GRAPHICS DEVICE INTERFACE

The **Graphics Device Interface (GDI)** is a crucial component of Microsoft Windows, responsible for **displaying graphics** on video displays and printers.

GDI plays a pivotal role in both user applications and the Windows operating system itself, handling the visual rendering of elements such as menus, scroll bars, icons, and mouse cursors.

This chapter provides a fundamental understanding of GDI, focusing on the **basics of drawing lines and filled areas.**

This foundational knowledge will serve as a stepping stone for subsequent chapters that delve into more **advanced GDI concepts**, including bitmap support, metafiles, and formatted text.

The GDI Philosophy

The **Graphics Device Interface (GDI)** is a fundamental component of Microsoft Windows, responsible for rendering graphics on video displays and printers.

GDI functions are exported from the **dynamic-link library GDI32.DLL.**

In Windows 98, GDI32.DLL utilizes the 16-bit dynamic-link library GDI.EXE for the implementation of many of its functions.

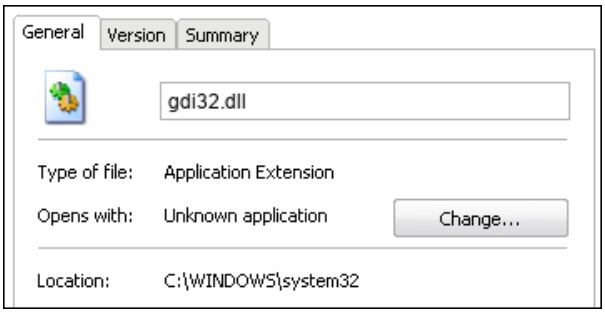
However, in Windows NT, GDI.EXE is only employed for 16-bit programs.

These dynamic-link libraries interact with device drivers for the video display and any connected printers.

The video driver interfaces with the video display hardware, while the printer driver translates GDI commands into codes or commands that the respective printers can interpret.

Consequently, different video display adapters and printers require specific device drivers.

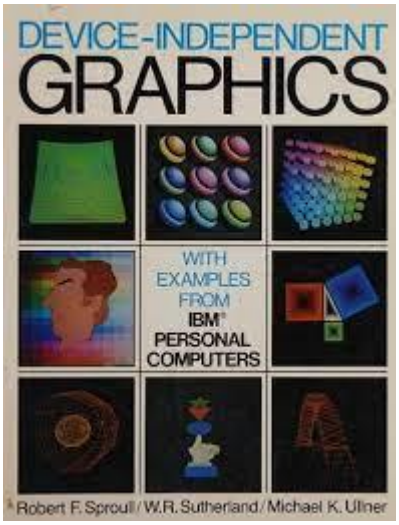
While Windows NT used a separate 16-bit dynamic-link library, GDI.EXE, for the implementation of GDI functions in 16-bit programs, this functionality has been integrated into GDI32.DLL in Windows 10 and 11.



Device-Independent Graphics

GDI is designed to support device-independent graphics, enabling Windows applications to function seamlessly on any compatible graphics output device.

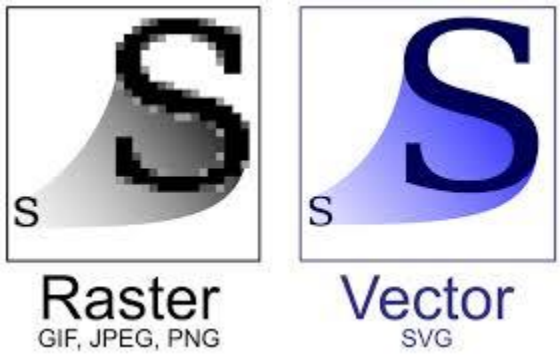
This goal is achieved by providing mechanisms that isolate programs from the unique characteristics of different output devices.



Raster vs. Vector Devices

Graphics output devices can be categorized into two main types: raster devices and vector devices.

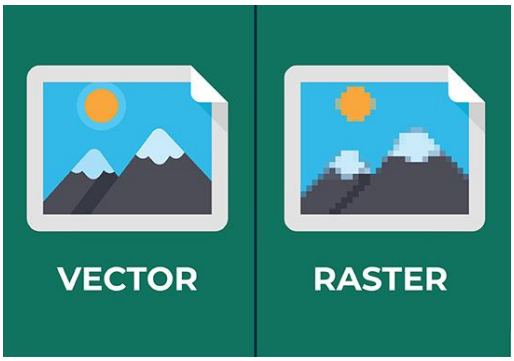
Raster devices, which include video display adapters, dot-matrix printers, and laser printers, represent images as a rectangular pattern of dots.



Vector devices, primarily limited to plotters these days, generate images using lines.

GDI as a High-Level Interface

Traditional computer graphics programming often relies solely on vectors, introducing an abstraction layer between the program and the hardware.



While output devices utilize pixels for graphics representation, the program doesn't directly interact with the hardware in terms of pixels.

The Windows GDI can be used as both a high-level vector drawing system and a relatively low-level pixel manipulation tool.

In this sense, GDI parallels C's position among programming languages.

C is renowned for its portability across different operating systems and environments, while also allowing programmers to perform low-level system functions often inaccessible in other high-level languages.

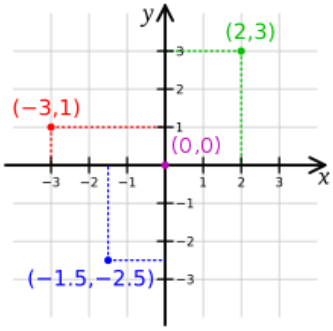
Just as C is sometimes considered a "high-level assembly language," GDI can be viewed as a high-level interface to the graphics device hardware.

Coordinate Systems

Windows employs a pixel-based coordinate system by default.

Traditional graphics languages typically use a "virtual" coordinate system with horizontal and vertical axes ranging from 0 to 32,767, for instance.

While some graphics languages restrict pixel coordinates, Windows GDI allows using both systems, along with additional coordinate systems based on physical measurements.



Programmers' Control

Programmers can opt for a virtual coordinate system to maintain a level of abstraction from the hardware or utilize the device coordinate system for closer hardware interaction.

Some programmers argue that using pixels signifies a departure from device independence.

However, as discussed earlier, this is not entirely true.

The key lies in using pixels in a device-independent manner.

This requires the graphics interface language to provide mechanisms for a program to determine the hardware characteristics of the device and make appropriate adjustments.

For instance, in the SYSMETS programs, the pixel size of a standard system font character was used to space text on the screen.

This approach **allowed the programs to adapt to different display adapters** with varying resolutions, text sizes, and aspect ratios.

Other methods for determining display sizes will be introduced in subsequent chapters.

Monochrome Displays

In the early days of Windows, many users ran the operating system with a monochrome display.

This meant that the display could only **display two colors: black and white.**

As a result, **GDI was designed** to allow programmers to write programs without having to worry about color.

Windows would **automatically convert** any colors used in the program to shades of gray.

Color Displays

In the early days of personal computing, **color displays were a luxury reserved for high-end workstations and graphics design studios.**

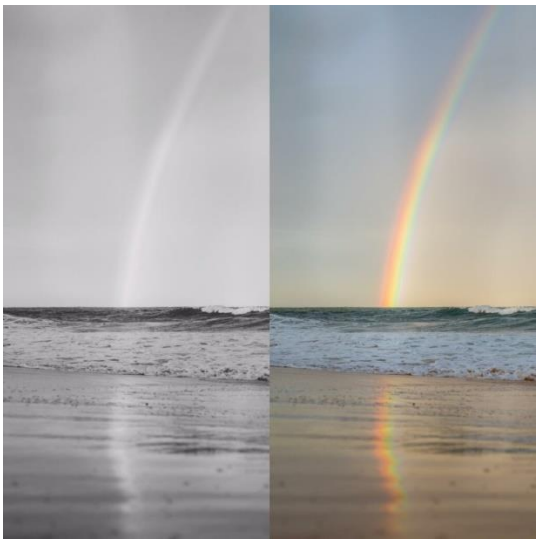
For most users, the world of computing was awash in shades of gray.

However, with the relentless advancement of technology and the decreasing cost of color components, color displays gradually became more accessible, eventually becoming the standard for personal computers.

Today, modern video displays used with Windows 10 and Windows 11 are capable of rendering millions of colors, commonly referred to as "true color."

This vast color palette allows for vibrant, photorealistic visuals and a rich, immersive computing experience.

The transition from monochrome to true color has revolutionized the way we interact with computers, transforming them from mere text-based machines into powerful tools for creativity and entertainment.

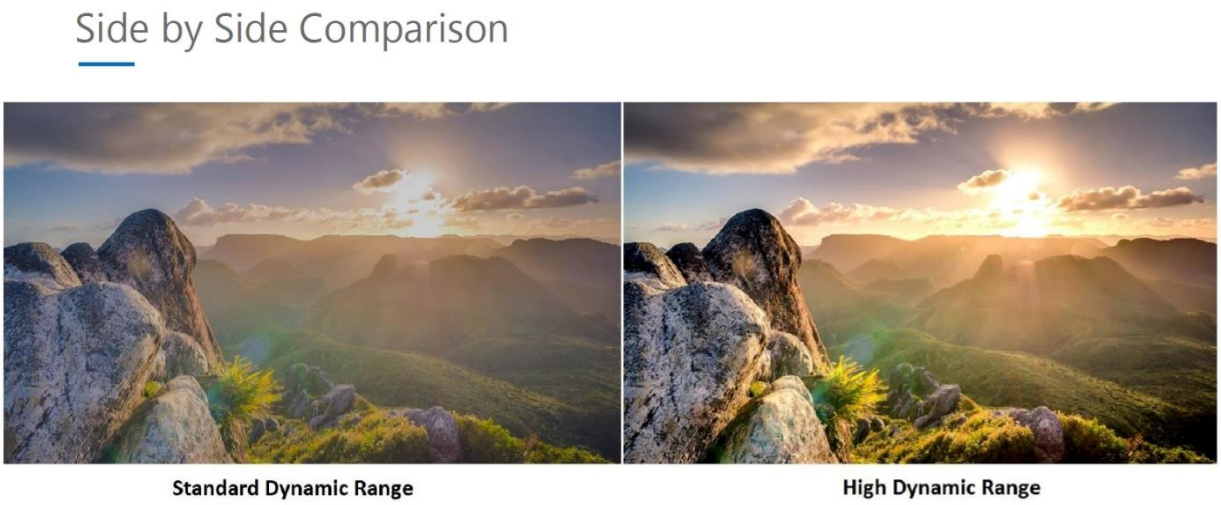


While true color is now the norm, it's worth noting that not all displays are created equal.

Some displays offer wider color gamuts, capable of reproducing a broader range of colors than standard models.

This enhanced color fidelity is particularly beneficial for professional applications like graphic design and video editing, where color accuracy is crucial.

Moreover, recent advancements in display technology have introduced features like high dynamic range (HDR), which further expands the color and contrast capabilities of displays.



HDR displays can render a wider range of brightness and luminance levels, resulting in more realistic and lifelike images.

Inkjet vs. Laser Printers

Inkjet printers have brought low-cost color printing to the masses, but many users still prefer black-only laser printers for high-quality output.

It is possible to use these devices blindly, but your program can also determine how many colors are available on the particular output device and take best advantage of the hardware.

Device Dependencies

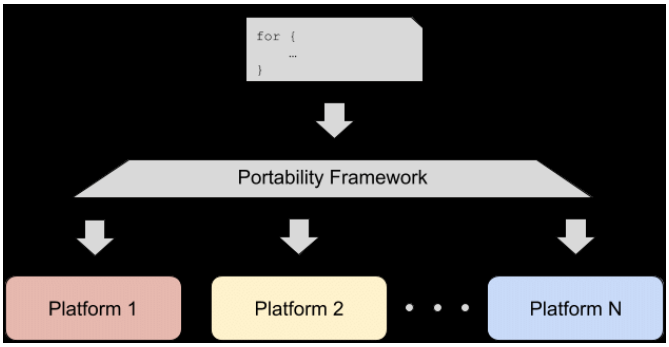
Just as you can write C programs that have subtle **portability problems** when they run on other computers, you can also inadvertently let device dependencies creep into your Windows programs.

That's part of the price of not being fully insulated from the hardware.

Device dependencies are any hardware or software requirements that a Windows program needs to function properly. These dependencies can be related to specific input devices, output devices, or software libraries that are not part of the Windows operating system itself.

Why are device dependencies a problem?

Portability issues: If a program relies on specific hardware or software that is not available on all computers, it may not run or function correctly on those systems. This can make it difficult to distribute and maintain programs that have device dependencies.



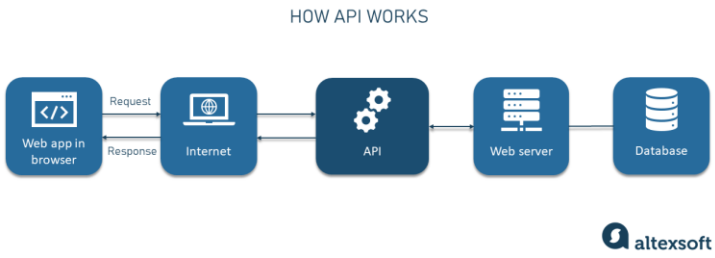
Compatibility issues: Device dependencies can also lead to compatibility issues between different versions of Windows. For example, a program that relies on a specific hardware feature that was introduced in Windows 10 may not work properly on Windows 8.1 or earlier versions of the operating system.

Security vulnerabilities: Device dependencies can also introduce security vulnerabilities into programs. For example, a program that relies on a specific software library may be vulnerable to attacks if that library has a known security flaw.

How to avoid device dependencies?

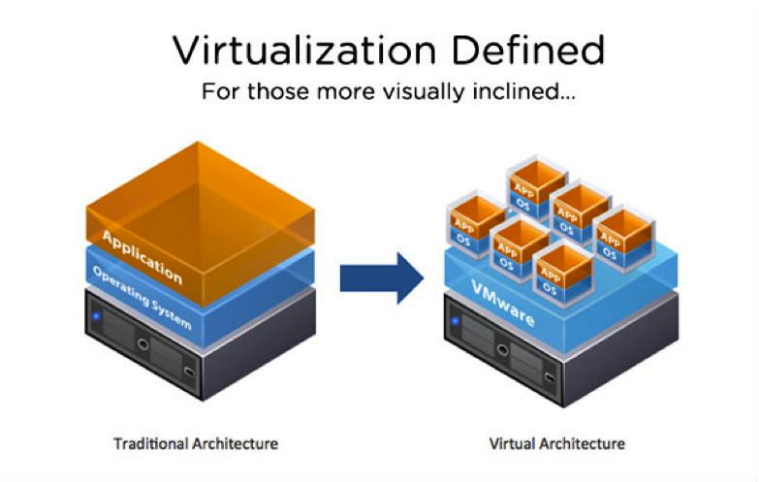
To avoid device dependencies in your Windows programs, you should:

Use standard APIs: When developing Windows programs, it is important to use standard APIs (Application Programming Interfaces) that are provided by the Windows operating system. These APIs are designed to work with a wide variety of hardware and software, so they are less likely to cause portability or compatibility issues.



Test on a variety of hardware: It is also important to test your Windows programs on a variety of hardware to ensure that they work properly on different systems. This will help you to identify and fix any device dependencies that may be present in your code.

Use virtualization technologies: Virtualization technologies, such as Hyper-V, can be used to create virtual machines that allow you to test your programs on a variety of hardware configurations without having to install the software on each machine individually.



Animation Support

GDI is **generally a static display system** with only limited animation support.

If you need to write sophisticated animations for games, you should explore Microsoft DirectX, which provides the support you'll need.

In summary, it is important to consider the following when writing graphics programs for Windows:

- The color capabilities of the display device.
- The type of output device (e.g., printer).
- The limitations of GDI.
- GDI is a powerful tool for creating graphics programs for Windows.
- GDI is designed to be device independent, so that programs can run on a variety of hardware.
- GDI has some limitations, such as its limited animation support.
- If you need to write sophisticated animations, you should explore Microsoft DirectX.

TYPES OF GDI FUNCTION CALLS

The GDI function calls can be broadly categorized into the following groups:

Device Context Management:

BeginPaint and EndPaint: These functions are part of the USER module and are used to obtain and release a device context during the WM_PAINT message.

GetDC and ReleaseDC: These functions are used to obtain and release a device context during other messages.

Device Context Information Access:

GetTextMetrics: This function retrieves information about the dimensions of the currently selected font in the device context.

DEVCAPS1: This program obtains more general device context information.

Drawing Functions:

TextOut: This function displays text in the client area of the window.

Other drawing functions: GDI provides functions for drawing lines, filled areas, and other graphical elements.

Device Context Attribute Management:

SetTextColor: This function specifies the color of text drawn using TextOut and other text output functions.

SetTextAlign: This function informs GDI that the starting position of the text string in TextOut should be the right side of the string rather than the left.

GDI Object Manipulation:

CreatePen, CreatePenIndirect, and ExtCreatePen: These functions create logical pens, which define the attributes of lines drawn using GDI.

Pen Selection and Deselection: Pens are selected into the device context using their handle and deselected when no longer needed. Destroying pens is crucial to release allocated memory.

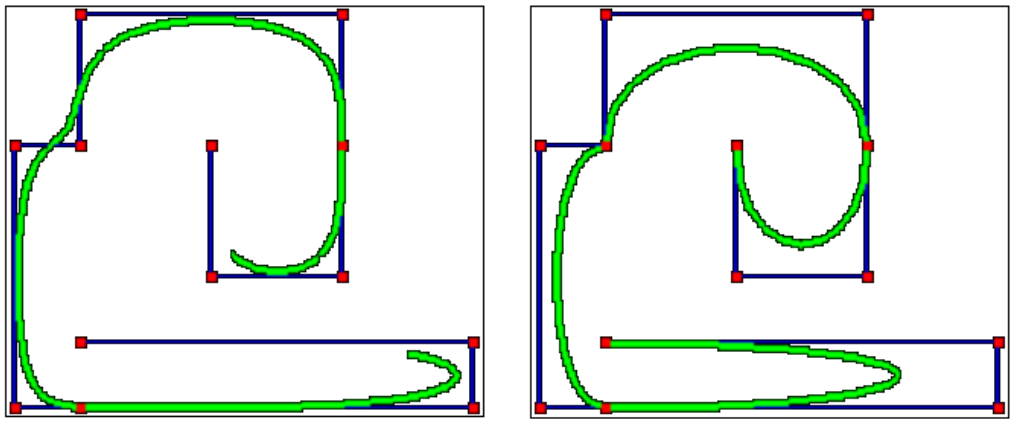
Brushes, Fonts, Bitmaps: GDI objects also include brushes for filling enclosed areas, fonts for text rendering, and bitmaps for image display.

These categories provide a comprehensive overview of the GDI function calls and their respective purposes.

GDI PRIMITIVES

Lines and Curves

Lines are the fundamental building blocks of any vector graphics drawing system. GDI supports a variety of line types, including straight lines, rectangles, ellipses (including circles), arcs (partial curves on an ellipse's circumference), and Bezier splines.



B - Spline Bezier Spline

If you need to draw a different type of curve, you can approximate it using a polyline, a series of very short lines that define the curve's shape. GDI renders lines using the current pen selected in the device context.

Filled Areas

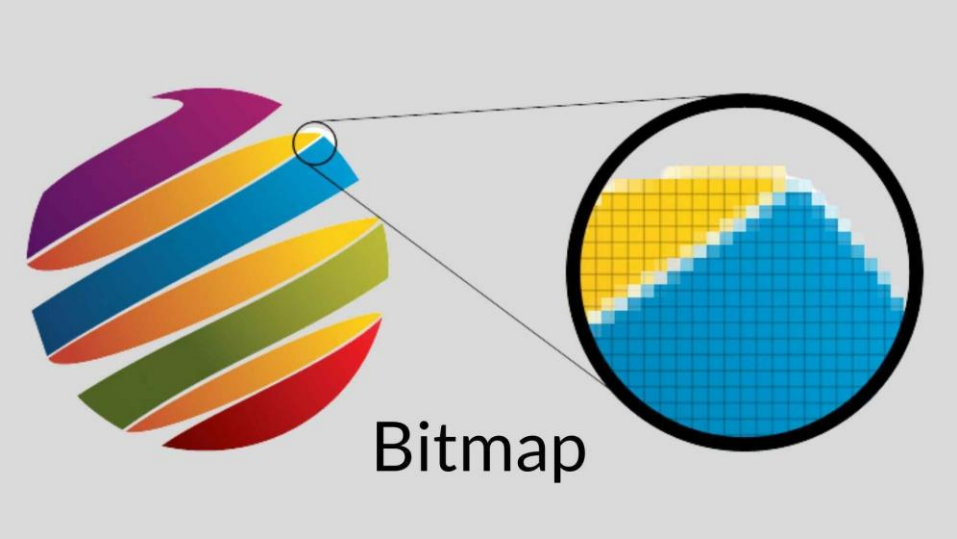
When a series of lines or curves encloses an area, you can instruct GDI to fill that area with the current GDI brush object.

This brush can be a solid color, a pattern (such as a series of horizontal, vertical, or diagonal hatch marks), or a bitmapped image that is replicated vertically or horizontally within the area.

Bitmaps

A bitmap, also known as a raster image, is a rectangular array of bits that correspond to the pixels of a display device.

Bitmaps are the foundation of raster graphics and are commonly used for displaying complex images, including real-world scenes, on the video display or printer.



Bitmaps are also employed for displaying small images that require rapid rendering, such as icons, mouse cursors, and toolbar buttons.

GDI supports two types of bitmaps: device-dependent bitmaps, which are GDI objects, and device-independent bitmaps (DIBs), which were introduced in Windows 3.0 and can be stored in disk files. Bitmaps will be discussed in detail in Chapters 14 and 15.

Text

Text, unlike other aspects of computer graphics, is not entirely mathematical; it is rooted in centuries of traditional typography, considered an art form by many typographers and design enthusiasts.

Consequently, text is often the most complex component of any computer graphics system, but it is also the most crucial aspect, assuming literacy remains the norm.

Among the largest data structures in Windows are those used to define GDI font objects and retrieve font information.

Starting with Windows 3.1, GDI began supporting TrueType fonts, which are based on filled outlines that can be manipulated using other GDI functions.

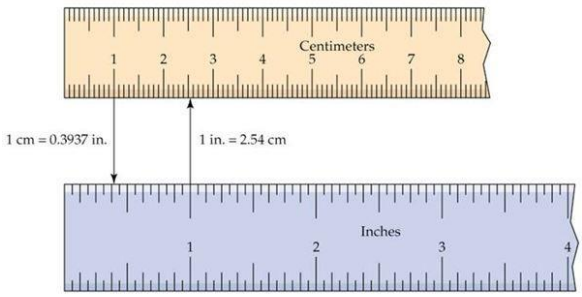
For backward compatibility and low memory requirements, Windows 98 continues to support older bitmap-based fonts. Fonts will be discussed in detail in Chapter 17.

These four categories encompass the primary GDI primitives and provide a solid foundation for creating a wide range of graphical elements.

Mapping Modes and Transforms

While the default drawing unit is pixels, GDI allows you to draw in other units, such as inches, millimeters, or any custom unit you define.

This is achieved through GDI mapping modes, which establish a relationship between device coordinates (pixels) and logical coordinates (your chosen units).



Windows NT also supports a traditional "world transform" represented by a 3x3 matrix.

This transform enables skewing and rotation of graphics objects, providing more flexibility in positioning and manipulating graphical elements.

Metafiles

A metafile is a file that contains a description of an image, rather than the image itself.

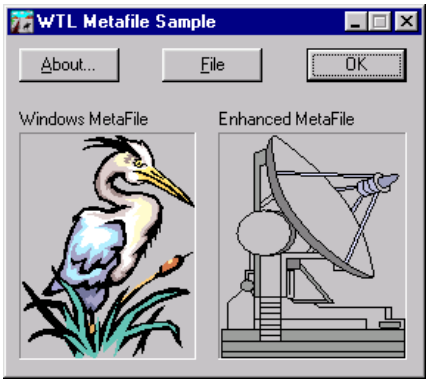
Metafiles are often used to store vector graphics, which can be scaled to any size without losing quality.

Emojis, on the other hand, are raster graphics, which means that they are made up of a grid of pixels.

This makes them less scalable than vector graphics, but it also means that they can be displayed more quickly and efficiently.

Metafiles are essentially collections of GDI commands stored in a binary format.

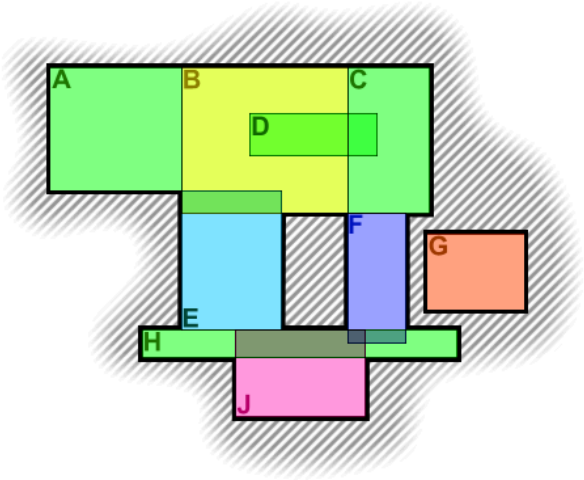
They are primarily used to transfer representations of vector graphic drawings through the clipboard, allowing for seamless exchange of graphics data between applications.



A metafile can be played back on any device that supports GDI, ensuring consistent rendering across different systems.

Regions

A region in GDI represents a complex area of any shape and is typically defined as a Boolean combination of simpler regions.



These complex regions can be stored internally in GDI as a series of scan lines derived from their original definition.

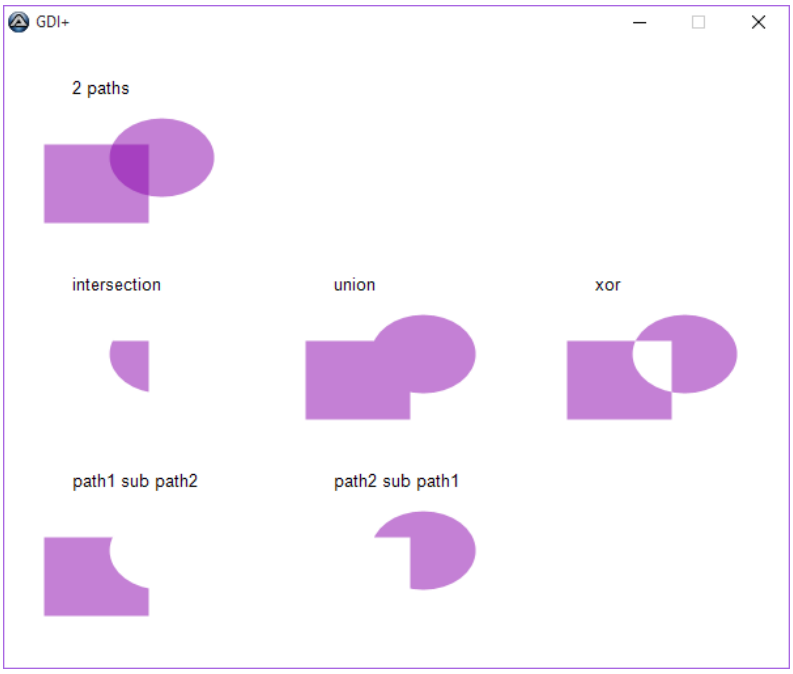
Regions are particularly useful for outlining, filling, and clipping operations.

Paths

Similar to regions, **paths are collections of straight lines and curves** stored internally in GDI.

They serve a similar purpose to regions, being used for drawing, filling, and clipping.

Additionally, paths can be converted to regions, providing flexibility in manipulating graphical shapes.



Clipping

Clipping is a technique that restricts drawing to a specific section of the client area.

The clipping area can be either rectangular or non-rectangular, and it is typically defined using a region or a path.

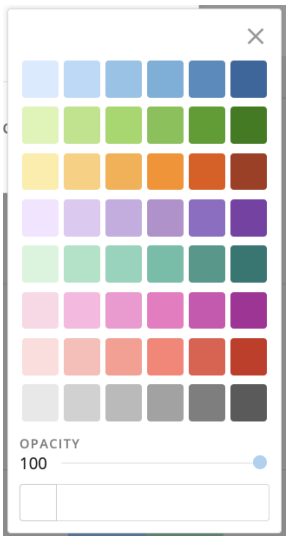
Clipping is valuable for preventing graphics from overlapping or extending beyond desired boundaries.

Palettes

Palettes are custom color sets used to enhance the visual appeal of graphics, particularly on displays that support a limited number of colors.

Windows reserves a subset of these colors for system use, while the remaining colors can be customized to accurately represent the colors of real-world images stored in bitmaps.

Palettes are primarily relevant for older systems with limited color capabilities.



Printing

While this chapter focuses on graphics display, most of the concepts covered can be applied to printing as well.

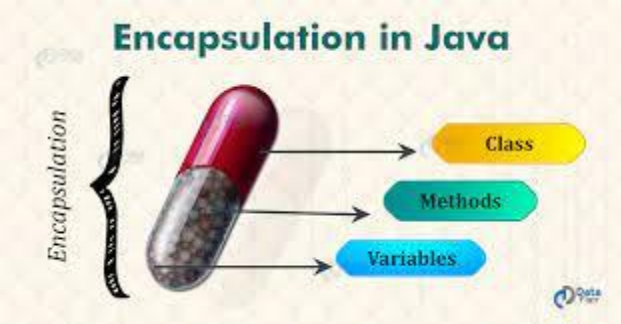
GDI provides a comprehensive set of functions for controlling printing output, allowing you to print text, graphics, and other visual elements with precision.

Printing-specific topics, such as printer drivers and page layout, will be discussed in more detail in Chapter 13.

DEVICE CONTEXT

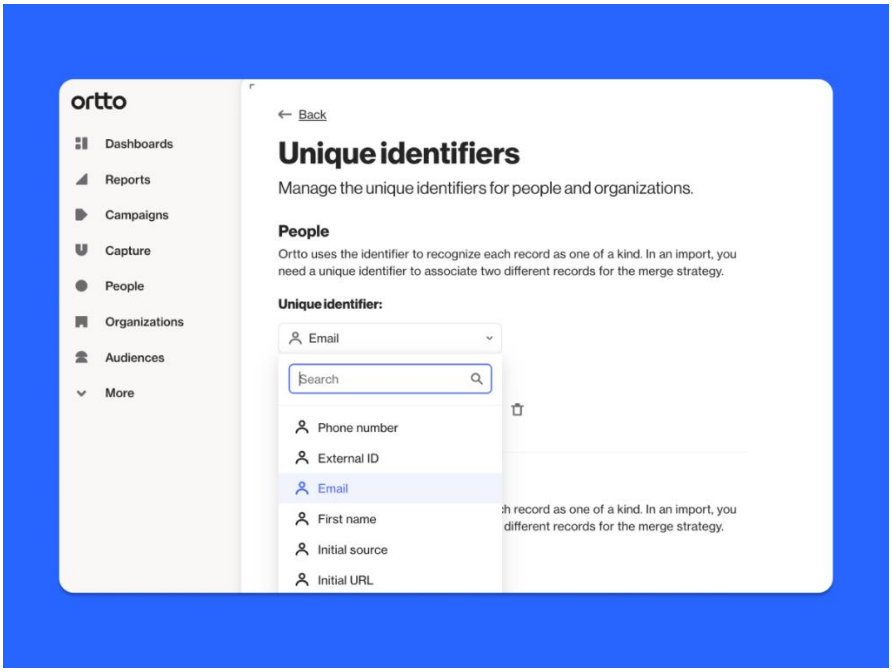
The device context (DC) is a fundamental concept in Windows GDI, acting as a bridge between your application and the graphics output device.

It encapsulates the necessary information and attributes for rendering graphics onto a specific device, such as the video display or a printer.



Obtaining a Device Context Handle

Before you can start drawing graphics, you need to obtain a device context handle. This handle serves as a unique identifier for the device context and provides access to its attributes and drawing functions.



Methods for Obtaining Device Context Handles

Windows offers several methods for acquiring device context handles:

BeginPaint and EndPaint: This method is commonly used for handling the WM_PAINT message, which informs your application that the client area of the window needs to be repainted. BeginPaint retrieves the device context handle and validates the invalid region of the client area. EndPaint releases the device context handle.

GetDC and ReleaseDC: This method allows you to obtain a device context handle directly for the client area of a specified window. The handle needs to be released using ReleaseDC when no longer needed.

GetWindowDC and ReleaseDC: This method is similar to GetDC, but it retrieves a device context handle that encompasses the entire window, including its non-client areas like the title bar, menu, scrollbars, and frame.

Considerations for Obtaining Device Context Handles

When obtaining a device context handle while processing a WM_PAINT message, it's crucial to release it before exiting the window procedure to avoid memory leaks.

For printer device contexts, the rules for releasing the handle are more flexible, as printing operations typically involve multiple contexts.

Device Context Attributes

The device context holds a collection of attributes that determine how GDI functions operate on the target device. These attributes include:

Font: Specifies the font to be used for text rendering.

Text color: Defines the color of text drawn using GDI functions.

Background color: Sets the color of the background area behind drawn text.

Intercharacter spacing: Adjusts the spacing between characters in drawn text.

Pen: Defines the characteristics of lines drawn using GDI functions, including line width, style, and color.

Brush: Determines the appearance of filled areas, such as patterns or images.

Clipping region: Defines the area within which drawing operations will be confined.

Modifying Device Context Attributes

To modify device context attributes, you can use specific GDI functions that target each attribute.

For instance, SetTextColor changes the color of text drawn using GDI functions, while SetBkColor alters the background color.

```
hdc = BeginPaint (hwnd, &ps) ;
//other program lines
EndPaint (hwnd, &ps) ;
```

In this code, the BeginPaint function returns a device context handle to the variable hdc. The hwnd parameter is the handle of the window for which you are obtaining the device context. The &ps parameter is a pointer to a PAINTSTRUCT structure, which contains information about the painting operation, such as the invalid region of the window's client area.

The [other program lines] section is where you would put your drawing code. Once you have finished drawing, you call the EndPaint function to release the device context handle. The hwnd parameter is the same as the one you passed to BeginPaint, and the &ps parameter is the same pointer to the PAINTSTRUCT structure.

This is a common pattern for drawing in Windows applications. It ensures that your application only draws in the invalid region of the window, and that it releases the device context handle when it is no longer needed.

Here is an example of how to obtain a device context handle for the client area of a window using BeginPaint and EndPaint:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg) {
        case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);

            // Draw graphics using hdc

            EndPaint(hwnd, &ps);
            return 0;
        }
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            return 0;
        }
    }

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

Here is an example of how to obtain a device context handle for the entire window using GetDC and ReleaseDC:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg) {
        case WM_PAINT:
        {
            HDC hdc = GetDC(hwnd);

            // Draw graphics using hdc

            ReleaseDC(hwnd, hdc);
            return 0;
        }
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            return 0;
        }
    }

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

Additional Notes

- You should always release device context handles as soon as you are finished using them. This will prevent memory leaks and other problems.
- If you obtain a device context handle while processing a message, you should release it before exiting the window procedure.
- You can use the GetDCEx function to obtain a device context handle with more advanced options.
- For more information on device contexts, please refer to the Microsoft Windows documentation.
- The device context plays a pivotal role in Windows GDI, providing a mechanism for applications to communicate with graphics output devices and control the rendering of graphics.
- Understanding how to obtain, manage, and modify device context handles and attributes is essential for effective graphics programming in Windows.

The different device context handles in Windows:

BeginPaint and EndPaint: These functions are used to obtain a device context handle for the client area of a window. The BeginPaint function returns a handle to the device context, and the EndPaint function releases the handle.

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg) {
        case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);

            // Draw graphics using hdc

            EndPaint(hwnd, &ps);
            return 0;
        }
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            return 0;
        }
    }

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

GetDC and ReleaseDC: These functions are used to obtain a device context handle for the entire window, including the non-client area. The GetDC function returns a handle to the device context, and the ReleaseDC function releases the handle.

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg) {
        case WM_PAINT:
        {
            HDC hdc = GetDC(hwnd);

            // Draw graphics using hdc

            ReleaseDC(hwnd, hdc);
            return 0;
        }
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            return 0;
        }
    }

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

GetWindowDC: This function is used to obtain a device context handle for the entire window, including the non-client area. It is similar to GetDC, but it is less commonly used.

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg) {
        case WM_PAINT:
        {
            HDC hdc = GetWindowDC(hwnd);

            // Draw graphics using hdc

            ReleaseDC(hwnd, hdc);
            return 0;
        }
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            return 0;
        }
    }

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

CreateDC: This function is used to obtain a device context handle for a specific device, such as the display or a printer. The CreateDC function returns a handle to the device context, and the DeleteDC function releases the handle.

```
hdc = CreateDC(TEXT("DISPLAY"), NULL, NULL, NULL);

[other program lines]

DeleteDC(hdc);
```

CreateIC: This function is used to obtain an "information context" handle for a specific device. The CreateIC function returns a handle to the information context, and the DeleteDC function releases the handle. You cannot use an information context handle to draw on the device.

```
hdc = CreateIC(TEXT("DISPLAY"), NULL, NULL, NULL);

[other program lines]

DeleteDC(hdc);
```

CreateCompatibleDC: This function is used to obtain a "memory device context" handle. The CreateCompatibleDC function returns a handle to the memory device context, and the DeleteDC function releases the handle. You can select a bitmap into the memory device context and use GDI functions to draw on the bitmap.

```
hdcMem = CreateCompatibleDC(hdc);

[other program lines]

DeleteDC(hdcMem);
```

CreateMetaFile: This function is used to obtain a metafile device context handle. The CreateMetaFile function returns a handle to the metafile device context, and the CloseMetaFile function releases the handle. You cannot use a metafile device context handle to draw on the device.

```
hdcMeta = CreateMetaFile(pszFilename);

[other program lines]

hmf = CloseMetaFile(hdcMeta);

DeleteDC(hdcMeta);
```

In addition to these device context handles, there are also several other specialized device context handles, such as the printer device context handle and the enhanced metafile device context handle. These handles are discussed in more detail in Chapter 13 and Chapter 18 of the book.

Creating a Device Context for the Entire Display

You can obtain a device context handle for the entire display by calling CreateDC with the following parameters:

```
hdc = CreateDC(TEXT("DISPLAY"), NULL, NULL, NULL);
```

This device context handle can be used to draw on the entire display. However, it is generally considered impolite to write outside your own window.

Creating an Information Context

Sometimes you need only to obtain some information about a device context and not do any drawing. In these cases, you can obtain a handle to an "information context" by using CreateIC. The arguments are the same as for the CreateDC function. For example:

```
hdc = CreateIC(TEXT("DISPLAY"), NULL, NULL, NULL);
```

You can't write to the device by using this information context handle.

Creating a Memory Device Context

When working with bitmaps, it can sometimes be useful to obtain a "memory device context".

A memory device context is a device context that resides in memory rather than on a physical device.

This can be useful for storing and manipulating bitmaps without having to write them directly to the display.

To create a memory device context, you can call CreateCompatibleDC with a handle to another device context as an argument. For example:

```
hdcMem = CreateCompatibleDC(hdc);
```

You can then select a bitmap into the memory device context and use GDI functions to draw on the bitmap.

Creating a Metafile Device Context

A **metafile is a collection of GDI function calls encoded in binary form.**

You can create a metafile by obtaining a metafile device context.

To create a metafile device context, you can call CreateMetaFile with the name of the metafile as an argument. For example:

```
hdcMeta = CreateMetaFile(pszFilename);
```

During the time the metafile device context is valid, any GDI calls you make using hdcMeta are not displayed but become part of the metafile.

When you call CloseMetaFile, the device context handle becomes invalid. The function returns a handle to the metafile (hmf).

Releasing Device Context Handles

When you are finished using a device context handle, you should always release it.

This will free up the resources associated with the handle and make it available for other applications to use.

To release a device context handle, you can use the DeleteDC function. For example:

```
DeleteDC(hdc);
```

You can also release a device context handle that was obtained with GetDC or GetWindowDC by calling ReleaseDC. For example:

```
ReleaseDC(hwnd, hdc);
```

Obtaining Device Context Information

The GetDeviceCaps function is used to retrieve information about a device context.

The function takes two arguments: a handle to a device context and an index that specifies the information to retrieve. The function returns the requested information as an integer value.

Commonly Used Device Context Information

Here are some of the most commonly used device context information indices:

- HORZRES: Retrieves the width of the device in pixels.
- VERTRES: Retrieves the height of the device in pixels.
- LOGPIXELSX: Retrieves the horizontal resolution of the device in pixels per inch.
- LOGPIXELSY: Retrieves the vertical resolution of the device in pixels per inch.
- PHYSICALWIDTH: Retrieves the width of the device in physical units (e.g., millimeters).
- PHYSICALHEIGHT: Retrieves the height of the device in physical units (e.g., millimeters).
- RASTERCAPS: Retrieves the raster capabilities of the device. This information can be used to determine whether the device can draw bitmaps, lines, and other graphics objects.

Example Code

The following code snippet retrieves the width of the video display in pixels:

```
HDC hdc = GetDC(NULL);
int width = GetDeviceCaps(hdc, HORZRES);
ReleaseDC(NULL, hdc);
```

The following code snippet retrieves the vertical resolution of the printer in pixels per inch:

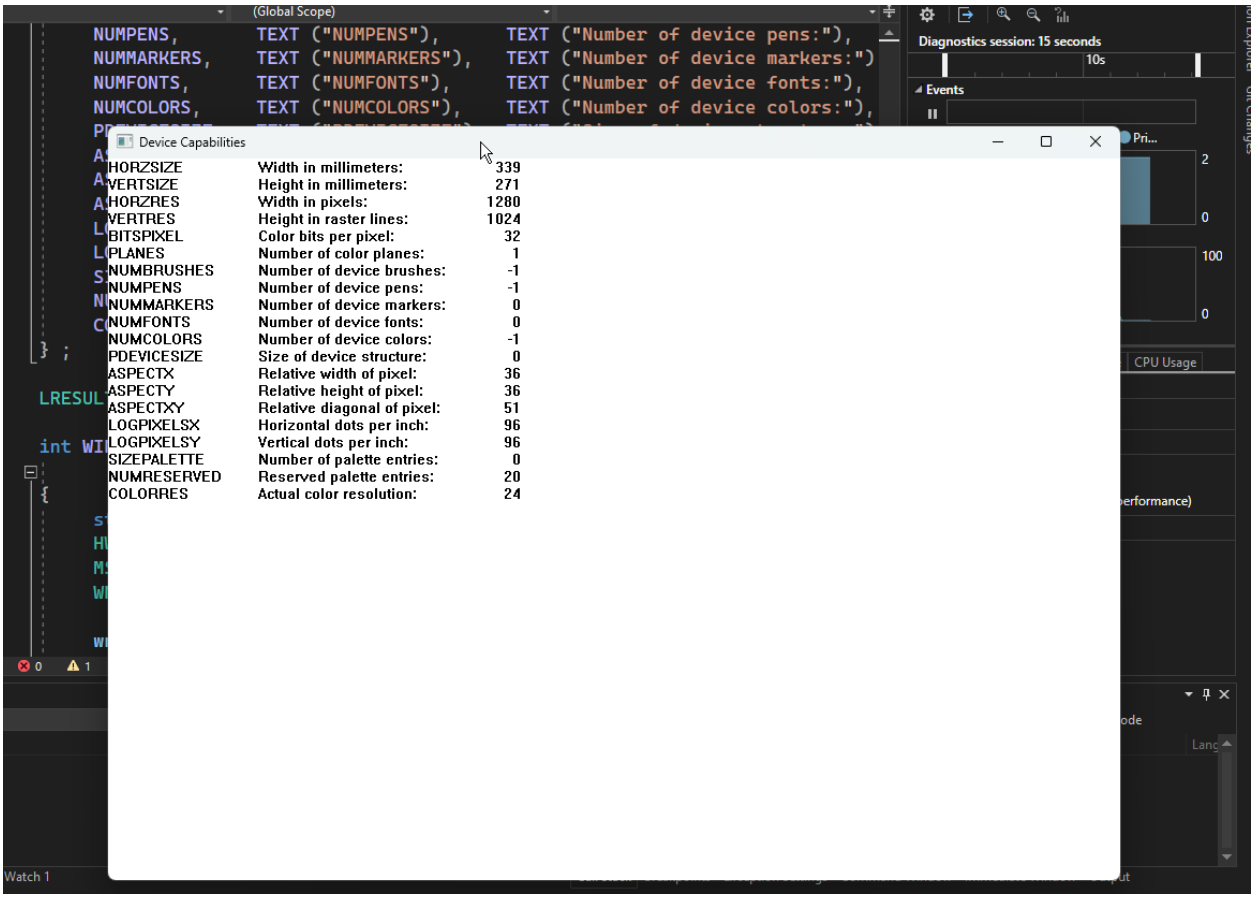
```
HDC hdc = CreateDC(TEXT("DISPLAY"), NULL, NULL, NULL);
int resolution = GetDeviceCaps(hdc, LOGPIXELSY);
DeleteDC(hdc);
```

The GetDeviceCaps function is used to retrieve information about a specific device context. If you want to retrieve information about the system as a whole, you can use the GetSystemMetrics function.

The GetDeviceCaps function returns an integer value, which may need to be converted to a more meaningful unit of measurement (e.g., millimeters, inches).

For more information on the GetDeviceCaps function, please refer to the Microsoft Windows documentation.

DEVCAPS1 PROGRAM



Code in chapter 5 folder.

Introduction

The **DEVCAPS1 program** is a Windows application that displays information about the video display device. The program uses the **GetDeviceCaps function to retrieve information** such as the width and height of the display, the number of colors that can be displayed, and the resolution of the display.

Program Structure

The program consists of **two main functions: WinMain and WndProc**. The WinMain function is responsible for initializing the application and creating the main window. The WndProc function is responsible for processing messages sent to the main window.

WinMain Function

The WinMain function first registers a window class for the main window. The window class defines the style of the window, the procedure that will be called to process messages sent to the window, and other information.

Next, the WinMain function creates the main window. The CreateWindow function is used to create the window.

The function takes several arguments, including the name of the window class to use, the text to display in the title bar of the window, the style of the window, the initial position of the window, the initial size of the window, and the instance handle of the application.

After the main window has been created, the WinMain function shows the window and updates it. The ShowWindow function is used to show the window, and the UpdateWindow function is used to update the window's contents.

Finally, the WinMain function enters a message loop. The message loop is responsible for processing messages sent to the application.

The GetMessage function is used to retrieve messages from the message queue, and the DispatchMessage function is used to send the messages to the appropriate window procedure.

WndProc Function

The WndProc function is responsible for processing messages sent to the main window. The function takes five arguments: the handle of the window, the message type, the wParam parameter, and the lParam parameter.

The WndProc function first checks the message type. If the message type is WM_CREATE, the function retrieves the text metrics for the current device context and calculates the character width and height.

If the message type is WM_PAINT, the function paints the contents of the window. The function first obtains a device context for the window.

Then, the function loops through the devcaps array, which contains information about the device capabilities.

For each item in the array, the function displays the label, description, and value of the device capability.

Finally, if the message type is WM_DESTROY, the function posts a WM_QUIT message to the message queue, which causes the application to terminate.

The DEVCAPS1 program is a simple example of how to use the GetDeviceCaps function to retrieve information about the video display device. The program can be used to learn more about the capabilities of the video display device and to troubleshoot problems with the display.

THE SIZE OF THE DEVICE

The size of the device is an important factor to consider when writing Windows applications. The size of the device will affect how you draw graphics, how you position windows, and how you handle user input.

Pixel Dimensions and Metrical Dimensions

The pixel dimensions of a device is the total number of pixels that the device displays horizontally and vertically.

The metrical dimensions of a device is the size of the display area of the device in inches or millimeters.

Resolution

Resolution is a measure of how many pixels are contained in a given area of a display. The resolution of a device is typically given as the number of pixels per inch (PPI).

Most Video Displays Have Square Pixels

Most video displays used with Windows today have square pixels. This means that the width of each pixel is equal to its height. However, this was not always the case. In the early days of Windows, some video displays had non-square pixels.

The Aspect Ratio of a Video Display

The aspect ratio of a video display is the ratio of its width to its height. The most common aspect ratio for video displays is 4:3. However, there are also some video displays with a 16:9 aspect ratio.

Windows Applications Should Not Assume a Specific Aspect Ratio

Windows applications should not assume that the video display has a specific aspect ratio. Instead, applications should use the GetDeviceCaps function to retrieve the dimensions of the display and adjust their layout accordingly.

Pixel Dimensions and Metrical Dimensions of Common Video Displays

The following table shows the pixel dimensions and metrical dimensions of some common video displays:

Pixel Dimensions	Metrical Dimensions	Resolution (PPI)
640 by 480 pixels	12.8 inches by 9.6 inches	50 PPI
800 by 600 pixels	15.3 inches by 11.5 inches	67 PPI
1024 by 768 pixels	19.2 inches by 14.4 inches	75 PPI
1280 by 1024 pixels	21.3 inches by 16.0 inches	96 PPI
1600 by 1200 pixels	25.6 inches by 19.2 inches	120 PPI

Determining the Pixel Dimensions of a Video Display

It is easy for a user to determine the pixel dimensions of a video display. To do so, they can run the Display applet in Control Panel and select the Settings tab. The pixel dimensions will be displayed in the area labeled Screen Area.

Traditional vs. Current Resolution

The traditional definition of resolution, as described in the text, is the number of pixels per inch. However, in the context of modern computer displays, the term "resolution" is often used to refer to the overall pixel dimensions of the display, such as 1920x1080 or 3840x2160.

Obtaining Pixel Dimensions

A **Windows application can obtain the pixel dimensions** of the display from GetSystemMetrics with the SM_CXSCREEN and SM_CYSCREEN arguments.

As you'll note from the **DEVCAPS1 program**, a program can obtain the same values from GetDeviceCaps with the HORZRES and VERTRES arguments.

HORZSIZE and VERTSIZE

The first two device capabilities, **HORZSIZE and VERTSIZE, are documented** as "Width, in millimeters, of the physical screen" and "Height, in millimeters, of the physical screen".

These seem like straightforward definitions until one begins to think through their implications.

How Does Windows Know the Monitor Size?

For example, given the nature of the interface between video display adapters and monitors, how can Windows really know the monitor size?

And what if you have a laptop (in which the video driver conceivably could know the exact physical dimensions of the screen) and you attach an external monitor to it? And what if you attach a video projector to your PC?

Windows' "Standard" Display Size

The HORZSIZE and VERTSIZE device capabilities are used to retrieve the width and height, in millimeters, of the physical screen.

In Windows 10 and 11, these values are derived from the HORZRES, VERTRES, LOGPIXELSX, and LOGPIXELSY values.

The HORZRES device capability is used to retrieve the width, in pixels, of the physical screen.

The VERTRES device capability is used to retrieve the height, in pixels, of the physical screen.

The LOGPIXELSX device capability is used to retrieve the horizontal resolution of the physical screen, in pixels per inch.

The LOGPIXELSY device capability is used to retrieve the vertical resolution of the physical screen, in pixels per inch.

The HORZSIZE and VERTSIZE values are calculated using the following formulas:

```
HORZSIZE = HORZRES * 25.4 / LOGPIXELSX
VERTSIZE = VERTRES * 25.4 / LOGPIXELSY
```

Where:

- HORZSIZE is the width of the physical screen, in millimeters.
- HORZRES is the width of the physical screen, in pixels.
- LOGPIXELSX is the horizontal resolution of the physical screen, in pixels per inch.
- VERTSIZE is the height of the physical screen, in millimeters.
- VERTRES is the height of the physical screen, in pixels.
- LOGPIXELSY is the vertical resolution of the physical screen, in pixels per inch.

Example:

- Consider a display with a resolution of 1920x1080 pixels and a horizontal resolution of 96 PPI and a vertical resolution of 96 PPI.

```
HORZSIZE = 1920 * 25.4 / 96 = 508 mm
VERTSIZE = 1080 * 25.4 / 96 = 300 mm
```

Therefore, the HORZSIZE and VERTSIZE values for this display would be 508 millimeters and 300 millimeters, respectively.

Display Applet and System Font

When you use the Display applet of the Control Panel to select a pixel size of the display, you can also select a size of your system font.

The reason for this option is that the font used for the 640 by 480 display may be too small to read when you go up to 1024 by 768 or beyond.

Instead, you'll want a larger system font. These system font sizes are referred to on the Settings tab of the Display applet as Small Fonts and Large Fonts.

Point Size and Typography

In traditional typography, the size of the characters in a font is indicated by a "point size."

A point is approximately 1/72 inch and in computer typography is often assumed to be exactly 1/72 inch.

Point Size and TEXTMETRIC Structure

In theory, the point size of a font is the distance from the top of the tallest character in the font to the bottom of descenders in characters such as j, p, q, and y, excluding accent marks.

For example, in a 10-point font this distance would be 10/72 inch. In terms of the TEXTMETRIC structure, the point size of the font is equivalent to the tmHeight field minus the tmInternalLeading field.

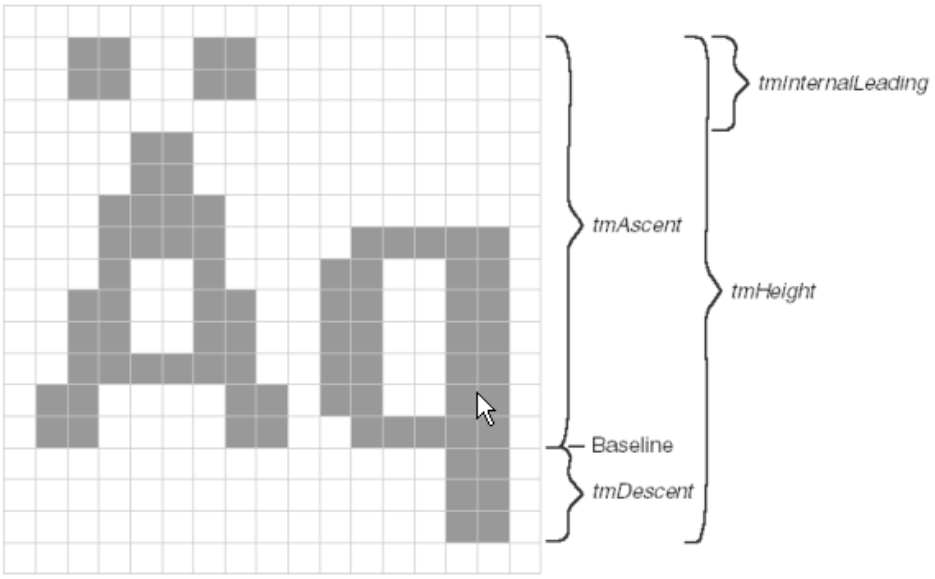


Figure 5–3. *The small font and the TEXTMETRIC fields.*

Font Size in Real-World Typography

In real-world typography, the point size of a font is not always an exact measure of the actual size of the font characters.

Font designers often make deliberate adjustments to the size of the characters to achieve a desired aesthetic or to accommodate the specific features of the font.

TMHeight and Line Spacing

The tmHeight field of the TEXTMETRIC structure indicates the recommended vertical spacing between successive lines of text.

This spacing is also typically measured in points. For instance, a 12-point line spacing means that the baselines of consecutive lines of text should be 12/72 (or 1/6) inch apart.

Using a 10-point line spacing for a 10-point font is generally not advisable, as it could cause the lines of text to overlap.

Standard Font Size and Line Spacing

This book is printed using a 10-point font with a 13-point line spacing.

A 10-point font is considered comfortable for reading extended periods of time.

Fonts with a point size much smaller than 10 can be difficult to read for extended durations.

Windows System Font

The default font used by Windows, regardless of whether it's the "small font" or the "large font," and irrespective of the chosen video pixel dimension, is assumed to be a 10-point font with a 12-point line spacing.

This might seem counterintuitive, as the system fonts are labeled as "small font" and "large font" even though they're both 10-point fonts.

The Role of Display Resolution

The key to understanding this apparent inconsistency lies in the concept of display resolution. When you select the small font or the large font in the Display applet of the Control Panel, you're essentially selecting an assumed dots per inch (DPI) value for the video display.

Selecting the small font instructs Windows to assume a DPI of 96 dots per inch, while selecting the large font tells Windows to assume a DPI of 120 dots per inch.

Understanding Figure 5-3

Figure 5-3 illustrates the small font, which is based on a display resolution of 96 DPI. Despite being a 10-point font, the actual height of the characters is slightly larger than 10 points.

This is because the designer of the font intentionally made the characters slightly larger to improve readability at a lower DPI.

Line Spacing and Figure 5-4

The line spacing for the small font is 12 points, which translates to 16 pixels when multiplied by 96 DPI. This value is represented by tmHeight.

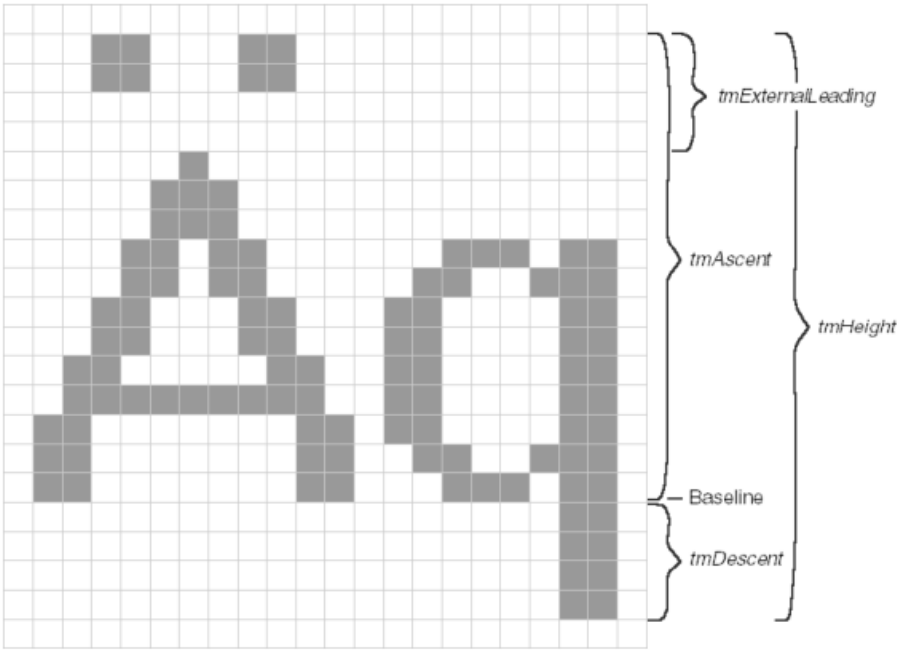


Figure 5–4. *The large font and the FONTMETRIC fields.*

Figure 5-4 shows the large font, which is based on a resolution of 120 DPI.

Conclusion

Again, it's a 10-point font, and the actual height of the characters is slightly larger than 10 points. The 12-point line spacing is equivalent to 20 pixels, which is represented by tmHeight.

Font size, line spacing, and display resolution are all interrelated factors that influence the appearance and readability of text on a screen.

The Windows system font, despite being labeled as "small" or "large," is always a 10-point font, but the actual size and spacing of the characters are adjusted based on the assumed DPI of the video display.

UNDERSTANDING DEVICE CAPABILITIES

Device capabilities are a set of values that provide information about the hardware and software capabilities of a computer system.

These capabilities can be retrieved using the GetDeviceCaps function in Windows applications.

Logical Pixels vs. Physical Pixels

The LOGPIXELSX and LOGPIXELSY device capabilities represent the logical resolution of the display in dots per inch (DPI), as opposed to the physical resolution.

Logical pixels are essentially a virtual representation of the display resolution that allows Windows to adjust the appearance of text and graphics based on the user's preferences.

HORZSIZE and VERTSIZE

The HORZSIZE and VERTSIZE device capabilities provide the assumed width and height, in millimeters, of the physical screen.

However, these **values are not directly related to the actual physical dimensions** of the screen.

Instead, they are calculated based on the pixel dimensions of the display and the logical resolution.

Calculating Logical Size from Physical Dimensions

The formulas used to calculate the logical size from the physical dimensions are:

```
Horizontal Size (mm) = 25.4 × Horizontal Resolution (pixels)/ Logical Pixels X (dots per inch)
Vertical Size (mm) = 25.4 × Vertical Resolution (pixels)/ Logical Pixels Y (dots per inch)
```

The 25.4 constant is used to convert from inches to millimeters.

The Importance of Logical Resolution

The reason why Windows uses logical resolution instead of physical resolution is to ensure that text and graphics appear consistently across different display configurations.

By adjusting the logical resolution based on the user's preferences, Windows ensures that a 10-point font will always appear the same size, regardless of the actual physical dimensions of the display.

Adapting to Different Display Configurations

Consider a 17-inch monitor with an actual display size of approximately 12 inches by 9 inches.

If you were to run Windows with the minimum required pixel dimensions of 640 by 480, the actual resolution would be 53 dots per inch.

A 10-point font, which would be perfectly readable on paper, would appear only 7 pixels in height on the screen, making it difficult to read.

Impact of Font Size and Resolution on Readability

Conversely, connecting a video projector with a much larger display area to your PC would result in a lower resolution, making it impractical to display a small font like a 10-point font.

By using logical resolution, Windows can adjust the size of text and graphics to ensure optimal readability across different display configurations.

Understanding the relationship between device capabilities, logical resolution, and physical dimensions is crucial for developing applications that can adapt to different display configurations in Windows.

By using the LOGPIXELSX, LOGPIXELSY, HORZSIZE, and VERTSIZE device capabilities, developers can ensure that their applications provide a consistent and user-friendly experience across a wide range of devices.

The Importance of a 10-Point Font

A 10-point font serves as a crucial reference point for ensuring readable text on a video display. Since a 10-point font is considered comfortably readable in printed form, it's essential that it remains readable on a screen.

Windows 98 Approach

In Windows 98, the HORZSIZE and VERTSIZE device capabilities are derived from the pixel dimensions of the display and the logical resolution (LOGPIXELSX and LOGPIXELSY).

This approach ensures that the size of text and graphics scales appropriately based on the user's preferences and the physical dimensions of the display.

Windows NT Approach

Windows NT employs a different strategy for defining HORZSIZE and VERTSIZE. Instead of relying on the actual physical dimensions of the display, these values are fixed to represent a standard monitor size. This approach was maintained for compatibility with older applications and hardware configurations.

Implications of Fixed HORZSIZE and VERTSIZE

The fixed HORZSIZE and VERTSIZE values in Windows NT can lead to inconsistencies with the values obtained from GetDeviceCaps using HORZRES, VERTRES, LOGPIXELSX, and LOGPIXELSY.

This discrepancy arises from the different methods used to calculate the logical size of the display.

Obtaining Actual Physical Dimensions

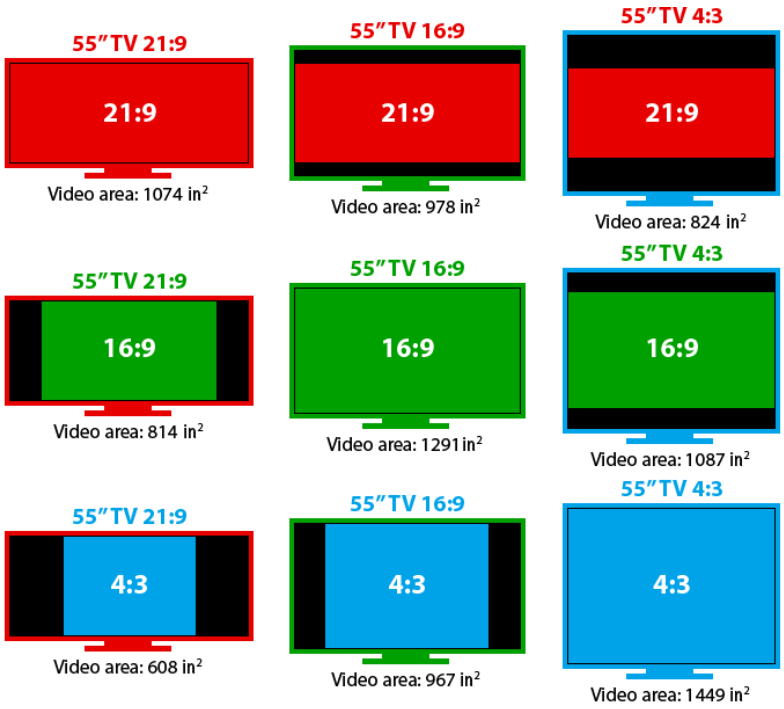
If an application requires the precise physical dimensions of the video display, the most reliable approach is to prompt the user through a dialog box.

This method ensures that the application obtains the actual dimensions from the user, eliminating any potential discrepancies.

ASPECTX, ASPECTY, and ASPECTXY

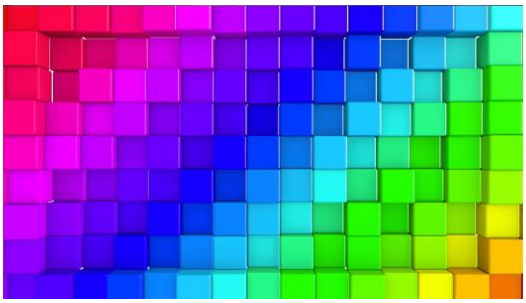
The ASPECTX, ASPECTY, and ASPECTXY device capabilities provide information about the relative width, height, and diagonal size of each pixel.

These values are relevant for applications that need to account for the aspect ratio of the display, such as video playback or image processing.

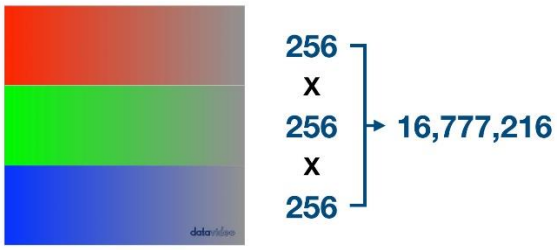


Color Depth and Pixel Bits

Color depth, also known as bits per pixel (bpp), refers to the number of bits used to represent the color of each pixel on a video display.



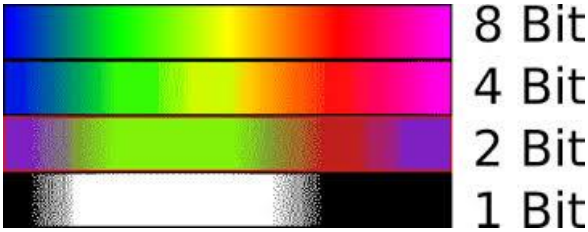
A higher color depth allows for a wider range of colors to be displayed, resulting in more vibrant and realistic images.



Common Color Depth Values

1-bit color: This is the simplest form of color display, capable of displaying only black and white pixels. It requires one bit per pixel.

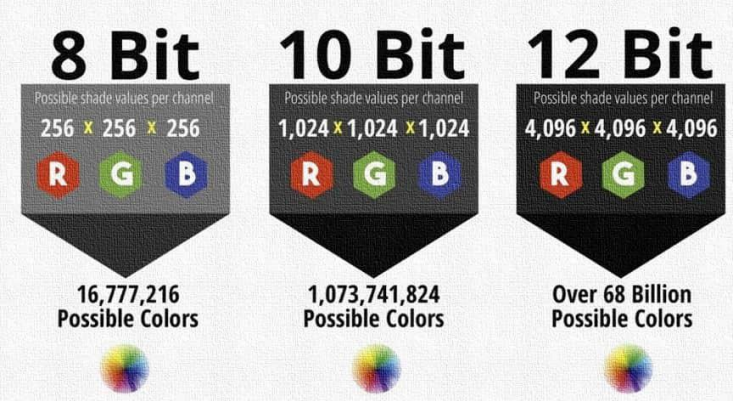
2-bit color: This color depth allows for four unique colors: black, white, red, and green. It requires two bits per pixel.



4-bit color: This color depth allows for 16 unique colors, typically a combination of black, white, red, green, blue, yellow, cyan, and magenta. It requires four bits per pixel.

8-bit color (High Color): This color depth, also known as 256 colors, allows for 256 unique colors. It is commonly used for basic graphics and video playback. It requires eight bits per pixel.

16-bit color (True Color): This color depth, also known as 32,768 colors, allows for a wider range of colors than 8-bit color. It is often used for high-quality graphics and video editing. It requires 16 bits per pixel.



24-bit color (True Color): This color depth, also known as full color, is the standard for most modern video displays. It allows for over 16 million unique colors, resulting in very realistic and detailed images. It requires 24 bits per pixel.

Color Palettes

In some cases, particularly with older video adapters, color palettes are used to manage the limited number of colors available.

A color palette is a table that defines the actual colors associated with each color index.

Windows programs can manipulate the color palette to customize the colors displayed on the screen.

Determining Color Depth and Color Count

The GetDeviceCaps function in Windows can be used to retrieve information about the color capabilities of a video adapter.

The PLANES and BITSPIXEL indices provide the number of color planes and the number of bits per pixel, respectively.

The iColors formula:

```
(iColors = 1 << (iPlanes * iBitsPixel))
```

Can be used to calculate the total number of colors that can be simultaneously rendered on the video adapter.

NUMCOLORS Index

The NUMCOLORS index of GetDeviceCaps can provide the number of colors obtainable with the video adapter, but its reliability varies depending on the color depth.

For 256-color video adapters, it returns the number of colors reserved by Windows, leaving the remaining colors to be managed by the palette manager.

For high-color and full-color display resolutions, NUMCOLORS often returns -1, making it less reliable for determining color count.

A more in-depth explanation of COLORREF values and how they represent RGB colors in Windows GDI:

COLORREF Structure

In Windows GDI, a COLORREF value is a 32-bit unsigned long integer that represents a specific color using the RGB color model. The RGB color model combines red, green, and blue intensities to create a wide range of colors.

The 32 bits of a COLORREF value are structured as follows:

```
Bits 31-24: Red intensity (8 bits)
Bits 23-16: Green intensity (8 bits)
Bits 15-8: Blue intensity (8 bits)
Bits 7-0: Unused (always zero)
```

This arrangement allows for 256 (8 bits) possible values for each color component, resulting in a total of 256 * 256 * 256 = 16,777,216 (approximately 16 million) unique colors.

RGB Macro for Creating COLORREF Values

The RGB macro in the WINGDI.H header file provides a convenient way to create COLORREF values from red, green, and blue intensity values. The syntax is:

```
RGB(red, green, blue);
```

For instance, **RGB(255, 255, 0) represents yellow**, as it combines maximum red and green intensities. Setting all three arguments to 0 **produces black**, while setting all three to 255 produces white.

Extracting Color Components from COLORREF

The GetRValue, GetGValue, and GetBValue macros can be used to extract the red, green, and blue intensity values from a COLORREF value.

These macros can be useful when working with functions that return RGB color values.

Dithering on Limited-Color Displays

On video adapters with limited color capabilities (16 or 256 colors), Windows employs a technique called dithering to simulate a wider range of colors.

Dithering involves strategically interspersing pixels of different colors to create an illusion of a wider color palette.

Determining Pure Non-Dithered Color

The GetNearestColor function can be used to determine the closest pure non-dithered color of a particular dithered color value.

This is useful when you want to display a color without the dithering effect.

Example Code

Here's an example of how to use the RGB macro and GetNearestColor function:

```
COLORREF yellowColor = RGB(255, 255, 0); // Create a COLORREF value for yellow
COLORREF pureColor = GetNearestColor(hdc, yellowColor); // Get the closest pure non-dithered color
```

This code snippet creates a COLORREF value for yellow and then uses GetNearestColor to determine the closest pure non-dithered color representation of yellow on the current video adapter.

In summary, COLORREF values play a crucial role in representing colors in Windows GDI.

They provide a standardized way to specify RGB colors and interact with various GDI functions that deal with color manipulation.

Understanding the structure and usage of COLORREF values is essential for developing graphics-intensive applications using Windows GDI.

Device Context Attribute	Default	Function(s) to Change	Function to Obtain
Mapping Mode	MM_TEXT	SetMapMode	GetMapMode
Window Origin	(0, 0)	SetWindowOrgEx	OffsetWindowOrgEx
Viewport Origin	(0, 0)	SetViewportOrgEx	OffsetViewportOrgEx
Window Extents	(1, 1)	SetWindowExtEx	SetMapMode
Viewport Extents	(1, 1)	SetViewportExtEx	SetMapMode
Pen	BLACK_PEN	SelectObject	SelectObject
Brush	WHITE_BRUSH	SelectObject	SelectObject
Font	SYSTEM_FONT	SelectObject	SelectObject
Bitmap	None	SelectObject	SelectObject
Current Position	(0, 0)	MoveToEx	LineTo
Background Mode	OPAQUE	SetBkMode	GetBkMode
Background Color	White	SetBkColor	GetBkColor
Text Color	Black	SetTextColor	GetTextColor
Drawing Mode	R2_COPYPEN	SetROP2	GetROP2
Stretching Mode	BLACKONWHITE	SetStretchBltMode	GetStretchBltMode
Polygon Fill Mode	ALTERNATE	SetPolyFillMode	GetPolyFillMode
Intercharacter Spacing	0	SetTextCharacterExtra	GetTextCharacterExtra
Brush Origin	(0, 0)	SetBrushOrgEx	GetBrushOrgEx
Clipping Region	None	SelectObject	SelectClipRgn

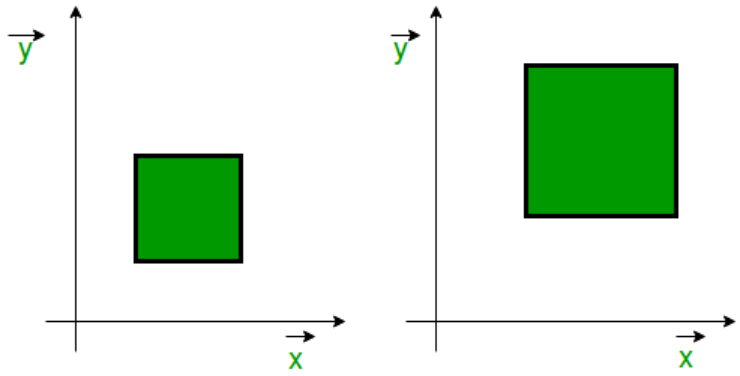
The terms provided above are related to the device context (DC) in Windows GDI (Graphics Device Interface). The DC is a data structure that encapsulates various attributes and settings that control how GDI functions interact with the display.

Mapping Mode

Defines the relationship between logical coordinates and device coordinates.

Logical coordinates are abstract units used to specify positions and dimensions, while device coordinates are physical pixels on the display.

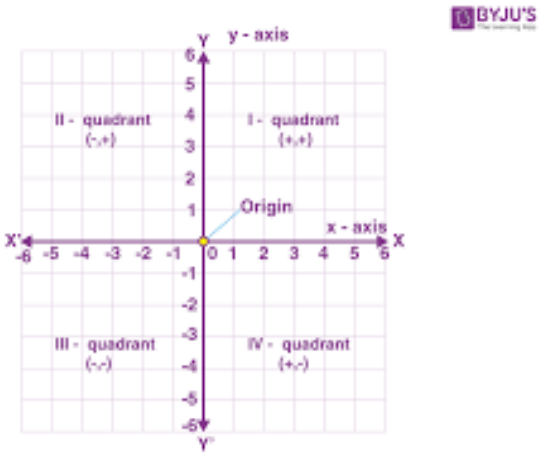
Different mapping modes provide different scaling and translation factors between these two coordinate systems.



Window Origin and Viewport Origin

The window origin and viewport origin are offsets that determine the starting point for mapping logical coordinates to device coordinates.

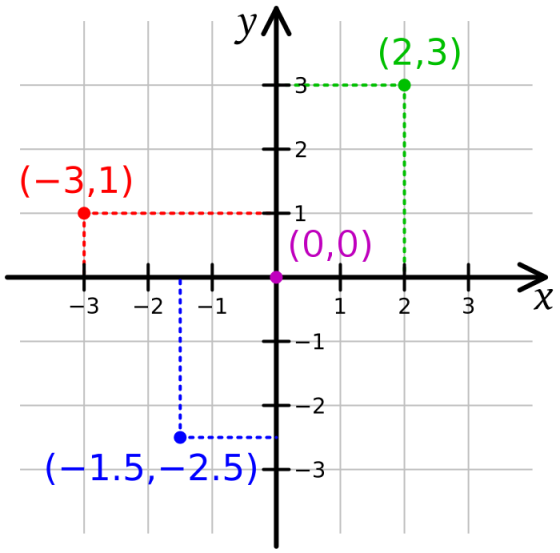
The window origin affects the entire window, while the viewport origin affects a specific rectangular region within the window.



Window Extents and Viewport Extents

Window extents and viewport extents define the size of the logical coordinate space and the visible portion of the window, respectively.

Window extents specify the width and height of the logical coordinate space, while viewport extents specify the width and height of the visible area within the window.



Pen and Brush

A pen is an object that defines the style and attributes of lines drawn using GDI functions. It specifies properties like line width, color, and pattern.

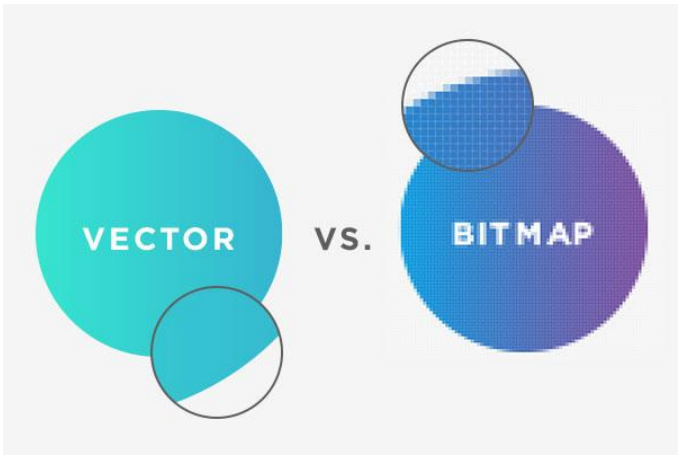
A brush is an object that defines the style and attributes of filled areas using GDI functions. It specifies properties like fill color, pattern, and transparency.

Font

A font defines the appearance of text characters. It specifies properties like typeface, size, weight, and style.

Bitmap

A bitmap is a rectangular image represented as a grid of pixels. It can be displayed, copied, or manipulated using GDI functions.



Current Position

The current position is the point where the next GDI drawing operation will begin. It is typically set using functions like MoveToEx and updated as drawing operations are performed.

Background Mode

Background mode determines how the background color is handled when drawing shapes and text. OPAQUE mode fills the background with the specified color, while TRANSPARENT mode allows the underlying background to show through.

Background Color and Text Color

Background color specifies the color used to fill the background of the drawing area, while text color specifies the color of text drawn using GDI functions.

Drawing Mode

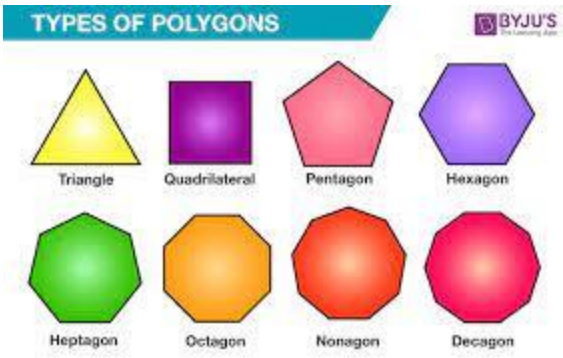
Drawing mode defines how source pixels are combined with destination pixels when drawing shapes and text. It determines how colors are composited and blended.

Stretching Mode

Stretching mode determines how an image is scaled when displayed using the StretchBlt function. It specifies how to handle color interpolation and maintain aspect ratio.

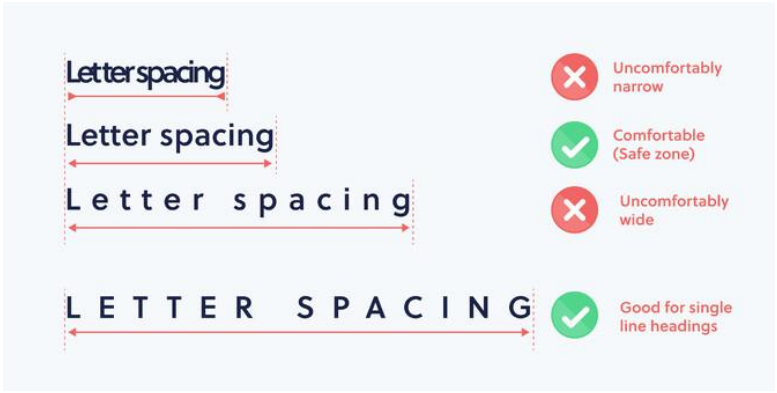
Polygon Fill Mode

Polygon fill mode specifies how the interior of a polygon is filled when drawn using the PolyFill function. It determines how the filling algorithm handles edges and intersections.



Intercharacter Spacing

Intercharacter spacing adjusts the horizontal spacing between characters when drawing text using GDI functions. It can be used to control text density and readability.

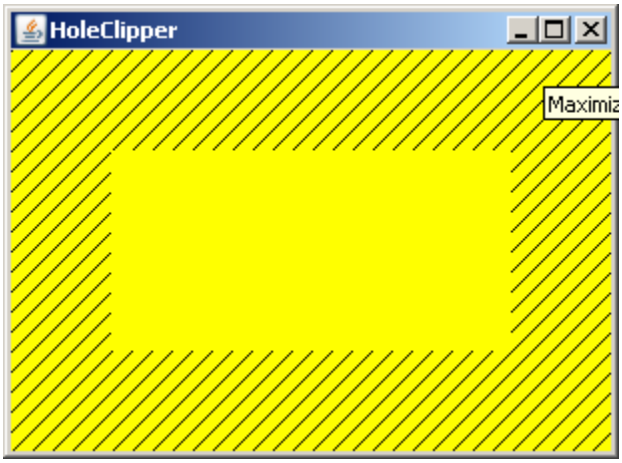


Brush Origin

Brush origin specifies the offset applied to the pattern of a brush when drawing filled shapes. It allows for tiling or repositioning the brush pattern.

Clipping Region

Clipping region defines a rectangular area that restricts the drawing area. GDI functions only draw within the specified clipping region, preventing drawing outside the defined boundaries.



These device context attributes are crucial for controlling the appearance and behavior of graphics operations in Windows GDI.

Understanding their purpose and usage is essential for developing graphics-intensive applications using Windows GDI.

SAVING DEVICE CONTEXT IN WINGDI

Default Device Context Behavior

By default, when you obtain a device context (DC) using GetDC or BeginPaint, Windows provides a temporary DC with default values for all its attributes.

Any modifications you make to these attributes are lost when you release the DC with ReleaseDC or EndPaint.

Saving Device Context Attributes

If your application requires non-default DC attributes, you'll need to initialize the DC's attributes every time you obtain a new DC handle.

This can be cumbersome and inefficient, especially if you frequently need to use specific attribute settings.

CS_OWNDC Window Class Style

To overcome this limitation, you can utilize the CS_OWNDC flag when registering your window class.

This flag instructs Windows to create a private DC for each window instance, ensuring that the DC persists even after the window is destroyed.

Benefits of CS_OWNDC

Using CS_OWNDC offers several advantages:

Attribute Persistence: Changes made to the DC's attributes remain in effect until explicitly modified, eliminating the need to reinitialize the DC's attributes with each WM_PAINT message.

Performance Optimization: By avoiding repeated DC initialization, you reduce the overhead associated with creating and destroying temporary DCs, improving application performance.

Initializing the Device Context

When using CS_OWNDC, you only need to initialize the DC's attributes once, typically in response to the WM_CREATE message:

```
case WM_CREATE:
  hdc = GetDC(hwnd);
  // Initialize device context attributes
  ReleaseDC(hwnd, hdc);
```

Once initialized, the DC's attributes remain valid until you explicitly change them. This approach is more efficient and convenient for applications that require consistent DC configurations.

Device Context Persistence with CS_OWNDC

The CS_OWNDC window class style instructs Windows to create a private device context (DC) for each window instance.

This private DC persists even after the window is destroyed, unlike the temporary DCs obtained with GetDC or BeginPaint.

Using CS_OWNDC eliminates the need to reinitialize DC attributes with every WM_PAINT message, improving performance for graphics-intensive applications.

Saving and Restoring Device Context State

The SaveDC and RestoreDC functions allow you to save and restore the state of a device context.

This is useful when you need to temporarily modify the DC's attributes, perform some drawing operations, and then restore the DC to its original state.

Saving Multiple States

You can call SaveDC multiple times before calling RestoreDC.

Each SaveDC call creates a new state on the DC stack.

When you call RestoreDC, it restores the DC to the state at the top of the stack.

Common Usage Pattern

The most common usage pattern for SaveDC and RestoreDC involves saving the DC state before making changes and then restoring the state after performing the desired operations.

This ensures that the DC is restored to its original state regardless of how many times SaveDC is called.

Restoring to the Most Recent State

Calling RestoreDC with -1 restores the DC to the state saved by the most recent SaveDC call. This is equivalent to popping the top state from the stack.

CS_OWNDC and Device Context Handles

Even if you use CS_OWNDC, you should still release the device context handle before exiting the window procedure.

This ensures that all resources associated with the DC are properly cleaned up.

Conclusion

Saving device contexts using the CS_OWNDC flag can significantly simplify and optimize your GDI programming by eliminating the need for repeated DC initialization and allowing you to maintain persistent DC configurations.

This approach is particularly beneficial for applications that frequently use non-default DC attributes.

Saving and restoring device context state is a valuable technique for manipulating device context attributes and maintaining the desired graphical environment within your Windows GDI applications.

Understanding the usage of SaveDC and RestoreDC allows you to effectively control the appearance and behavior of your graphics operations.

```
HDC hdc = GetDC(hwnd); // Get the device context

// Save the current device context state
int savedDC = SaveDC(hdc);

// Modify some device context attributes
SetTextColor(hdc, RGB(255, 0, 0)); // Set text color to red

// Perform drawing operations using the modified attributes
TextOut(hdc, 10, 10, _T("Hello, World!"), _tcslen(_T("Hello, World!")));

// Restore the saved device context state
RestoreDC(hdc, savedDC);

// Cleanup
ReleaseDC(hwnd, hdc);
```

In this example, the **current device context state** is saved before modifying the text color and drawing a text string.

Then, the saved state is restored before releasing the device context. This ensures that the device context's attributes are reverted to their original values.

Continued in chapter 5 part 2…