Standardized graphics on the IBM Personal Computer

by T. B. Clarkson III

Although acknowledged to be an effective means of communicating information, graphics has not progressed more rapidly in the burgeoning use of personal computers due to the lack of standards for both writing and running graphics applications. A graphics standard—the Virtual Device Interface (VDI)—has been proposed for national use and is in the process of being adopted. An implementation of the VDI is currently available for the IBM Personal Computer. This paper briefly traces the history of graphics as used with personal computers, explores the difficulties that standardization efforts have met, explains the VDI model, and shows how this model operates in the IBM Personal Computer environment to make graphics a natural extension of the operating system.

Graphics is well recognized as an effective means of communication. Some studies[1] indicate that a graphic message is consciously recognized more than twice as fast as a text message. However, for years the use of graphics was limited to sophisticated mainframe systems because of the expense involved; complex graphic images require both extensive processing power and vast amounts of memory.

Graphics quickly moved into the microcomputer environment when the size of processors and the price of memory decreased significantly;[2] however, widespread implementation has been slowed by the plethora of incompatible devices and noncommunicating software. Currently, graphics implementations are either written as a program for a specific device or custom-installed for each graphics device.

Despite the difficulties, graphics for use with personal computers seems to be here to stay. Certainly there is user demand for graphics, from simple menu pictorial symbols, or icons, to elaborate window systems for concurrent display of data from different processes. But before personal computer graphics can really become widespread, a standard architecture is needed for writing graphics applications and having them communicate with the huge diversity of available hardware. IBM has chosen to offer the Virtual Device Interface (VDI) implemented by Graphic Software Systems, Inc., for use on its line of personal computers.

The Graphics Development Toolkit, or VDI, and device driver technology provide the needed framework to permit graphics to easily penetrate all facets of the IBM Personal Computer (IBM PC) user interaction. The VDI forms an extension to the operating system to provide communications with graphic devices on a logical basis, much as FORTRAN programs read from and write to logical units. In this case, however, the user is no longer restricted to traditional alphanumeric text. Graphics can become as easy and natural a way of communicating with the user as alphanumeric messages and menus have been in the past.

Graphics enters the personal computer world

Mainframe graphics software has been in existence almost as long as electronic computers.[3] However, because of the high system cost in terms of memory requirements and computing power, the use of graphics was limited to critical tasks (air traffic con-

trol) or tasks in which graphics significantly increased the performance-to-cost ratio (computer-aided design).

The technological developments that allowed the development of the personal computer—compact packaging and low memory costs—have also allowed graphics to become an integral part of the capabilities of the personal computer. Converting numerical data into charts or graphs, combining text and drawings, changing type styles, and even playing games are all examples of graphics applications in a personal computer environment.

The key use for graphics in a PC environment is data representation.

With the advent of a variety of graphic pointing devices, such as the joystick, mouse, digitizing tablet, touch screen, touch tablet, and track ball, the selection of operations became easier for the novice user. With such a device, the user need only move the cross hairs to the picture of the desired operation and select it. Graphics provides an immediately accessible front end to applications when integrated into the program in the original design.[4]

Video games have done a great deal to influence the public's high expectations of graphics on personal computers. Many electronic games evidence high-quality, ingenious graphics and animation. Most people expect their business programs to have graphics of a quality equal to or higher than that of a Centipede™ or Donkey Kong™ game.[5]

However, the requirements of most games that call for video animation are often met by special-purpose processors with graphics images recorded as raster screen representations. Further, simultaneous production of game images on different types of devices is not necessary. There is clearly more to microcomputer graphics than games, and to try to play a game where the graphic images are produced on a plotter is irrelevant.

Historically, the key use for graphics in a personal computer environment has been data representation. As programs such as electronic spreadsheets and forecast models proliferate, personal computer users want the same kind of easily grasped graphic output that is available on mainframe computers. The big problem, of course, is that there is no professional data processing staff to help the user couple his application with graphics. In a personal computer environment, graphics programs must be much simpler to use, or as a better alternative, integrated into the application design.

Barriers to personal computer graphics

With the user demand for increased "friendliness" came a demand for better, higher-resolution color graphics. IBM has taken great strides in meeting this demand with the introduction of the Enhanced Graphics Adapter and Display and the Professional Graphics Display and Controller for the IBM PC. In the past, however, several problems arose in implementing graphics on personal computers. The first was the limit on memory. Despite dramatic improvements in price/performance, memory can still account for a considerable portion of the cost of a personal computer. For every addressable point, or pixel, on the screen, at least one bit of memory is required. When one considers that the Enhanced Graphics Adapter with a monochrome display has a resolution of 640 × 200 pixels, it can be seen that 16000 bytes of memory are required for a single image. The Professional Graphics Controller, when fully configured, uses 307200 bytes of memory for display storage.

There is a second problem: It is difficult to translate from the picture elements of a display screen to those of a printer when the two devices have different resolutions and aspect ratios, as they almost always do. For instance, a screen might have a resolution of 640 × 200 with an aspect ratio of 4 × 3, but a printer might have a resolution of 480 × 751 with an aspect ratio of 3 × 4.

The conversion results in a very poor match between the display screen and the hard copy device. This is because such "screen dumps" merely duplicate the screen resolution, which is usually far inferior to that of the output device. Whereas a screen may be able to display a horizontal resolution of 640 points across a width of, say, ten inches, many dot matrix printers are capable of imaging 200 dots per inch or 1600 dots per line—more than two and one-half times better resolution in the horizontal direction alone.

As the use of personal computers for data representation grows in the business environment, high-quality hard copy in the form of 35-mm slide output will become increasingly important, not only for the resolution it affords but for the impact of color it adds. Achieving presentation quality requires the use of either high-resolution or anti-aliasing techniques, both of which use memory intensively. There needs to be a fast, economical way to create both bit-mapped screen images and high-resolution output.

Yet a third problem crops up: the nontrivial matter of software responsible for controlling the input and output devices. For mainframe computers, each independent software developer wrote a library of input and output device-controlling software, called device drivers, for this purpose, closely matching the application to the supported devices. The result was a specialized, limited-use system.

For personal computers, it is frequently the end user who must coordinate the screen, the application software, input/output peripherals, and device drivers. When this task becomes too difficult, the user simply does not bother, instead buying what is easiest to configure.

Even if an applications programmer does spend time writing tailored controlling software, new developments or improvements in existing devices are difficult to address. Most applications restrict device-dependent programming to a limited number of modules. But these low-level routines must be rewritten for each application and for each supported device. Having to write custom software for each device not only limits the number of devices supported by an applications package, but it forces a certain downward compatibility on manufacturers. Better hardware will not be built if no software is available to run on it. The result is apt to be little incentive for innovation.

The Virtual Device Interface

From this background, it can be seen (1) why better graphics has not been available for personal computers and (2) why stronger efforts have not been made to standardize the production of graphics programs.

To assist in solving these problems, the American National Standards Institute (ANSI) has formed a technical committee (X3H3) to develop computer graphics standards. The first standard to be accepted was the Graphical Kernel System (GKS), adopted in October 1984 and previously adopted by the International Standards Organization. GKS defines graphics functions at the programmer level, with specification of how those functions are assessed through high-level programming languages.

Three other proposals are being considered for standardization by ANSI: Programmer's Hierarchical Interactive Graphical Standard (PHIGS), Virtual Device Metafile (VDM), and Virtual Device Interface (VDI).[6] This paper addresses the VDI produced by Graphic Software Systems for the IBM Personal Computer.

The role of the VDI is analogous to that of the BIOS.

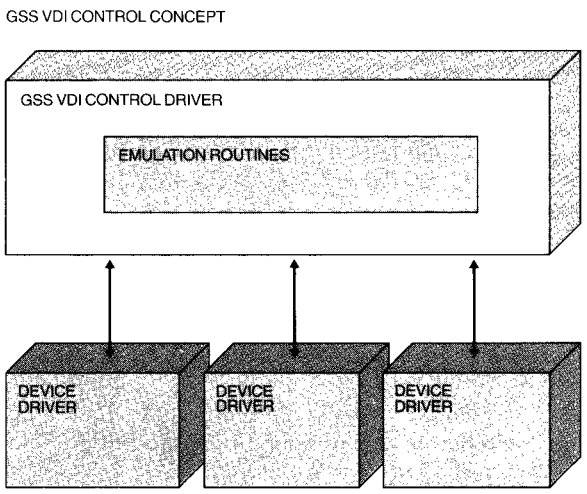
The VDI constitutes a single standard for relaying input to a program and graphics information to output devices. The VDI conceptual model specifies the logical capabilities of both input devices (keyboards, mice, joysticks, etc.) and output devices (screens, printers, plotters, cameras, etc.).

The role of the VDI is analogous to that of the BIOS (Basic Input/Output System) in a portable operating system. The BIOS is the specialized, hardware-specific layer that performs all logical system functions such as displaying data on the screen, reading from and writing to disk, and accepting keyboard input.

The role of the VDI is to provide device independence by creating a logical graphics device interface. Such an interface allows an application to control any graphics peripheral, regardless of individual peculiarities. For input devices, the VDI specifies the kinds of actions, such as pointing or string input, which an input device should be capable of performing. Similarly for output devices, the VDI specifies conceptual capabilities, such as the drawing of lines and polygons.

ANSI has received proposals for a VDI from a number of manufacturers and, as noted, is in the process of

Figure 1 VDI structural dichotomy



formulating a draft standard. Because industry needs are so immediate and because final adoption of the ANSI VDI standard may be more than a year away, the VDI was developed to satisfy the current needs of industry.

The Graphics Development Toolkit (the VDI) developed by Graphic Software Systems provides for device independence through the following capabilities:

- Device driver management
- Coordinate transformation
- Text models
- Character I/O
- Emulation of certain graphics primitives
- Device inquiry
- Error reporting

The implementation of the VDI is functionally divided into two parts (see Figure 1): a VDI controller and the various device drivers. The VDI controller is responsible for device driver management, coordinate transformations, and emulation; the drivers are responsible for all graphics tasks. The user can think of the VDI as a black box, without knowing about this separation, since divisions of labor within the VDI are transparent. What is important is to understand that the VDI serves as a standardized interface between drivers, application program, and operating system.

Unless otherwise specified, VDI in this paper will always refer to the VDI controller, as distinguished from the drivers.

Device driver management. A key point of the VDI is the fact that the application talks only to the VDI and never to the actual devices. The insulation of the application from all hardware is a central advantage of the VDI device driver management. This makes applications device-independent; applications can run on any device for which there is a driver resident in the IBM Personal Computer Disk Operating System (PC DOS), the operating system of the Personal Computer. When a new device is developed, its driver can easily be added. This means that VDI-based applications can run on devices that will be developed in the future, capitalizing on increased resolution, color, and performance.

Another key feature of the VDI is that it loads the required drivers dynamically (as needed), minimizing the memory requirements for systems having many peripheral devices. In addition, the drivers look like standard DOS drivers (in the IBM PC environment) and are loaded as if they were part of the operating system. The user is never aware of device driver changes; all graphics invocation is completely transparent. The VDI can dynamically configure the collection of devices that constitute a workstation, and have multiple drivers resident and active simultaneously (for simultaneous use of, say, a joystick, a keyboard, and a printer). In order to load a device driver, the VDI receives requests from the application program through a language-specific VDI binding (Figure 2).

To facilitate use of the Graphics Development Toolkit, it is accessible from programs written in FORTRAN, Pascal, BASIC, C, and Macro Assembler. Since the VDI has not been officially adopted, the language bindings to the VDI are unique to the GSS implementation.

The language binding issues system calls to the VDI. VDI calls stay the same from system to system. Above the VDI call level there can be a tremendous amount of variation from application to application—different high-level languages, different program sizes, etc. Below the VDI calls, at the hardware and device driver level, there is also tremendous variation. However, the VDI interface always remains the same.

Coordinate transformation. The VDI deals strictly in what is called normalized device coordinate (NDC) space, which the VDI defines as a Cartesian coordinate space bounded by 0 and 32767 (see Figure 3). This universal NDC space allows graphics information to be developed for all devices in an identical way, regardless of the device used.

Input and output devices, however, form images in coordinate spaces appropriate to the particular device. It is part of the job of the VDI to translate, or transform, normalized device coordinates to the actual device coordinates used by the peripheral device. This built-in transformation capability of the VDI, from NDC space to device-specific space, frees the application for application-related processing.

Transformation and emulation of graphics primitives (discussed below) are the primary responsibilities of the VDI. Often other VDI implementations place additional responsibilities with the VDI controller, greatly retarding throughput. As a result of the design for the VDI, the graphics can run at very high speeds.

To maintain this speed and to isolate the VDI from application-specific tasks, special graphics manipulations such as scaling, rotation, segment picking, etc. are handled at a higher level. Programs that perform these tasks include (1) a level "mb" implementation of the ANSI Graphical Kernel System; (2) Plotting System, a library of generalized plotting and charting routines; and (3) Graphical File System, a utility to read and interpret graphics and alphanumeric data stored in the proposed ANSI Virtual Device Metafile standard. These programs, also developed by Graphic Software Systems, are being distributed by IBM under the IBM logo.

Emulation. In all current cases, the capabilities offered by a graphics device are only a subset of the capabilities offered by VDI. In order to ensure total portability, i.e., the ability to have different devices operate with a system despite each having a different command format, the VDI offers guaranteed emulation of many functions.

For instance, if the VDI issues the command "Draw a polygon," and the device does not include the primitive, "Draw polygon," the VDI will see that the polygon gets drawn using primitives the device does have. It might accomplish "Draw a polygon" by piecing together a series of "Draw line" commands. Emulation is totally transparent to the user, who only knows that his command has been carried out. Guaranteed emulation is a key requirement of any VDI implementation, since it ensures true portability across all devices.

Device inquiry. While the VDI provides device independence, it also enables application software to query device capabilities for tailoring program execution. For example, a charting program can inquire whether a color display is available. If it is, color can be used to differentiate data; if it is not, shade patterns or line styles can be substituted.

Figure 2 Schematic of FORTRAN application talking to VDI

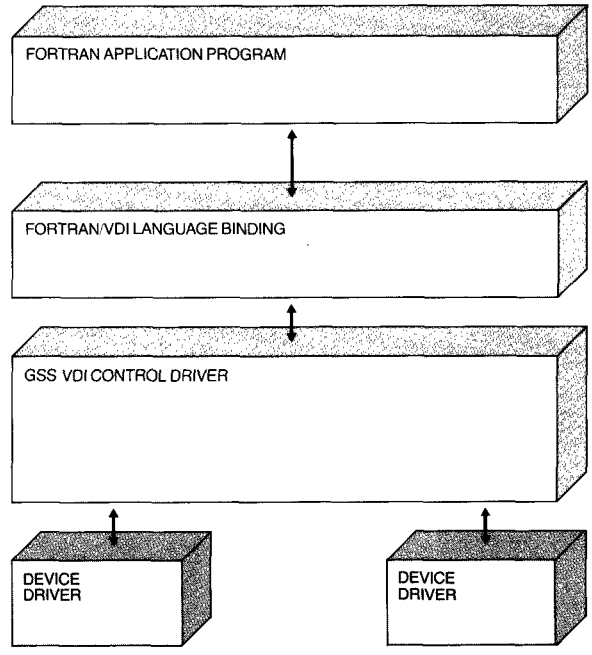
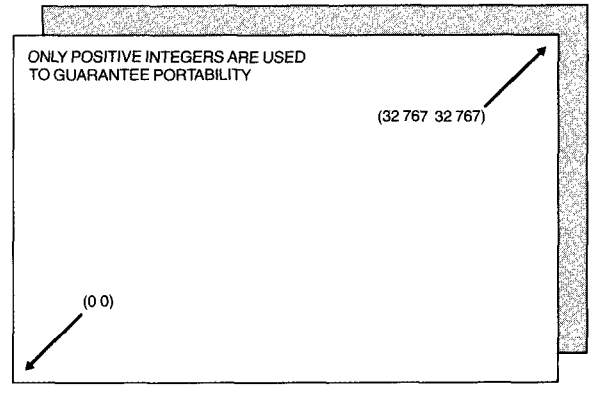


Figure 3 Illustration of NDC space



Error reporting. The VDI has a comprehensive set of error codes that inform the application program of any unusual condition. The application program has the responsibility for checking the error status and attempting error recovery, or at least informing the user.

VDI functions

The VDI graphics capabilities can be divided into six functional areas:

1. Graphics
2. Alpha text
3. Cursor text
4. Inquiry
5. Input
6. Control

Graphics functions. The VDI achieves device-independent graphic standardization by supporting input and output graphics primitives, as well as individual attribute control. A graphics primitive is defined as a function that generates graphic objects on the display surface. Lines, markers, circles, and arcs are examples. An attribute is a characteristic of that primitive (color, style, line style, or line width).

The primitives of the VDI are polyline, polymarker, filled areas, cell arrays, arcs, circles, bars, and pie slices. *A polyline* is a line connecting a specified series of points. Symbols used to highlight the points are called *polymarkers*. When a polyline encloses an area, the area can be filled. A *cell array* is a collection of points used to form a pattern that can be used to fill an area. An *arc* is a segment of a circle; some devices are capable of producing arcs but not circles. A *circle* is a collection of points equidistant from a central point; circles can be emulated by successive arcs. *Bars* are rectilinear areas, frequently filled. *Pie slices* are sectors of a circle, sometimes formed from arcs and line segments to the focus of the arc, frequently used in the pie charts. Pie slices are treated as filled areas. All graphics primitives are specified by a set of coordinates in the NDC space.

The VDI attributes are divided into character attributes (for text), polyline attributes, polymarker attributes, and fill attributes. For example, graphics text attributes control height, baseline rotation, color, font, and alignment. Polyline attributes control type, width, and color. Fill attributes control type, style, and color.

The text model is chosen by the application writer and offers an extra element of flexibility in fitting the application to particular requirements.

Graphics text can be rotated, scaled, aligned, positioned, and colored like any other graphics primitive.

Alpha text functions. Alpha text is not rotatable or scalable like graphics text, but can be precisely positioned on device-unit-addressable pixel boundaries. It provides a high-speed path to hardware text in printers for mixing word processing with graphics.

Alpha text attributes include variable line spacing, subscripting, superscripting, underlining, overstriking, text quality, and color. Alpha text enables the application writer to tailor the application to particular application requirements.

It is the VDI inquiry function that ensures intelligent device independence.

Cursor text functions. Used only with cathode ray tubes (CRTs) in the generation of screen menus and forms, cursor text is available in only one size and one font, and is character-cell-positionable. Cursor text displays a text string on a fixed rectangular grid and is used commonly for the filling out of forms. It is not rotatable and cannot be combined with graphics text, alpha text, or graphics primitives. Its attributes are reverse video, underline, bold, blink, and color.

Inquiry. For the graphics, alpha text, and cursor text functions, the VDI provides an extensive set of inquiry facilities to provide the user with information on current system status, current attribute settings, active devices, and device/system capabilities. Inquiry functions are provided as listed in Table 1.

It is the VDI inquiry function that ensures intelligent device independence. The application can adapt itself to the peculiarities of the device and make adjustments to overcome device limitations.

Input functions. Input functions relay information from the operator to the application via the device. There are four types of graphics input functions:

Table 1 Characteristics determined by inquiry functions

For graphics primitives
- current polyline attributes
- current polymarker attributes
- current fill-area attributes
- current graphics text attributes

For alpha text
- capabilities
- position
- font availability
- string length
- attributes

For cursor text
- number of addressable character cells
- current cursor address
- video capabilities

Workstation capabilities
- maximum addressable width
- maximum addressable height
- scaling capabilities
- width of one pel
- height of one pel
- number of character heights
- number of line types
- number of line widths
- number of marker types
- number of marker sizes
- number of graphic text fonts
- number of patterns
- number of hatch styles
- number of predefined colors
- number of generalized drawing primitives
- list of generalized drawing primitives
- attributes of each primitive
- color capability
- text rotation capability
- fill area capability
- pel operation capability
- total number of colors available
- locator capability
- valuator capability
- number of choices available
- string input capability
- workstation type
- device type
- number of writing modes available
- highest level of input mode available
- text alignment capability
- inking capability as output echo
- rubberbanding capability
- maximum addressable NDC units on x axis
- maximum addressable NDC units on y axis
- version of the driver
- minimum graphic character height in NDC units
- maximum graphic character width in NDC units
- minimum line width in NDC units
- maximum line width in NDC units
- minimum marker height in NDC units
- maximum marker height in NDC units

1. *Locator.* The locator input reports the point location of the graphics input device (such as a tablet, mouse, or joystick) in NDC units.
2. *Valuator.* The valuator input function returns a scalar value corresponding to the status of a valuator device (potentiometer, slide control, etc.).
3. *Choice.* The input choice function returns the status of a choice device (such as a switch or function key).
4. *String.* String input allows text input from a keyboard or other text-character device.

Input functions operate in two modes: request and sample. In the sample mode, the input is returned immediately. In the request mode, the operator must complete the input function by indicating that the request is complete.

Control functions. The control functions include initialization of a graphics device, termination of graphics operations to a device, clearing the display surface of a workstation, and displaying pending graphics. These functions are invoked by the commands that are now described.

Open workstation. This command links an application program to the actual physical device that will be used. As written in a FORTRAN program the call is VOPNWK (workin, devhdl, workot), where workin is an array containing environmental information for the device being opened; devhdl is the device reference returned to VDI.SYS, the control driver; and workot is an array containing information about the capabilities of the device being opened. Note that redirection at the system level through the DOS SET command is possible, so although the output of an application program may have been designed for a display, it is easily and correctly directed to another device, such as a pen plotter.

These device capabilities returned by the opening of a workstation are used by VDI emulation routines to

determine the most proficient way of producing graphics output.

The OPEN WORKSTATION command dynamically loads a graphics device driver, if it is not already present in memory, initializes the graphics device, and sets attribute defaults. This command is always the first graphics operation performed to be sent to a device.

Close workstation. This command terminates graphics operations to a device. It is always the last graphics operation performed to be sent to a device. The FORTRAN call is VCLSWK (devhdl), where devhdl is the device identifier returned from OPEN WORKSTATION.

Any application can use any device as long as an appropriate device driver exists.

Clear workstation. This command clears the surface of the workstation: clears a CRT screen, prompts for new paper on a plotter, or sends all pending graphics to a printer and advances to top-of-form. The FORTRAN call is VCLRWK (devhdl), where devhdl refers to the device identifier returned from OPEN WORKSTATION.

Update workstation. This command displays all pending graphics. The FORTRAN call to VUPDWK (devhdl) displays all pending graphics on the workstation specified by devhdl. For printers, this causes the current picture to be printed and the printer to advance to top-of-form.

The notion of a "workstation" is fundamental to the VDI. A workstation is any logical entity from which the application receives input or to which output is directed. A workstation can be a display screen and keyboard, a mouse, or a plotter. At the programmer level, a workstation can refer to a complete work site, eliminating the need to reference each device explicitly. At the VDI level, however, each device is always designated explicitly, although the designation may be transparent to the user.

Techniques of invocation under the operating system of the IBM Personal Computer

When it is remembered that all VDI-level operations discussed thus far are happening at the operating system level, one sees that a tremendous amount of graphics power and flexibility is effectively built in as far as the user is concerned. There is no need to worry about specific hardware which may or may not be available. *With the VDI, applications deal strictly with the VDI binding and not at all with devices.* Any application can use any device as long as an appropriate device driver exists.[7]

We will now see exactly how this is accomplished at the operating system level. Figure 4 shows how VDI is related to the operating system, the application, and the device drivers.

The application program with high-level language calls to VDI functions is linked with the appropriate VDI language-binding library and is a link-time, application-developer-controlled activity. Everything below the language binding is a run-time, application-transparent activity. The language binding issues VDI calls to the VDI controller, which then performs driver selection, transformations, invocation of the driver, etc. The VDI calls never change to the application; they look like calls to the operating system.

When the user starts DOS (boots it), a file called CONFIG.SYS is loaded. The CONFIG.SYS file specifies boot-time device drivers for DOS, among which can be VDI device drivers. The user can edit the CONFIG.SYS file to include them. The last VDI driver specified must be the VDI controller.

VDI device drivers may be "grouped." Grouped drivers share memory space and thus are not resident at the same time; however, drivers from different groups may be resident (and open) at the same time. There is no limit (other than memory space) to the number of groups that may be specified.

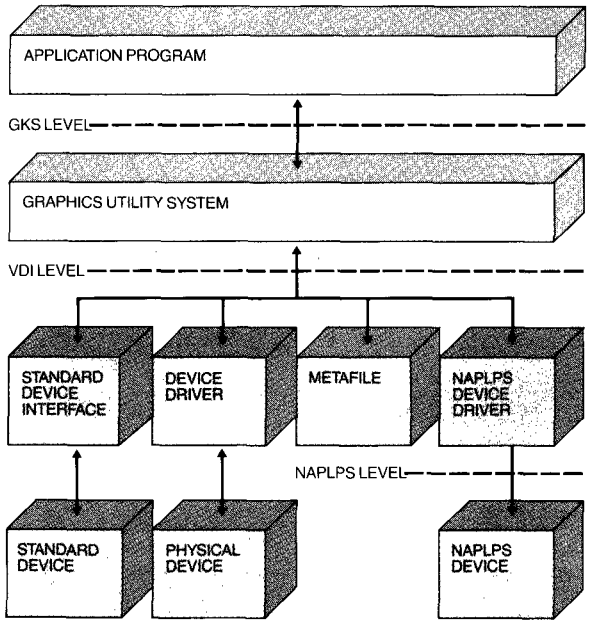
When booting, DOS sequentially loads each of the drivers specified in the CONFIG.SYS file. After it loads a driver, it invokes that driver with a special one-time INIT call to determine, among other things, the amount of memory the driver requires.

VDI device drivers, when invoked with the INIT call, tell DOS that they only need a small amount of memory, since all they leave resident is a 100-byte-long header. When the VDI controller is finally loaded (towards the end of the processing by DOS of the CONFIG.SYS file at boot time) and hit with the INIT call, it computes the amount of memory required by the largest VDI driver in each group. When the VDI controller returns to DOS from the INIT call, it reserves memory for itself and that total.

Thus, after DOS is fully booted, enough space has automatically been allocated in the area of the VDI controller for a full complement of VDI device drivers, with no subsequent user concern for memory allocation or management. When an OPEN WORKSTATION command is given by an application, the VDI controller brings in the appropriate device driver (if it is not already loaded and if no other driver is already loaded in its group area) and starts it.

The key point to remember is that VDI and DOS are one and the same as far as the application is concerned. Device selection is fully automatic and transparent to the user. The application never deals with devices—only with VDI.

Figure 4 Schematic showing software layers at OS level: application program on top, VDI bindings, VDI calls, etc.



Emulation example

To better understand what goes on at the operating system level when the VDI is active, let's look at an example of a typical VDI function—emulation. Suppose that a user writing in FORTRAN wants to create a filled circle, with center coordinates of $X = 30\,000$ NDC units; $Y = 2000$ NDC units; R (radius) = 4000 NDC units. Because the circle is located near the end of the x axis, when the circle is drawn it will be clipped to the edge of NDC space. The figure will be drawn on a pen plotter that cannot draw filled circles on its own.

Appearance attributes of a circle include interior style, fill style, and color. Interior style is set through a call to VSFINT (devhdl, stylin), where devhdl is the device identifier returned by OPEN WORKSTATION and stylin is an integer specifying the interior fill style. Acceptable values are 0 for hollow fill (outline only), 1 for solid fill, 2 for pattern fill, and 3 for hatch fill. Pattern and hatch fill patterns are specified by another call to SET FILL STYLE INDEX. Color is selected by a call to SET FILL COLOR INDEX.

The user first issues a command to the VDI routine that draws a circle: VCIRCL (devhdl, X, Y, radius), where devhdl is the device identifier returned when the device was opened, X and Y are the center coordinates of the circle, and radius is the radius of the circle as measured along the x axis.

The FORTRAN language binding used by the application would format this information into five arrays of the appropriate form for the VDI:

CONTROL[] INTIN[] PTSIN[] INTOUT[] PTSOUT[]

The CONTROL array is used to specify the desired VDI function opcode and the length counts of the other passed arrays. INTIN is an array of integer input parameters, while INTOUT will contain output parameters. PTSIN is an array of X, Y pair coordinate data, with PTSOUT reserved for output coordinates in a similar fashion. For a circle, the CONTROL and PTSIN arrays are meaningful and contain the following:

CONTROL [0]—opcode = 11
CONTROL [1]—number of input vertices = 3
CONTROL [2]—length of INTIN array = 0
CONTROL [5]—device identifier = devhdl
CONTROL [6]—subopcode for circle = 4

Figure 5 Program to draw a filled circle

```
c *** program to open a vdi device, draw a circle and close
c *** the vdi device
      implicit integer * 2 (a-z)
      dimension workin(19), workout(66), echoxy(2)
      character * 10 dummy
      data workin /0,1,1,1,1,1,1,1,1,1,
     +      68,73,83,80,76,65,89,32/
      data echoxy /0,0/
c *** open the device
      status = vopnwk(workin,devhandle,workout)
c *** set the fill color to blue
      status = vsfcol(devhandle,4)
c *** set the interior style to solid
      status = vsfint(devhandle,1)
c *** draw a filled circle
      status = vcircl(devhandle,3000,2000,4000)
c *** do a read to wait for viewing
      status = vrqstr(devhandle,2,0,echoxy,dummy)
c *** close the device
      status = vclswk(devhandle)
      stop
      end
```

PTSIN [0]—x coordinate of center
PTSIN [1]—y coordinate of center
PTSIN [2]—x coordinate of point on circumference
PTSIN [3]—y coordinate of point on circumference
PTSIN [4]—radius
PTSIN [5]—0

The VDI would then direct the device driver to draw the circle. The device, not having filled-circle capabilities, would essentially respond: "I don't know how." The VDI circle emulation routine, em_circle (em), would then be summoned. This routine uses the parameters specified in the DRAW CIRCLE function, as well as parameters returned when the device was opened, to construct a polyline representing the edge of the circle, clipped where appropriate. Since values in the VDI must be in the range 0–32 767, coordinate values cannot be less than 0 in the Y direction, nor greater than 32 767 in the X direction, so a polyline is constructed along the two edges of device area and an arc is constructed between them.

Finally, the VDI would direct the driver to fill the circle and would again call the em_circle routine with a solid fill style. The plotter pen width would dictate the spacing necessary to produce a solid, uniform filled area.

After these emulation routines, all transparent to the user, have been summoned, the figure is complete. The small FORTRAN program in Figure 5 illustrates the above example.

Concluding remarks

The Graphics Development Toolkit (the VDI) is apt to have the same kind of impact on the personal computer environment for graphics that the processor-independent (BIOS-based) operating system had on general applications, enabling graphics to be integrated with almost any generic application function.

Applications written to a VDI standard are portable from one machine to another—as well as to devices offered in the future. If the machines have the same operating system, the object code is portable; if they have different operating systems, the high-level source code is portable.

Concentrating graphics functionality at the operating system level allows applications to be upgraded easily and to take advantage of new or improved devices. Additionally, a low-capability device can transparently support high-level VDI graphics functions through emulation.

If the "open system" concept is furthered, more application programs will be developed with graphics as a natural and logical extension rather than as an afterthought.

The VDI promotes development of a wider range of input and output devices, all of which can be handled with the same conceptual model without regard for device-specific constraints.

The VDI frees computer development, since graphic application software developed for one graphics computer will work with others.

The VDI frees IBM PC graphics from the encumbrance of device dependence, allowing the field to expand and blossom as a dynamic industry.

One may expect that such low-level routines as those contained in the VDI will soon migrate into device

controller chips, with the consequent development of devices which can accept direct VDI input.

As for the VDI itself, after it is standardized the VDI may be extended to include enhanced font technology, advanced raster techniques, and, as the power of personal computers improves, graphic segments.

In summation, the IBM Personal Computer Graphics Development Toolkit (VDI) permits an application program to be isolated from the operating environment in which it is run.

Cited references and notes

1. Studies done by Wharton Applied Research Center, Wharton School, University of Pennsylvania, Philadelphia, under grant from the Audio Visual Division of 3M Corporation. Reviewed in *Computer Graphics News* (March/April 1982).
2. A. Bechtolsheim and F. Baskett, "High-performance raster graphics for microcomputer systems," *Computer Graphics* **14,** No. 3, 43–47 (July 1980).
3. Whirlwind example from J. D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Co., Reading, MA (1982), p. 18.
4. W. K. English, D. C. Englebart, and M. L. Berman, "Display selection techniques for text manipulation," *IEEE Transactions on Human Factors in Electronics* **HFE-8,** No. 1, 21–31 (1967).
5. Centipede™ and Donkey Kong™ are software games available from Atari, Inc., P.O. Box 2943, South San Francisco, CA 94080.
6. GKS is known as ANS X3.124. PHIGS will be available soon in working draft form for public review in 1985. VDM is known as dpANS X3.122, with final approval expected in early 1985. VDI is available in draft form for public review in 1985. At its June 1984 meeting in Genodet, France, the International Organization for Standardization (ISO) WGZ (Graphics) voted the following formal titles for the projects formerly known as VDM and VDI:

Long title:	Information Processing—Computer Graphics—Metafile for Transfer and Storage of Picture Description Information
Short title:	Computer Graphics Metafile
Abbreviation:	CGM
Long title:	Information Processing—Computer Graphics—Interface Techniques for Dialogues with Graphical Devices
Short title:	Computer Graphics Interface
Abbreviation:	CGI

   It is expected that ANSI will adopt ISO nomenclature when these standards are adopted.
7. The following devices are supported by the IBM Personal Computer Graphics Development Toolkit (VDI): IBM Game Adapter and IBM PCjr and PC Joystick, IBM Color/Graphics Monitor Adapter—High Resolution 2 Color, IBM Color/Graphics Monitor Adapter—Medium Resolution 4 Color, IBM Enhanced Graphics Adapter with Monochrome Monitor, IBM Enhanced Graphics Adapter with Enhanced Color Display, IBM Enhanced Graphics Adapter—Medium Resolution 16 Color, IBM Enhanced Graphics Adapter—High Resolution 16 Color, IBM Professional Graphics Display and Controller—256 Color, IBM Compact Printer, IBM Color Printer, IBM Graphics Printer, IBM PCjr—High Resolution 4 Color, IBM PCjr—Medium Resolution 16 Color, IBM PCjr—Low Resolution 16 Color, IBM 7372 Color Plotter, IBM 7371 Color Plotter, and IBM Virtual Device Metafile (VDM).

Reprint Order No. G321-5233.

Thomas B. Clarkson III *Graphic Software Systems, Inc., 25117 SW Parkway, Wilsonville, Oregon 97070.* Mr. Clarkson is President of GSS and co-founder of the corporation. Before becoming involved with the enterprise, he played a key role in enhancing the Plot-10 interactive graphics software library of Tektronix. Prior to that, he was a software project leader and computer analyst at Jet Propulsion Laboratory, developing programs for the Galileo project to Jupiter. He had earlier been a research assistant at the Georgia Institute of Technology and a programmer at International Energy Conservation Systems, Inc., in Atlanta. Mr. Clarkson earned a Bachelor's degree in physics from Wake Forest University and a Master of Science degree in information and computer science from the Georgia Institute of Technology. He is a member of the ANSI X3H3 Graphics Standards Committee, ACM SIGGRAPH, and the National Computer Graphics Association.