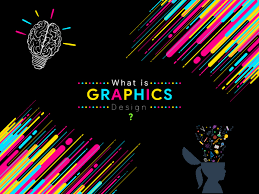
GRAPHICS DEVICE INTERFACE

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.

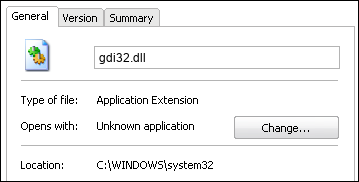
However, in Windows NT, GDI.EXE is only employed for 16-bit programs.

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.

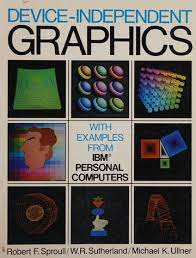
While Windows NT used a separate 16-bit dynamic-link library, GDI.EXE, for the implementation of GDI functions in 16-bit programs, this functionality has been integrated into GDI32.DLL in Windows 10 and 11.



Device-Independent Graphics

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

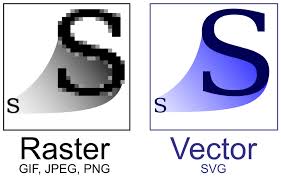
This goal is achieved by providing mechanisms that isolate programs from the unique characteristics of different output devices.



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.

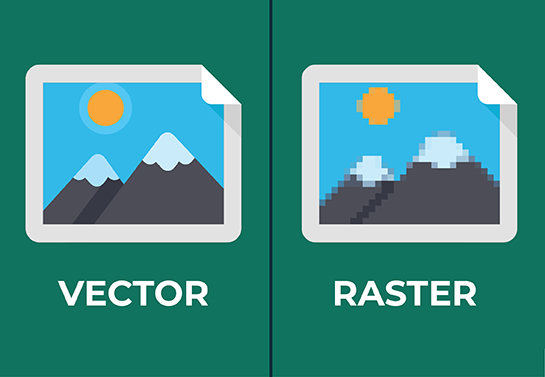


Vector devices, primarily limited to plotters these days, generate images using lines.



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.

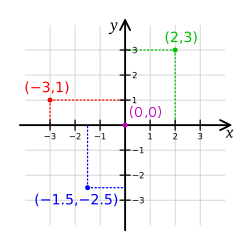
Just as C is sometimes considered a "high-level assembly language," GDI can be viewed as a high-level interface to the graphics device hardware.

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.

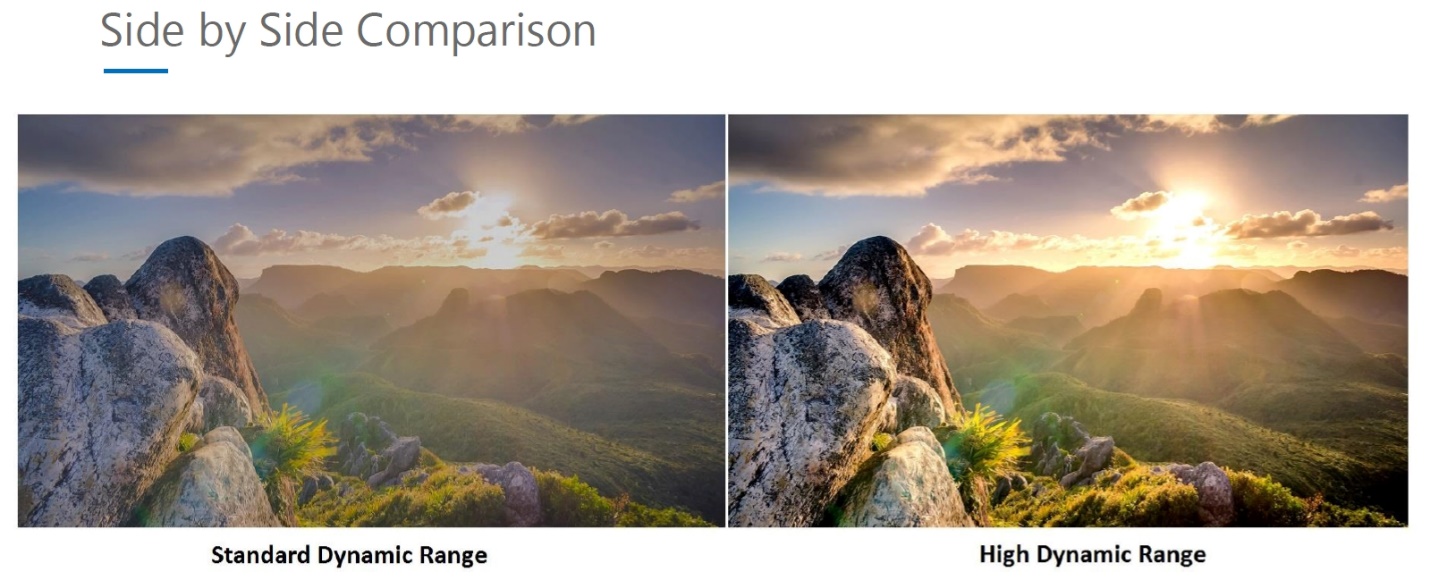


While true color is now the norm, it's worth noting that not all displays are created equal.

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

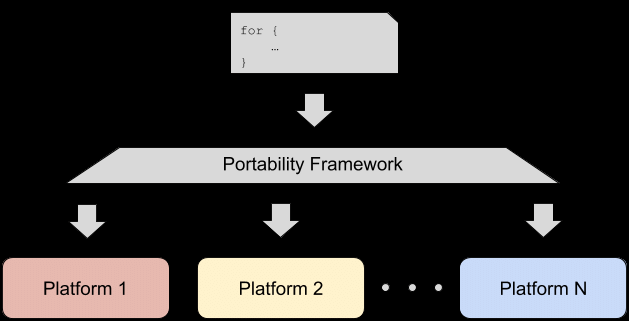
Just as you can write C programs that have subtle portability problems when they run on other computers, you can also inadvertently let device dependencies creep into your Windows programs.

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.



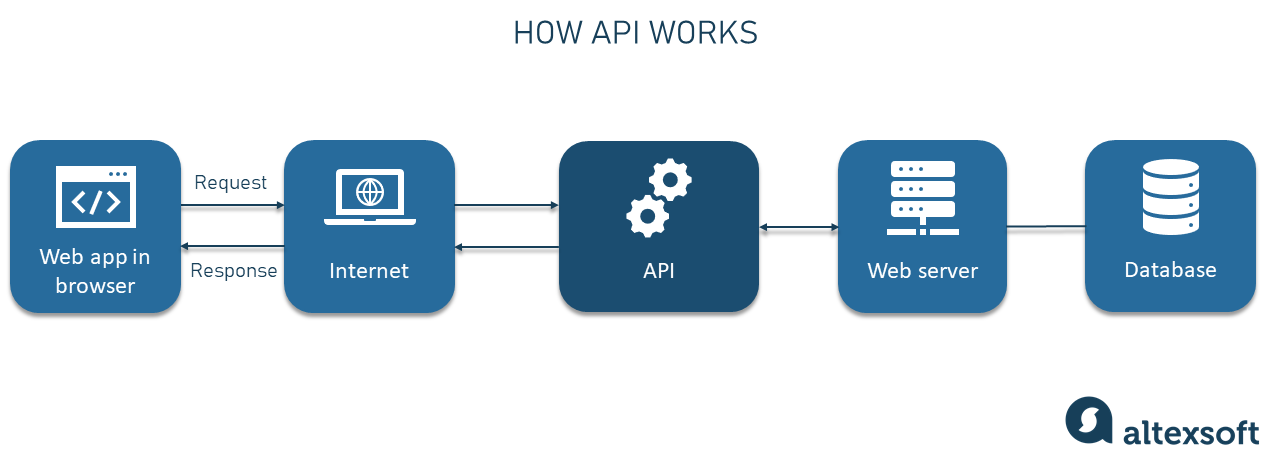
Security vulnerabilities: Device dependencies can also introduce security vulnerabilities into programs. For example, a program that relies on a specific software library may be vulnerable to attacks if that library has a known security flaw.



***How to avoid device dependencies?***

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

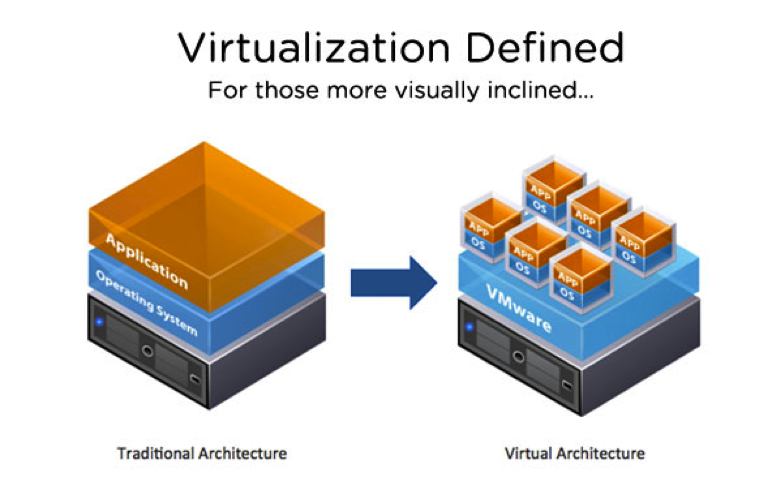
Use standard APIs: When developing Windows programs, it is important to use standard APIs (Application Programming Interfaces) that are provided by the Windows operating system. These APIs are designed to work with a wide variety of hardware and software, so they are less likely to cause portability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

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.

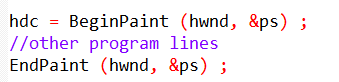
Brush: Determines the appearance of filled areas, such as patterns or images.

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.



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.

In summary, 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.