based metafiles, alleviating you from file I/O concerns.

While the old metafile format remains functional, the newer Enhanced Metafile (EMF) format is generally recommended due to its advancements in functionality and compatibility.

METAFILE.C PROGRAM

```c
/* METAFILE.C - Metafile Demonstration Program(c) Charles Petzold, 1998 */
#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("Metafile");
    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("Metafile Demonstration"),
                        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);
    }

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static HMETAFILE hmf;
    static int cxClient, cyClient;
    HBRUSH hBrush;
    HDC hdc, hdcMeta;
    int x, y;
    PAINTSTRUCT ps;

    switch (message) {
        case WM_CREATE:
            hdcMeta = CreateMetaFile(NULL);
            hBrush = CreateSolidBrush(RGB(0, 0, 255));
            Rectangle(hdcMeta, 0, 0, 100, 100);
            MoveToEx(hdcMeta, 0, 0, NULL);
            LineTo(hdcMeta, 100, 100);
            MoveToEx(hdcMeta, 0, 100, NULL);
            LineTo(hdcMeta, 100, 0);
            SelectObject(hdcMeta, hBrush);
            Ellipse(hdcMeta, 20, 20, 80, 80);
            hmf = CloseMetaFile(hdcMeta);
            DeleteObject(hBrush);
            return 0;
        case WM_SIZE:
            cxClient = LOWORD(lParam);
            cyClient = HIWORD(lParam);
            return 0;

        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            SetMapMode(hdc, MM_ANISOTROPIC);
            SetWindowExtEx(hdc, 1000, 1000, NULL);
            SetViewportExtEx(hdc, cxClient, cyClient, NULL);
            for (x = 0; x < 10; x++)
                for (y = 0; y < 10; y++) {
                    SetWindowOrgEx(hdc, -100 * x, -100 * y, NULL);
                    PlayMetaFile(hdc, hmf);
                }
            EndPaint(hwnd, &ps);
            return 0;

        case WM_DESTROY:
            DeleteMetaFile(hmf);
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc(hwnd, message, wParam, lParam);
}
```

Metafile Creation:

The program begins by creating a metafile, which functions like a recording device for graphics commands. It's analogous to recording a song instead of playing individual notes each time.

This metafile is stored in memory, and its handle is kept for later use.

Drawing on the Metafile:

During the WM_CREATE message, the program draws shapes onto the metafile. It creates a blue brush, draws a rectangle, two diagonal lines forming an X, then uses the blue brush to fill an ellipse within the rectangle.

These drawing commands are recorded within the metafile, not directly rendered onto the screen.

Playing the Metafile:

The WM_PAINT message triggers the actual rendering of the metafile's contents.

The program obtains the window's device context (DC), which acts as a canvas for drawing.

It sets up an anisotropic mapping mode, allowing for flexible scaling of the image.

It then enters a loop, iterating 100 times to create a tiled effect.

In each iteration, it shifts the viewport origin within the logical coordinate system.

It then plays the metafile onto the device context, effectively drawing the recorded shapes multiple times in a grid-like pattern.

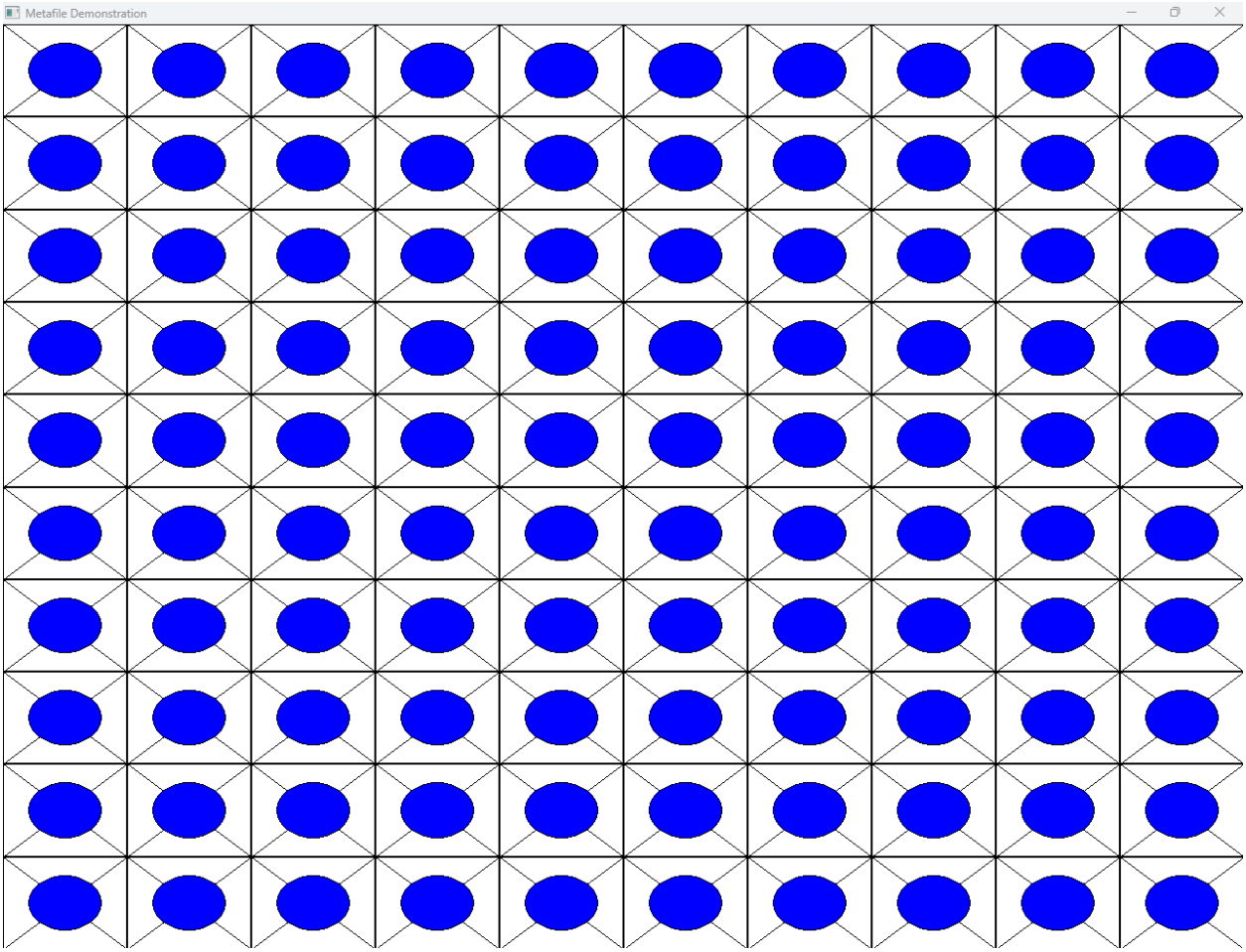
Metafile Handling:

After the drawing is complete, the metafile is closed, finalizing its contents.

During the WM_DESTROY message, the metafile is deleted to release its resources.

Key Points:

- ✓ Metafiles offer a way to store and replay graphics commands, providing flexibility in rendering images.
- ✓ They can be stored in memory or on disk.
- ✓ The program demonstrates a basic usage of metafiles, creating one in memory and replaying it multiple times to create a tiled pattern.
- ✓ Old Metafile Format: The program uses the old metafile format (WMF). While functional, the newer Enhanced Metafile (EMF) format offers advantages and is generally recommended.
- ✓ Anisotropic Mapping Mode: This mode allows flexible scaling, but requires careful management to ensure consistent results across different window sizes and resolutions.
- ✓ Code Structure: The code follows standard WinAPI window procedure patterns, making it approachable for those familiar with WinAPI development.



A program, referred to as "METAFILE," to store metafiles on disk rather than in memory. This approach is particularly advantageous for handling large metafiles due to its reduced memory footprint. However, it comes with the trade-off of requiring disk access each time the metafile is played.

To implement the transition from an in-memory metafile to a disk-based one, the key modification involves replacing the NULL argument in the CreateMetaFile call with a filename during the WM_CREATE processing.

Subsequently, when the WM_CREATE message concludes, the program deletes the metafile handle using DeleteMetaFile. Importantly, this deletion only affects the handle, leaving the actual disk file intact.

During the processing of the WM_PAINT message, the program retrieves a metafile handle to the existing disk file using the GetMetaFile function.

This handle can then be used to play the metafile as before. Upon completion of the WM_PAINT message, the metafile handle is deleted using DeleteMetaFile.

When handling the WM_DESTROY message, the program doesn't need to delete the metafile handle explicitly since it was already deleted at the end of both WM_CREATE and WM_PAINT messages.

However, it is crucial to delete the disk file associated with the metafile using DeleteFile(szFileName) unless there is a specific intention to retain the file.

To provide an alternative approach, the notes suggest the possibility of treating a metafile as a programmer-defined resource.

In such a scenario, the metafile can be loaded as a data block, and a metafile handle can be created using SetMetaFileBitsEx(iSize, pData).

This function allows the program to define a metafile using a block of data.

A corresponding function, GetMetaFileBitsEx, is available to copy the contents of a metafile to a block of memory.

```
// ... Other includes and definitions ...

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static HMETAFILE hmf;
    static int cxClient, cyClient;
    static TCHAR szFileName[] = TEXT("example.wmf");  // Replace with your desired filename
    HBRUSH hBrush;
    HDC hdc, hdcMeta;
    int x, y;
    PAINTSTRUCT ps;

    switch (message) {
        case WM_CREATE:
            hdcMeta = CreateMetaFile(szFileName);  // Specify the filename
            // ... drawing operations ...
            hmf = CloseMetaFile(hdcMeta);
            DeleteObject(hBrush);
            return 0;

        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            SetMapMode(hdc, MM_ANISOTROPIC);
            SetWindowExtEx(hdc, 1000, 1000, NULL);
            SetViewportExtEx(hdc, cxClient, cyClient, NULL);
            for (x = 0; x < 10; x++)
                for (y = 0; y < 10; y++) {
                    SetWindowOrgEx(hdc, -100 * x, -100 * y, NULL);
                    PlayMetaFile(hdc, hmf);
                }
            EndPaint(hwnd, &ps);
            DeleteMetaFile(hmf);  // Delete the metafile handle
            return 0;

        case WM_DESTROY:
            DeleteFile(szFileName);  // Delete the disk file
            PostQuitMessage(0);
            return 0;
    }

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

In this modified version, the filename "example.wmf" is used for illustration purposes. Ensure that the filename aligns with your application's requirements. The DeleteFile function is called during WM_DESTROY to remove the disk file associated with the metafile, providing a clean and resource-efficient approach.

Challenges with Old Metafiles:

Size Determination: Determining the rendered size of an old metafile from its handle alone is challenging, often requiring internal metafile inspection.

Clipboard Flexibility Limitations: Directly placing metafile handles on the clipboard restricts scaling and mapping control for recipient programs.

The METAFILEPICT Structure:

Purpose: Bridges the gap between old metafiles and clipboard flexibility.

Fields:

- mm: The metafile's intended mapping mode (e.g., MM_ISOTROPIC, MM_ANISOTROPIC).
- xExt: The width of the metafile image in logical units.
- yExt: The height of the metafile image in logical units.
- hMF: The handle to the actual metafile.

```
typedef struct tagMETAFILEPICT {
    LONG mm;       // Mapping mode
    LONG xExt;     // Width of the metafile image
    LONG yExt;     // Height of the metafile image
    HMETAFILE hMF; // Handle to the metafile
} METAFILEPICT;
```

Declare a variable, fill its fields and then use it for clipboard operations or other purposes.

```
 9
10  METAFILEPICT mfp;
11  mfp.mm = MM_ANISOTROPIC;
12  mfp.xExt = 100;
13  mfp.yExt = 50;
14  mfp.hMF = hMetafile; // Assuming hMetafile holds a valid metafile handle
```

Clipboard Interaction:

Copying to Clipboard:

- Create a METAFILEPICT structure.
- Set the mapping mode, width, and height based on metafile properties.
- Store the metafile handle within the structure.
- Place the METAFILEPICT structure on the clipboard.

Pasting from Clipboard:

- Retrieve the METAFILEPICT structure from the clipboard.
- Access the metafile handle, mapping mode, width, and height.
- Adjust viewport extents and mapping mode as needed before playback.
- Play the metafile using PlayMetaFile.

Benefits:

Size Information: Provides recipient programs with metafile dimensions for scaling and layout decisions.

Mapping Mode Flexibility: Allows recipient programs to control scaling and coordinate mapping for optimal rendering.

Enhanced Clipboard Compatibility: Ensures seamless metafile exchange across different applications.

Key Considerations:

Enhanced Metafiles (EMF): Modern applications often favor EMFs for expanded features and compatibility.

Performance: Consider alternative clipboard formats (e.g., bitmaps) for large metafiles or performance-critical scenarios.

Compatibility: Be mindful of potential compatibility issues when working with older metafile formats and clipboard operations.

UNDERSTANDING MAPPING MODES AND VIEWPORTS:

Mapping Modes: Define how logical coordinates (used in drawing functions) map to device coordinates (pixels on screen).

Viewport: The rectangular region on a device context where images are displayed.

Window Ext and Viewport Ext: Control the mapping between logical and device coordinates.

METAFILEPICT's Role in Metafile Exchange:

Bridging the Gap: Acts as a container for metafile handles and essential information, facilitating seamless clipboard exchange.

Structure: Contains fields for mapping mode (mm), image dimensions (xExt, yExt), and the metafile handle itself (hMF).

Interpreting xExt and yExt:

Non-Isotropic/Anisotropic Modes: Directly indicate image size in units of the specified mapping mode.

Isotropic/Anisotropic Modes: Serve as guides for recipient programs to set appropriate viewport extents, ensuring proper scaling and aspect ratio preservation.

Recommendations for Metafile Creation:

Favor Isotropic/Anisotropic Modes: Grant recipient programs greater flexibility in adjusting scaling and mapping.

Set Explanatory xExt and yExt Values: Provide informative hints for viewport adjustments.

Avoid Internal SetViewportExtEx Calls: Ensure compatibility and adaptability for recipient programs.

Recipient Program Responsibilities:

Read METAFILEPICT Fields: Extract mapping mode, dimensions, and handle.

Set Mapping Mode and Viewport Extents: Based on display surface and desired rendering behavior.

Play Metafile: Using PlayMetaFile, respecting the communicated mapping and sizing information.

Additional Considerations:

Window Extent: Always stored within metafiles using Isotropic/Anisotropic modes, governing drawing coordinates within the metafile itself.

Alternative Formats: Consider bitmaps or Enhanced Metafiles (EMFs) for large metafiles or frequent clipboard operations, as they may offer performance and compatibility advantages.

Compatibility: Be mindful of potential issues when working with older metafile formats and clipboard operations.

Mapping Modes and Their Nuances:

MM_TEXT: Ideal for text and simple graphics where precise pixel alignment is crucial.

MM_LOMETRIC and MM_HIMETRIC: Well-suited for measurements and technical drawings in centimeters or millimeters.

MM_TWIPS: Useful for ensuring consistent appearance across devices with different resolutions, often employed in desktop publishing.

MM_ISOTROPIC: Excels in maintaining aspect ratios for images and graphics that must scale proportionally, preventing distortions.

MM_ANISOTROPIC: Offers maximum flexibility in scaling, allowing independent control over X and Y dimensions, but requires careful aspect ratio management to avoid unintended distortions.

Image Dimensions and Communicating Intent:

Non-Isotropic/Anisotropic Modes: xExt and yExt directly convey image size in specific units, aiding recipient programs in accurately allocating display space.

MM_ANISOTROPIC:

- Zero values signal complete freedom for the recipient to determine size and aspect ratio.
- Positive values propose a size in MM_HIMETRIC units, providing a visual suggestion while allowing for adjustments.

MM_ISOTROPIC:

- Positive values suggest both size and aspect ratio, promoting consistency in rendering.
- Negative values convey aspect ratio prioritization while leaving size determination to the recipient, fostering adaptability to different display contexts.

Internal Metafile Calls and Recipient Control:

SetWindowExtEx and SetWindowOrgEx: Establish logical coordinates within the metafile for MM_ISOTROPIC and MM_ANISOTROPIC, ensuring proper drawing interpretation.

Avoiding SetMapMode, SetViewportExtEx, and SetViewportOrgEx: Grants recipient programs greater flexibility in scaling and mapping decisions, aligning with display capabilities and visual preferences.

Memory-Based Metafiles for Clipboard Efficiency:

Offer superior performance and compatibility for clipboard exchange due to their direct accessibility in memory, minimizing overhead and potential file system issues.

Beyond the Basics:

Enhanced Metafiles (EMFs): Consider using EMFs for more complex drawings, enhanced features, and broader compatibility, especially in modern Windows environments.

Bitmaps: Alternatively, use bitmaps for scenarios where absolute pixel accuracy is paramount, or when dealing with raster-based images that cannot be adequately represented as vector graphics.

Compatibility Testing: When working with older metafile formats or clipboard operations, conduct thorough testing to ensure consistent behavior across different platforms and applications.

Creating the Metafile:

```
// Create a memory-based metafile for efficient clipboard exchange
hdcMeta = CreateMetaFile(NULL);

// If using MM_ISOTROPIC or MM_ANISOTROPIC, set window extents first
if (mappingMode == MM_ISOTROPIC || mappingMode == MM_ANISOTROPIC) {
    SetWindowExtEx(hdcMeta, desiredWidth, desiredHeight, NULL);
}

// Optionally set the window origin (applies to all mapping modes)
SetWindowOrgEx(hdcMeta, windowOriginX, windowOriginY, NULL);

// Perform GDI drawing operations on the metafile device context
// ... (Your drawing code here)

// Close the metafile and obtain its handle
hmf = CloseMetaFile(hdcMeta);
```

Explanation:

- CreateMetaFile(NULL) generates a memory-based metafile, suitable for clipboard operations.
- SetWindowExtEx establishes logical coordinates within the metafile for MM_ISOTROPIC and MM_ANISOTROPIC, ensuring proper drawing interpretation.
- SetWindowOrgEx optionally sets the origin of the drawing space for all mapping modes.
- GDI drawing calls are performed on the metafile's device context (hdcMeta), recording drawing commands.
- CloseMetaFile finalizes the metafile and returns its handle (hmf).

Preparing the METAFILEPICT Structure:

```
// Allocate global memory for the METAFILEPICT structure
hGlobal = GlobalAlloc(GHND | GMEM_SHARE, sizeof(METAFILEPICT));
pMFP = (LPMETAFILEPICT)GlobalLock(hGlobal);

// Fill the structure's fields
pMFP->mm = mappingMode;
pMFP->xExt = suggestedWidth;
pMFP->yExt = suggestedHeight;
pMFP->hMF = hmf;
```

Explanation:

- GlobalAlloc reserves memory accessible to other processes for clipboard exchange.
- GlobalLock obtains a pointer to the allocated memory.
- The METAFILEPICT structure is populated with:
  - ✓ mm: The mapping mode used within the metafile.
  - ✓ xExt: Suggested width (interpreted based on mapping mode).
  - ✓ yExt: Suggested height (interpreted based on mapping mode).
  - ✓ hMF: The handle to the created metafile.

Copying to Clipboard:

```
// Copy the METAFILEPICT structure to the clipboard
OpenClipboard(hwndOwner); // Replace hwndOwner with your window handle
EmptyClipboard();
SetClipboardData(CF_METAFILEPICT, hGlobal);
CloseClipboard();
```

Explanation:

- OpenClipboard grants clipboard access.
- EmptyClipboard clears previous clipboard contents.
- SetClipboardData places the METAFILEPICT structure (and its associated metafile) on the clipboard.
- CloseClipboard releases clipboard access.

Additional Considerations:

Error Handling: Implement robust error checking for functions like CreateMetaFile, GlobalAlloc, and clipboard operations to ensure program stability.

Compatibility: Be mindful of potential compatibility issues when working with older metafile formats and clipboard operations.

Alternative Formats: Consider Enhanced Metafiles (EMFs) for more complex scenarios and broader compatibility.

Retrieving the Metafile from the Clipboard:

Open the Clipboard: Access clipboard contents using OpenClipboard.

Retrieve METAFILEPICT Structure: Obtain a pointer to the METAFILEPICT structure using GetClipboardData(CF_METAFILEPICT).

Extract Information: Extract the following information from the structure:

- mm: The mapping mode used within the metafile.
- xExt: Suggested width (interpreted based on mapping mode).
- yExt: Suggested height (interpreted based on mapping mode).
- hMF: The handle to the metafile itself.

Playing the Metafile:

1.

Set Mapping Mode:

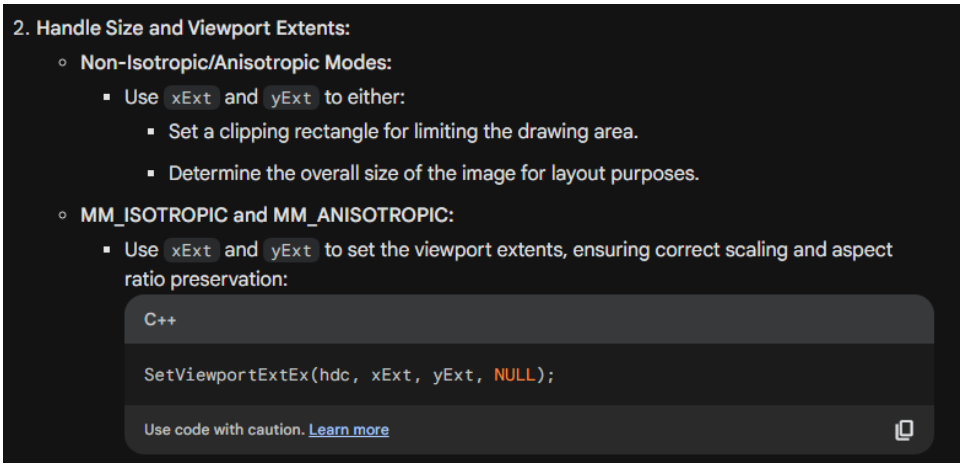
To set the appropriate mapping mode based on the mm value retrieved from the METAFILEPICT structure, you can use the SetMapMode function.

The mm value represents the desired mapping mode, such as MM_TEXT, MM_LOMETRIC, MM_HIMETRIC, MM_LOENGLISH, MM_HIENGLISH, MM_TWIPS, MM_ISOTROPIC, or MM_ANISOTROPIC.

By calling SetMapMode with the appropriate mapping mode value, you can configure the device context for the desired coordinate system and scaling.

2.

Handle Size and Viewport Extents:



When working with non-isotropic or anisotropic mapping modes, the xExt and yExt values from the METAFILEPICT structure can be utilized in different ways:

- Clipping Rectangle: You can use the xExt and yExt values to define a clipping rectangle that limits the drawing area within the metafile. By setting the clipping region using the coordinates (0, 0, xExt, yExt), you can ensure that any drawing operations stay within the specified bounds.
- Image Size and Layout: Alternatively, the xExt and yExt values can be used to determine the overall size of the image contained in the metafile. This information can be useful for layout purposes, allowing you to position and align the metafile image correctly within your application's interface.

By leveraging the xExt and yExt values, you can adapt the drawing environment to the specific requirements of the metafile, ensuring proper mapping and handling of its contents. This flexibility enables you to create accurate and visually appealing representations of the metafile when playing it back or incorporating it into your application.

3.

Play the Metafile:

Use PlayMetaFile to execute the drawing commands stored within the metafile:

```
PlayMetaFile(hdc, pMFP->hMF);
```

4.

Release Resources:

To properly release resources related to the metafile and clipboard, you can follow these steps:

Close Metafile: After you have finished working with the metafile, it is important to finalize it by calling the CloseMetaFile function. This action ensures that any pending operations or modifications to the metafile are completed and that the resources associated with it are properly released.

Close Clipboard: Once you have finished using the clipboard to copy or paste the metafile, it is essential to release access to the clipboard by calling the CloseClipboard function. This action ensures that other programs can access and modify the clipboard contents as needed.

Key Considerations:

- Recipient Control: The recipient program has flexibility in interpreting xExt and yExt to adjust scaling and viewport behavior based on its display capabilities and visual preferences.
- Mapping Mode Impact: The chosen mapping mode significantly affects how logical coordinates in the metafile map to device coordinates on the screen, influencing the final rendering.
- Error Handling: Incorporate robust error checking for clipboard and metafile operations to safeguard program stability.
- Compatibility: Be mindful of potential compatibility issues, especially when working with older metafile formats or across different platforms.

Retrieving the Metafile:

```
OpenClipboard(hwnd);  // Access clipboard contents
hGlobal = GetClipboardData(CF_METAFILEPICT);  // Retrieve metafile handle
pMFP = (LPMETAFILEPICT)GlobalLock(hGlobal);  // Lock memory for access
```

Preparing the Device Context:

```
SaveDC(hdc);  // Preserve current device context settings
SetMapMode(pMFP->mm);  // Set mapping mode based on metafile
```

Handling Size and Viewport Based on Mapping Mode:

Non-Isotropic/Anisotropic Modes:

```
if (pMFP->mm != MM_ISOTROPIC && pMFP->mm != MM_ANISOTROPIC) {
    // Either set a clipping rectangle:
    LPtoDP(hdc, (LPPOINT)&pMFP->xExt, 1);  // Convert logical to device units
    SetClipRect(hdc, 0, 0, pMFP->xExt, pMFP->yExt);

    // Or simply store the size for layout purposes:
    int imageWidth = pMFP->xExt;
    int imageHeight = pMFP->yExt;
}
```

MM_ISOTROPIC and MM_ANISOTROPIC Modes:

```
void SetViewportExtFromMetafile(HDC hdc, LPMETAFILEPICT pMF, int cxClient, int cyClient) {
    int xExt = pMF->xExt;
    int yExt = pMF->yExt;

    if (xExt == 0 || yExt == 0) {  // No suggested size, use client area
        xExt = cxClient;
        yExt = cyClient;
    }

    SetViewportExtEx(hdc, xExt, yExt, NULL);  // Set viewport extents
}
```

Playing the Metafile:

```
PlayMetaFile(hdc, pMFP->hMF);  // Execute drawing commands
```

Releasing Resources:

```
RestoreDC(hdc, -1);  // Restore original device context settings
GlobalUnlock(hGlobal);  // Unlock memory
CloseClipboard();  // Release clipboard access
```

Key Points:

Mapping Mode Significance: The chosen mapping mode dictates how logical coordinates in the metafile map to device coordinates on the screen, significantly impacting rendering.

Recipient Flexibility: Recipient programs can adapt scaling and viewport behavior based on display capabilities and visual preferences.

Units Conversion: LPtoDP is crucial for converting logical coordinates to device units when setting clipping rectangles or interpreting dimensions.

Error Handling: Incorporate error checking for clipboard and metafile operations to ensure program stability.

Compatibility: Be mindful of potential issues when working with older metafile formats or across different platforms.

Retrieving the Metafile from the Clipboard:

OpenClipboard(hwnd): Accesses the clipboard's contents.

hGlobal = GetClipboardData(CF_METAFILEPICT): Retrieves the handle to the METAFILEPICT structure, which holds information about the metafile.

pMFP = (LPMETAFILEPICT)GlobalLock(hGlobal): Locks the memory block associated with the structure to enable access to its data.

Preparing the Device Context for Playback:

SaveDC(hdc): Saves the current device context's settings, ensuring they can be restored later.

SetMapMode(pMFP->mm): Sets the mapping mode of the device context to match the mapping mode used within the metafile. This is crucial for accurate rendering.

Handling Size and Viewport Based on Mapping Mode:

Non-Isotropic/Anisotropic Modes (MM_TEXT, MM_LOMETRIC, MM_HIMETRIC, MM_TWIPS):

Setting a Clipping Rectangle:

LPtoDP(hdc, (LPPOINT)&pMFP->xExt, 1);: Converts xExt and yExt from logical units to device units, ensuring proper scaling and alignment of the clipping rectangle.

SetClipRect(hdc, 0, 0, pMFP->xExt, pMFP->yExt);: Establishes the clipping rectangle to confine the metafile's drawing to the specified area.

Storing Size for Layout:

Retrieves xExt and yExt to determine the overall size of the image for layout purposes within your application.

MM_ISOTROPIC and MM_ANISOTROPIC Modes:

SetViewportExtFromMetafile Function:

Responsible for determining appropriate viewport extents based on the metafile's suggestions and available client area:

If xExt or yExt is zero, it defaults to using the client area's dimensions for viewport extents.

Otherwise, it employs the provided xExt and yExt values.

SetViewportExtEx(hdc, xExt, yExt, NULL);: Sets the viewport extents, controlling how logical coordinates in the metafile map to device coordinates on the screen.

Playing the Metafile:

PlayMetaFile(hdc, pMFP->hMF): Executes the drawing commands stored within the metafile, rendering the image onto the device context.

Releasing Resources:

RestoreDC(hdc, -1): Restores the original device context settings, ensuring subsequent drawing operations are unaffected.

GlobalUnlock(hGlobal): Unlocks the memory block associated with the METAFILEPICT structure.

CloseClipboard(): Releases clipboard access, allowing other programs to interact with it.

```
// Retrieving the Metafile from the Clipboard
OpenClipboard(hwnd);
hGlobal = GetClipboardData(CF_METAFILEPICT);
pMFP = (LPMETAFILEPICT)GlobalLock(hGlobal);

// Preparing the Device Context for Playback
SaveDC(hdc);
SetMapMode(pMFP->mm);

// Handling Size and Viewport Based on Mapping Mode
if (pMFP->mm != MM_ISOTROPIC && pMFP->mm != MM_ANISOTROPIC) {
    // Non-Isotropic/Anisotropic Modes: Setting a Clipping Rectangle
    LPtoDP(hdc, (LPPOINT)&pMFP->xExt, 1);
    SetClipRect(hdc, 0, 0, pMFP->xExt, pMFP->yExt);
} else {
    // MM_ISOTROPIC and MM_ANISOTROPIC Modes: Setting Viewport Extents
    SetViewportExtFromMetafile(hdc, pMFP, cxClient, cyClient);
}

// Playing the Metafile
PlayMetaFile(hdc, pMFP->hMF);

// Releasing Resources
RestoreDC(hdc, -1);
GlobalUnlock(hGlobal);
CloseClipboard();

// Function for Handling Viewport Extents in MM_ISOTROPIC or MM_ANISOTROPIC Modes
void SetViewportExtFromMetafile(HDC hdc, LPMETAFILEPICT pMF, int cxClient, int cyClient) {
    int xExt = pMF->xExt;
    int yExt = pMF->yExt;

    if (xExt == 0 || yExt == 0) {
        xExt = cxClient;
        yExt = cyClient;
    }

    SetViewportExtEx(hdc, xExt, yExt, NULL);
}
```

Key Points:

The mapping mode plays a pivotal role in determining how logical coordinates in the metafile translate to device coordinates on the screen, significantly impacting visual output.

Recipient programs have flexibility in adapting scaling and viewport behavior based on their display capabilities and desired visual outcomes.

LPtoDP is crucial for appropriate units conversion when working with logical coordinates.

Always incorporate error handling for clipboard and metafile operations to safeguard program stability.

Be mindful of potential compatibility issues, especially with older metafile formats or across different platforms.

UNDERSTANDING PREPAREMETAFILE:

Central Role: This function meticulously prepares a device context for flawless metafile playback, ensuring faithful visual representation by addressing scaling and aspect ratio concerns.

Mapping Mode Harmony: It initiates its task by establishing alignment between the device context's mapping mode and the metafile's mapping mode. This synchronization is paramount, as mapping modes dictate how logical coordinates within the metafile transform into device coordinates on the screen, profoundly influencing rendering.

Addressing Isotropic and Anisotropic Modes:

Flexibility Focus: The function devotes particular attention to MM_ISOTROPIC and MM_ANISOTROPIC modes, as they grant enhanced control over scaling and aspect ratio preservation. This adaptability proves invaluable for metafiles necessitating precise visual consistency across diverse display environments.

```c
void PrepareMetaFile(HDC hdc, LPMETAFILEPICT pmfp, int cxClient, int cyClient)
{
    int xScale, yScale, iScale;
    SetMapMode(hdc, pmfp->mm);

    if (pmfp->mm == MM_ISOTROPIC || pmfp->mm == MM_ANISOTROPIC)
    {
        if (pmfp->xExt == 0)
        {
            SetViewportExtEx(hdc, cxClient, cyClient, NULL);
        }
        else if (pmfp->xExt > 0)
        {
            SetViewportExtEx(hdc,
                pmfp->xExt * GetDeviceCaps(hdc, HORZRES) /
                GetDeviceCaps(hdc, HORZSIZE) / 100,
                pmfp->yExt * GetDeviceCaps(hdc, VERTRES) /
                GetDeviceCaps(hdc, VERTSIZE) / 100,
                NULL);
        }
        else if (pmfp->xExt < 0)
        {
            xScale = 100 * cxClient * GetDeviceCaps(hdc, HORZSIZE) /
                GetDeviceCaps(hdc, HORZRES) / (-pmfp->xExt);
            yScale = 100 * cyClient * GetDeviceCaps(hdc, VERTSIZE) /
                GetDeviceCaps(hdc, VERTRES) / (-pmfp->yExt);
            iScale = min(xScale, yScale);

            SetViewportExtEx(hdc,
                -pmfp->xExt * iScale * GetDeviceCaps(hdc, HORZRES) /
                GetDeviceCaps(hdc, HORZSIZE) / 100,
                -pmfp->yExt * iScale * GetDeviceCaps(hdc, VERTRES) /
                GetDeviceCaps(hdc, VERTSIZE) / 100,
                NULL);
        }
    }
}
```

Interpreting Extent Values:

Three-Pronged Approach: The function carefully examines the extent values (xExt and yExt) within the metafile to determine appropriate viewport extents.

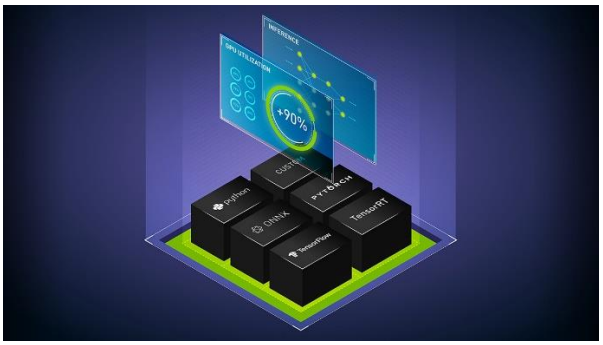
No Size or Aspect Ratio Guidance: If xExt is 0, it means there are no suggested dimensions. The function adjusts the viewport extents to match the dimensions of the provided client area, ensuring seamless integration with the surrounding content.

Explicit Size Recommendation: When xExt is greater than 0, it indicates a suggested image size specified in 0.01mm units. The function uses the GetDeviceCaps function to determine the screen's resolution and pixel density. It then calculates the viewport extents that align with the intended dimensions of the metafile.

Aspect Ratio Preservation Priority: In cases where xExt is less than 0, the metafile prioritizes maintaining the aspect ratio while deferring the determination of the size. The function calculates scaling factors (xScale and yScale) that ensure the desired aspect ratio is preserved, resulting in proportional rendering. It then establishes the viewport extents using these scaling factors and the negative extent values, guaranteeing both accurate scaling and aspect ratio fidelity.

Additional Considerations:

Device Capabilities Awareness: The function's use of GetDeviceCaps demonstrates its ability to adapt to different display environments. It can gracefully handle varying screen resolutions and pixel densities, ensuring consistent results across different systems.



Dual-Purpose Flexibility: The code is designed to handle both suggested size and aspect ratio scenarios, making it versatile enough to accommodate metafiles with different requirements. It can adjust the viewport extents accordingly, ensuring the metafile is displayed correctly.

Mapping Mode Impact: The function emphasizes the importance of synchronizing the mapping mode to achieve the desired visual results. Understanding mapping modes is crucial for controlling how the metafile content is rendered.

Error Handling Imperative: Although not explicitly shown in the provided code, it is essential to implement robust error-checking mechanisms for clipboard and metafile operations. Error handling ensures program stability and resilience by handling potential issues that may arise during these operations.

Compatibility Vigilance: Developers should be cautious about potential compatibility challenges when working with older metafile formats or across different platforms. Thorough testing and adherence to format specifications are key to ensuring successful usage of metafiles across multiple platforms.

Executing the Metafile:

Optional Viewport Origin: After careful preparation, you have the option to set a specific viewport origin using SetViewportOrgEx. This allows you to control where the image will be positioned within the client area, enabling precise placement.

Metafile Playback: The important PlayMetaFile function takes the spotlight, executing the drawing commands that were recorded in the metafile. It draws the image onto the prepared device context, bringing it to life visually.

Restoring Device Context: Once the metafile has been displayed on the screen, RestoreDC gracefully reverts the device context back to its original state. This ensures that subsequent drawing operations are not affected by the specific settings of the metafile, preserving the integrity of the context.

Releasing Resources:

Memory Unlock: The GlobalUnlock function releases the lock on the memory block that held the metafile data. This allows other programs to access and use that memory region, ensuring efficient utilization of system resources.

Clipboard Closure: The final step involves calling CloseClipboard. This closes the clipboard, allowing other applications to access and interact with its contents. By responsibly releasing the clipboard, resources can be shared harmoniously among different applications.

Enhanced Metafile Considerations:

Effortless Conversion: Windows takes on the responsibility of converting between different metafile formats, such as older formats and enhanced formats. This conversion happens automatically, without requiring manual intervention. It simplifies workflows and promotes compatibility by seamlessly handling the format conversion.

Automatic Translation: When an application places a metafile format on the clipboard, Windows actively translates it into the format requested by the receiving application. This ensures that the data exchange between applications is smooth and avoids any format mismatches.

Viewport Origin Control: Developers can precisely position metafile rendering by strategically adjusting the viewport origin.

Enhanced Metafile Convenience: Windows gracefully handles format conversions for enhanced metafiles, streamlining development efforts.

Resource Management: Releasing memory and closing the clipboard are paramount for responsible resource utilization and system harmony.

```
// (Optionally) Set viewport origin:
if (desiredOriginX != 0 || desiredOriginY != 0) {
    SetViewportOrgEx(hdc, desiredOriginX, desiredOriginY, NULL);
}

// Play the metafile:
PlayMetaFile(pMFP->hMF);

// Restore device context:
RestoreDC(hdc, -1);

// Release resources:
GlobalUnlock(hGlobal);
CloseClipboard();

// Note: For enhanced metafiles, Windows handles format conversions automatically.
```

Key Takeaways:

- Viewport Origin Control: Developers have the ability to precisely position the rendering of metafiles by adjusting the viewport origin as needed.
- Enhanced Metafile Convenience: Windows simplifies development efforts by handling format conversions for enhanced metafiles automatically, saving developers from the manual conversion process.
- Resource Management: Releasing memory and closing the clipboard are important steps for responsible use of system resources and ensuring smooth operation between different applications.

ADVANCED METAFILES(EMF)

32-bit Introduction: Enhanced metafiles (EMFs) emerged in 32-bit Windows environments to address limitations of their predecessors, Windows metafiles (WMFs).

Functionality Expansion: They offer a suite of new functions, data structures, a distinct clipboard format (CF_ENHMETAFILE), and the .EMF file extension.

Header Enhancements: EMFs boast a significantly enriched header that stores a wealth of information, including:

- Device context settings
- Image dimensions
- Size of embedded objects
- Color management data
- Thumbnail preview
- Application-specific data

Playback Optimization: This wealth of header information empowers applications to tailor playback strategies for optimal rendering, ensuring accurate and efficient display.

Compatibility Considerations:

Format Translation: Windows provides functions for converting between EMF and WMF formats, enabling compatibility with older applications. However, conversions might not always be seamless, as WMF lacks support for certain advanced features introduced in EMF.

Basic Procedure:

Metafile Creation: Employ CreateEnhMetaFile to initiate a new enhanced metafile, specifying a device context and optional filename.

Drawing Operations: Execute standard drawing commands within the metafile context, meticulously recording them for later playback.

Metafile Closure: Call CloseEnhMetaFile to finalize the metafile, sealing its contents and returning a handle for subsequent interaction.

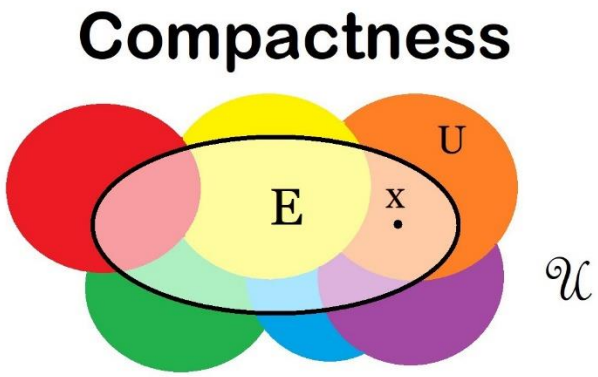
Playback: Utilize PlayEnhMetaFile to render the captured drawing commands onto a device context, bringing the image to life.

Advantages of Enhanced Metafiles:

Advanced Graphics Support: EMFs gracefully accommodate a broader spectrum of graphics features, including paths, regions, and enhanced text handling.

Scalability: They excel in preserving image quality across diverse display resolutions and output devices.

Compactness: EMFs often boast smaller file sizes compared to WMFs, conserving storage space.



Extensibility: Their extensible structure invites application-specific data inclusion, fostering customization and flexibility.

Stay tuned for code examples to illustrate these concepts in action!

EMF1.C PROGRAM

Window Class Registration (WinMain):

Clarity and Efficiency: The program carefully registers a window class named "EMF1" with specific attributes and behaviors. This registration process allows smooth interaction with the Windows operating system, serving as the foundation for creating and handling windows and messages.

Default Values: By using macros like CW_USEDEFAULT and NULL for window size and positioning, the program follows a common practice of accepting default values provided by the system. This approach often simplifies development and ensures compatibility with various screen resolutions.

Message Loop (WinMain):

Responsiveness Cornerstone: The message loop serves as the central component of any Windows application. It continuously listens for and responds to various events, including user input, window resizing, and system notifications. This perpetual cycle ensures that the program remains responsive and adaptable to changing conditions.

Enhanced Metafile Creation (WndProc, WM_CREATE):

Device Context Canvas: The CreateEnhMetaFile function creates a special canvas designed to capture and store graphics commands. This device context serves as the foundation for drawing visual elements within the enhanced metafile.

Drawing Commands: The program uses functions like Rectangle, MoveToEx, and LineTo to skillfully draw a rectangle and two diagonal lines within the metafile. These basic drawing commands are the essential tools for creating visual elements in Windows programming.

Preserving the Masterpiece: CloseEnhMetaFile securely saves the metafile's contents and provides a handle that allows access to the captured artwork. This handle acts as a key, enabling future playback and manipulation of the metafile's visual elements.

Metafile Playback (WndProc, WM_PAINT):

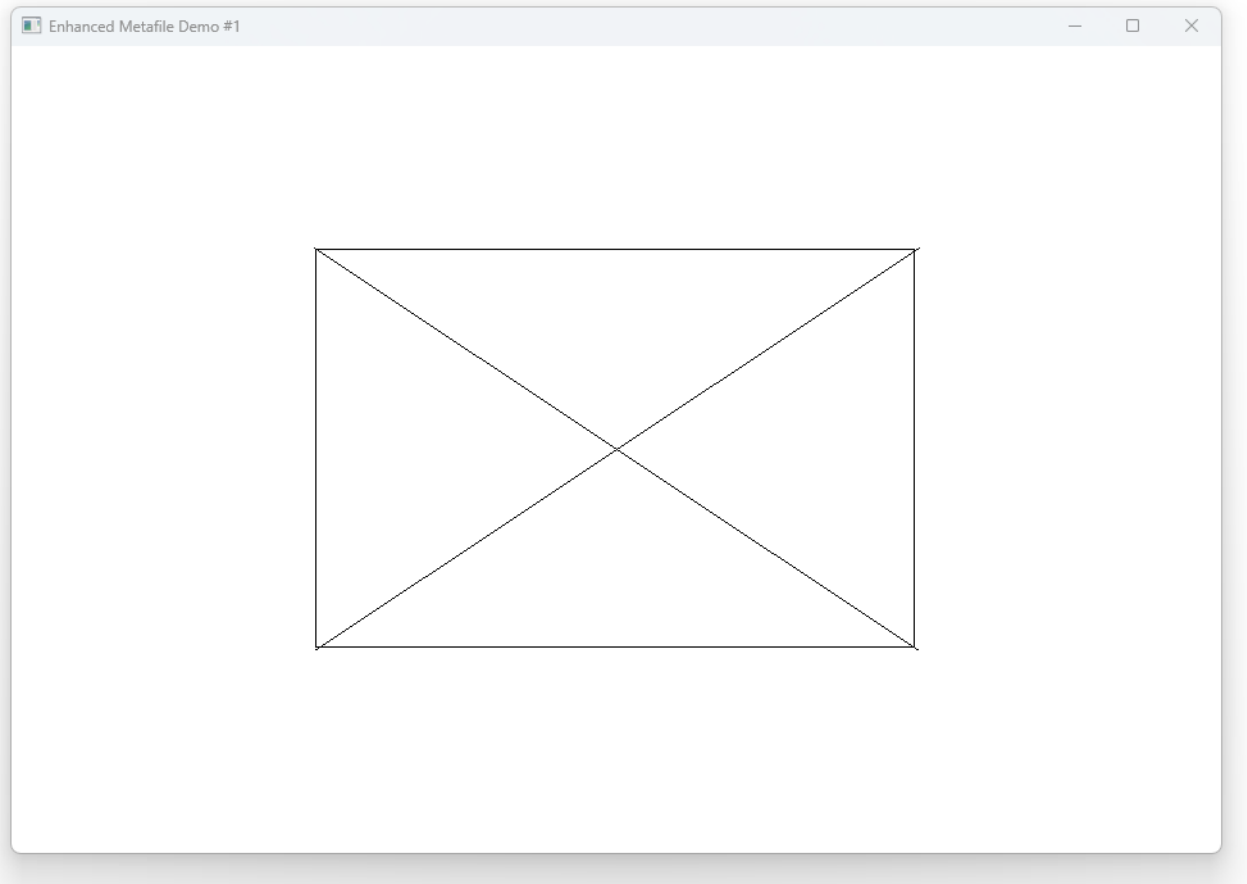
Painting Canvas: The BeginPaint function provides the program with access to a brush that is ready for painting within the window's client area. This brush is applied to a canvas known as the device context, which eagerly awaits the strokes of visual expression.

Focused Display: The program intentionally limits the playback area to the central region of the window, ensuring that the visual experience is concentrated and focused. This attention to detail demonstrates the ability to position and scale metafile content within a window.

Reawakening the Art: PlayEnhMetaFile takes the spotlight, skillfully replaying the recorded drawing commands onto the device context. This process brings the rectangle and lines to life on the window's canvas, showcasing the enduring capability of metafiles to preserve and reproduce visual elements.

Resource Conservation (WndProc, WM_DESTROY):

Return to the Void: The program responsibly frees up the resources associated with the metafile using DeleteEnhMetaFile. This ensures proper memory management and a clean exit from the program. Cleaning up resources is crucial for maintaining system performance and preventing memory leaks.



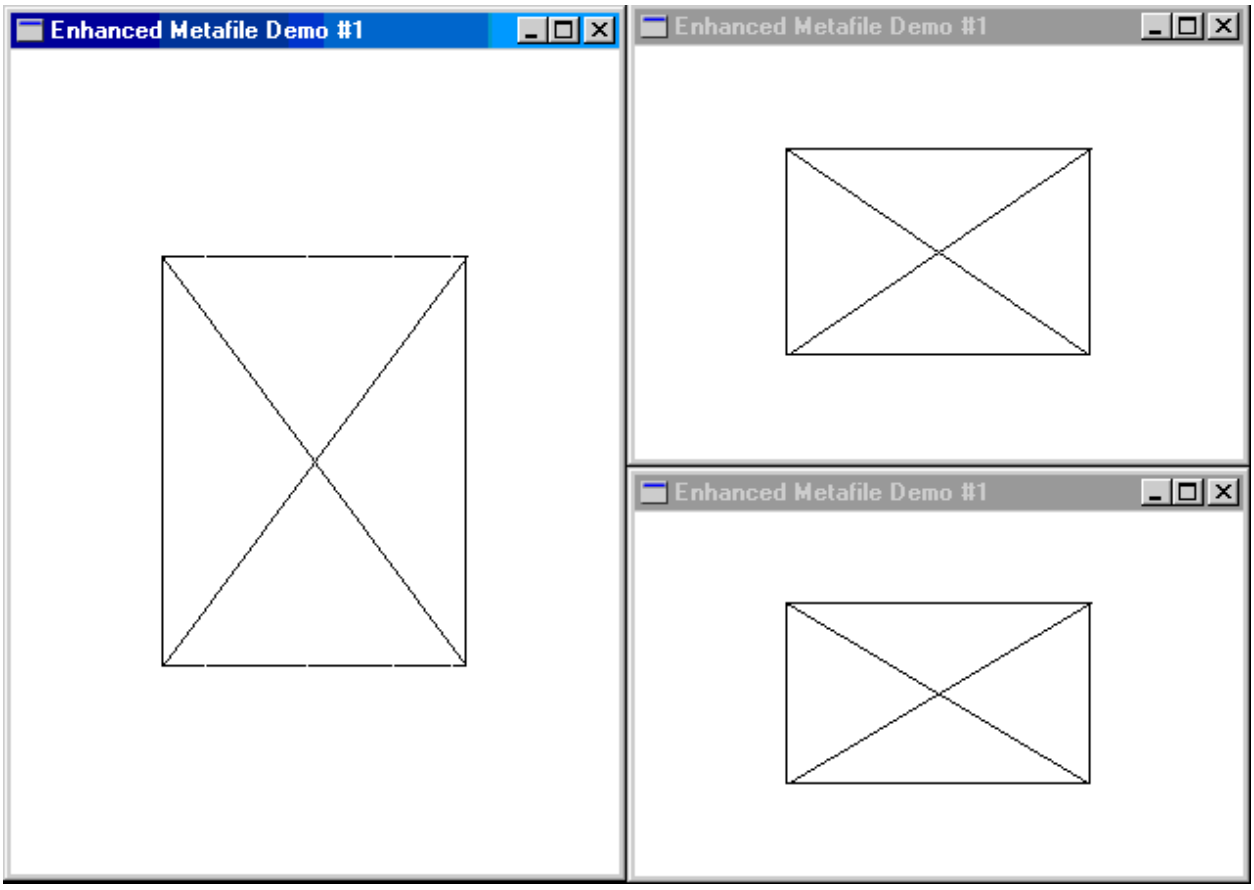
Metafile Creation (WM_CREATE):

Initiation: The CreateEnhMetaFile function begins the creation of the metafile, acting like a magical brush. While the arguments are currently set to NULL, they can be customized for more advanced usage later.

Canvas for Drawing: The CreateEnhMetaFile function returns a special device context handle that serves as a dedicated canvas for drawing within the metafile.

Capturing Artistry: The program skillfully draws a rectangle and two diagonal lines on this special canvas. These drawings are then translated into binary form and stored within the metafile, preserving the visual expression.

Sealing the Masterpiece: CloseEnhMetaFile securely finalizes the metafile's contents, providing a valuable handle that allows access to the captured artistry. This handle acts as a key to unlock and explore the metafile's contents in the future.



Metafile Playback (WM_PAINT):

Window Dimensions: The program carefully determines the size of its client window using a RECT structure, which is similar to an architect's blueprint.

Centered Focus: The program adjusts the blueprint to create a rectangle that occupies the central area of the window, ensuring a focused viewing experience.

Reawakening Beauty: PlayEnhMetaFile takes the spotlight, skillfully replaying the recorded drawing commands onto the window's device context. This process brings the rectangle and lines to life, demonstrating the enduring ability of metafiles to preserve and reproduce visual elements.

Stretching for Perfection: GDI (Graphics Device Interface), the master of graphics, gracefully stretches the image to fit perfectly within the designated rectangle. This ensures a seamless visual presentation and allows metafiles to adapt and maintain their integrity on various canvases.

Beyond the Basics:

- Pathway to Complexity: This program introduces basic concepts of EMF (Enhanced Metafile Format), laying the foundation for exploring more advanced features like gradient fills, transparency, and image embedding. These capabilities provide a wide range of possibilities for creating visually impressive and interactive graphics experiences.
- Cross-Platform Considerations: Although EMFs are well-suited for Windows, it's important to note that other platforms have their own alternative metafile formats, such as SVG (Scalable Vector Graphics). Understanding these differences is essential for designing graphics that can seamlessly work across different platforms.
- Metafiles as Recorders: Enhanced metafiles efficiently record drawing commands and store them as binary data, allowing for playback at a later time.
- Flexible Display: Metafiles can be displayed within specified rectangles, adjusting their proportions to fit different canvases and maintaining their visual integrity.
- GDI's Role: The Graphics Device Interface (GDI) plays a crucial role in both the creation and playback of metafiles. It ensures that the visual elements are harmoniously rendered and provides the necessary functionality for working with metafiles.

Coordinate Flexibility:

The precise coordinates used during metafile creation don't dictate the final displayed size. It's their relative proportions that define the image's shape. This offers flexibility in adjusting coordinates without compromising the image.

Stretching and Distortion:

GDI automatically stretches the image to fit the specified rectangle during playback, potentially causing distortion. This behavior has both advantages and disadvantages:

- Advantage: Ensures complete image filling within a user-defined area, suitable for scenarios like image embedding in word processing.
- Disadvantage: Can compromise aspect ratios crucial for accurate visual representation, such as in police sketches or images requiring precise dimensions.

Maintaining Aspect Ratios and Metrical Sizes:

Techniques exist to preserve aspect ratios or metrical sizes when necessary. These techniques are essential for maintaining visual integrity in specific contexts.

Line-Rectangle Mismatch:

The lines don't perfectly intersect the rectangle's corners due to a Windows metafile coordinate storage issue. This chapter will address potential solutions for this visual discrepancy.

The EMF1 program provides valuable insights into the usage of enhanced metafiles. Here are some key takeaways:

Coordinate Independence: In EMF1, the specific coordinates used in drawing functions like Rectangle and LineTo within the metafile are not crucial. The relationship between the coordinates is what matters in defining the image. Modifying the coordinates uniformly or applying constant offsets produces the same result. It emphasizes that the relative positions of the coordinates are essential rather than their absolute values.

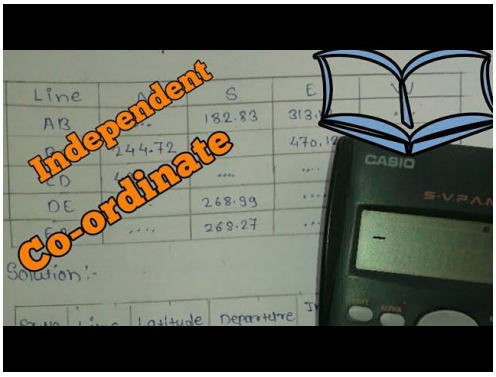
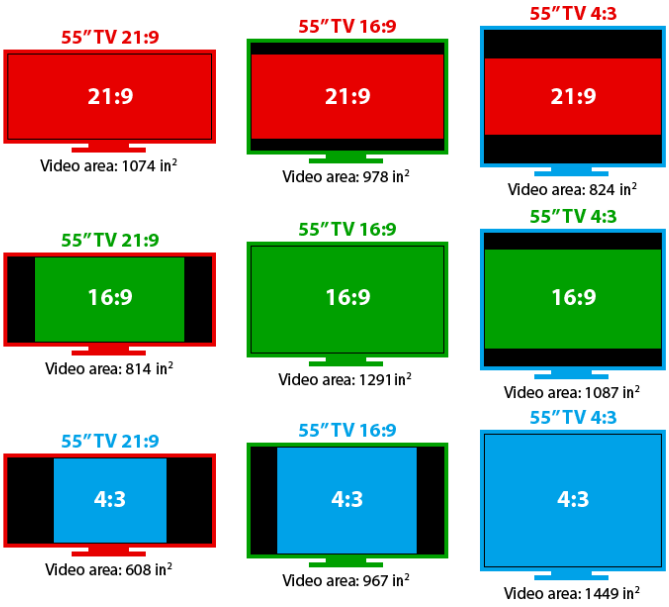
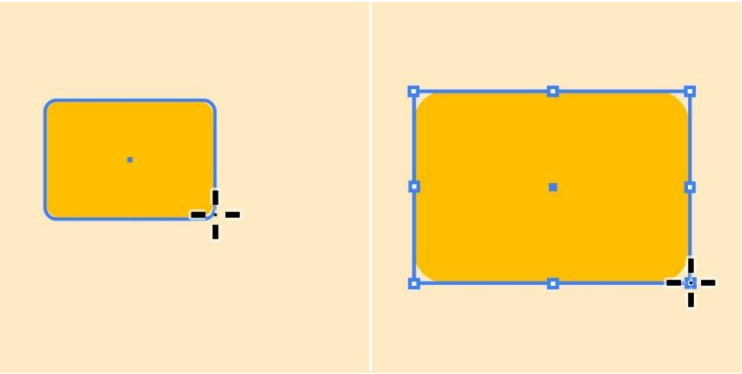


Image Stretching: When playing back the metafile using PlayEnhMetaFile, the image is stretched to fit the specified rectangle. This can sometimes result in distortion, as the original metafile coordinates might imply a square image, but the playback can yield different proportions. However, this flexibility allows for adjusting the image to fit specific rectangles or maintaining the aspect ratio of the original image, depending on the desired visual outcome.

Preserving Aspect Ratio and Size: Depending on the context, it may be important to preserve the aspect ratio or metrical size of the original image. For critical visual information, such as a police sketch, maintaining the original aspect ratio is crucial to accurately represent the subject. In other scenarios, ensuring the image retains a specific size, like two inches high, can be important and should not be altered during playback.



Corner Alignment: A notable issue in the EMF1 program is that the lines drawn in the metafile may not precisely meet the corners of the rectangle during playback. This stems from a problem in how Windows stores rectangle coordinates in the metafile. A solution to this problem can be addressed later by applying appropriate fixes.



In summary, the EMF1 program highlights the flexibility and considerations when working with enhanced metafiles. It demonstrates the independence of coordinates, the ability to stretch and adjust images during playback, the importance of preserving aspect ratios or metrical sizes, and the need to address potential alignment issues.

Key Points:

- Enhanced metafiles offer a robust format for storing and replaying graphical operations, surpassing the capabilities of Windows Metafiles.
- They provide a special device context for recording drawing commands, which are then encoded and stored within the metafile.
- The PlayEnhMetaFile function renders the metafile's contents onto a specified device context, allowing flexible display and scaling.
- The example demonstrates the creation and playback of a simple metafile containing a rectangle and diagonal lines, highlighting the fundamental concepts of using metafiles.
- Image Fidelity vs. Flexibility: Developers must carefully balance the need for visual accuracy with the flexibility of size adjustments, choosing appropriate techniques based on specific use cases.
- Understanding Coordinate Systems: Contextualizing coordinates within metafiles and playback scenarios is crucial for achieving desired visual outcomes.
- Addressing Windows-Specific Issues: Awareness of potential quirks like the line-rectangle mismatch is essential for crafting visually flawless metafile experiences.

EMF2 PROGRAM

Window Creation and Registration:

The program follows standard Windows conventions, creating a window class and registering it to define its visual and behavioral characteristics.

Metafile Handling (WM_CREATE):

Disk-Based Creation: Unlike EMF1, this program uses CreateEnhMetaFile to create a disk-based metafile named "emf2.emf". This allows for easier examination of the metafile's contents without retaining it in memory.

Platform-Specific Adjustment: The program adjusts the rectangle coordinates based on the operating system (Windows 98 or Windows NT) to ensure consistent rendering across platforms.

Drawing Commands: The program replicates the rectangle and line-drawing commands from EMF1, storing them within the metafile.

Immediate Deletion: The metafile is closed using CloseEnhMetaFile and promptly deleted using DeleteEnhMetaFile. This creates the disk-based file for inspection without retaining the metafile handle for later playback within the window.

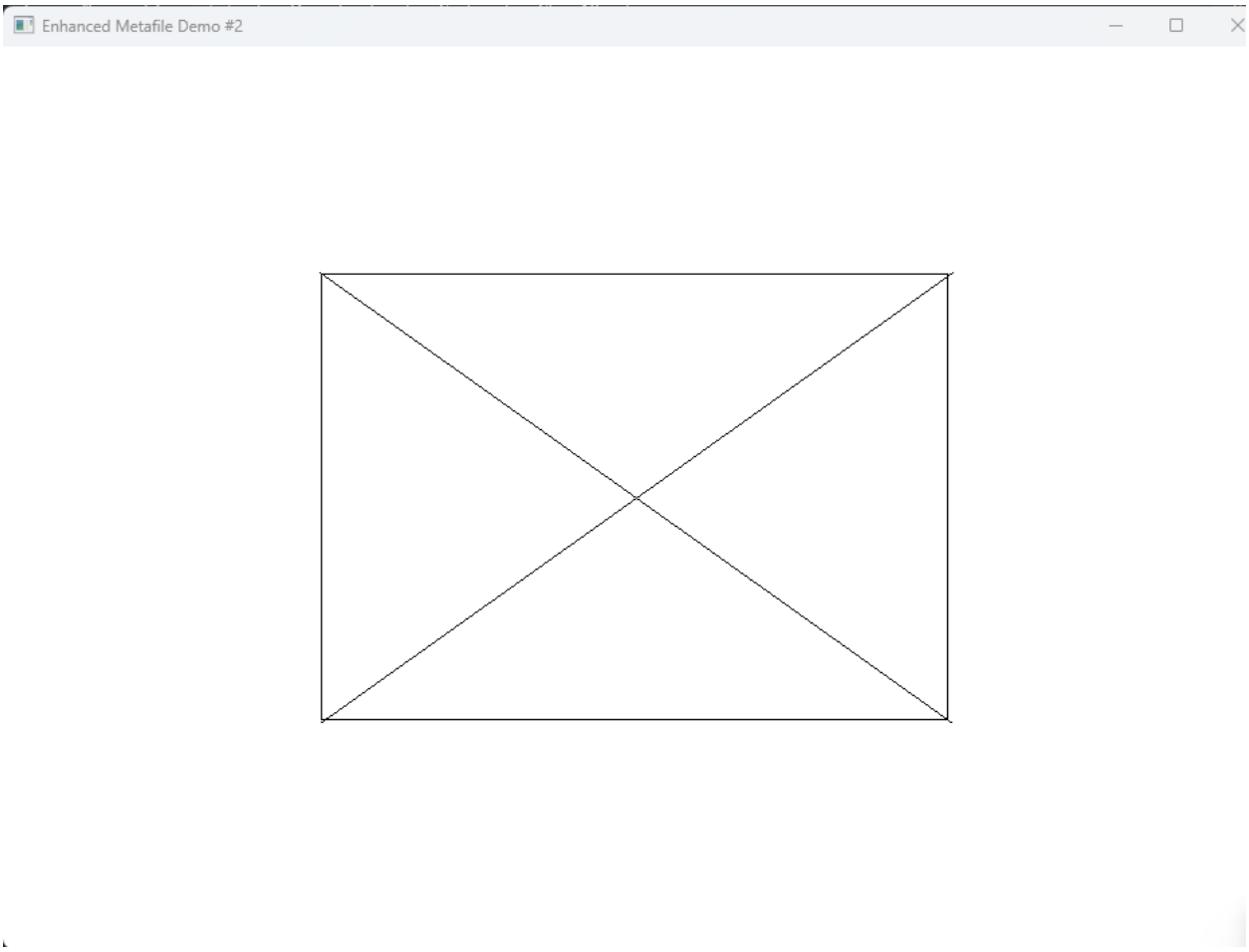
Playback from Disk (WM_PAINT):

Loading from Disk: During the paint cycle, the program retrieves the previously created metafile from disk using GetEnhMetaFile.

Centered Playback: It defines a rectangle that centers the image within the window, similar to EMF1.

Playing the Metafile: The PlayEnhMetaFile function is used to render the contents of the metafile onto the window's device context, bringing the stored drawing commands to life.

Memory Management: The metafile handle is released using DeleteEnhMetaFile to ensure proper memory cleanup.



Key Points:

Focus on Metafile Creation: The primary purpose of this program is to demonstrate the creation and structure of disk-based metafiles, rather than emphasizing their playback within a window.

Disk Storage: The use of a disk-based metafile facilitates examination of its contents to gain a deeper understanding of their format and structure.

Platform-Specific Considerations: The program highlights the importance of accounting for potential rendering differences across operating systems when working with metafiles.

I want you to really understand this subtopic, so am going to be repeating what we just learnt above…

Metafile Creation (WM_CREATE):

Specifying Information:

- The CreateEnhMetaFile function is called with carefully chosen arguments to specify the metafile's nature and metadata:
- Null Device Context: While a device context handle can be provided for metrical information, EMF2 sets it to NULL, relying on GDI's defaults based on the video device context.
- File-Based Metafile: Instead of a memory metafile, EMF2 creates a disk-based metafile named "EMF2.EMF" for later inspection.
- Auto-Calculated Dimensions: The rectangle argument is also NULL, allowing GDI to determine the metafile's dimensions automatically.
- Descriptive Text: The program includes a descriptive text string, identifying the application and image content within the metafile.

Drawing Commands:

- The program executes GDI drawing functions using the metafile device context handle, recording the commands within the metafile.

Metafile Closure and Deletion: Preserving Essence, Releasing Form

- Finalizing the Masterpiece: CloseEnhMetaFile gracefully seals the metafile, providing its handle as proof of its existence.
- Releasing the Scaffolding: DeleteEnhMetaFile quickly removes the metafile's memory-bound form, preserving its essence within the disk-based file, similar to a sculptor discarding molds while keeping the sculpture intact.

Metafile Playback: A Gallery Exhibit

Retrieving from the Vault: During the paint cycle, GetEnhMetaFile acts as a curator, retrieving the metafile from its disk-based gallery and providing its handle for display.

Rendering the Image: A Performance of Light and Color: PlayEnhMetaFile conducts a symphony of stored drawing commands, creating a vibrant display within the specified canvas, bringing the artist's vision to life.

Memory Management: A Respect for Resources: DeleteEnhMetaFile diligently cleans up after the performance, ensuring optimal memory allocation for future exhibitions.

Key Takeaways:

- Disk-Based Persistence: A Legacy of Visuals: The disk-based metafile remains as evidence of the program's artistry, available for examination and reuse even after its initial creation.
- Memory Optimization: A Steward of Resources: The careful deletion of metafile handles reflects a commitment to efficient memory usage, acknowledging the limited resources available.
- Descriptive Metadata: A Guide to the Collection: The embedded text description within the metafile archive helps understand its contents, providing guidance and context.
- Flexible Handling: A Versatile Repertoire: Metafiles adaptively handle creation, storage, retrieval, and playback, offering a dynamic and adaptable approach to managing graphics, much like a versatile theater troupe ready to perform diverse productions.

Accessing the Clipboard's Metafile:

Opening the Clipboard: The OpenClipboard function grants access to the system's clipboard, unlocking its contents for retrieval.

Retrieving the Metafile Picture Structure: GetClipboardData specifically targets data in the CF_METAFILEPICT format, representing a metafile stored on the clipboard.

Locking the Data: GlobalLock secures the retrieved data for exclusive use, preventing conflicts with other applications.

Preserving Device Context and Setting Mapping Mode:

Saving Current Attributes: SaveDC snapshots the current settings of the device context, ensuring they can be restored later for consistent rendering.

Aligning with Metafile's Mapping Mode: SetMappingMode synchronizes the device context's mapping mode to match the metafile's, aligning their coordinate systems for accurate playback.

Handling Clipping and Viewport Extent:

1. Non-Isotropic/Anisotropic Mapping Modes:

If the metafile's mapping mode isn't MM_ISOTROPIC or MM_ANISOTROPIC, a clipping rectangle is set based on the xExt and yExt values from the metafile picture structure.

LPtoDP converts these logical coordinates to device units, ensuring alignment with the physical display.

2. Isotropic/Anisotropic Mapping Modes:

For these modes, xExt and yExt guide viewport extent adjustments.

A provided function (not shown) likely uses cxClient and cyClient to determine appropriate viewport dimensions if xExt and yExt lack size suggestions.

```
void PrepareMetaFile(HDC hdc, LPMETAFILEPICT pmfp, int cxClient, int cyClient) {
    int xScale, yScale, iScale;

    SetMapMode(hdc, pmfp->mm);

    if (pmfp->mm == MM_ISOTROPIC || pmfp->mm == MM_ANISOTROPIC) {
        if (pmfp->xExt == 0) {
            SetViewportExtEx(hdc, cxClient, cyClient, NULL);
        } else if (pmfp->xExt > 0) {
            SetViewportExtEx(hdc,
                pmfp->xExt * GetDeviceCaps(hdc, HORZRES) /
                    (GetDeviceCaps(hdc, HORZSIZE) * 100),
                pmfp->yExt * GetDeviceCaps(hdc, VERTRES) /
                    (GetDeviceCaps(hdc, VERTSIZE) * 100),
                NULL);
        } else if (pmfp->xExt < 0) {
            xScale = 100 * cxClient * GetDeviceCaps(hdc, HORZSIZE) /
                         (GetDeviceCaps(hdc, HORZRES) * -pmfp->xExt);
            yScale = 100 * cyClient * GetDeviceCaps(hdc, VERTSIZE) /
                         (GetDeviceCaps(hdc, VERTRES) * -pmfp->yExt);
            iScale = min(xScale, yScale);

            SetViewportExtEx(hdc,
                -pmfp->xExt * iScale * GetDeviceCaps(hdc, HORZRES) /
                    (GetDeviceCaps(hdc, HORZSIZE) * 100),
                -pmfp->yExt * iScale * GetDeviceCaps(hdc, VERTRES) /
                    (GetDeviceCaps(hdc, VERTSIZE) * 100),
                NULL);
        }
    }
}
```

The function PrepareMetaFile is designed to configure a device context (HDC) for rendering a metafile on a specified area. Let's break down its functionality in more detail:

Setting Mapping Mode:

The function begins by setting the mapping mode of the device context (hdc) based on the value stored in the metafile picture structure (pmfp->mm). The mapping mode determines how logical units are mapped to physical units during drawing operations.

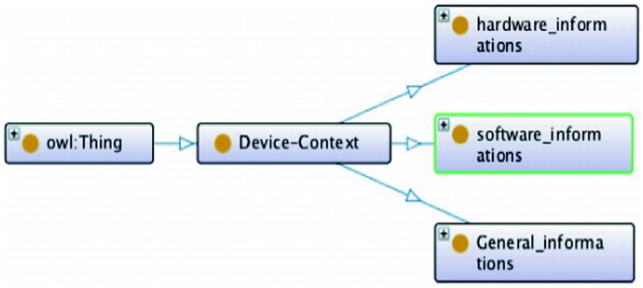
Handling Isotropic and Anisotropic Mapping Modes:

- If the mapping mode is either MM_ISOTROPIC or MM_ANISOTROPIC, the function adjusts the viewport extent to match the dimensions specified in the metafile picture structure (pmfp->xExt and pmfp->yExt).
- If pmfp->xExt is 0, it sets the viewport extent directly to the dimensions of the client area (cxClient and cyClient).
- If pmfp->xExt is positive, it calculates the viewport extent based on the metafile dimensions and the device capabilities.
- If pmfp->xExt is negative, it calculates scaling factors (xScale, yScale) based on the client area and metafile dimensions. The minimum scaling factor (iScale) is determined, and the viewport extent is adjusted accordingly.

Viewport Configuration for Other Mapping Modes:

If the mapping mode is neither MM_ISOTROPIC nor MM_ANISOTROPIC, the function sets up a clipping rectangle for other mapping modes. The clipping rectangle is defined based on the dimensions specified in the metafile picture structure.

In summary, the PrepareMetaFile function dynamically configures the device context to accommodate different mapping modes and adjusts the viewport extent accordingly. It handles isotropic and anisotropic mapping modes by ensuring proper scaling and alignment of the metafile on the specified client area. The function provides flexibility for rendering metafiles in various scenarios by adapting the rendering environment based on the characteristics of the metafile and the device context.



Key Considerations:

Importance of Mapping Mode: Accurate rendering relies on proper synchronization of mapping modes between the metafile and device context.

Viewport Adjustments: For MM_ISOTROPIC and MM_ANISOTROPIC modes, viewport extent modifications ensure proper image scaling and aspect ratio preservation.

Clipping for Other Modes: Non-isotropic/anisotropic modes often employ clipping rectangles to constrain image boundaries.

Data Integrity: Proper locking and unlocking of shared clipboard data safeguards its integrity and prevents conflicts.

```
// Open the clipboard and access the metafile picture structure:
OpenClipboard(hwnd);
hGlobal = GetClipboardData(CF_METAFILEPICT);
pMFP = (LPMETAFILEPICT)GlobalLock(hGlobal);

// Save the device context attributes and set the mapping mode:
SaveDC(hdc);
SetMappingMode(pMFP->mm);

// Handle mapping modes for clipping or viewport extent:
if (pMFP->mm != MM_ISOTROPIC && pMFP->mm != MM_ANISOTROPIC) {
    // Set clipping rectangle for other mapping modes:
    RECT clipRect;
    clipRect.left = 0;
    clipRect.top = 0;
    clipRect.right = pMFP->xExt;
    clipRect.bottom = pMFP->yExt;
    LPtoDP(hdc, (LPPOINT)&clipRect, 2);  // Convert to device units
    SetClipRect(hdc, &clipRect);
} else {
    // Set viewport extent for MM_ISOTROPIC or MM_ANISOTROPIC:
    SetViewportExtEx(hdc, pMFP->xExt, pMFP->yExt, NULL);
    SetViewportOrgEx(hdc, 0, 0, NULL);
    SetWindowExtEx(hdc, cxClient, cyClient, NULL);
    SetWindowOrgEx(hdc, 0, 0, NULL);
}
```

This code is similar to the one above. Both code snippets seem to be addressing similar tasks related to preparing a device context for rendering a metafile.

The second snippet (PrepareMetaFile function) is more structured as a standalone function, making it reusable and potentially easier to understand. The first snippet appears to be a part of a larger program but lacks the encapsulation of a function. The differences are:

Function Name:

The first snippet is part of a code block and doesn't have a distinct function name, while the second snippet is encapsulated in a function named PrepareMetaFile.

Parameter Differences:

The second snippet (PrepareMetaFile function) takes additional parameters (int cxClient, int cyClient) representing the client area dimensions.

Scaling Factor Handling:

In the second snippet, there's a section that calculates scaling factors (xScale, yScale, iScale) based on the metafile's dimensions and the client area dimensions. This scaling is then used to set the viewport extent.

Additional Insights:

Error Handling: Consider implementing error checking for clipboard operations and device context functions to ensure robustness.

Alternative Data Handling: While GlobalLock is used here, GlobalLockZero or other memory management techniques could be explored based on specific needs.

Function Clarity: Providing the referenced function for viewport extent adjustment would enhance code clarity and understanding.

Enhanced Metafile Convenience for the above program:

The code demonstrates working with the older metafile format. Enhanced metafiles offer automatic conversion between formats when placed on or retrieved from the clipboard, simplifying clipboard interactions.

EMF3 PROGRAM

1. Window Creation and Registration:

Creates a standard window class named "EMF3" with white background and default icons/cursor.

Displays an error message if running on a non-Windows NT system, as EMF features might be limited.

2. Handling WM_CREATE Message:

Here's a simplified explanation of the WM_CREATE message handling portion of EMF3.C:

WM_CREATE Message: When the window is created, it receives a special message called WM_CREATE. This message triggers the window's creative process.

Metafile Creation: In response to the WM_CREATE message, the code creates an enhanced metafile. This metafile acts as a canvas to capture and store drawing commands for later playback. It is given the name "emf3.emf" and includes descriptive text.

Selecting Tools: The code chooses a blue brush and a red pen as the tools to paint on the metafile. The brush provides color, while the pen creates outlines and highlights.

Drawing a Rectangle: Using the selected tools, the code draws a rectangle on the metafile. The rectangle's coordinates define its size and position, forming the foundation of the visual composition.

Adding Crossed Lines: To enhance the visual complexity, the code draws two diagonal lines within the rectangle using the red pen. These lines intersect, creating an X shape that adds interest and suggests movement.

Resource Management: The code restores the default pen and brush to conserve resources for future use. This ensures that these tools remain available for other artistic tasks.

Closing and Releasing: The code closes the metafile, finalizing its contents. Instead of keeping it in memory, the code releases the metafile and saves it to disk using the DeleteEnhMetaFile function. This preserves the artwork for future access and exhibitions.

3. Handling WM_PAINT and WM_DESTROY Messages:

Here's a simplified explanation of the WM_PAINT and WM_DESTROY handling in EMF3.C:

WM_PAINT Handling:

Canvas Preparation: When the window receives a WM_PAINT message, the code prepares the canvas by determining the region available for painting, known as the client area.

Creating a Focused Frame: Within the client area, the code creates a smaller rectangle that is centered and covers 50% of the window's dimensions. This focused frame acts as a stage to showcase the artwork.

Unveiling the Metafile: The code retrieves the stored metafile, "emf3.emf", from disk. This brings the artwork back to life in preparation for display.

Playing the Metafile: Using the PlayEnhMetaFile function, the code projects the metafile's contents onto the canvas. This includes the vibrant blue rectangle and dynamic X, captivating the viewer.

Releasing Resources: After the metafile is played, the code gracefully deletes its handle to free up memory. This ensures resources are released for future use.

WM_DESTROY Handling:

Signaling the Finale: When the window receives a WM_DESTROY message, indicating its closure, the code prepares for the program's conclusion.

Graceful Exit: The code sends a message, PostQuitMessage(0), to signal the end of the program. This triggers the application's components to close harmoniously.

Conclusion: With the final message conveyed, the program exits, leaving behind the memory of its artistic expression. The window fades away, concluding its impact on the viewer's experience.

Key Points:

- Disk-Based Metafile Storage: The metafile is created and saved to disk immediately in WM_CREATE, rather than being held in memory for later playback.
- Centered Playback: The metafile is intentionally played back within a smaller centered rectangle, showcasing potential scaling and positioning capabilities.
- Resource Management: The code carefully deletes both pen and brush objects after use, as well as the metafile handle after playback, to avoid memory leaks.
- No Direct Playback: Unlike EMF1 and EMF2, EMF3 doesn't directly play the metafile within WM_CREATE. It saves it to disk and then retrieves it during WM_PAINT for playback.

Insights:

- This approach highlights the flexibility of metafiles for storage and delayed playback.
- It demonstrates drawing basic shapes and lines within a metafile.
- It reinforces the importance of proper resource management in graphics programming.

GDI Function Context Sensitivity:

Device Context-Specific Functions: Some functions in GDI (Graphics Device Interface) are tied to a specific device context. These functions are responsible for directly affecting the visual output, such as drawing shapes and lines. When these functions are used with a metafile device context, their instructions are recorded within the metafile instead of being immediately rendered. This allows the recorded instructions to be played back on different devices or at a later time.

Device Context-Independent Functions: On the other hand, there are functions in GDI that create graphics objects like pens and brushes. These functions operate independently of any specific device context. Instead, they create abstract definitions of these objects within GDI's internal memory. These object definitions are stored and can later be associated with a device context when rendering is required.

Object Creation and Metafile Interactions:

Initial Object Creation: When you call functions like CreateSolidBrush or ExtCreatePen, it doesn't immediately impact the metafile. These functions simply define the objects without any immediate visual effects.

Selecting Objects into Metafile Device Context: The important step comes when you use the SelectObject function to associate a created object with the metafile's device context. This action triggers GDI (Graphics Device Interface) to encode both the instructions for creating the object and the SelectObject call within the metafile. This ensures that a complete record of the visual elements and their attributes is stored in the metafile.

Metafile Content Insights:

Hexadecimal Dump Exploration: By examining the hexadecimal dump of the EMF3.EMF file (Figure 18-8), you can analyze the precise sequence of encoded calls within the metafile. This exploration reveals the specific instructions recorded by GDI, shedding light on how object creation and selection are intertwined within the metafile.

Understanding Metafile Structure: When delving into the structure of the metafile, you gain insights into how GDI meticulously captures and preserves drawing commands. The metafile structure provides a framework that allows these commands to be stored in a format that enables their subsequent playback on different devices and under various rendering conditions.

```
0000  01 00 00 00 88 00 00 00 60 00 00 00 60 00 00 00  ........`...`...
0010  CC 00 00 00 CC 00 00 00 B8 0B 00 00 B8 0B 00 00  ................
0020  E7 18 00 00 E7 18 00 00 20 45 4D 46 00 00 01 00  ........ EMF....
0030  88 01 00 00 0F 00 00 00 03 00 00 00 12 00 00 00  ................
0040  64 00 00 00 00 00 00 00 00 00 04 00 00 00 03 00  d...............
0050  40 01 00 00 F0 00 00 00 00 00 00 00 00 00 00 00  @...............
0060  00 00 00 00 45 00 4D 00 46 00 33 00 00 00 45 00  ....E.M.F.3...E.
0070  4D 00 46 00 20 00 44 00 65 00 6D 00 6F 00 20 00  M.F. .D.e.m.o. .
0080  23 00 33 00 00 00 00 00 27 00 00 00 18 00 00 00  #.3.....'.......
0090  01 00 00 00 00 00 00 00 00 00 FF 00 00 00 00 00  ................
00A0  25 00 00 00 0C 00 00 00 01 00 00 00 5F 00 00 00  %............_...
00B0  34 00 00 00 02 00 00 00 34 00 00 00 00 00 00 00  4.......4.......
00C0  34 00 00 00 00 00 00 00 00 00 01 00 05 00 00 00  4...............
00D0  00 00 00 00 FF 00 00 00 00 00 00 00 00 00 00 00  ................
00E0  25 00 00 00 0C 00 00 00 02 00 00 00 2B 00 00 00  %...........+...
00F0  18 00 00 00 63 00 00 00 63 00 00 00 C6 00 00 00  ....c...c.......
0100  C6 00 00 00 1B 00 00 00 10 00 00 00 64 00 00 00  ............d...
0110  64 00 00 00 36 00 00 00 10 00 00 00 C8 00 00 00  d...6...........
0120  C8 00 00 00 1B 00 00 00 10 00 00 00 C8 00 00 00  ................
0130  64 00 00 00 36 00 00 00 10 00 00 00 64 00 00 00  d...6.......d...
0140  C8 00 00 00 25 00 00 00 0C 00 00 00 07 00 00 80  ....%...........
0150  28 00 00 00 0C 00 00 00 02 00 00 00 25 00 00 00  (...........%...
0160  0C 00 00 00 00 00 00 80 28 00 00 00 0C 00 00 00  ........(.......
0170  01 00 00 00 0E 00 00 00 14 00 00 00 00 00 00 00  ................
0180  10 00 00 00 14 00 00 00                          ........
```

Figure 18–8. *A hexadecimal dump of EMF3.EMF.*

Key Takeaways:

- GDI functions exhibit context sensitivity, with some operating directly on device contexts and others creating device-independent objects.
- Object creation functions like CreateSolidBrush and ExtCreatePen only affect the metafile when used in conjunction with SelectObject to associate objects with the metafile device context.
- Analyzing metafile content, such as through hexadecimal dumps, provides valuable insights into the interplay of GDI functions and object handling within metafiles.

Metafile Growth and Structural Differences:

Expanded Size: EMF3.EMF is larger than EMF2.EMF due to additional records for object creation, selection, and deletion, reflecting the use of custom pens and brushes.

Bound Adjustment: The rclBounds field in the header accounts for a wider pen, shifting the image's boundaries.

Handle Increase: The nHandles field indicates a greater number of GDI object handles in EMF3.EMF, corresponding to the added pen and brush.

Object Creation Records:

Brush Construction: The EMR_CREATEBRUSHINDIRECT record at offset 0x0088 captures the creation of a blue brush with solid fill style, mirroring the CreateSolidBrush call.

Pen Crafting: The EMR_EXTCREATEPEN record (offset 0x00AC) reflects the creation of a thick red pen, aligning with the ExtCreatePen call.

Object Selection and Management:

Assignment to Metafile DC: Records for SelectObject calls integrate these created objects into the metafile device context, ensuring their presence in the visual output.

Resource Stewardship: DeleteObject records demonstrate responsible cleanup of objects after use, preventing potential memory leaks.

Pen Construction and Selection:

The EMR_EXTCREATEPEN record meticulously captures the creation of a thick red pen, mirroring the ExtCreatePen call. The record's structure, including repeated size fields and a PS_GEOMETRIC style, hints at GDI's internal mechanisms for managing pen attributes.

This record is followed by an EMR_SELECTOBJECT call, integrating the pen into the metafile device context and ensuring its visual impact.

Drawing and Restoring Defaults:

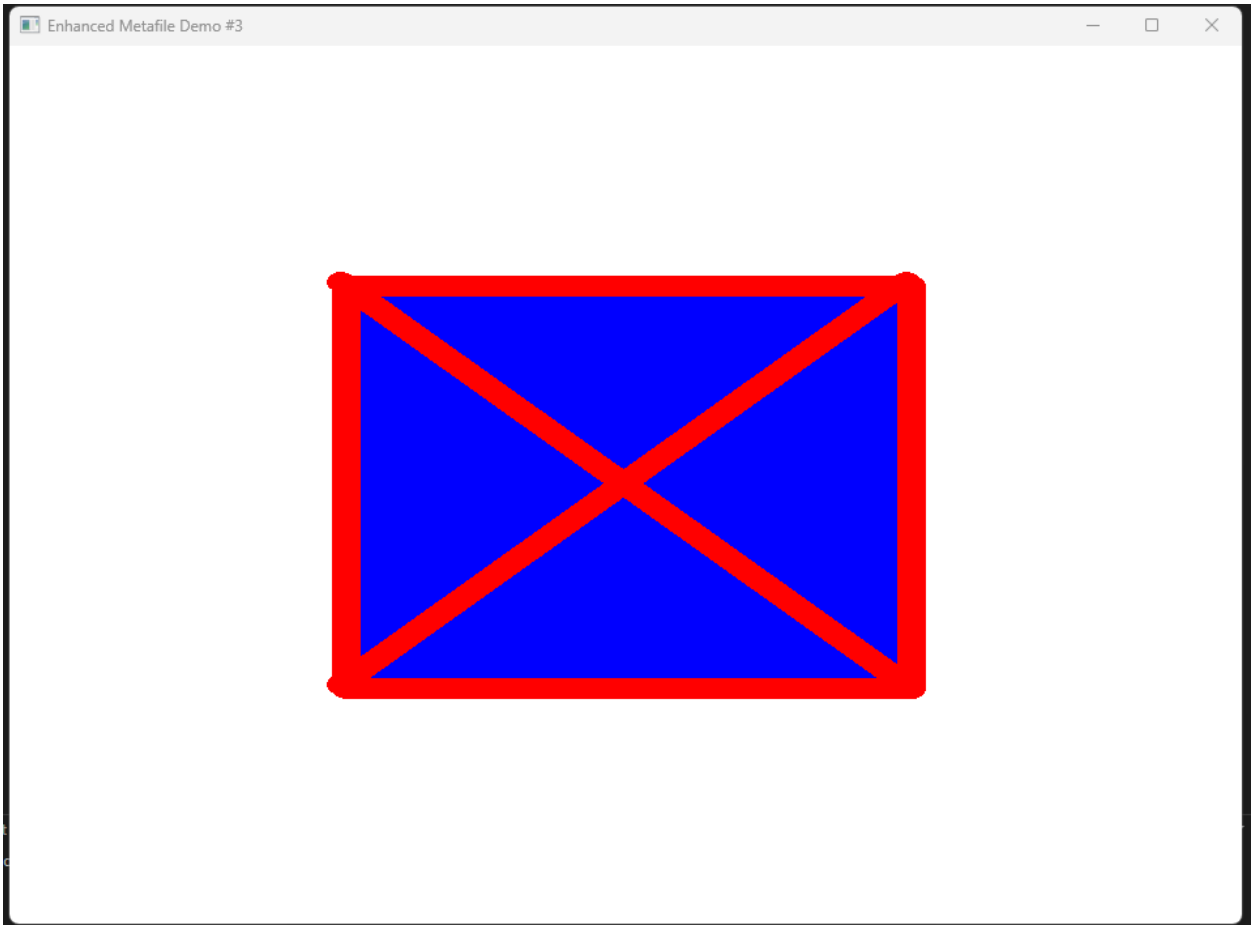
The subsequent records for EMR_RECTANGLE, EMR_MOVETOEX, and EMR_LINETO mirror those found in EMF2.EMF, reflecting the core drawing commands responsible for the visual content.

After drawing, EMR_SELECTOBJECT records with high-bit-set arguments restore stock objects (the black pen and white brush), indicating a return to default drawing settings.

Object Cleanup and Metafile Termination:

EMR_DELETEOBJECT records diligently clean up the custom pen and brush, demonstrating responsible resource management within the metafile and preventing potential memory issues.

The metafile concludes with an EMF_EOF record, signaling its completion and readiness for playback.



Key Takeaways:

- Metafiles serve as comprehensive records of drawing commands, object definitions, and resource management actions for later playback.
- Non-default objects in metafiles receive unique identifiers, facilitating their selection and manipulation.
- Stock objects in metafiles are identified by a high-bit-set flag in EMR_SELECTOBJECT records.
- Analyzing the structure of metafiles provides insights into GDI's internal workings and best practices for object handling.
- Proper object deletion within metafiles is important for efficient resource management and preventing memory issues.

Further Exploration:

- Investigate the impact of custom extended pen styles on metafile structure and playback.
- Research the purpose of repeated size fields within the EMR_EXTCREATEPEN record to gain deeper insights into GDI's implementation.
- Explore how metafile playback interacts with system resources, particularly in regards to object creation and deletion, to gain practical knowledge for graphics programming.

Tell me more. What are some other best practices for efficient resource management within metafiles? How can the unique identifiers assigned to non-default objects in metafiles be used for manipulation? Can you provide more examples of stock objects and their identification in metafiles?

Here are some additional details on resource management, object manipulation using unique identifiers, and examples of stock objects in metafiles:

Best Practices for Efficient Resource Management within Metafiles:

- Properly release and delete objects: When you no longer need an object in a metafile, make sure to release and delete it to free up system resources. This helps prevent memory leaks and improves overall performance.
- Limit unnecessary object creation: Avoid creating unnecessary objects within metafiles. Instead, reuse existing objects whenever possible to minimize resource consumption.
- Use stock objects when suitable: Utilize stock objects provided by GDI, such as stock pens, brushes, and fonts, which are pre-defined objects with shared properties. Stock objects can help reduce the size of the metafile and improve playback efficiency.

Manipulation of Non-Default Objects using Unique Identifiers:

The unique identifiers assigned to non-default objects in metafiles can be used to manipulate these objects during playback. By referencing the identifier, you can select and modify specific objects within the metafile. For example, you can change the properties of a particular brush or pen, or apply transformations to a specific region within the metafile.

Examples of Stock Objects and Identification in Metafiles:

- Stock Pens: GDI provides a set of pre-defined pens, such as the black pen (identified with a specific identifier), which can be used for drawing lines of a specific color and thickness.
- Stock Brushes: There are stock brushes available, such as the white brush, that can be used for filling shapes with a specific color.
- Stock Fonts: GDI offers stock fonts with different styles (e.g., bold, italic) and attributes (e.g., serif, sans-serif) that can be selected and used for text rendering.

EMF4 PROGRAM

Key Functionality:

Integrating Bitmaps into Metafiles: EMF4 showcases the ability to seamlessly incorporate bitmaps into enhanced metafiles, expanding the range of visual elements that can be captured and replayed. This demonstrates the flexibility of metafiles to represent diverse graphical content.

Window Creation and Initialization:

The code begins by registering a window class and creating the main application window, establishing the foundation for visual output.

WM_CREATE Message Handler: Capturing Bitmap Content:

Metafile Creation: A new enhanced metafile called "emf4.emf" is created using CreateEnhMetaFile, which sets it up to record drawing instructions.

Bitmap Retrieval: The standard "close" button bitmap is obtained from system resources using LoadBitmap. This bitmap will be integrated into the metafile.

Bitmap Information: The dimensions of the loaded bitmap are retrieved using GetObject to ensure accurate rendering within the metafile.

Memory Device Context: A temporary canvas, called a memory device context, compatible with the metafile device context, is created using CreateCompatibleDC. This canvas will be used for bitmap manipulation.

Bitmap Selection: The loaded bitmap is associated with the memory device context by using SelectObject, preparing it for transfer to the metafile.

Bitmap Transfer and Recording: The bitmap is carefully copied from the memory device context to the metafile device context at coordinates (100, 100) and resized to 100x100 pixels using StretchBlt. This action is recorded within the metafile, preserving the drawing instruction for later playback.

Resource Cleanup: The memory device context and the bitmap are released using DeleteDC and DeleteObject respectively, demonstrating responsible resource management.

Metafile Finalization: The creation of the metafile is completed using CloseEnhMetaFile, which returns a handle to it. The metafile is then immediately deleted from memory using DeleteEnhMetaFile, but its content, including the captured bitmap, remains intact within the "emf4.emf" file for future playback.

WM_PAINT Message Handler: Recreating the Scene:

Painting Preparation: The painting process begins with BeginPaint, preparing the canvas for visual expression.

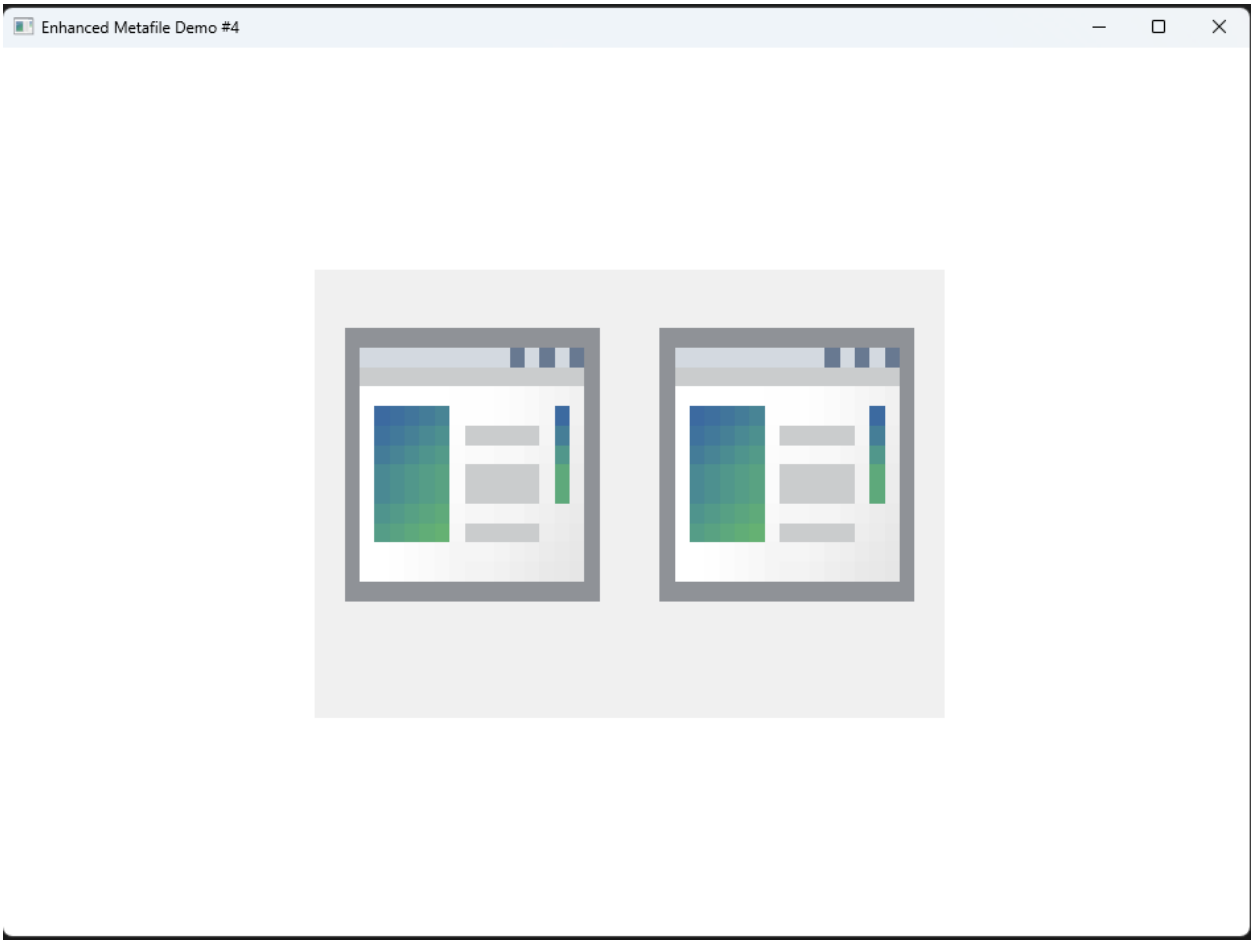
Window Dimensions: The dimensions of the window's client area are obtained using GetClientRect, providing the size of the area where the metafile will be displayed.

Centering the Metafile: The retrieved rectangle is adjusted to position the metafile content at the center of the window, ensuring visual balance and harmony.

Metafile Retrieval: The "emf4.emf" metafile is retrieved from disk using GetEnhMetaFile, ready to release its recorded drawing commands.

Playing the Metafile: The recorded drawing instructions, including the embedded bitmap, are faithfully reproduced within the specified rectangle using PlayEnhMetaFile. This process brings the visual essence of the metafile, including the bitmap, to life on the window.

Resource Management: The metafile is released using DeleteEnhMetaFile, ensuring responsible resource management. The painting process concludes with EndPaint, completing the visual performance.



Key Takeaways:

- Versatility of Metafiles: Metafiles are capable of capturing and storing various graphical elements, including both traditional drawing commands and bitmaps.
- Role of Memory DCs: Memory device contexts (DCs) are valuable intermediaries that facilitate the seamless transfer of bitmaps to metafile DCs for recording. They enable efficient integration of bitmaps into the metafile.
- Persistence of Metafile Content: Even after deletion from memory, metafiles retain their captured content, serving as enduring blueprints for visual reproduction. This means that the visual elements stored within the metafile can be retrieved and replayed at a later time.
- Playback Flexibility: Metafiles offer flexibility in playback, allowing for adjustments in size and position to suit different display contexts. This flexibility enables the metafile content to adapt and be visually expressed in various ways.

Challenges and Solutions:

Capturing Bitmap-Specific Actions: While typical bitmap display involves functions that don't directly operate on the metafile DC, GDI employs clever strategies to record essential information for playback.

Key Strategies:

Embracing Indirect Objects: GDI recognizes that functions like LoadBitmap, GetObject, and CreateCompatibleDC aren't directly applicable to metafile DCs. Instead, it stores the resulting data as indirect objects, which are referenced later when needed.

Preserving Bitmap Data: The bitmap's raw data is encapsulated within the metafile itself, ensuring its availability during playback, even if the original bitmap file isn't present.

Recording Essential Details: GDI meticulously records the bitmap's dimensions and other pertinent attributes for accurate rendering during playback.

Prioritizing Playback Actions: Metafiles primarily focus on storing the drawing commands required to recreate the visual output, rather than capturing the entire process of loading and manipulating bitmaps.

Structure of EMF4.EMF (Insights from Figure 18-10):

Bitmap Data Storage: The metafile embeds the bitmap data itself, ensuring self-containment.

StretchBlt as the Pivot: The StretchBlt call serves as the central recorded action, referencing the stored bitmap data and specifying its placement within the metafile.

Omission of Memory DC Details: The memory DC creation and bitmap selection aren't explicitly recorded, as they aren't directly relevant to visual output.

Playback Process:

Metafile Reconstruction: When a metafile is played back, GDI extracts the embedded bitmap data and recreates it in memory.

Bitmap Rendering: The StretchBlt command then accesses this recreated bitmap and renders it onto the target device context, orchestrating the visual symphony.

```
0000  01 00 00 00 88 00 00 00 64 00 00 00 64 00 00 00  ........d...d...
0010  C7 00 00 00 C7 00 00 00 35 0C 00 00 35 0C 00 00  ........5...5...
0020  4B 18 00 00 4B 18 00 00 20 45 4D 46 00 00 01 00  K...K... EMF....
```

```
0030  F0 0E 00 00 03 00 00 00 01 00 00 00 12 00 00 00  ................
0040  64 00 00 00 00 00 00 00 00 00 04 00 00 00 03 00  d...............
0050  40 01 00 00 F0 00 00 00 00 00 00 00 00 00 00 00  @...............
0060  00 00 00 00 45 00 4D 00 46 00 34 00 00 00 45 00  ....E.M.F.4...E.
0070  4D 00 46 00 20 00 44 00 65 00 6D 00 6F 00 20 00  M.F. .D.e.m.o. .
0080  23 00 34 00 00 00 00 00 4D 00 00 00 54 0E 00 00  #.4.....M...T...
0090  64 00 00 00 64 00 00 00 C7 00 00 00 C7 00 00 00  d...d...........
00A0  64 00 00 00 64 00 00 00 64 00 00 00 64 00 00 00  d...d...d...d...
00B0  20 00 CC 00 00 00 00 00 00 00 00 00 00 00 80 3F  ...............?
00C0  00 00 00 00 00 00 00 00 00 00 80 3F 00 00 00 00  ...........?....
00D0  00 00 00 00 FF FF FF 00 00 00 00 00 6C 00 00 00  ............l...
00E0  28 00 00 00 94 00 00 00 C0 0D 00 00 28 00 00 00  (...........(...
00F0  16 00 00 00 28 00 00 00 28 00 00 00 16 00 00 00  ....(...(.......
0100  01 00 20 00 00 00 00 00 C0 0D 00 00 00 00 00 00  .. .............
0110  00 00 00 00 00 00 00 00 00 00 00 00 C0 C0 C0 00  ................
0120  C0 C0 C0 00 C0 C0 C0 00 C0 C0 C0 00 C0 C0 C0 00  ................
. . . .
0ED0  C0 C0 C0 00 C0 C0 C0 00 C0 C0 C0 00 0E 00 00 00  ................
0EE0  14 00 00 00 00 00 00 00 10 00 00 00 14 00 00 00  ................
```

Figure 18–10. *A partial hexadecimal dump of EMF4.EMF.*

This is useless information that you might not want to read.

Header (0000-002F):

Metafile Signature: The file begins with "01 00 00 00" (little-endian DWORD), identifying it as a Placeable Metafile (PMF).

Bounds and Frame Rectangles: These fields outline the image's spatial dimensions and frame size, crucial for positioning and scaling.

Enhanced Metafile Signature: The 4-byte sequence "20 45 4D 46" designates it as an Enhanced Metafile (EMF).

Version Number: A single byte (01) specifies the EMF format version.

Metafile Description (0030-007F):

Size and Number of Records: These fields convey the metafile's overall size and the quantity of drawing records it contains.

NumPalEntries: A 0 value here indicates the absence of a color palette.

BytesPerPixel: A value of 3 suggests a 24-bit color depth.

Description String: A null-terminated Unicode string identifies the metafile as "EMF4" and includes a "Demo #4" label.

StretchBlt Record (0080-0EE0):

Record Type: The value 0x4D signals a StretchBlt drawing command.

Record Size: A DWORD value (0x00004EE0) denotes the size of this record.

Destination and Source Rectangles: These define the target area for the bitmap within the metafile DC and the source rectangle within the bitmap itself.

Raster Operation (ROP) Code: A value of 0xCC0020 specifies a source copy operation, indicating that pixel data will be directly copied from the bitmap.

Source Bitmap Handle: A value of 0x0D00C0 likely references an indirect object representing the embedded bitmap data.

Bitmap Dimensions: Values of 28x16 pixels suggest the dimensions of the embedded bitmap.

Bitmap Color Format: A value of 1 (0x010020) suggests a 1-bit monochrome format.

Bitmap Data: The extensive sequence of "C0 C0 C0 00" bytes represents the actual bitmap data, stored in a compressed format optimized for metafiles.

End of File Record (0EE0-0EEF):

Record Type: The value 0x0E marks the end of the metafile.

Record Size: A DWORD value (0x00000014) indicates the size of this final record.

Key Observations:

The metafile faithfully captures the StretchBlt operation, incorporating the bitmap data as an indirect object.

The absence of records for LoadBitmap, CreateCompatibleDC, and GetObject aligns with GDI's focus on storing playback-centric drawing commands.

The bitmap's dimensions and color format are meticulously recorded for accurate rendering during playback.

Contextual Insights:

While we don't have the full metafile content, understanding its structure and key elements sheds light on how GDI effectively stores bitmap-related drawing instructions for later reconstruction.

Let’s go in-depth abit more…

Metafiles as Portable Drawing Orchestras:

Conciseness and Flexibility: Metafiles are highly efficient at storing drawing instructions by focusing on the essential elements needed for visual reproduction. They achieve this by using device-independent representations, which allow for seamless portability across various display environments.

Under the Hood of the EMR_STRETCHBLT Record: This important record contains the hidden details of how GDI skillfully captures bitmap-related drawing operations within the compact structure of a metafile.

DIBs: The Cross-Platform Canvas for Bitmaps:

Device Independence for Universal Compatibility: Device-Independent Bitmaps (DIBs) play a crucial role in storing bitmaps within metafiles. Unlike device-specific bitmaps (DDBs), DIBs are not tied to specific devices, allowing for smooth playback across different graphics contexts and hardware configurations.

A Structured Symphony of Pixel Data: Examining the EMR_STRETCHBLT record, we encounter a meticulously organized DIB. It begins with a 40-byte BITMAPINFOHEADER, acting as an introduction, providing important details about the bitmap's dimensions, color depth, and other essential attributes. Following this graceful introduction is the pixel data itself, arranged in a harmonious grid of 22 rows, each consisting of 40 pixels. Each pixel elegantly occupies 4 bytes in this 32-bit-per-pixel masterpiece.

GDI's Playback Strategies: A Dual-Path Encore:

Pathway 1: The Direct Approach with StretchDIBits: In this approach, GDI directly utilizes the embedded DIB data for efficient rendering. The StretchDIBits function takes the lead and skillfully paints the bitmap onto the target device context, playing a central role in this performance.

Pathway 2: The Reconstructed DDB Performance: Alternatively, GDI can choose a more intricate choreography. It begins by using CreateDIBitmap to meticulously reconstruct a device-dependent bitmap (DDB) from the DIB, as if creating a tangible canvas from a blueprint. Then, in a grand finale, GDI employs a memory device context as a temporary stage and expertly executes StretchBlt to coordinate the display of the bitmap. This culminates in a captivating visual symphony.

The Encore: Key Takeaways for a Standing Ovation:

Metafiles: The Ultimate Performance Capture: Metafiles reign supreme as a portable and efficient format for recording and replaying graphical performances, ensuring a consistent visual spectacle across diverse platforms.

DIBs: The Versatile Cross-Platform Canvas: DIBs serve as the metafile's preferred medium for preserving bitmap data, guaranteeing flexibility and adaptability in the face of varied display environments.

GDI: The Maestro of Playback Strategies: GDI gracefully wields multiple techniques to render DIBs during playback, striking a delicate balance between efficiency and device compatibility, ensuring a flawless visual experience for every audience.

Key Takeaways:

- Metafiles ingeniously capture bitmap-related drawing instructions while maintaining a playback-centric approach.
- Indirect objects and embedded bitmap data play crucial roles in preserving visual content for later reproduction.
- Understanding metafile structure and playback mechanisms is essential for effective graphics programming.

EMF5 PROGRAM

Setting the Stage:

Creating a Window: The program starts by crafting a window like a blank canvas, ready for visual expression.

Loading the Metafile: It retrieves a pre-recorded metafile named "emf3.emf," which holds a collection of drawing instructions like a meticulously crafted recipe book.

Responding to the Paint Request:

WM_PAINT Message: When the window needs to be painted (like an artist preparing their tools), the WM_PAINT message springs into action.

Clearing the Canvas: The program begins by clearing the window's drawing surface, creating a fresh starting point.

Defining the Display Area: It carefully outlines a rectangular region within the window, like an art gallery curator choosing the perfect exhibition space.

Unveiling the Masterpiece, One Stroke at a Time:

Enumerating the Metafile: Here's where the magic unfolds! The program embarks on an elegant journey through the metafile, taking each drawing instruction one at a time.

EnumEnhMetaFile Function: This function acts as a guide, leading the program through the metafile's contents.

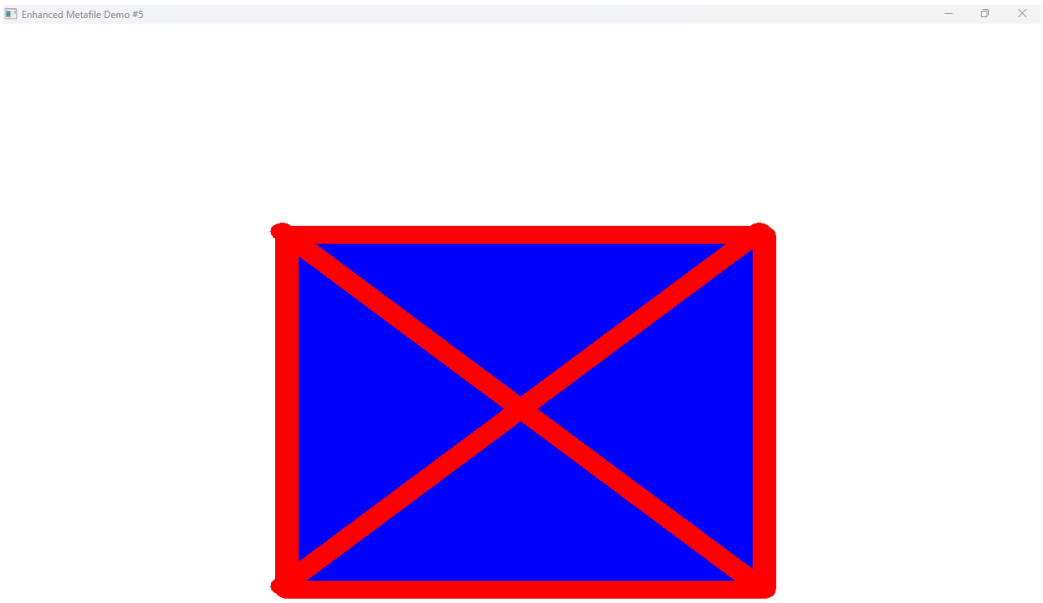
EnhMetaFileProc Callback: With each step, this function is summoned to play back a single drawing record, like an artist carefully applying brushstrokes to a canvas.

PlayEnhMetaFileRecord Function: This function meticulously executes the current drawing command, bringing the visual element to life within the defined display area.

Cleaning Up and Saying Farewell:

Releasing Resources: Once the entire metafile has been rendered, the program gracefully releases its hold on the metafile, like an artist carefully cleaning their brushes.

Responding to Closure: If the user decides to close the window, the WM_DESTROY message gracefully handles the farewell, ensuring a smooth ending to the visual experience.



Key Takeaways:

- Metafile Enumeration: This technique offers a powerful way to interact with metafiles at a granular level, enabling customized playback and even modifications to individual drawing commands.
- Playback Control: By carefully selecting a display area, you can orchestrate how and where the metafile's visual elements unfold, tailoring the artistic experience.
- Resource Management: Remember to release resources like metafiles when they're no longer needed, ensuring a clean and efficient performance.

Playing a Metafile: Two Different Paths:

PlayEnhMetaFile: This is like a conductor leading an orchestra through an entire symphony in one go. It plays back all the drawing instructions in a metafile at once.

EnumEnhMetaFile: This is like a music teacher carefully guiding a student through each note of a piece. It processes a metafile one record at a time, offering more control.

EnumEnhMetaFile: Taking a Closer Look:

Calling the Enumeration Function: When you use EnumEnhMetaFile, it calls a special function (EnhMetaFileProc in this case) for every single record in the metafile, giving you a chance to interact with each instruction.

Playing Each Record: Inside the enumeration function, PlayEnhMetaFileRecord is used to execute individual drawing commands, like a musician playing each note of a musical score.

EMF5: Recreating EMF3's Results with More Control:

Same Output, Different Approach: EMF5 uses EnumEnhMetaFile and PlayEnhMetaFileRecord to achieve the same visual output as EMF3, but with more granular control over the playback process.

Key Benefits of Enumeration:

Examination and Modification: You can inspect and even modify metafile records during enumeration, opening up possibilities for customization and filtering.

Playing a Selected Subset: By selectively passing records to PlayEnhMetaFileRecord, you can choose which parts of the metafile to render, like a DJ mixing tracks from a playlist.

Playing by the Rules:

No Direct Metafile Record Modification: While enumeration gives you access to individual records, it's important to respect their integrity and avoid direct modification. There are other ways to create variations, as we'll see in EMF6.

EMF6 PROGRAM

Setting the Stage:

Creating a Window: The program starts by creating a window, like an empty canvas ready for painting.

Loading the Metafile: It retrieves the "emf3.emf" file, which contains drawing instructions, just like a recipe book.

Responding to the Paint Request:

WM_PAINT Message: When the window needs to be painted, the WM_PAINT message kicks in, like a signal to start the artistic process.

Clearing the Canvas: The program prepares a fresh drawing surface by clearing any previous content.

Defining the Display Area: It outlines a rectangular region within the window, like a curator choosing the perfect spot for an exhibition.

Transforming Rectangles into Ellipses:

Enumerating the Metafile: The program embarks on a journey through the metafile, exploring each drawing instruction one by one.

EnhMetaFileProc Callback: For each record, this function acts as a magic filter, turning rectangles into ellipses. Here's how it works:

Copying the Record: It creates a temporary copy of the current metafile record, like making a photocopy of a recipe.

Identifying Rectangles: It checks if the record holds a rectangle drawing command (EMR_RECTANGLE).

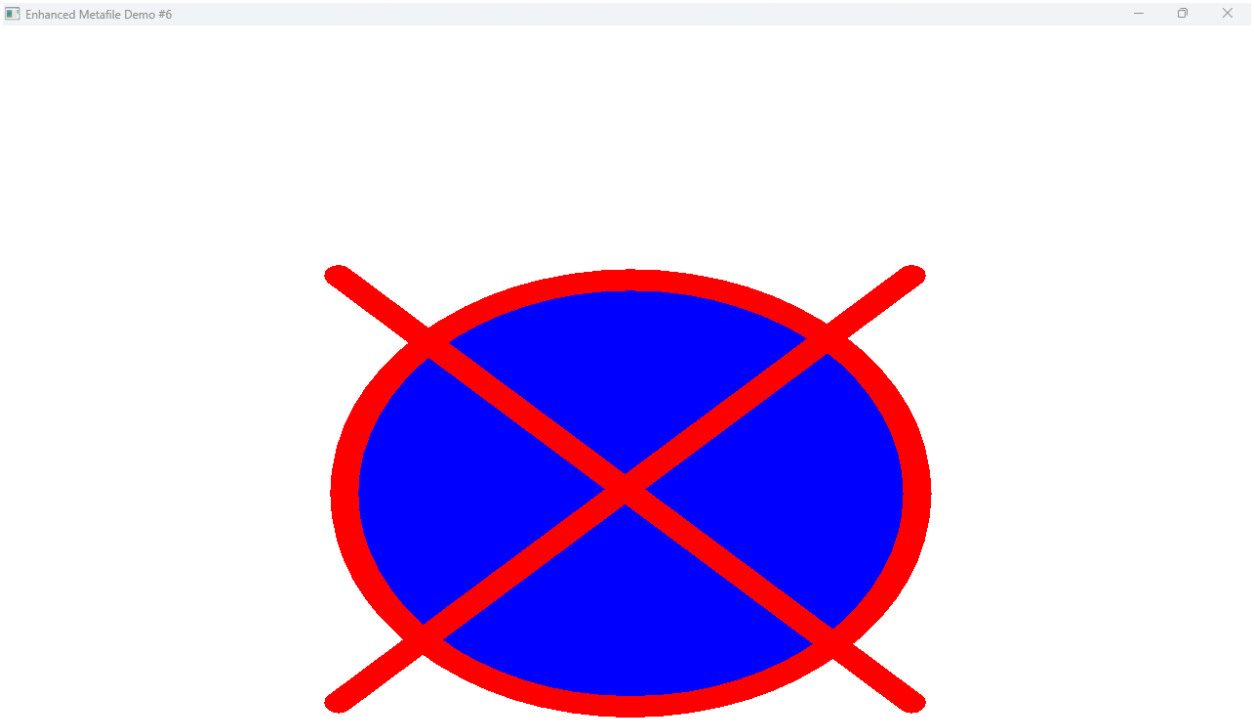
Morphing into Ellipses: If it's a rectangle, the magic happens: the function changes the record type to EMR_ELLIPSE, transforming the rectangle into an ellipse in the recipe.

Playing the Modified Record: The transformed record is then played back, gracefully executing the modified drawing command, like a chef following the altered recipe.

Cleaning Up and Finishing the Masterpiece:

Releasing Resources: Once the modified metafile has been rendered, the program releases its hold on the metafile, like a chef cleaning up after a meal.

Handling Closure: If the user decides to close the window, the WM_DESTROY message gracefully concludes the visual experience.



Modifying Metafiles During Playback:

Copy and Modify: Instead of directly altering original metafile records, which can compromise their integrity, EMF6 demonstrates the effective approach of creating copies and modifying those copies for a safer and more controlled process.

Record Transformation: Specifically, it transforms rectangles into ellipses by:

- Copying the current metafile record.
- Checking if the record type is EMR_RECTANGLE.
- If so, changing the type to EMR_ELLIPSE, effectively morphing the shape.
- Playing back the modified record to render the ellipse.

Adding Records for Extra Effects:

Slip in New Records: You can create even more dynamic effects by inserting additional records during enumeration. For example, you could play back the original rectangle record and then modify it to an ellipse record, resulting in both shapes being drawn.

Managing GDI Objects During Enumeration:

Handle Table: GDI keeps track of GDI objects (like pens, brushes, etc.) used in a metafile through a structure called HANDLETABLE.

Handle Storage: It stores handles to these objects in the objectHandle array within this structure, allowing for efficient retrieval and management during playback.

Array Size: The size of the array is determined by the nHandles field, which is 1 more than the number of GDI objects in the metafile.

First Element: The first element of the array always holds a handle to the metafile itself.

Object Numbering: Each GDI object is assigned a unique object number starting from 1.

Handle Assignment: When an object-creation record (like EMR_CREATEBRUSHINDIRECT) is played back, GDI creates the object and stores its handle in the corresponding element of the objectHandle array.

Handle Retrieval: When a SELECTOBJECT record is encountered, GDI retrieves the object handle from the array based on the object number and uses it to select the object for subsequent drawing operations.

Object Information: You can use functions like GetObjectType and GetObject to access information about the GDI objects stored in the handle table.

Key Takeaways:

Metafile Enumeration: It's a powerful tool for not only playing back metafiles but also modifying and customizing their content during the rendering process.

Record Manipulation: By carefully working with metafile records and GDI objects, you can achieve a wide range of visual effects and transformations.

Creative Possibilities: This flexibility opens up opportunities for dynamic and engaging graphical experiences.

Key Takeaways:

- Metafile Transformations: EMF6 demonstrates how metafile enumeration can be creatively used to modify and transform graphical elements during playback, opening up possibilities for dynamic visual effects.
- Preserving Integrity: Instead of directly modifying the original metafile records, EMF6 makes temporary copies and applies changes to those copies, ensuring the original metafile remains intact.
- Creative Control: This approach highlights the flexibility and potential for artistic expression that can be achieved through metafile enumeration and record manipulation.

Sometimes, I just feel like repeating notes. 👍 🤯

EMF7 PROGRAM

EMF7 does differently compared to EMF3 TO EMF6:

1. Metafile Embedding:

Core Feature: EMF7 demonstrates the concept of embedding images within an existing metafile, a powerful technique for creating composite graphics.

Process: It achieves this by:

- Creating a new metafile device context (hdcEMF).
- Using EnumEnhMetaFile to play back records from the original metafile (hemfOld) onto this new metafile device context.
- Injecting additional drawing commands (like the ellipse in this case) directly onto the metafile device context during enumeration.
- Closing the metafile device context, resulting in a new metafile (emf7.emf) that combines the original content with the embedded image.

2. Selective Rendering:

Customization: EMF7 doesn't simply play back all records from the original metafile. It selectively renders them using the EnhMetaFileProc callback function.

Record Filtering: The callback function filters out the EMR_HEADER and EMR_EOF records, which aren't essential for visual content.

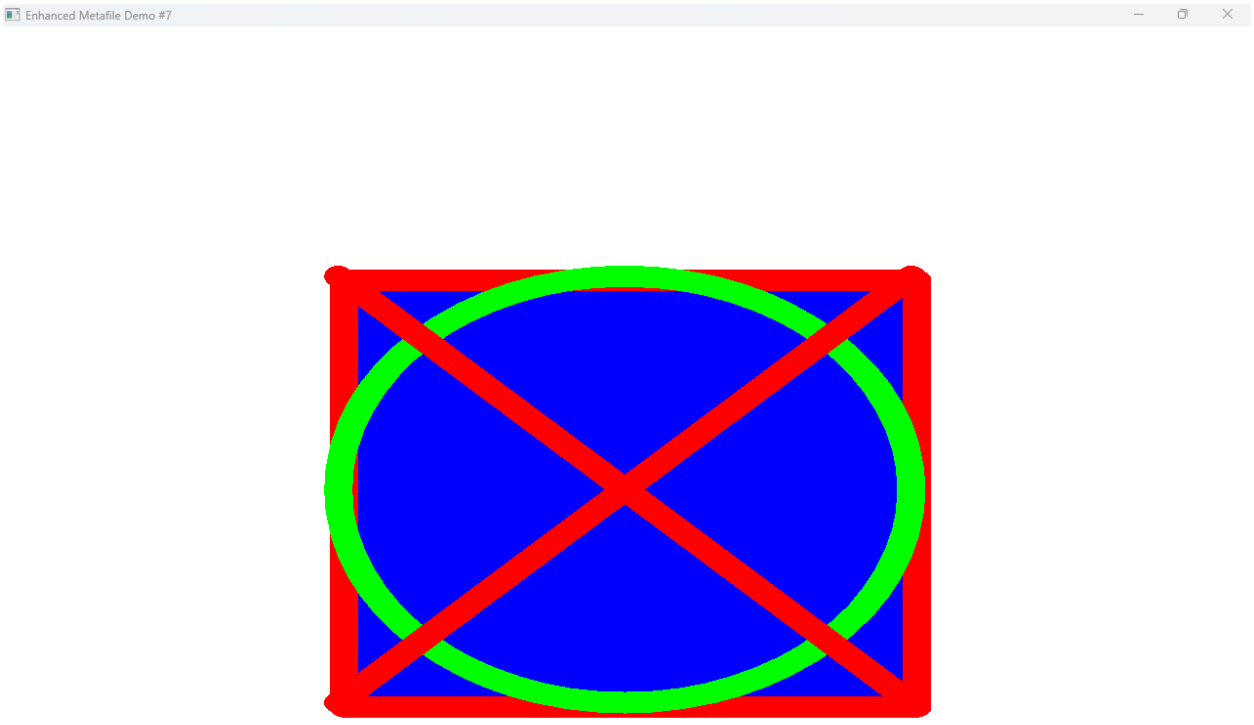
3. Conditional Embedding:

Contextual Insertion: It embeds the ellipse only when a specific condition is met: when a rectangle record (EMR_RECTANGLE) is encountered.

Flexibility: This conditional embedding allows for more tailored modifications and customizations within the metafile.

4. Metafile Preservation:

Non-Destructive: It's important to note that the original metafile (emf3.emf) remains intact. EMF7 creates a new metafile (emf7.emf) that incorporates the embedded image while preserving the original.



Key Differences in EMF7:

Metafile Embedding: It creates a new metafile (EMF7.EMF) that incorporates content from the existing EMF3.EMF, along with an embedded ellipse.

Selective Rendering: It filters out non-essential records during enumeration to streamline the new metafile.

Conditional Embedding: The ellipse is embedded only when a rectangle record is encountered, demonstrating a conditional approach.

Preservation of Original Metafile: EMF3.EMF remains untouched, highlighting the non-destructive nature of metafile embedding.

WM_CREATE Processing:

Retrieve Original Metafile: Obtains a handle to EMF3.EMF using GetEnhMetaFile.

Get Header Information: Retrieves the metafile header using GetEnhMetaFileHeader to access the rclBounds field for later use.

Create New Metafile: Creates a new disk-based metafile named EMF7.EMF using CreateEnhMetaFile.

Enumerate and Embed: Uses EnumEnhMetaFile to:

- Process records from EMF3.EMF.
- Play back selected records onto the new metafile's device context.
- Inject the ellipse drawing commands when a rectangle record is encountered.
- Close and Clean Up: Closes the new metafile using CloseEnhMetaFile and releases handles to both metafiles.

EnhMetaFileProc Callback:

Record Handling:

- Plays back records that aren't EMR_HEADER or EMR_EOF using PlayEnhMetaFileRecord.
- Draws a green ellipse with a transparent interior when a Rectangle record is encountered.

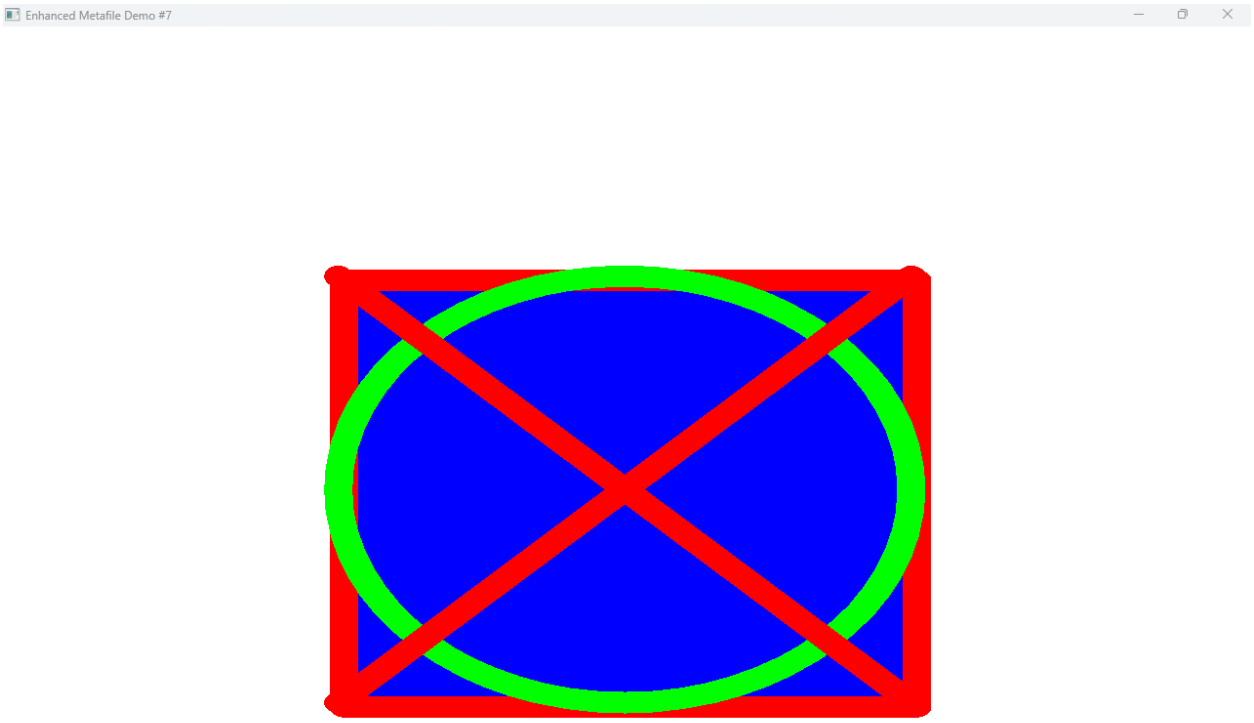
State Management: Saves and restores previous pen and brush handles to ensure context consistency.

WM_PAINT Handling:

Playback: Simply plays back the newly created EMF7.EMF using PlayEnhMetaFile.

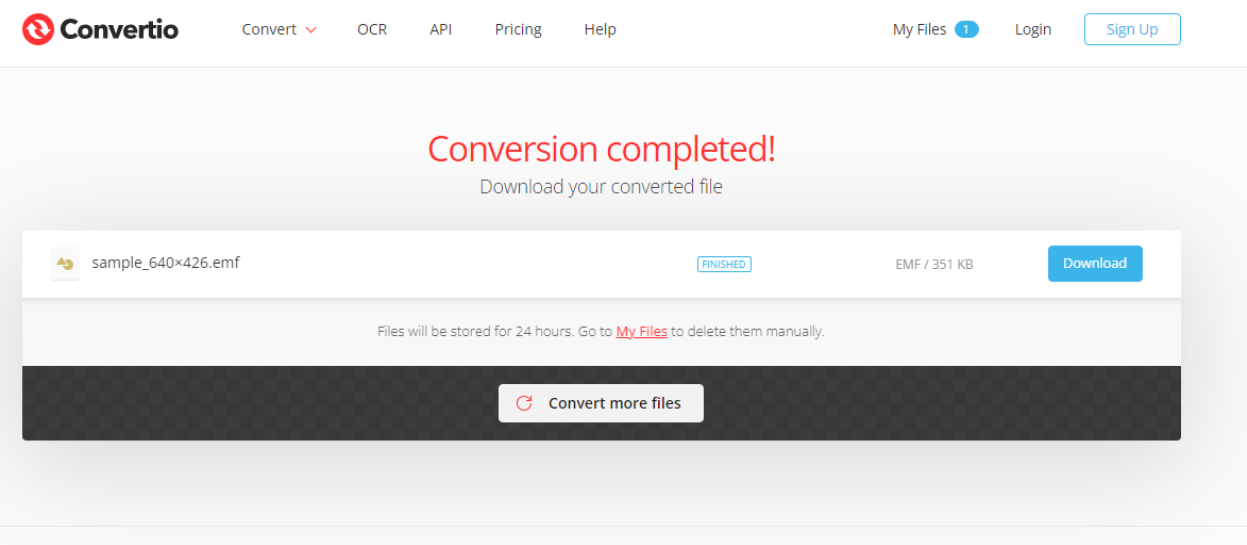
Output:

The display shows the rectangle, followed by the ellipse, and then the crisscrossing lines, confirming the successful embedding of the ellipse within the original content.



EMFVIEW PROGRAM

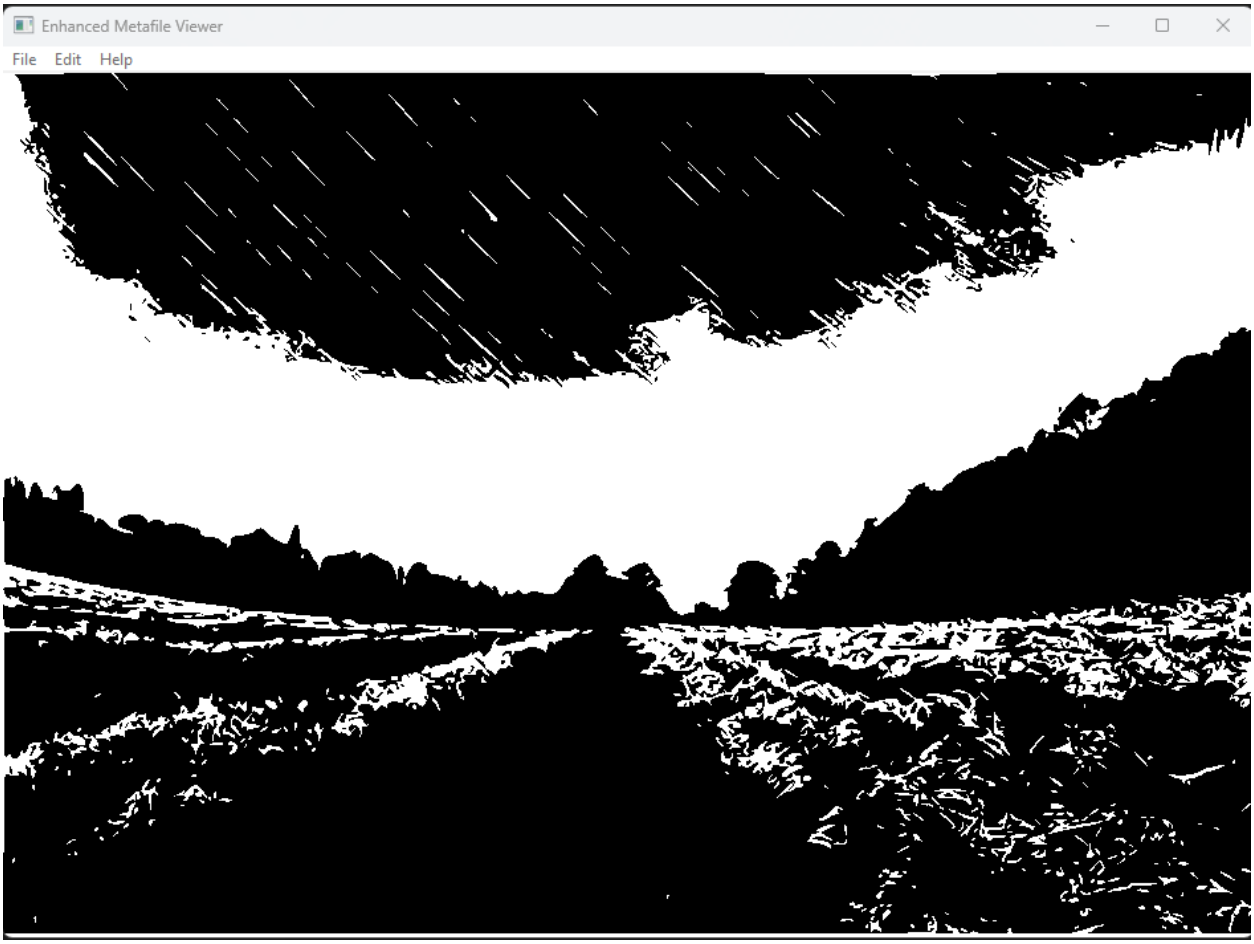
Note the first thing I did was convert the regular .bmp file to .emf file using:



…or any other tool online.

Loaded as a regular .bmp

Loaded as a .emf file:



The about:

The "EMFVIEW" program is an application that allows users to view, manipulate, and print Enhanced Metafiles (EMF) on Windows. An Enhanced Metafile is a file format used to store vector graphics, which means that it can scale without losing image quality. The program is written in C and utilizes the Windows API for its functionality.

Upon launching the program, a window titled "Enhanced Metafile Viewer" appears. The user interface provides various options and features, accessible through a menu bar. Here is an in-depth explanation of the program's key functionalities:

Opening and Viewing Metafiles:

The "File" menu allows users to open existing EMF files through the "Open" option.

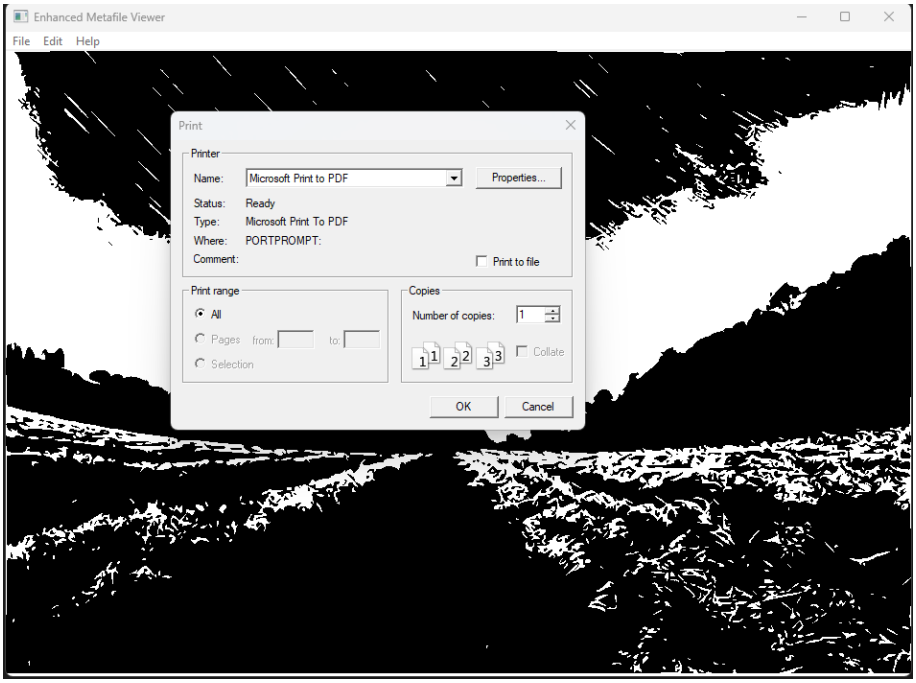
When a file is loaded, the program uses the GetEnhMetaFile function to obtain the enhanced metafile's handle, allowing it to be displayed within the program's window.

Saving Metafiles:

Users can save the currently loaded EMF file or a modified version using the "Save As" option in the "File" menu. This functionality is achieved through the CopyEnhMetaFile function to create a copy of the metafile.

Printing Metafiles:

The program facilitates the printing of metafiles through the "Print" option in the "File" menu. It uses the Windows Printing API, with functions like PrintDlg to obtain the printer device context and StartDoc, StartPage, and EndPage to manage the printing process.



- Initiation: Triggered by the "Print" command in the menu.
- Dialog and Dimensions: Displays the common printer dialog box and obtains printable area dimensions.
- Stretching: Stretches the metafile to fill the printable area, ensuring full utilization of space.
- Window Display: Metafiles are similarly stretched to fit within the program's window.

Viewing Metafile Properties:

The "File" menu includes an option to display properties of the loaded metafile. This information includes details such as bounds, frame dimensions, resolution, size, records, handles, and palette entries.

Clipboard Operations:

The program supports clipboard operations such as copy, cut, paste, and delete for the loaded metafile. It uses functions like CopyEnhMetaFile and SetClipboardData to handle these operations.

User Interface and Interaction:

The application features a graphical user interface with options accessible through menus. It utilizes standard Windows controls and handles user interactions, such as opening files, copying/pasting to the clipboard, and printing.

Palette Handling:

The program includes functionality for handling palettes associated with metafiles. It creates and selects palettes using the CreatePalette function, helping to ensure accurate color representation during printing and viewing.

EMFVIEW ensures accurate color display, even on systems with limited color palettes.

Mechanism:

- Extracts a palette from the metafile using CreatePaletteFromMetaFile.
- Activates the palette with SelectPalette and RealizePalette.
- Handles WM_QUERYNEWPALETTE and WM_PALETTECHANGED messages for optimal color management.

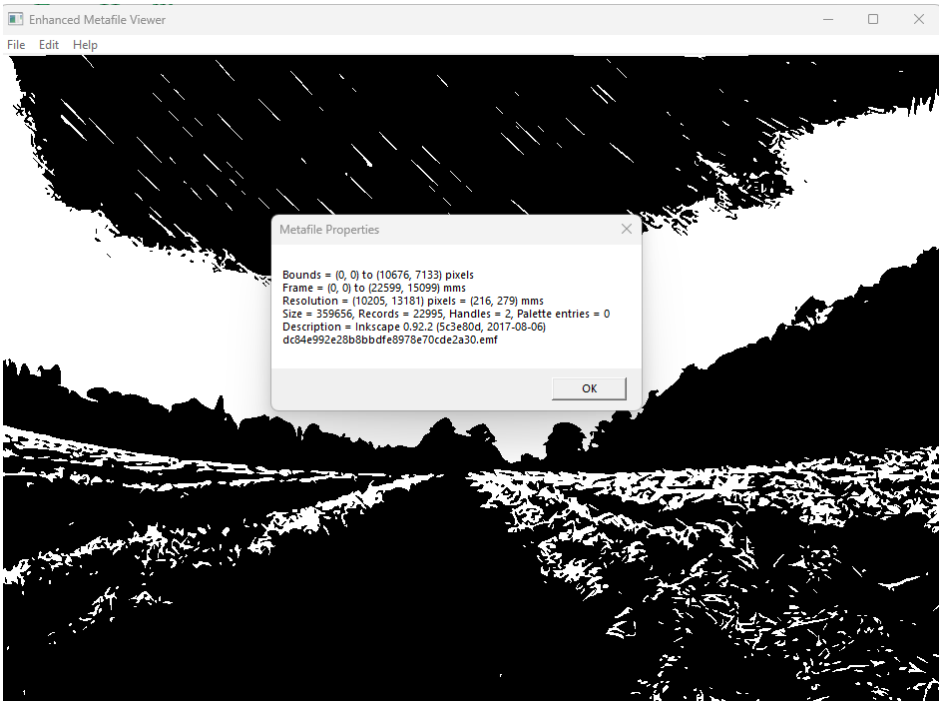
Error Handling:

The program incorporates error handling mechanisms, such as displaying a message box in case of issues like failure to load or save a metafile.

Properties Information:

Available through the "Properties" item in the "File" menu.

Displays a message box with details extracted from the metafile header, providing valuable insights into the file's characteristics.



Vector Image Printing Considerations:

Thin Lines: Vector images with very thin lines (e.g., those in EMF2.EMF) might appear nearly invisible on high-resolution printers.

Recommendation: For better visibility in printouts, consider using wider pens (e.g., 1-point width), as demonstrated in the ruler image example.

Additional Insights:

- Print Dialog Flexibility: The common printer dialog box empowers users to select desired printer settings and preferences.
- Visual Consistency: Stretching metafiles to fill both the printable area and window maintains a consistent visual experience across different contexts.
- Header Information Value: The metafile header often contains valuable metadata, such as dimensions, resolution, and color information, accessible through the "Properties" function.
- Vector Image Printing Best Practices: Adjusting pen widths for printing is crucial for optimal visibility and clarity of vector images on various printer resolutions.

In summary, EMFVIEW is a versatile tool for working with Enhanced Metafiles, providing users with the ability to view, manipulate, and print these vector graphics files in a Windows environment. The program's code demonstrates the use of various Windows API functions for graphics handling, file operations, and user interface interactions.

Metafile Scaling Advantages:

Vector Graphics: Metafiles primarily consist of vector graphics primitives (lines, shapes, fonts), which can be scaled gracefully without significant loss of fidelity, unlike bitmaps.

Versatility: This scalability makes metafiles adaptable for various contexts, including resizing within documents and maintaining clarity on different output devices.

Potential Limitations:

Extreme Compression: Metafiles can still exhibit visual artifacts when severely compressed, especially for dense line patterns or dithering effects.

Embedded Bitmaps and Fonts: Metafiles containing bitmaps or raster fonts inherit those formats' limitations when scaling.

Application Considerations:

Arbitrarily Scaling: While metafiles generally offer flexibility, some applications (e.g., storing signatures) require preserving original proportions to maintain accuracy.

Aspect Ratio Preservation: In such cases, it's essential to maintain the aspect ratio to prevent distortions.

Accurate Display Techniques:

Metafile Header: Information within the metafile header, including dimensions and resolution, is crucial for precise display control.

Bounding Rectangle: The PlayEnhMetaFile function's bounding rectangle parameter influences display size and aspect ratio.

Header-Based Rectangle Setting: Accurate display involves setting the rectangle structure based on metafile header information.

Sample Program Structure:

Shell Program (EMF.C): Contains printing logic and serves as a foundation for subsequent examples.

Resource Script (EMF.RC): Stores resources like menus and dialog boxes.

Header File (RESOURCE.H): Includes resource definitions.

Key Takeaways:

- Metafiles offer significant advantages in scalability and adaptability for various display and printing scenarios.
- Understanding potential limitations and proper techniques for accurate display is essential for effective metafile usage.
- Accessing header information and carefully setting bounding rectangles are key to achieving desired display results.

EMF8.C and EMF.C: DEMONSTRATES ACCURATE DISPLAY OF A 6-INCH RULER METAFILE.

Think of metafiles like recipes for drawing pictures.

Regular image formats (JPEG, PNG) are like pre-cooked meals: You just open them and see the finished picture. You can't really change them much.

Metafiles are like the recipe: They tell you how to draw the picture step-by-step, using lines, shapes, and colors.

Here's the key difference:

With a regular image: If you want to make the picture bigger, it might get blurry or pixelated. You can't really change its details.

With a metafile: You can stretch it to any size without losing quality! The recipe just uses bigger or smaller numbers for the lines and shapes. You can even change the colors or add new details easily.

Here are some real-world uses of metafiles:

Logos: Company logos often use metafiles because they can be resized for different uses without losing their sharpness.

Maps: Online maps can use metafiles to draw the map features (streets, buildings, etc.) because you can zoom in and out without it getting blurry.

Printers: Printers use metafiles to understand how to draw the pages you send them, even if you change the font size or layout.

So, to sum it up:

- Metafiles are like recipes for drawing pictures.
- They can be resized without losing quality.
- They are used in logos, maps, and printing.

Enhanced Metafile Fundamentals:

Vector-Based Graphics: Unlike bitmap formats (e.g., JPEG, PNG), which store pixel-by-pixel information, EMFs store a series of drawing commands that can be executed to reproduce the image.

Scalability: This vector-based nature grants EMFs exceptional scalability. They can be resized without losing clarity, making them ideal for scenarios demanding adaptability.

Device Independence: EMFs maintain consistent quality across different devices and resolutions, ensuring the intended visual appearance is preserved.

EMF8.C PROGRAM

Deeper Exploration of Key Functions:

DrawRuler:

This function is responsible for drawing the ruler with tick marks and numbers on a device context (HDC). It uses various GDI functions such as CreatePen, Rectangle, MoveToEx, LineTo, CreateFontIndirect, and TextOut to create the visual elements of the ruler.

Pen Selection: The choice of a black pen with a 1-point width aligns with typical ruler design conventions, ensuring visual clarity and familiarity.

Tick Mark Variability: The use of different tick mark lengths effectively communicates various measurement increments, enhancing the ruler's usability.

Font Choice: "Times New Roman" is a classic and easily readable font, well-suited for displaying numerical values in a clear and concise manner.

CreateRoutine:

This function is called to create the EMF file. It starts by creating an enhanced metafile (EMF) using the CreateEnhMetaFile function. The EMF file is given the name "emf8.emf" and a description "EMF8\0EMF Demo #8\0".

The function then retrieves the device capabilities (GetDeviceCaps) to determine the size and resolution of the device. It calls the DrawRuler function to draw the ruler on the EMF. Finally, it closes the EMF file with CloseEnhMetaFile and deletes the temporary EMF handle with DeleteEnhMetaFile.

DPI Awareness: The conversion of dimensions to dots per inch is crucial for ensuring accurate scaling and rendering of the ruler, especially when displayed on devices with varying resolutions.

PaintRoutine:

This function is called to paint the contents of the EMF file onto the specified device context (HDC) within the given area. It retrieves the handle to the EMF file using GetEnhMetaFile and obtains the header information of the EMF file using GetEnhMetaFileHeader.

The dimensions of the image are calculated based on the bounds of the EMF file. The function then uses PlayEnhMetaFile to render the EMF file onto the device context within the specified rectangle. Finally, it deletes the EMF handle with DeleteEnhMetaFile.

Header Information: Extracting the ruler's dimensions from the metafile header enables precise placement and sizing within the window's client area, maintaining visual consistency.

Centered Rendering: The calculation of a centered rectangle for metafile playback contributes to visual balance and symmetry, enhancing the overall presentation.

Contextual Significance:

Metafile Versatility: The EMF8.C program effectively showcases the versatility of EMFs for storing and reproducing graphical content in various contexts.

Graphics Programming Foundation: Understanding the principles demonstrated in this program serves as a valuable foundation for further exploration of graphics programming techniques and Windows API graphics capabilities.

Additional Considerations:

- Alternative Metafile Formats: While EMFs offer distinct advantages, other metafile formats, such as Windows Metafiles (WMF) or Scalable Vector Graphics (SVG), have their own unique properties and use cases.
- Metafile Handling in Modern Systems: Contemporary operating systems and graphics frameworks often provide advanced tools and libraries for working with metafiles, streamlining their creation, manipulation, and integration into applications.
- Performance Optimization: In scenarios where performance is critical, careful consideration should be given to the potential overhead associated with metafile creation and playback, as well as potential optimizations for specific use cases.

EMF.C PROGRAM

The code includes necessary header files and defines external functions.

The WinMain function is the entry point of the program. It registers a window class, creates a window, and enters the message loop.

The WndProc function is the window procedure that handles various messages sent to the window, such as WM_CREATE, WM_COMMAND, WM_SIZE, WM_PAINT, and WM_DESTROY.

The WM_CREATE message is handled by calling the CreateRoutine function to perform any necessary initialization.

The WM_COMMAND message is handled to process menu commands. In particular, the IDM_PRINT command triggers the PrintRoutine function, which initiates the printing process. The IDM_EXIT command closes the application, and the IDM_ABOUT command displays information about the program.

The WM_SIZE message is handled to update the client area dimensions when the window is resized.

The WM_PAINT message is handled by obtaining a device context (hdc), calling the PaintRoutine function to perform painting operations, and then releasing the device context.

The WM_DESTROY message is handled to post a quit message and exit the message loop, terminating the program.

The PrintRoutine function is responsible for printing the contents of the window. It displays a print dialog, retrieves the printer device context, determines the page dimensions, starts a print job, and calls the PaintRoutine function to draw the content onto the printer device context.

The CreateRoutine function is an external function that is responsible for any necessary initialization. Its implementation is not provided in the code you shared.

The PaintRoutine function is an external function that performs the actual drawing operations. Its implementation is not provided in the code you shared.

Key Concepts:

Device Contexts (DCs): Abstract drawing surfaces, crucial for both screen and printer output in Windows graphics programming.

Message Loop: The heart of a Windows application, responsible for receiving and processing user input, system events, and drawing requests.

Window Procedure (WndProc): A function that handles messages specific to a window, defining its behavior and interactions.

Printing Pipeline: Involves obtaining a printer DC, initiating the printing process, rendering content using appropriate functions, and finalizing the document.

Code Structure:

Modular Design: The separation of core functions (WinMain, PrintRoutine, WndProc) promotes code clarity and maintainability.

External Dependencies: The reliance on application-specific files for EMF content and drawing logic highlights the framework's versatility.

Error Handling: The inclusion of error-checking mechanisms, such as the message box for printing errors, enhances user experience and aids in troubleshooting.

Additional Considerations:

Historical Context: The Windows NT requirement reflects the specific EMF capabilities of that operating system at the time of the program's creation.

Compatibility and Adaptation: Understanding the program's structure and dependencies is essential for adapting it to modern operating systems or alternative graphics libraries.

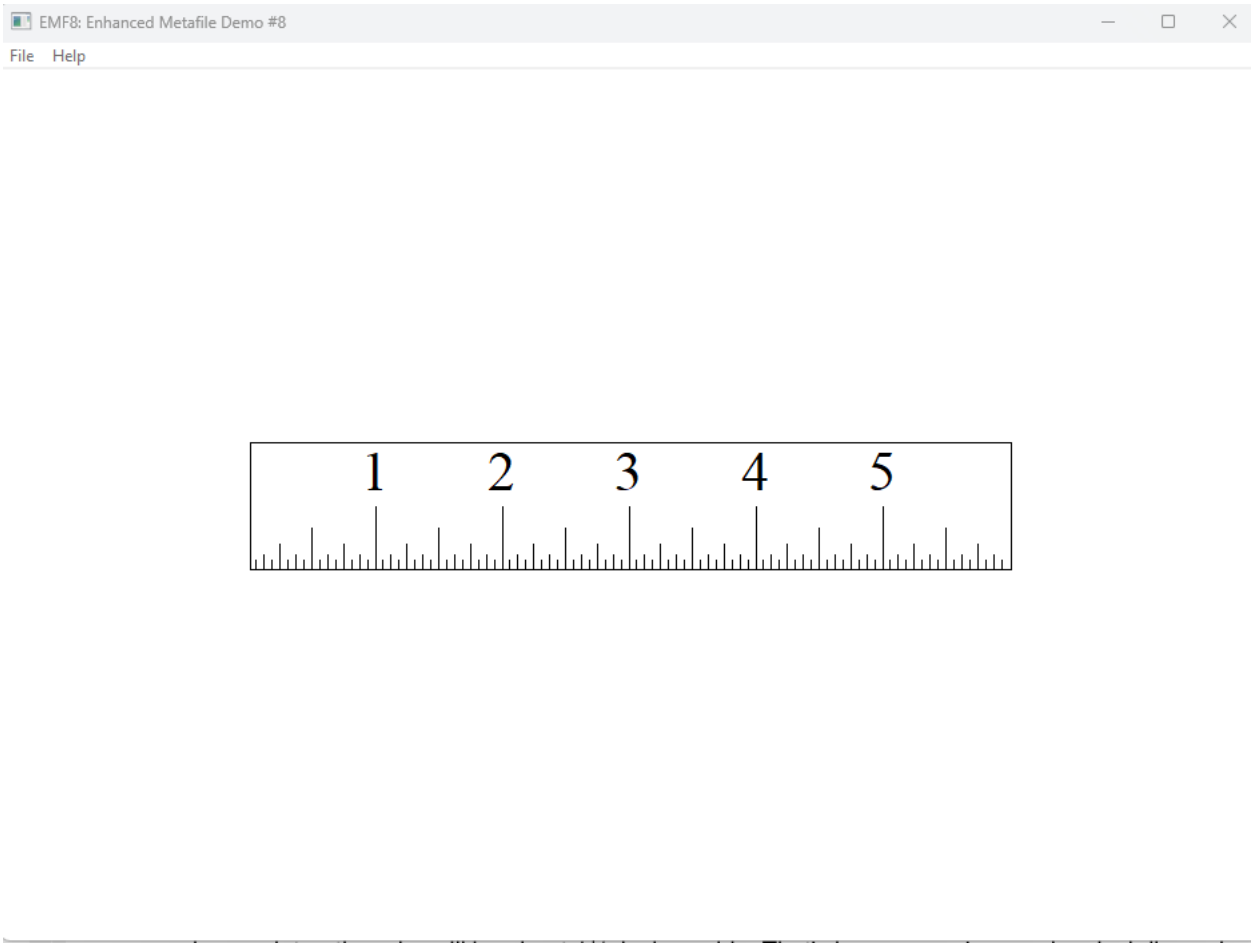
Customization Potential: The framework's modular nature invites experimentation with different EMF creation and playback techniques, fostering a deeper understanding of vector graphics principles.

Further Exploration:

Graphics Programming: Investigate Windows API graphics functions for drawing lines, shapes, text, and images, as well as advanced techniques like coordinate transformations and color management.

Metafile Formats: Explore the properties and use cases of other metafile formats, such as Windows Metafiles (WMF) and Scalable Vector Graphics (SVG), to choose the most suitable format for different applications.

Cross-Platform Development: Consider graphics libraries like Qt or wxWidgets for creating cross-platform applications with support for vector graphics and metafiles.



RECAP OF THE PROGRAM ABOVE:

Metafile Creation and Playback:

When: The program creates a metafile containing a ruler image during the WM_CREATE message.

How: It calls an external function named CreateRoutine to generate the metafile.

Where: The metafile is displayed both on-screen (during WM_PAINT) and printed (when the user selects the print command).

Resolution and Scaling:

Challenge: Accurate rendering of the ruler requires knowledge of device resolution, as printers often have higher resolution than displays.

Reference Device: The CreateEnhMetaFile function uses a reference device context (usually the video display) to determine device characteristics.

Calculating DPI: The code obtains device dimensions in millimeters and pixels, and calculates dots per inch (DPI) for accurate scaling.

Header Information:

Bounding Rectangle: The PaintRoutine function retrieves the metafile header using GetEnhMetaFileHeader.

Purpose: The header contains the rclBounds field, which specifies the image's size in pixels.

Centering: This information is used to center the ruler image within the client area, ensuring proper visual alignment.

Screen vs. Print Discrepancy:

On-screen Display: The ruler drawn with the video display's resolution as reference might not perfectly match a physical ruler's size due to display limitations discussed in Chapter 5.

Printing Issue: When printed on a high-resolution printer (e.g., 300 DPI), the ruler appears 11/3 inches wide instead of the intended 6 inches because the pixel-based size was used.

Metafile Header Rectangles:

rclBounds: Used in EMF8, specifies the image size in pixels, but leads to inaccurate printing on different resolutions.

rclFrame: Contains the image size in 0.01 millimeters, offering device-independent information for accurate scaling.

Reference Device Context:

Determines the relationship between pixel and millimeter dimensions in the header.

In this case, the video display acts as the reference, leading to pixel-based size in EMF8.

Solution and Next Steps:

EMF9 program utilizes the rclFrame information from the metafile header for accurate ruler rendering on various devices.

This example highlights the importance of using the appropriate rectangle field and understanding the reference device context impact.

Further Exploration:

- Experiment with EMF9 to see how the ruler scales accurately on different resolutions.
- Investigate other metafile formats and their header information for flexible image handling.
- Learn more about reference device contexts and their role in scaling and device independence.

Key Takeaways from EMF8 program:

- Metafile Flexibility: Metafiles, like EMFs, offer versatility in displaying and printing images due to their device-independent nature.
- DPI Awareness: Accurate rendering requires understanding device resolution and scaling images accordingly.
- Header Metadata: Metafile headers contain valuable information, such as bounding rectangles, which can be used for precise positioning and scaling of images.

EMF9 PROGRAM

```
// Program Title: Enhanced Metafile Demo #9
// Author: Charles Petzold, 1998

// Include necessary headers
#include <windows.h>
#include <string.h>

// Define window class and title
TCHAR szClass[] = TEXT("EMF9");
TCHAR szTitle[] = TEXT("EMF9: Enhanced Metafile Demo #9");

// Function to create the window
void CreateRoutine(HWND hwnd) {
    // Implementation not provided in the code snippet
    // This function would typically handle window creation, initialization, etc.
}

// Function to paint the window
void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea) {
    ENHMETAHEADER emh;
    HENHMETAFILE hemf;
    int cxMms, cyMms, cxPix, cyPix, cxImage, cyImage;
    RECT rect;

    // Get device characteristics
    cxMms = GetDeviceCaps(hdc, HORZSIZE);
    cyMms = GetDeviceCaps(hdc, VERTSIZE);
    cxPix = GetDeviceCaps(hdc, HORZRES);
    cyPix = GetDeviceCaps(hdc, VERTRES);

    // Load an enhanced metafile from a file
    hemf = GetEnhMetaFile(TEXT("..\\emf8\\emf8.emf"));

    // Retrieve the header information of the metafile
    GetEnhMetaFileHeader(hemf, sizeof(emh), &emh);

    // Calculate the image size in pixels based on device characteristics
    cxImage = emh.rclFrame.right - emh.rclFrame.left;
    cyImage = emh.rclFrame.bottom - emh.rclFrame.top;
    cxImage = cxImage * cxPix / cxMms / 100;
    cyImage = cyImage * cyPix / cyMms / 100;

    // Calculate the position to center the image in the window
    rect.left = (cxArea - cxImage) / 2;
    rect.right = (cxArea + cxImage) / 2;
    rect.top = (cyArea - cyImage) / 2;
    rect.bottom = (cyArea + cyImage) / 2;

    // Play the enhanced metafile on the specified rectangle
    PlayEnhMetaFile(hdc, hemf, &rect);

    // Delete the metafile handle to free resources
    DeleteEnhMetaFile(hemf);
}
```

Imagine a digital ruler that always measures correctly, even when printed on different devices:

- This code makes it happen by using a special format called an Enhanced Metafile (EMF).
- It's like storing the ruler's drawing instructions instead of a simple picture.
- The instructions include measurements in millimeters, ensuring accuracy.

Here's how it works:

Load the Ruler's Blueprint: The code opens the EMF file, which contains the ruler's design and measurements.

Check the Output Device: It examines the screen or printer to figure out its resolution (how many dots or pixels it can display per inch).

Calculate the Right Size: Using the device's resolution, it converts the ruler's measurements from millimeters to pixels to ensure it looks correct on that specific device.

Position the Ruler: It calculates the best spot to center the ruler within the window or page.

Draw the Ruler: It follows the EMF's instructions to carefully draw the ruler lines, numbers, and tick marks in the correct size and position.

Clean Up: It tidies up by closing the EMF file to save memory.

The result: A ruler that always measures 6 inches wide and 1 inch tall, whether you see it on the screen or print it on paper!

1. Application Structure:

The code seems to be structured around two main functions: CreateRoutine and PaintRoutine. The former is likely responsible for creating and initializing the window, while the latter handles the painting of the window with an enhanced metafile.

2. Window Creation (CreateRoutine):

Unfortunately, the details of the CreateRoutine function are not provided in this snippet. In a typical Windows application, this function would involve creating a window, setting up the necessary structures, and handling any initialization required for the application.

3. Metafile Loading and Drawing (PaintRoutine):

The PaintRoutine function is more detailed and appears to be responsible for loading an enhanced metafile and drawing it on the window's device context.

Metafile Loading:

- The code loads an enhanced metafile from a file using the GetEnhMetaFile function. The filename is specified as "..\emf8\emf8.emf".
- The GetEnhMetaFileHeader function retrieves information about the metafile, particularly the dimensions and other characteristics.

Device Characteristics:

- The code then retrieves characteristics of the device context (hdc) using functions like GetDeviceCaps. These characteristics include the size of the device in millimeters (cxMms, cyMms) and pixels (cxPix, cyPix).

Image Size Calculation:

- The dimensions of the metafile are adjusted based on the device characteristics to calculate the size of the image in pixels (cxImage, cyImage). This adjustment ensures that the image is scaled appropriately for the device.

Image Positioning:

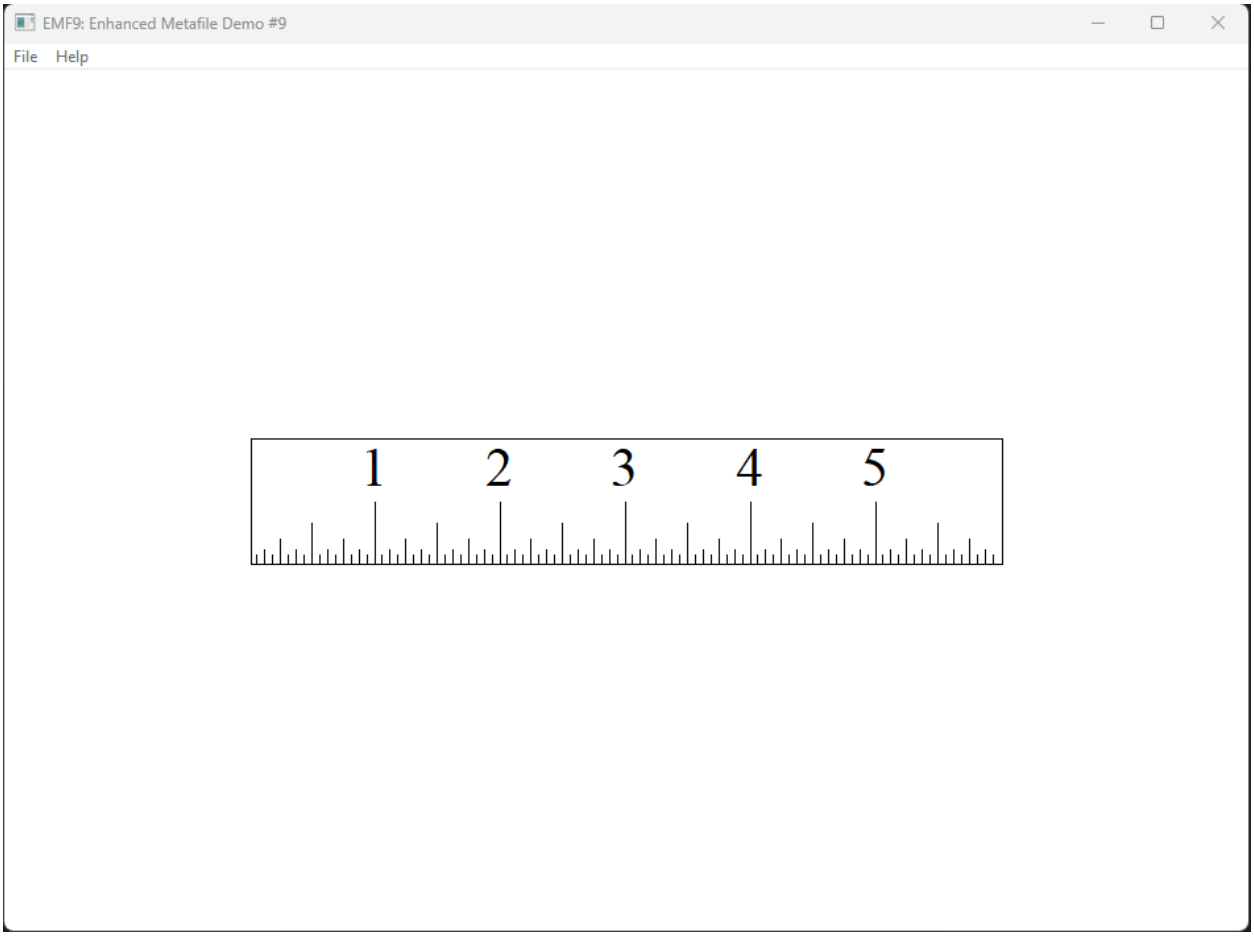
- The code calculates the rectangle (rect) where the metafile will be drawn. It aims to center the image within the window, considering both the width and height.

Drawing Metafile:

- The PlayEnhMetaFile function is then used to draw the enhanced metafile on the specified rectangle within the window.

Cleanup:

- Finally, the DeleteEnhMetaFile function is called to release resources associated with the loaded metafile.



4. Conclusion:

In summary, this code segment is part of a larger Windows application that deals with enhanced metafiles. It illustrates the loading and drawing of a metafile on a window, considering device characteristics for appropriate scaling. The specific details of window creation and initialization are not provided in this snippet.

EMF10 PROGRAM

```c
// Program Title: Enhanced Metafile Demo #10
// Author: Charles Petzold, 1998

// Include necessary headers
#include <windows.h>

// Define window class and title
TCHAR szClass[] = TEXT("EMF10");
TCHAR szTitle[] = TEXT("EMF10: Enhanced Metafile Demo #10");

// Function to create the window
void CreateRoutine(HWND hwnd) {
    // Implementation not provided in the code snippet
    // This function would typically handle window creation, initialization, etc.
}

// Function to paint the window
void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea) {
    ENHMETAHEADER emh;
    float fScale;
    HENHMETAFILE hemf;
    int cxMms, cyMms, cxPix, cyPix, cxImage, cyImage;
    RECT rect;

    // Get device characteristics
    cxMms = GetDeviceCaps(hdc, HORZSIZE);
    cyMms = GetDeviceCaps(hdc, VERTSIZE);
    cxPix = GetDeviceCaps(hdc, HORZRES);
    cyPix = GetDeviceCaps(hdc, VERTRES);

    // Load an enhanced metafile from a file
    hemf = GetEnhMetaFile(TEXT("..\\emf8\\emf8.emf"));

    // Retrieve the header information of the metafile
    GetEnhMetaFileHeader(hemf, sizeof(emh), &emh);

    // Calculate the pixel size of the original 6-inch-wide image appropriate for the destination device context
    cxImage = emh.rclFrame.right - emh.rclFrame.left;
    cyImage = emh.rclFrame.bottom - emh.rclFrame.top;
    cxImage = cxImage * cxPix / cxMms / 100;
    cyImage = cyImage * cyPix / cyMms / 100;

    // Calculate a scaling factor (fScale) based on the minimum ratio of client area width/height to image width/height
    fScale = min((float)cxArea / cxImage, (float)cyArea / cyImage);

    // Increase the pixel dimensions of the image using the scaling factor
    cxImage = (int)(fScale * cxImage);
    cyImage = (int)(fScale * cyImage);

    // Calculate the bounding rectangle for displaying the metafile
    rect.left = (cxArea - cxImage) / 2;
    rect.right = (cxArea + cxImage) / 2;
    rect.top = (cyArea - cyImage) / 2;
    rect.bottom = (cyArea + cyImage) / 2;

    // Play the enhanced metafile on the specified rectangle
    PlayEnhMetaFile(hdc, hemf, &rect);

    // Delete the metafile handle to free resources
    DeleteEnhMetaFile(hemf);
}
```

Understanding the Program's Goals:

Preserve Aspect Ratio: The program prioritizes maintaining the ruler image's 6:1 width-to-height ratio, preventing distortions that would occur if it were simply stretched to fit any arbitrary space. This ensures the ruler's visual accuracy and consistency across different display areas.

Adapt to Display Constraints: It adapts to the available client area (the visible portion of a window or the printable area of a page), ensuring the ruler fits within these boundaries without exceeding them. This adaptability makes it suitable for a variety of display environments.

Center Image for Visual Balance: The program centers the ruler within the available space, either horizontally or vertically depending on the client area's dimensions. This creates a visually pleasing and well-balanced presentation.

Key Steps and Concepts:

Leveraging Enhanced Metafiles (EMFs): The program demonstrates the power of EMFs, a file format that stores graphical instructions rather than raw image data. This enables consistent rendering across different devices and resolutions, ensuring the ruler's accuracy.

Respecting Device Capabilities: It emphasizes the importance of understanding device characteristics, such as resolution and size, to ensure proper scaling and display of the ruler. This highlights the need for software to adapt to different hardware environments.

Calculating Optimal Scaling:

The program employs a clever approach to scaling:

- It determines a scaling factor that respects both the client area's dimensions and the ruler's aspect ratio.
- This ensures the ruler fits within the available space without stretching or compressing its shape.

Centering for Visual Harmony:

The calculation of a centered rectangle for image placement demonstrates attention to visual presentation. Centering the ruler creates a balanced and aesthetically pleasing layout.

Utilizing Key Functions: The program effectively leverages Windows API functions for metafile handling and image rendering:

- GetEnhMetaFile retrieves the ruler's design and measurements from the EMF file.
- GetDeviceCaps gathers information about the output device.
- PlayEnhMetaFile follows the EMF's instructions to draw the ruler accurately.

In Conclusion:

This program showcases crucial concepts in graphics programming:

- ✓ Preserving aspect ratios during resizing to maintain image integrity.
- ✓ Adapting to varying display environments for flexibility and compatibility.
- ✓ Utilizing device-independent graphics formats for consistent rendering.
- ✓ Understanding device capabilities for accurate image scaling and positioning.

EMF11 PROGRAM

Key Concepts:

Mapping Modes and Metafiles: The program explores the challenges and potential pitfalls of using mapping modes with metafiles, which can introduce complexities when scaling and displaying images across different devices.

Enhanced Metafile (EMF) Creation: It creates an EMF file named "emf11.emf" to store the ruler's graphical instructions.

Mapping Mode Experimentation: It attempts to use the MM_LOENGLISH mapping mode to draw the ruler in inches directly within the EMF, but highlights potential issues with this approach.

CreateRoutine:

- Creates a metafile device context (hdcEMF) using CreateEnhMetaFile.
- Sets the mapping mode to MM_LOENGLISH for drawing in inches.
- Calls DrawRuler to create the ruler graphics within the metafile.
- Closes the metafile using CloseEnhMetaFile and retrieves its handle (hemf).
- Deletes the metafile handle to free resources.

DrawRuler:

- Draws a black rectangle, tick marks, and numbers to represent a 6-inch ruler.
- Uses a 1-point pen width for accuracy in inches.
- Adjusts rectangle coordinates for visual consistency across Windows versions.
- Creates a font with a height of half the ruler's height for proper scaling.
- Aligns numbers at the bottom and center of each inch.

PaintRoutine:

- Retrieves device characteristics for scaling calculations.
- Loads the EMF file using GetEnhMetaFile.
- Retrieves the EMF header information.
- Calculates the ruler's dimensions in pixels based on device characteristics.
- Determines a centered rectangle for displaying the ruler within the available area.
- Plays the metafile using PlayEnhMetaFile to render the ruler.
- Deletes the metafile handle to free resources.

Key Considerations:

- Mapping Mode Confusion: The program highlights that using mapping modes with metafiles can lead to unexpected results when played back on different devices.
- Manual Calculations: It emphasizes the importance of understanding device characteristics and performing manual scaling calculations to ensure consistent rendering across various output contexts.
- Pixel-Based Approach: The previous programs (EMF8-EMF10) demonstrate a more reliable approach using pixel-based calculations for device-independent ruler display.

Mapping Mode Considerations:

The choice of mapping mode can have a significant impact on the visual representation of a metafile. For example, different mapping modes can affect the scaling, positioning, and orientation of objects within the metafile. It's important to understand and consider the behavior of different mapping modes to achieve the desired results.

When selecting a mapping mode, it's essential to take into account the characteristics of the target output device. Different devices may have different resolutions, aspect ratios, or physical dimensions. Choosing an appropriate mapping mode for each output device can help ensure consistent and accurate rendering across different platforms and devices.

Rectangle Adjustments:

In the code example provided, there are adjustments made to rectangle coordinates in the DrawRuler function. These adjustments are likely done to compensate for differences in how different versions of Windows handle the drawing of rectangles. The specific reasons behind these adjustments are not explained in the code snippet, but they are likely related to variations in coordinate systems or rendering behaviors across different Windows versions. These adjustments aim to achieve visual consistency and compatibility across different Windows platforms.

Alternative Approaches:

While the code demonstrates the use of mapping modes to simplify metafile creation, there are alternative approaches to achieve device-independent graphics rendering. One such approach is using matrix transformations, which allow you to apply scaling, rotation, translation, and other transformations to objects.

This can be useful for achieving more complex transformations or effects. Another approach is to implement custom scaling functions that adjust the coordinates and dimensions of objects based on the target device's characteristics. Exploring these alternative techniques can provide more flexibility and control in graphics rendering.

Scalability:

The code examples provided in the snippet focus on fixed-size rulers. However, extending the concepts to dynamically scalable graphics elements would enhance the program's versatility. By incorporating techniques like variable scaling factors or user-defined scaling, you can create graphics that adapt to different display scenarios or user preferences. This can be particularly useful in applications or interfaces that need to support different screen resolutions or device sizes.

Performance Considerations:

It's important to consider the potential performance implications of using mapping modes within metafiles, especially for complex graphics. Some mapping modes may involve additional calculations or transformations, which can impact rendering performance.

It's recommended to evaluate the performance of your graphics operations and consider optimizations if necessary. Techniques like caching or pre-rendering certain elements can help improve performance when working with large or complex metafiles.

Error Handling and User Feedback:

To enhance the code's robustness and user experience, it's valuable to include error handling mechanisms. This can involve checking for errors returned by functions, handling exceptions, or providing informative error messages to the user.

Additionally, providing feedback to the user during long operations, such as printing or rendering, can improve the perceived responsiveness of the program. Progress indicators or status messages can help keep the user informed about the ongoing operations and any potential issues encountered.

Conclusion:

While mapping modes can be useful in certain graphics scenarios, this program demonstrates that careful consideration and manual adjustments are often necessary when working with metafiles to ensure accurate and consistent rendering across different devices.

EMF12 PROGRAM

The code snippet provided is an example of an enhanced metafile (EMF) program. It consists of several functions that create and display a ruler using metafile operations. Let's break down the code and explain its functionality:

DrawRuler: This function is responsible for drawing the ruler using GDI (Graphics Device Interface) operations. It takes an HDC (device context) and the dimensions of the ruler as parameters. The function draws tick marks at different intervals and displays numbers on the ruler.

CreateRoutine: This function creates an enhanced metafile (EMF) by calling CreateEnhMetaFile. It retrieves the device capabilities (size and resolution) using GetDeviceCaps to determine the dimensions and units of the metafile. The DrawRuler function is then called to draw the ruler on the metafile. Finally, the metafile is closed, saved to a file, and deleted.

PaintRoutine: This function is responsible for painting the metafile on a window. It takes an HDC, the window's client area dimensions, and performs the following steps:

- Sets the mapping mode to MM_HIMETRIC, which establishes a logical coordinate system where 1 unit is equal to 0.01 millimeters.
- Sets the viewport origin to the bottom-left corner of the client area using SetViewportOrgEx.
- Converts the client area dimensions from device points to logical coordinates using DPtoLP.
- Retrieves the metafile (emf12.emf) using GetEnhMetaFile.
- Retrieves the metafile header using GetEnhMetaFileHeader to determine the dimensions of the metafile image.
- Calculates the destination rectangle (rect) where the metafile will be displayed. The rectangle is centered in the client area and sized according to the metafile dimensions.
- Finally, it uses PlayEnhMetaFile to render the metafile on the specified HDC within the destination rectangle.
- The line DeleteEnhMetaFile(hemf) after PlayEnhMetaFile appears to be a typographical error and should be removed.

```c
// Program Title: Enhanced Metafile Demo #12
// Author: Charles Petzold, 1998

// Include necessary headers
#include <windows.h>

// Define window class and title
TCHAR szClass[] = TEXT("EMF12");
TCHAR szTitle[] = TEXT("EMF12: Enhanced Metafile Demo #12");

// Function to draw a ruler
void DrawRuler(HDC hdc, int cx, int cy) {
    int iAdj, i, iHeight;
    LOGFONT lf;
    TCHAR ch;

    iAdj = GetVersion() & 0x80000000 ? 0 : 1;

    // Black pen with 1-point width
    SelectObject(hdc, CreatePen(PS_SOLID, cx / 72 / 6, 0));

    // Rectangle surrounding entire pen (with adjustment)
    Rectangle(hdc, iAdj, iAdj, cx + iAdj + 1, cy + iAdj + 1);

    // Tick marks
    for (i = 1; i < 96; i++) {
        if (i % 16 == 0) iHeight = cy / 2;          // inches
        else if (i % 8 == 0) iHeight = cy / 3;      // half inches
        else if (i % 4 == 0) iHeight = cy / 5;      // quarter inches
        else if (i % 2 == 0) iHeight = cy / 8;      // eighths
        else iHeight = cy / 12;                      // sixteenths

        MoveToEx(hdc, i * cx / 96, cy, NULL);
        LineTo(hdc, i * cx / 96, cy - iHeight);
    }

    // Create logical font
    FillMemory(&lf, sizeof(lf), 0);
    lf.lfHeight = cy / 2;
    lstrcpy(lf.lfFaceName, TEXT("Times New Roman"));
    SelectObject(hdc, CreateFontIndirect(&lf));
    SetTextAlign(hdc, TA_BOTTOM | TA_CENTER);
    SetBkMode(hdc, TRANSPARENT);

    // Display numbers
    for (i = 1; i <= 5; i++) {
        ch = (TCHAR)(i + '0');
        TextOut(hdc, i * cx / 6, cy / 2, &ch, 1);
    }
    // Clean up
    DeleteObject(SelectObject(hdc, GetStockObject(SYSTEM_FONT)));
    DeleteObject(SelectObject(hdc, GetStockObject(BLACK_PEN)));
}
// Function to create the window
void CreateRoutine(HWND hwnd) {
    HDC hdcEMF;
    HENHMETAFILE hemf;
    int cxMms, cyMms, cxPix, cyPix, xDpi, yDpi;
    hdcEMF = CreateEnhMetaFile(NULL, TEXT("emf12.emf"), NULL, TEXT("EMF13\0EMF Demo #12\0"));
    cxMms = GetDeviceCaps(hdcEMF, HORZSIZE);
    cyMms = GetDeviceCaps(hdcEMF, VERTSIZE);
    cxPix = GetDeviceCaps(hdcEMF, HORZRES);
    cyPix = GetDeviceCaps(hdcEMF, VERTRES);
    xDpi = cxPix * 254 / cxMms / 10;
    yDpi = cyPix * 254 / cyMms / 10;
    DrawRuler(hdcEMF, 6 * xDpi, yDpi);
    hemf = CloseEnhMetaFile(hdcEMF);
    DeleteEnhMetaFile(hemf);
}
// Function to paint the window
void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea) {
    ENHMETAHEADER emh;
    HENHMETAFILE hemf;
    POINT pt;
    int cxImage, cyImage;
    RECT rect;
    // Set the mapping mode to MM_HIMETRIC
    SetMapMode(hdc, MM_HIMETRIC);
    // Set the origin of the viewport
    SetViewportOrgEx(hdc, 0, cyArea, NULL);
    // Convert client coordinates to logical coordinates
    pt.x = cxArea;
    pt.y = 0;
    DPtoLP(hdc, &pt, 1);
    // Load an enhanced metafile
    hemf = GetEnhMetaFile(TEXT("emf12.emf"));
    GetEnhMetaFileHeader(hemf, sizeof(emh), &emh);
    // Calculate image dimensions
    cxImage = emh.rclFrame.right - emh.rclFrame.left;
    cyImage = emh.rclFrame.bottom - emh.rclFrame.top;
    // Calculate the bounding rectangle
    rect.left = (pt.x - cxImage) / 2;
    rect.top = (pt.y + cyImage) / 2;
    rect.right = (pt.x + cxImage) / 2;
    rect.bottom = (pt.y - cyImage) / 2;
    // Play the enhanced metafile on the specified rectangle
    PlayEnhMetaFile(hdc, hemf, &rect);
    // Delete the metafile handle to free resources
    DeleteEnhMetaFile(hemf);
}
```

Key Points:

- ✓ Mapping Mode for Metafile Creation: EMF12 uses MM_LOENGLISH for drawing the ruler in inches, simplifying metafile creation.
- ✓ Mapping Mode for Playback: MM_HIMETRIC is used during playback to align the metafile with the display's size and resolution.
- ✓ Viewport Adjustment: SetViewportOrgEx ensures correct vertical positioning.
- ✓ Device-Independent Scaling: The program aims to achieve device-independent scaling by combining mapping modes with DPI calculations.
- ✓ Potential Issues: Using mapping modes with metafiles can still lead to unexpected results on different devices, requiring careful testing and adjustments.

Overall:

The program demonstrates an approach to simplify metafile creation and scaling using mapping modes, but it highlights the complexities and potential pitfalls of this approach.

This code generates an EMF file containing a ruler and then displays the ruler by playing the metafile using PlayEnhMetaFile with the MM_HIMETRIC mapping mode.

By using MM_HIMETRIC, the logical units of the metafile and the destination rectangle are set to 0.01 millimeters, ensuring consistent scaling and positioning on different devices.

Working with mapping modes and metafiles. Here are a few additional points to consider:

Scaling and Resolution: The code snippet calculates the logical units of the metafile based on the device's resolution and size. This ensures that the metafile is scaled consistently across different devices. However, it's important to note that scaling can affect the visual quality and precision of the graphics. It's recommended to test and verify the desired scaling and resolution for your specific application.

Compatibility with Other Metafile Viewers: While the code demonstrates the creation and playback of metafiles using mapping modes, it's worth considering the compatibility of the generated metafiles with other metafile viewers or editors. Not all viewers may correctly interpret or support mapping modes. If compatibility with third-party applications is a concern, it's advisable to test the generated metafiles in different viewers to ensure proper rendering.

Error Handling: The code snippet provided does not include comprehensive error handling. When working with GDI functions, it's important to handle potential errors and failure cases appropriately. This includes checking the return values of functions, handling resource allocation failures, and providing meaningful error messages or fallback mechanisms when necessary.

Documentation and Code Organization: It's important to document the code and its functionality properly, especially if it's intended for use by other developers or as part of a larger project. Clear comments, function headers, and consistent coding conventions can greatly improve code readability and maintainability.

EMF13 PROGRAM

```c
// Program Title: Enhanced Metafile Demo #13
// Author: Charles Petzold, 1998

// Include necessary headers
#include <windows.h>

// Define window class and title
TCHAR szClass[] = TEXT("EMF13");
TCHAR szTitle[] = TEXT("EMF13: Enhanced Metafile Demo #13");

// Function to create the window
void CreateRoutine(HWND hwnd) {}

// Function to paint the window
void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea) {
    ENHMETAHEADER emh;
    HENHMETAFILE hemf;
    POINT pt;
    int cxImage, cyImage;
    RECT rect;

    // Set the mapping mode to MM_HIMETRIC
    SetMapMode(hdc, MM_HIMETRIC);

    // Set the origin to the lower left corner
    SetViewportOrgEx(hdc, 0, cyArea, NULL);

    // Convert client coordinates to logical coordinates
    pt.x = cxArea;
    pt.y = 0;
    DPtoLP(hdc, &pt, 1);

    // Load the enhanced metafile created by EMF11
    hemf = GetEnhMetaFile(TEXT("..\\emf11\\emf11.emf"));
    GetEnhMetaFileHeader(hemf, sizeof(emh), &emh);

    // Calculate image dimensions in 0.01 millimeters
    cxImage = emh.rclFrame.right - emh.rclFrame.left;
    cyImage = emh.rclFrame.bottom - emh.rclFrame.top;

    // Calculate the destination rectangle centered in the middle of the client area
    rect.left = (pt.x - cxImage) / 2;
    rect.top = (pt.y + cyImage) / 2;
    rect.right = (pt.x + cxImage) / 2;
    rect.bottom = (pt.y - cyImage) / 2;

    // Play the enhanced metafile on the specified rectangle
    PlayEnhMetaFile(hdc, hemf, &rect);

    // Delete the metafile handle to free resources
    DeleteEnhMetaFile(hemf);
}
```

The code snippet demonstrates how to create and paint a window using an enhanced metafile.

The CreateRoutine function is responsible for creating the window. However, in the provided code, the function is empty, indicating that no specific actions are performed during window creation.

The PaintRoutine function is responsible for painting the window. It takes four parameters: hwnd (the window handle), hdc (the device context), cxArea (the width of the client area), and cyArea (the height of the client area).

Here is a step-by-step breakdown of the PaintRoutine function:

Set the mapping mode to MM_HIMETRIC. This mapping mode ensures that the logical units used for drawing are in 0.01 millimeters, providing a consistent scale across devices.

Set the origin of the viewport to the lower-left corner of the client area using SetViewportOrgEx. This adjustment is necessary because the default origin is at the upper-left corner, but with the mapping mode set to MM_HIMETRIC, the y-axis increases upwards.

Convert the client coordinates (cxArea, 0) to logical coordinates using DPtoLP. This conversion ensures that the logical coordinates match the desired scale and units.

Load an enhanced metafile named "emf11.emf" using GetEnhMetaFile. This metafile was presumably created by a previous program (EMF11) and is now being used for rendering.

Retrieve the metafile header using GetEnhMetaFileHeader to obtain information about the metafile's dimensions and other properties. The header is stored in the emh variable.

Calculate the dimensions of the metafile image in 0.01 millimeters by subtracting the left coordinate from the right coordinate for the width and the top coordinate from the bottom coordinate for the height.

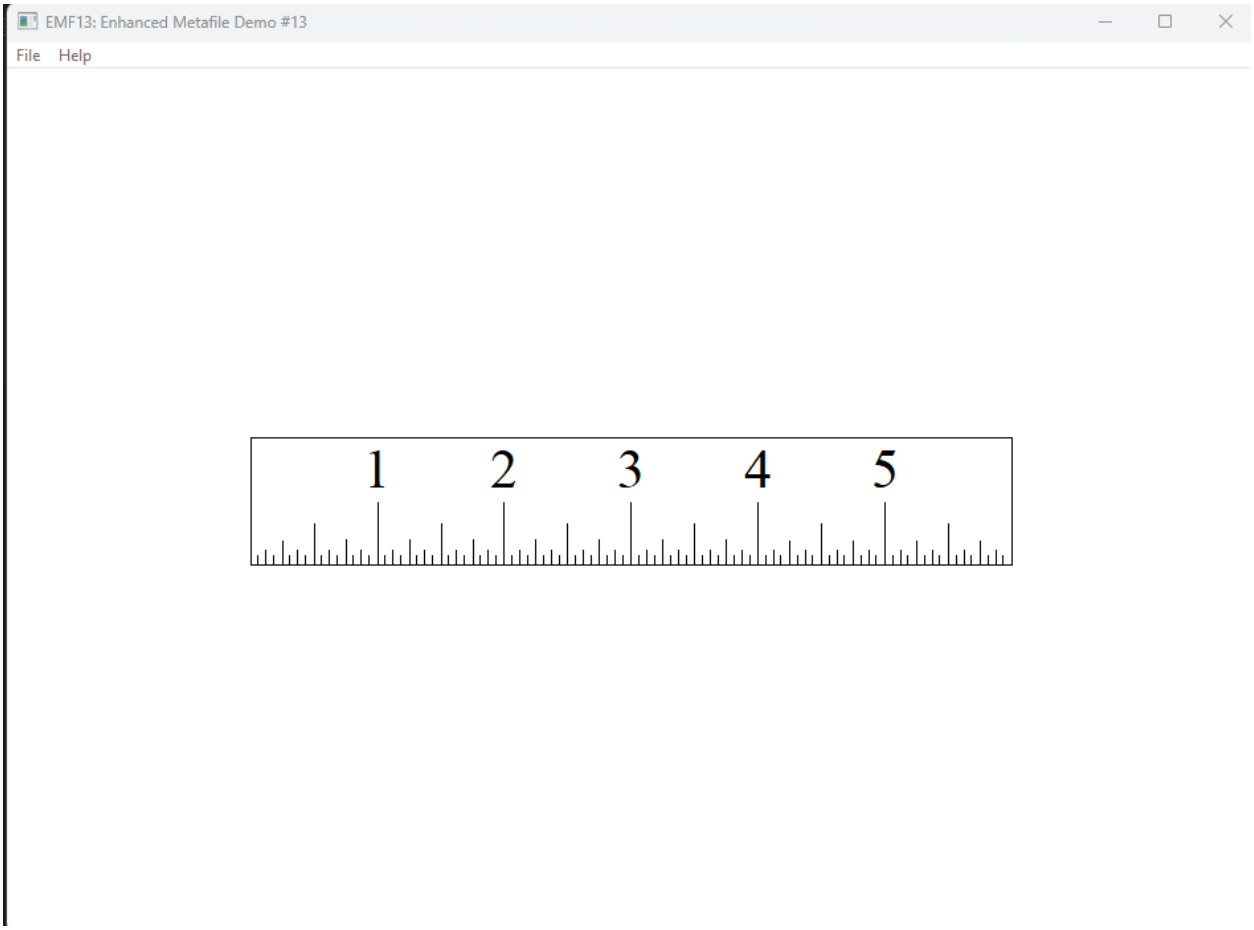
Calculate the destination rectangle where the metafile will be displayed. The rectangle is centered in the middle of the client area. The left and right coordinates are calculated based on the logical coordinates and the width of the metafile image. The top and bottom coordinates are calculated in a similar manner. The resulting rectangle is stored in the rect variable.

Use PlayEnhMetaFile to play the enhanced metafile on the specified rectangle within the device context. This function renders the metafile content onto the window.

Delete the metafile handle using DeleteEnhMetaFile to release the resources associated with the metafile.

This code snippet demonstrates the process of loading an enhanced metafile and rendering it on a window using a specific mapping mode. It showcases how to position and scale the metafile content within the window's client area.

In the EMF13 program, the ruler metafile is not created using a mapping mode. It is already created by EMF11. EMF13 simply loads the existing metafile and utilizes a mapping mode to calculate the destination rectangle, similar to how EMF11 operates.



Based on your explanation, we can establish a couple of principles:

Metafile Creation: When a metafile is created, GDI takes into account any embedded changes to the mapping mode to calculate the size of the metafile image in pixels and millimeters. This information is stored in the metafile header. The mapping mode in effect during the creation of the metafile is used to determine the image size.

Metafile Playback: When a metafile is played, GDI determines the physical location of the destination rectangle based on the mapping mode in effect at the time of the PlayEnhMetaFile call. The mapping mode used during playback determines the positioning and scaling of the metafile content within the destination rectangle. Once the mapping mode is set for playback, the metafile itself cannot change the location of the destination rectangle.

These principles *highlight the importance of mapping modes during both metafile creation and playback.* The mapping mode used during creation affects the size information stored in the metafile header, while the mapping mode used during playback affects the positioning and scaling of the metafile.

It's worth noting that the mapping modes used during creation and playback should be chosen carefully to ensure consistent and desired results. Changes to the mapping mode between creation and playback can lead to unexpected positioning and scaling of the metafile content.