The PC graphics handbook

The PC graphics handbook

Julio Sanchez & Maria P. Canton
Minnesota State University

CRC PRESS
Boca Raton London New York Washington, D.C.

Library of Congress Cataloging-in-Publication Data

Catalog record is available from the Library of Congress

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

Direct all inquiries to CRC Press LLC, 2000 N.W. Corporate Blvd., Boca Raton, Florida 33431.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation, without intent to infringe.

Visit the CRC Press Web site at www.crcpress.com

No claim to original U.S. Government works
International Standard Book Number 0-8493-1678-2
Printed in the United States of America 1 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

Preface

This book is about graphics programming on the Personal Computer. As the title indicates, the book's emphasis is on programming, rather than on theory. The main purpose is to assist program designers, systems and applications programmers, and implementation specialists in the development of graphics software for the PC.

PC graphics began in 1982 with the introduction of the IBM Color Graphics Adapter. In 20 or so years the technology has gone from simple, rather primitive devices, sometimes plagued with interference problems and visually disturbing defects, to sophisticated multi-processor environments capable of realistic 3D rendering and life-like animation. A machine that 20 years ago was hardly capable of displaying bar charts and simple business graphics now competes with the most powerful and sophisticated graphics workstations. During this evolution many technologies that originally showed great promise have disappeared, while a few others seem to hang on past the most optimistic expectations. Programming has gone from rather crude and primitive routines for rendering simple geometrical objects to modeling, rendering, and animating solid objects at runtime in a realistic manner.

What Is in the Book

In the complex graphics environment of the PC, covering the fundamentals of some technologies requires one or more full-sized volumes. This is the case with systems such as VGA, SuperVGA, XGA, DirectX, Direct 3D, and OpenGL. Thus, in defining the contents of this book our first task was to identify which systems and platforms are still relevant to the programmer. Our second task was to compress the coverage of the selected systems so that the entire PC graphics context would fit in a single volume.

The topic selection process entailed many difficult decisions: how much graphics theory should be in the book? Is DOS graphics still a viable field? Which portions of Direct3D are most important to the "average" programmer? Should OpenGL be included? In some cases the complexity and specialized application of a graphics technology determined our decision. For example, Direct3D immediate mode programming was excluded because of its difficulty and specialized application. Other platforms such as OpenGL are technologically evolved but not part of the PC mainstream. Our decisions led to the following structure:

- Part I of the book is an overview of PC graphics and a description of the theories that support the material covered.

- Part II is devoted to DOS graphics. In this part we discuss the VGA, XGA, and SuperVGA systems. DOS bitmapped graphics are also in this part.
- Part III is about Windows API graphics. Since Windows is a graphics environment the topics covered include an overview of general Windows programming. In Part III bitmapped graphics is revisited in the Windows platform.
- Part IV covers some portions of DirectX. In addition to a description of the DirectX platform itself and its relation to the COM, we cover DirectX and Direct3D retained mode. Direct3D immediate mode is excluded for the reasons previously mentioned.

Programming Environment

The principal programming environment is C++, but some primitive functions are coded in 80x86 Assembly Language. The high performance requirements of graphics applications sometimes mandate Assembly Language. Microsoft's Visual C++ Version 6.0 and MASM version 6.14 were used in developing the book.

We approach Windows programming at its most basic level, that is, using the Windows Application Programming Interface (API). We do not use the Microsoft Foundation Class Library (MFC) or other wrapper functions. It is our opinion that graphics programs cannot afford the overhead associated with higher-level development tools. Furthermore, DirectX provides no special support for MFC.

Although we do not use the wrapper functions we do use tools that are part of the Visual C++ development package. These include resource editors for creating menus, dialog boxes, icons, bitmaps, and other standard program components.

The Book's Software

The software for the book is furnished on-line at www.crcpress.com. The software package includes all the sample programs and projects developed in the text as well as several graphics libraries.

Table of Contents

Chapter 10 - VGA Geometrical Primitives

Chapter 13 - SuperVGA Programming

Part III - Windows API Graphics

Chapter 16 - Graphics Programming in Windows

Part IV - DirectX Graphics

Chapter 29 - Blitting

List of Tables

List of Illustrations

Part I

Graphics Fundamentals

Chapter 1

PC Graphics Overview

Topics:

- History and evolution of PC graphics
- Technologies
- Applications
- Development platforms
- The state-of-the-art

This first chapter is a brief historical summary of the evolution of PC graphics, a short list of graphics-related technologies and fields of application, and an overview of the state-of-the-art. A historical review is necessary in order to understand current PC graphics technologies. What the PC is today as a graphical machine is the result of a complex series of changes, often influenced by concerns of backward compatibility and by commercial issues. A review of graphics hardware technologies is also necessary because the graphics programmer usually works close to the metal. The hardware intimacy requires a clear understanding the how a binary value, stored in a memory cell, is converted into a screen pixel. The chapter also includes a description of some of the most important applications of computer graphics and concludes with a presentation of the graphics technologies and development platforms for the PC.

1.1 History and Evolution

The state-of-the-art computer is a graphics machine. It is typically equipped with a high-resolution display, a graphics card or integral video system with 3D capabilities, and a processor and operating system that support a sophisticated graphical user interface. This has not always been the case. In the beginning computers were text-based. Their principal application was processing text data. The typical source of input was a typewriter-like machine called a *teletype terminal* or TTY. Output was provided by a line printer that operated by means of a mechanical arrangement of small pins that noisily produced an approximate rendering of the alphabetic characters. It was not until the 1960s that *cathode-ray tube* technology (CRT) found its way from television into computers. We start at this technological point.

1.1.1 The Cathode-Ray Tube

The CRT display consists of a glass tube whose interior is coated with a specially formulated phosphor. When the phosphor-coated surface is struck by an electron beam it becomes fluorescent.. In computer applications CRT displays are classified into three groups: *storage tube*, *vector refresh*, and *raster-scan*.

The *storage tube CRT* can be used both as a display and as a storage device, since the phosphor image remains visible for up to 1 hour. To erase the image the tube is flooded with a voltage that turns the phosphor back to its dark state. One limitation is that specific screen areas cannot be individually erased. This determines that in order to make a small change in a displayed image, the entire CRT surface must be redrawn. Furthermore, the storage tube technology display has no color capabilities and contrast is low. This explains why storage tube displays have seldom been used in computers, and never in microcomputers.

Computers were not the first machines to use the cathode-ray tubes for graphic display. The oscilloscope, a common laboratory apparatus, performs operations on an input signal in order to display the graph of the electric or electronic wave on a fluorescent screen

The *vector-refresh display*, on the other hand, uses a short-persistence phosphor whose coating must be reactivated by the electron beam. This reactivation, called the *refresh*, takes place at a rate of 30 to 50 times per second. The vector-refresh system also requires a display file and a display controller. The display file is a memory area that holds the data and instructions for drawing the objects to be displayed. The display controller reads this information from the display file and transforms it into digital commands and data which are sent to the CRT. Figure 1-1 shows the fundamental elements of a vector refresh display system.

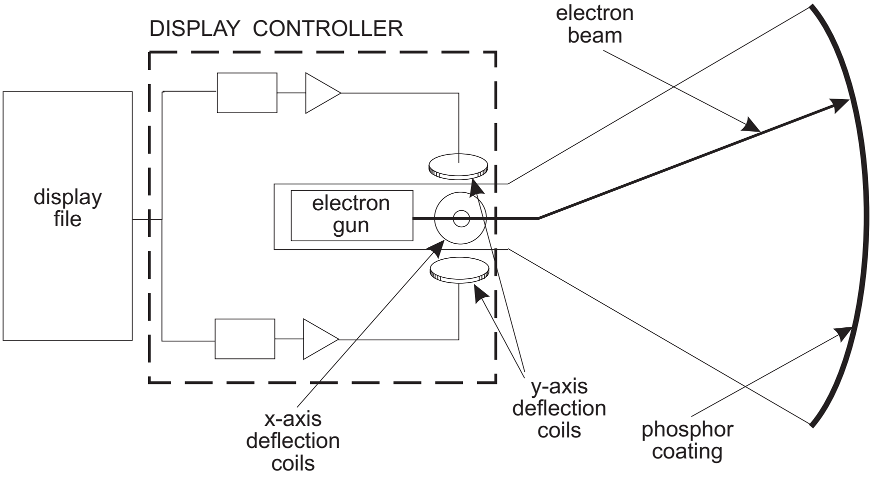


Figure 1-1 *Vector-Refresh Display*

The disadvantages of the vector-refresh CRT are its high cost and limited color capabilities. Vector refresh display technology has not been used in the PC.

During the 1960s Conrac Corporation developed a computer image processing technology, known as *raster-scan graphics*. Their approach took advantage of the methods of image rendering and refreshing used in television receivers. In a raster-scan display the electron beam follows a horizontal line-by-line path, starting at the top-left corner of the CRT surface. The scanning cycle takes place 50 to 70 times per second. At the start of each horizontal line the controller turns on the electron beam. The beam is turned off during the horizontal and vertical retrace cycles. The scanning path is shown in Figure 1-2.

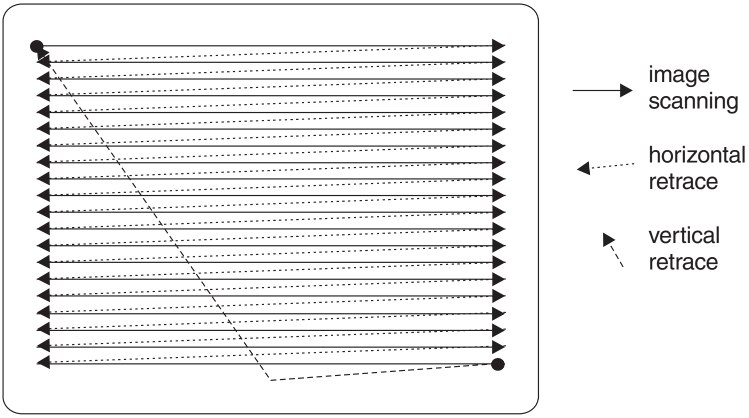


Figure 1-2 *A Raster-Scan System*

The raster-scan display surface is divided into a grid of individual dots, called *pixels*. The term pixel was derived from the words **pic**ture and **el**ements. In the memory-mapped implementation of raster-scan technology, an area of RAM is devoted to recording the state of each individual screen pixel. The simplest color-coding scheme consists of using a single bit to represent either a white or a black pixel. Conventionally, if the memory bit is set, the display scanner renders the corresponding pixel as white. If the memory bit is cleared, the pixel is left dark. The area of memory reserved for the screen display is usually called the *frame buffer* or the *video buffer*. Figure 1-3, on the following page, shows the elements of a memory-mapped video system.

Implementing color pixels requires a more elaborate scheme. In color systems the CRT is equipped with one electron gun for each color that is used to activate the pixels. Usually there are three color-sensitive electron guns: one for red, one for green, and one for blue. Data for each of the three colors must be stored separately. One approach is to have a separate memory map for each color. A more common solution is to devote bit fields or storage units to each color. For example, if one memory byte is used to encode the pixel's color attributes, three bits can be assigned to encode the red color, two bits to encode the green color, and three bits for the blue color. One possible mapping of colors to pixels is shown in Color Figure 1.

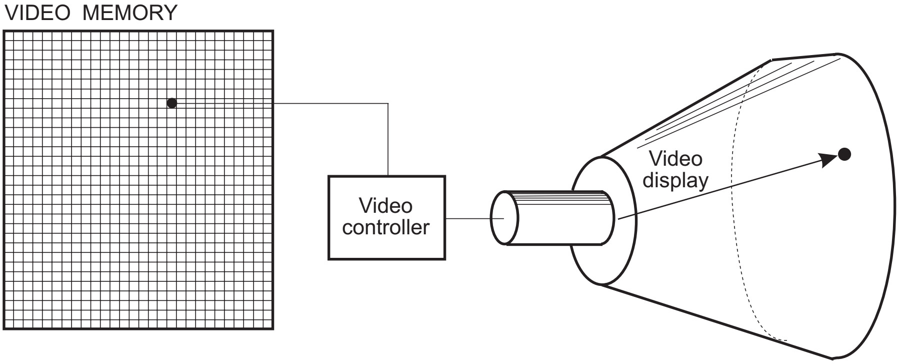


Figure 1-3 *A Memory-Mapped System*

In Color Figure 1 one memory byte has been divided into three separate bit fields. Each bit field encodes the color values that are used to render a single screen pixel. The individual bits are conventionally labeled with the letters R, G, and B, according to the color they represent. Since eight combinations can be encoded in a three-bit field, the blue and red color components can each have eight levels of intensity. In this example we have used a two-bit field to encode the green color; therefore it can only be rendered in four levels of intensity. The total number of combinations that can be encoded in 8 bits is 256, which is also the number of different color values that can be represented in one memory byte. The color code is transmitted by the display controller hardware to a *Digital-to-Analog converter* (DAC), which, in turn, transmits the color video signals to the CRT.

In the PC all video systems are raster-scan and memory mapped. The advantages of a raster-scan display are low cost, color capability, and easy programmability. One major disadvantage is the grainy physical structure of the display surface that results from the individual screen dots. Among other aberrations, the dot pattern causes lines that are not vertical, horizontal, or at exactly 45 degrees to exhibit a staircase effect. Raster-scan systems also have limitations in rendering animation. Two factors contribute to this problem: first, all the screen pixels within a rectangular area must be updated with each image change. Second, in order to ensure smoothness, the successive images that create the illusion of motion must be flashed on the screen at a fast rate. These constraints place a large processing load on the microprocessor and the display system hardware.

1.2 Short History of PC Video

The original IBM Personal Computer was offered in 1981 equipped with either a *Monochrome Display Adapter* (MDA), or a graphics system named the *Color/Graphics Monitor Adapter* (CGA). The rationale for having two different display systems was that users who intended to use the PC for text operations would prefer a machine equipped with the MDA video system, while those requiring graphics would like one equipped with the CGA card. But, in reality, the CGA graphics system provided only the most simple and unsophisticated graphics. The card was also plagued with interference problems which created a screen disturbance called "snow." However, the fact that the original IBM Personal Computer was furnished with an optional graphics system signaled that the industry considered video graphics as an essential part of microcomputing.

During the past 20 years PC video hardware has been furnished in an assortment of on-board systems, plug-in cards, monitors, and options manufactured and marketed by many companies. In the following sections we briefly discuss better known PC video systems. Systems that were short lived or that gained little popularity, such as the *PCJr*, the *IBM Professional Graphics Controller*, the *Multicolor Graphics Array*, and the *IBM Image Adapter A*, are not mentioned.

1.2.1 Monochrome Display Adapter

The original alphanumeric display card designed and distributed by IBM for the Personal Computer was sold as the *Monochrome Display and Printer Adapter* since it included a parallel printer port. The MDA could display the entire range of alphanumeric and graphic characters in the IBM character set, but did not provide pixel-level graphics functions. The MDA was compatible with the IBM PC, PC XT, and PC AT, and some of the earlier models of the PS/2 line. It could not be used in the PCjr, the PC Convertible, or in the microchannel PS/2 machines. The card required a special monochrome monitor of long-persistence (P39) phosphor. These monitors, which produced very pleasant text, were available with either green or amber screens. The video hardware was based on the Motorola 6845 CRT controller. The system contained 4K of on-board video memory, mapped to physical address B0000H.

The MDA was designed as a pure alphanumeric display: the programmer could not access the individual screen pixels. Video memory is mapped as a grid of character and attribute bytes. The character codes occupy the even-numbered bytes in adapter memory, and the display attributes the odd-numbered bytes. This special storage and display scheme was conceived to save memory space and to simplify programming. Figure 1-4 shows the cell structure of the MDA video memory space and the bitmap for the attribute cells.

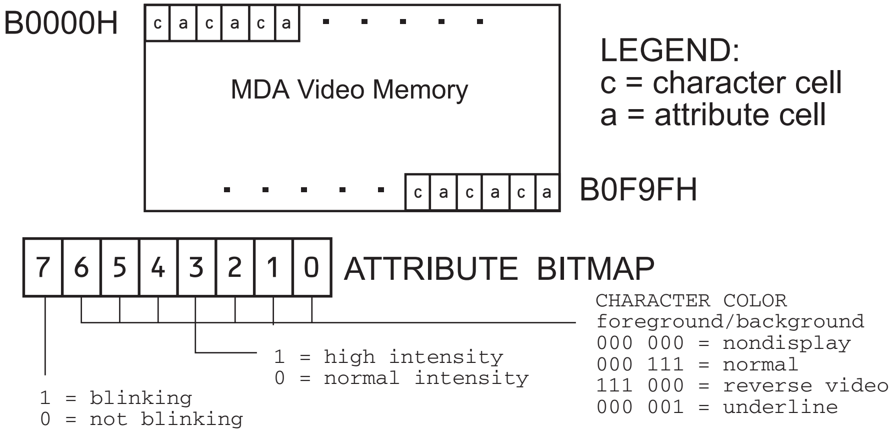


Figure 1-4 *Memory Mapping and Attributes in the MDA Adapter*

1.2.2 Hercules Graphics Card

An aftermaket version of the MDA, developed and marketed by Hercules Computer Technologies, was called the *Hercules Graphics Card* (HGC). HGA emulates the monochrome functions of the MDA, but can also operate in a graphics mode. Like the MDA, the HGC includes a parallel printer port. Because of its graphics capabilities, the Hercules card was often preferred over the IBM version. In the HGA the display buffer consists of 64K of video memory. In alphanumeric mode the system sees only the 4K required for text mode number 7. However, when the HGC is in the graphics mode, the 64K are partitioned as two 32K graphics pages located at physical addresses B0000H to B7FFFH and B8000H to BFFFFH. Graphic applications can select which page is displayed.

1.2.3 Color Graphics Adapter

The *Color Graphics Adapter* (CGA), released early in 1982, was the first color and graphics card for the PC. The CGA operates in seven modes which include monochrome and color graphics. Mode number 0 is a 40 columns by 25 rows monochrome alphanumeric mode. In Mode 0 text characters are displayed in 16 shades of grey. Characters are double width and 40 can be fitted on a screen line. Graphics mode number 6 provides the highest resolution, 640 horizontal by 200 vertical pixels.

One notable difference between the CGA and the MDA is the lower quality text characters of the color card. In a raster-scan display the visual quality of the text characters is related to the size of the respective character cells. In the MDA each character is displayed in a box of 9-by-14 screen pixels. In the CGA the character box is of 8-by-8 pixels. The resulting graininess of the CGA text characters was so disturbing that many users considered the card unsuitable for text operations.

The CGA was designed so that it could be used with a standard television set; however, it performed best when connected to an RGB color monitor. Timing and control signals were furnished by a Motorola 6845 CRT controller, identical to the one used in the MDA. The CGA contains 16K of memory, which is four times the memory in the MDA. This makes it possible for the CGA to simultaneously hold data for four full screens of alphanumeric text. The CGA video buffer is located at physical address B8000H. The 16K memory space in the adapter is logically divided into four 1K areas, each of which holds up to 2000 characters with their respective attributes. The memory-to-pixel mapping in the CGA is shown in Figure 1-5.

Video memory in the CGA text modes consists of consecutive character and attribute bytes, as in the MDA. The mapping of the attribute bits in the black and white alphanumeric modes is identical to the one used in the MDA, but in color alphanumeric modes the attribute bits are mapped differently.

The CGA suffers from a form of screen interference, popularly called snow. This irritating effect results from CGA's use of RAM chips (called dynamic RAMs) which are considerably slower than the static RAMs used in the MDA card. In a CGA system, if the CPU reads or writes to the video buffer while it is being refreshed by the CRT Controller, a visible screen disturbance takes place. The solution is to synchronize screen updates with the vertical retrace signal generated by the 6845 controller.

This is possible during a short time interval, called the *vertical retrace cycle*. Since the duration of the vertical retrace is barely sufficient to set a few pixels, rendering is considerably slowed down by this synchronization requirement. Furthermore, during screen scroll operations the display functions must be turned off while the buffer is updated. This causes a disturbing screen flicker.

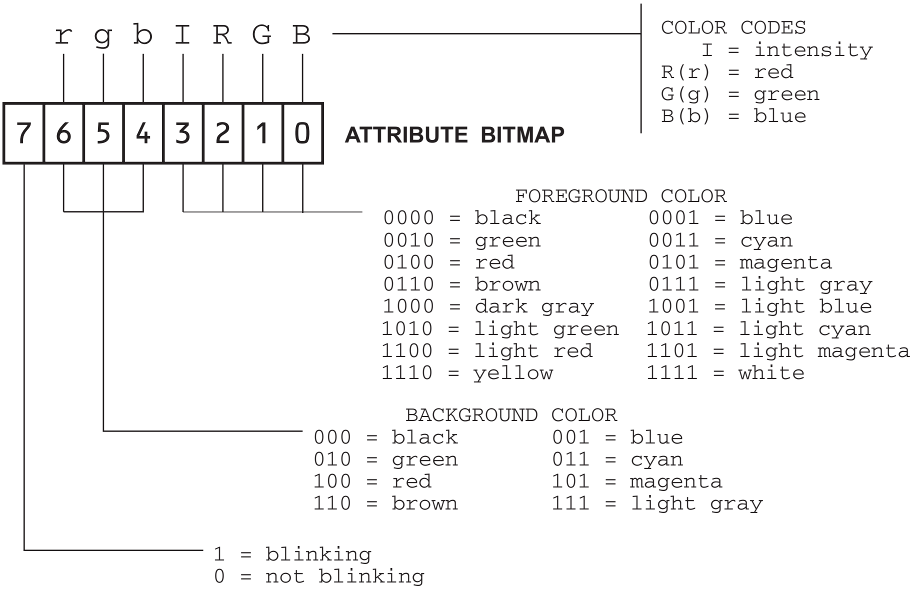


Figure 1-5 *Memory-to-Pixel Mapping in the CGA Color Alpha Modes*

1.2.4 Enhanced Graphics Adapter

The *Enhanced Graphics Adapter* (EGA) was introduced by IBM in 1984 as an alternative to the much maligned CGA card. The EGA could emulate most of the functions and all the display modes of both the CGA and the MDA. At the same time, EGA had a greater character definition in the alphanumeric modes than the CGA, higher resolution in the graphics modes, and was not plagued with the snow and flicker problems. EGA can drive an Enhanced Color Display with a maximum graphics resolution of 640-by-350 pixels.

EGA introduced four new graphics modes, sometimes called the *enhanced graphics modes.* These modes are numbered 13 through 16. The highest graphics resolution is obtained in the modes numbers 15 and 16, which displayed 640-by-350 pixels. The EGA used a custom video controller chip with different port and register assignments than those of the Motorola 6845 controller used in the MDA and CGA cards. The result is that programs that access the MDA and CGA 6845 video controller directly do not work on the EGA. EGA was furnished with optional on-board RAM in blocks of 64K. In the minimum configuration the card had 64K of video memory, and 256K in the maximum one.

EGA systems had several serious limitation. In the first place, EGA supported write operations to most of its internal registers, but not read operations. This made it virtually impossible for software to detect and preserve the state of the adapter, which in turn, made EGA unsuitable for memory resident applications or for multitasking or multiprogramming environments. Another limitation of the EGA is related to its unequal definitions in the vertical and horizontal planes; this problem is also present in the HGC and the CGA cards. In an EGA, equipped with a typical monitor, the vertical resolution in graphic modes 15 and l6 is approximately 54 pixels per inch and the horizontal resolution approximately 75 pixels per inch. This gives a ratio of vertical to horizontal definition of approximately 3:4. Although not as bad as the 2:3 ratio of the HGC, the disproportion still determines that a pixel pattern geometrically representing a square is displayed on the screen as a rectangle and the pattern of a circle is displayed as an ellipse. The geometrical aberration complicates pixel path calculations, which must take this disproportion into account and make the necessary adjustments.

1.3 PS/2 Video Systems

The PS/2 line of microcomputers was released by IBM in 1987. It introduced several new features, including a new system bus and board connectors, named the *microchannel architecture*, a 3.5-inch diskette drive with 1.44 megabytes of storage, and an optional multitasking operating system named OS/2, which is now virtually defunct. Machines of the PS/2 line came equipped with one of two new video graphics systems, while a third one was available as an option.

The new video standards for the PS/2 line were the *Multicolor Graphics Array* (MCGA), the *Video Graphics Array* (VGA), and the *8514/A Display Adapter*. The most notable improvement of the video hardware in the PS/2 systems was that IBM changed the display driver technology from digital to analog. The one drawback was that the monitors of the PC line were incompatible with the PS/2 computers, and vice versa. The main advantage of analog display technology is a much larger color selection. Another important improvement is their symmetrical resolution, that is, the screen resolution is the same in the vertical as in the horizontal planes. Symmetrical resolution simplifies programming by eliminating geometrical aberrations during pixel plotting operations. The aspect ratio of the PS/2 monitors is 4:3, and the best resolution is 640-by-480 pixels.

1.3.1 Video Graphics Array

Video Graphics Array (VGA) is the standard video display system for the IBM Personal System/2 computers models 50, 50z, 60, 70, and 80. IBM first furnished VGA on the system board. VGA comes with 256K of video memory, which can be divided into four 64K areas, called the *video maps* or *bit planes*. The system supports all the display modes of the MDA, CGA, and the EGA cards of the PC family. In addition, VGA introduced graphics mode number 18, with 640-by-480 pixel resolution in 16 colors. The effective resolution of the text modes is 720 by 400. In order to display text in a graphics mode, three text fonts with different box sizes could be loaded from BIOS into the adapter. VGA soon became available as an adapter card for non-IBM machines. The video technology introduced with VGA continues to be the PC video standard to this day.

1.3.2 8514/A Display Adapter

The 8514/A Display Adapter is a high-resolution graphics system designed for the PS/2 line. The tchnology was developed in the United Kingdom, at the IBM Hursley Laboratories. The 8514/A system comprises not only the display adapter, but also the 8514 Color Display and an optional Memory Expansion Kit. The original 8514/A is compatible only with PS/2 computers that use the microchannel bus. It is not compatible with machines of the PC line, with the PS/2 models 25 and 30, or with non-IBM computers that do not use the microchannel architecture. Other companies developed versions of 8514/A which can be used in machines based on the ISA or EISA bus architecture.

The 8514/A Display Adapter consists of two sandwiched boards designed to be inserted into the special microchannel slot that has the auxiliary video extension. The standard version comes with 512K of video memory. The memory space is divided into four maps of 128K each. In the standard configuration 8514/A displays in 16 colors, however, by installing the optional Memory Expansion Kit, video memory is increased to 1 megabyte. The 1 megabyte space is divided into eight maps, extending to 256 the number of available colors. The system is capable of four new graphic modes not available in VGA. IBM named them the *advanced function modes*. One of the new modes has 640-by-480 pixel definition, and the remaining three modes have 1024-by-768 pixels. 8514/A does not directly support the conventional alphanumeric or graphics modes of the other video standards, since it executes only in the advanced function modes. In a typical system VGA automatically takes over when a standard mode is set. The image is routed to the 8514/A monitor when an advanced function mode is enabled. An interesting feature of the 8514/A adapter is that a system containing it can operate with two monitors. In this case the usual setup is to connect the 8514 color display to the 8514/A adapter and a standard monitor to the VGA. Figure 1-6 shows the architecture of a VGA/8514A system.

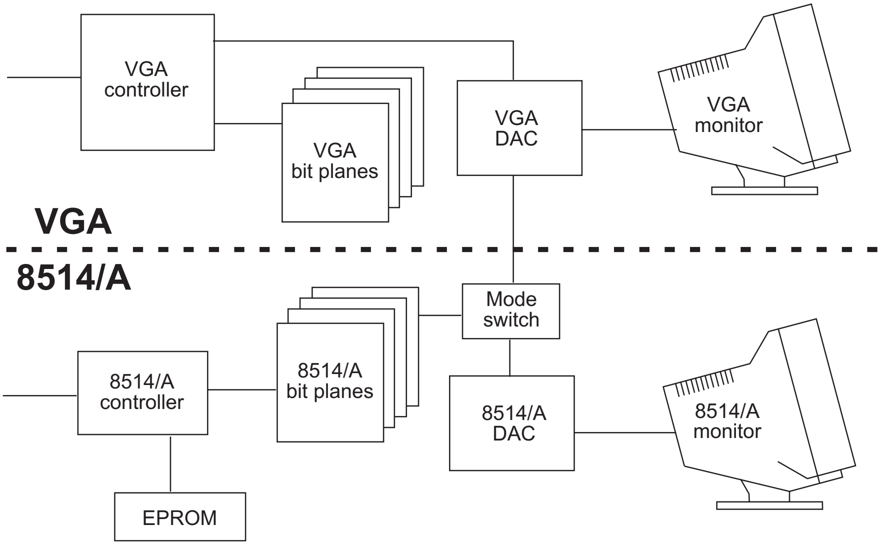


Figure 1-6 *Architecture of a VGA/8514A Video System*

A feature of 8514/A, which presaged things to come, is that it contains a dedicated graphics chip that performs as a graphics coprocessor. Unlike previous systems, in 8514/A the system microprocessor cannot access video memory; instead this function is left to the graphic coprocessor. The greatest advantage of this setup is that it improves performance by offloading the graphics functions from the CPU. The 8514/A can be programmed through a high-level graphics function package called the *Adapter Interface*, or AI. There are a total of 59 drawing primitives in the AI, accessible through a software interrupt.

Approximately 2 years after the introduction of 8514/A, IBM unveiled another high-performance, high-priced graphics board, designated the *Image Adapter/A*. The Image Adapter/A is compatible with the 8514/A at the Adapter Interface level but not at the register level. Image Adapter/A was short-lived due to its high price tag, as well as to the fact that shortly thereafter IBM released its new XGA technology.

1.3.3 Extended Graphics Array

In September 1990, IBM disclosed preliminary information on a new graphics standard designated the *Extended Graphics Array*, or XGA. Like its predecessor the 8514-A, XGA hardware was developed in the UK. Two XGA configurations were implemented: an adapter card and a motherboard version. In 1992, IBM released a non-interlaced version of the XGA designated as XGA-2 or XGA-NI (non-interlaced). The XGA adapter is compatible with PS/2 microchannel machines equipped with the 80386 or 486 CPU. The system is integrated in the motherboard of the IBM Models 90 XP 486, in the Model 57 SLC, and furnished as an adapter board in the Model 95 XP 486. In 1992, *Radius Incorporated* released the Radius XGA-2 Color Graphics Card for computers using the ISA or EISA bus. Other companies developed versions of the XGA system for microchannel and non-microchannel computers. XGA is still found today in some laptop computers. Figure 1-7 is a component diagram of the XGA system.

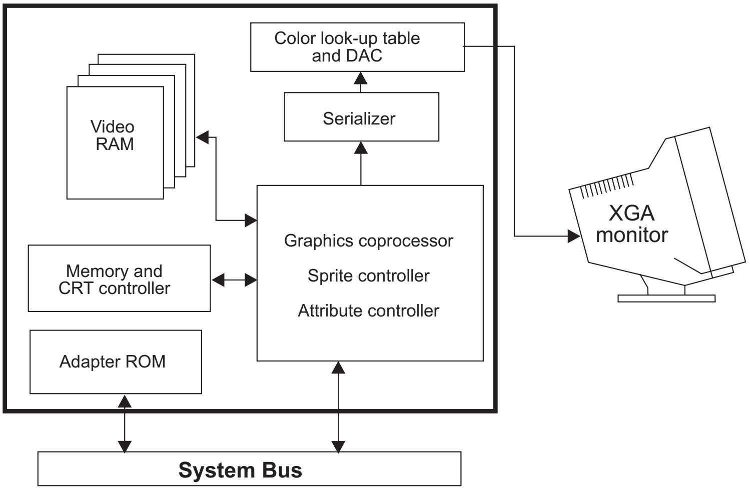


Figure 1-7 *XGA Component Diagram*

1.4 SuperVGA

The general characteristic of SuperVGA boards, as the name implies, is that they exceed the VGA standard in definition, color range, or both. The term SuperVGA is usually applied to enhancements to the VGA standard developed by independent manufacturers and vendors. A typical SuperVGA card is capable of executing, not only the standard VGA modes, but at least one additional mode with higher definition or greater color range than VGA. These modes are usually called the *SuperVGA Enhanced Modes*.

In the beginning, the uncontrolled proliferation of SuperVGA hardware led to compatibility problems. Lack of standardization and production controls led to a situation in which the features of a card by one manufacturer were often incompatible with those of a card produced by another company. This situation often led to the following problem: an application designed to take advantage of the enhancements in a particular SuperVGA system would not execute correctly in another systems. An attempt to solve this lack of standardization resulted in several manufacturers of SuperVGA boards forming the *Video Electronics Standards Association* (VESA). In October 1989, VESA made public its first SuperVGA standard. This standard defined several enhanced video modes and implemented a BIOS extension designed to provide a few fundamental video services in a hardware-compatible fashion.

1.4.1 SuperVGA Architecture

In VGA systems the video memory space extends from A0000H to BFFFFH. The 64K area starting at segment base A000H is devoted to graphics, while the 64K area starting at segment base B000H is devoted to alphanumeric modes. This makes a total of 128K memory space reserved for video operations. But the fact that systems could be set up with two monitors, one in an alphanumeric mode and the other one in a color mode, actually limited the graphics video space to 64K.

Not much video data can be stored in a 64K. For example, if each screen pixel is encoded in one memory byte, then the maximum screen data that can be stored in 65,536 bytes corresponds to a square screen with 256 pixels on each side. Thus, a VGA system in 640-by-480 pixels resolution, using one data byte per pixel, requires 307,200 bytes for storing a single screen. Consider that in the Intel segmented architecture of the original PCs each segment consisted of a 64K space. In this case addressing 307,200 pixels requires making five segment changes.

VGA designers were able to compress video data by implementing a latching scheme that resulted in a semi-planar architecture. For example, in VGA mode number 18, with a resolution of 640-by-480 pixels, each pixel can be displayed in 16 different colors. To encode 16 color combinations requires a 4-bit field, and a total memory space of 153,600 bytes. However, the latching mechanism allows mapping each of the four color attributes to the same base address, all apearing to be located in a common 64K address space.

When the VGA was first released, engineers noticed that some VGA modes contained surplus memory. For example, in modes with 640-by-480 pixels resolution the video data stored in each map takes up 38,400 bytes of the available 64K. This leaves

27,136 unused bytes. The original idea of enhancing the VGA system was based on using this surplus memory to store video data. It is possible to have an 800-by-600 pixel display divided into four maps of 60,000 bytes each, and yet not exceed the 64K space allowed for each color map, nor the total 265K furnished with the VGA system. To graphics systems designers, a resolution of 800 by 600 pixels, in 16 colors, appeared as a natural extension to VGA mode number 18. This new mode, later designated as mode 6AH by the VESA SuperVGA standard, could be programmed in a similar manner as VGA mode number 18. The enhancement, which could be achieved with minor changes in the VGA hardware, provided a 36 percent increase in the display area.

1.4.2 Bank-Switched Memory

The memory structure for VGA 256-color mode number 19 is based, not on a bitmapped multiplane scheme, but in a much simpler format that maps a single memory byte to each screen pixel. This scheme is shown in Figure 1-8.

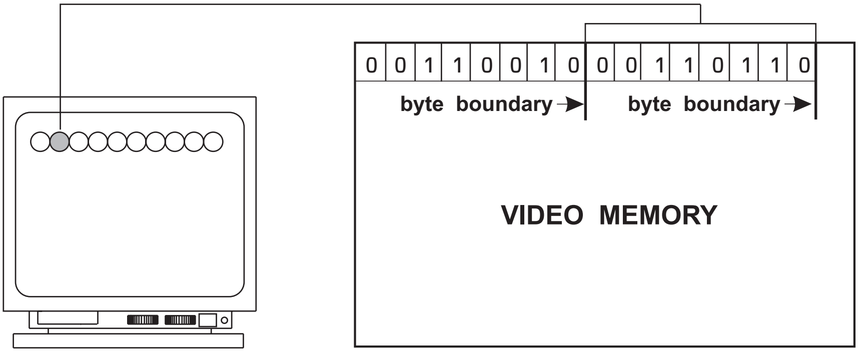


Figure 1-8 *Byte-to-Pixel Video Memory Mapping Scheme*

In byte-to-pixel mapping 256 color combinations can be directly encoded into a data byte, which correspond to the 256 DAC registers of the VGA hardware. The method is straightforward and uncomplicated; however, if the entire video space is to be contained in 64K, the maximum resolution is limited to 65,536 pixels. This means that a rectangular screen of 320-by-200 pixels nearly fills the allotted 64K.

In a segment architecture machine, if the resolution of a 256-color mode is to exceed 65,536 pixels it is necessary to find other ways of mapping video memory into 64K of system RAM. The mechanism adopted by the SuperVGA designers is based on a technique known as *bank switching*. In bank-switched systems the video display hardware maps several 64K-blocks of RAM to different locations in video memory. In the PC addressing of the multi-segment space is by means of a hardware mechanism that selects which video memory area is currently located at the system's aperture. In the SuperVGA implementation the system aperture is usually placed at segment base A000H. The entire process is reminiscent of memory page switching proposed in the LIM (Lotus/Intel/Microsoft) Extended Memory scheme. Figure 1-8 shows mapping of several memory banks to the video space and the map selection mechanism for CPU addressing.

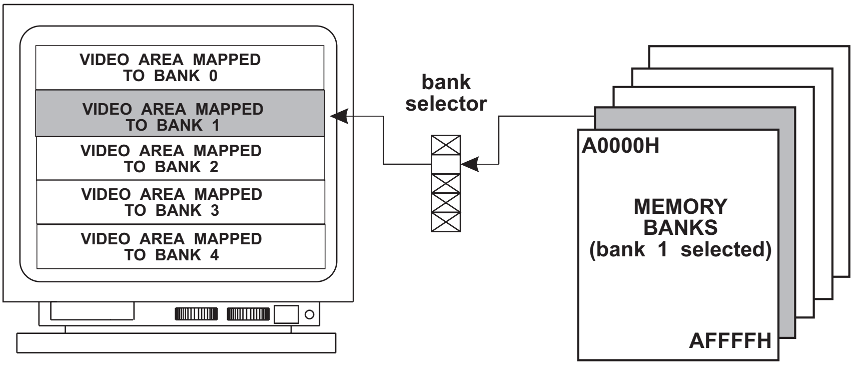

VIDEO AREA MAPPED TO BANK 0
VIDEO AREA MAPPED TO BANK 1
VIDEO AREA MAPPED TO BANK 2
VIDEO AREA MAPPED TO BANK 3
VIDEO AREA MAPPED TO BANK 4
bank selector
A0000H
MEMORY BANKS (bank 1 selected)
AFFFFH

Figure 1-9 *SuperVGA Banked-Switched Memory*

In the context of video system architecture, the term *aperture* is often used to denote the CPU window into the system's memory space. For example, if the addressable area of video memory starts at physical address A0000H and extends to AFFFFH, we say that the CPU has a 64K aperture into video memory (10000H = 64K). In Figure 1-10 we see that the bank selector determines which area of video memory is mapped to the processor's aperture. This determines the video display area that can be updated by the processor. In other words, in the memory banking scheme the processor cannot access the entire video memory at once. In the case of Figure 1-10, the graphics hardware has to perform five bank switches in order to update the entire screen.

1.4.3 256-Color Extensions

The SuperVGA alternative for increasing definition beyond the VGA limit is based on the banking mechanism shown in Figure 1-8. This scheme, in which a memory byte encodes the 256 color combinations for each screen pixel, does away with the programming complications that result from mapping pixel colors to bit fields, as in the high-resolution VGA modes previously mentioned. At the same time, bank switching introduces some new complexities of its own, one of which is the requirement of a bank selection device. In summary, the SuperVGA approach to extending video memory on the PC has no precedent in CGA, EGA, or VGA systems. It is not interleaved nor does it require memory planes or pixel masking. Although it is similar to VGA mode number 19 regarding color encoding, VGA mode number 19 does not use bank switching.

1.5 Graphics Coprocessors and Accelerators

A group of video systems based on dedicated graphics chips is perhaps the one most difficult to characterize and delimit. They can be roughly described as those systems in which graphics performance is enhanced by means of specialized graphics hardware that operates independently from the CPU. The enormous variations in the functionalities and design of graphics accelerators and coprocessors makes it impossible to list the specific features of these systems. Here we mention a few systems of historical interest in the evolution of PC graphics.

1.5.1 The TMS340 Coprocessor

One of the first full-featured dedicated graphics coprocessors used in the PC was the TMS 340 graphics coprocessor developed by Texas Instruments. The chip was introduced in 1986 and an upgrade, labeled TMS 34020, in 1990. The project was not a commercial success and in 1993 Texas Instruments started discouraging the development of new products based on the TMS340 chips. However, from 1988 to 1993 these coprocessors were incorporated into many video products, including several high-end video adapters, some of which were capable of a resolution of 1280-by-1024 pixels in more than 16 million colors. These products, now called *true color* or 24-bit color cards, furnished photographic-quality images. The image quality of coprocessor-based systems was often sufficient for image editing, prepress, desktop publishing, CAD, and other high-end graphics applications.

Not all coprocessor-based graphics systems marketed at the time used the TMS 340. For example, the *Radius Multiview 24* card contained three 8514/A-compatible chips, while the *RasterOps Paintboard PC* card was based on the S3. But it is safe to state that the TMS 340 and its descendants dominated the true-color field at the time; of ten true color cards reviewed in the January 1993 edition of *Windows Magazine*, seven were based on the TMS 340.

The TMS 340 was optimized for graphics processing in a 32-bit environment. The technology had its predecessors in the TI's 320 line of digital signal processing chips. The following are the distinguishing features of the TMS340 architecture:

1. The instruction set includes both graphics and general-purpose instructions. This made the TMS340 a credible stand-alone processor.

2. The internal data path is 32-bits wide and so are the arithmetic registers. The physical address range is 128 megabytes.

3. Pixel size is programmable at 1, 2, 4, 8, 16, or 32 bits.

4. Raster operations includes 16 boolean and 6 arithmetic options.

5. The chip contains 30 general purpose 32-bit registers. This is approximately four times as many registers as in an Intel 80386.

6. The 512-byte instruction cache allows the CPU to place a considerable number of instructions in the TMS340 queue while continuing to execute in parallel.

7. The coprocessor contains dedicated graphics instructions to draw single pixels and lines, and to perform twodimensional pixels array operations, such as pixBlts, area fills, and block transfers, as well as several auxiliary graphics functions.

The limited commercial success of the TMS 340-based systems is probably due to the slow development of graphics applications that took advantage of the chip's capabilities. Systems based on the TM 340 sold from $500 to well over $1000 and they had little commercial software support. The most important consequence of this technology was demonstrating that the PC was capable of high-quality, high-performance graphics.

1.5.2 Image Properties

An image is a surrogate of reality. Its main purpose it to convey visual information to the viewer. In computer technology the graphics image is usually a dot pattern displayed on a CRT monitor. Some of the characteristics of the computer image can be scientifically measured or at least evaluated objectively. But the human element in the perception of the graphic image introduces factors that are not easily measured. For example, aesthetic considerations can help us decide whether a certain graphic image "looks better" than another one, yet another image can give us an eyestrain headache that cancels its technological virtues.

Brightness and Contrast

Luminance is defined as the light intensity per unit area reflected or emitted by a surface. The human eye perceives objects by detecting differences in levels of luminance and color. Increasing the brightness of an object also increases the acuity with which it is perceived. However, it has been found that the visibility or legibility of an image is more dependent on contrast than on its absolute color or brightness.

The visual acuity of an average observer sustains an arc of approximately 1 minute. Therefore, the average observer can resolve an object that measures 5 one-thousands of an inch across when the image is displayed on a CRT and viewed at a distance of 18 inches. However, visual acuity falls rapidly with decreased luminance levels and with reduced contrast. This explains why ambient light, reflected off the surface of a CRT, decreases legibility.

A peculiarity of human vision is the decreasing ability of the eye to perceive luminance differences or contrasts as the absolute brightness increases. This explains why the absolute luminance values between object and background are less important to visual perception than their relative luminance, or contrast.

Color

Approximately three-fourths of the light-perceiving cells in the human eye are color-blind, which determines that luminance and contrast are more important to visual perception than color. Nevertheless, color is generally considered a valuable enhancement to the graphics image. The opinion is probably related to the popular judgment that color photography, cinematography, and television are to be preferred over the black-and-white versions.

Resolution

The quality of a raster-scan CRT is determined by the total number of separately addressable pixels contained per unit area. This ratio, called the resolution, is usually expressed in pixels-per-inch. For example, a CRT with 8-inch rows containing a total of 640 pixels per row has a horizontal resolution of 80 pixels per inch, while a CRT measuring 6 inches vertically and containing a total of 480 pixels per column has a vertical resolution of 80 pixels per inch.

Aspect Ratio

The aspect ratio of a CRT display is the relation between the horizontal and vertical dimensions of the image area. For example, a viewing surface measuring 8 inches horizontally and 6 inches vertically, is said to have a 4:3 aspect ratio. An 8t inch by 6 inch viewing surface has a 1:1 aspect ratio. Figure 1-10, on the following page, shows a CRT with a 4:3 aspect ratio.

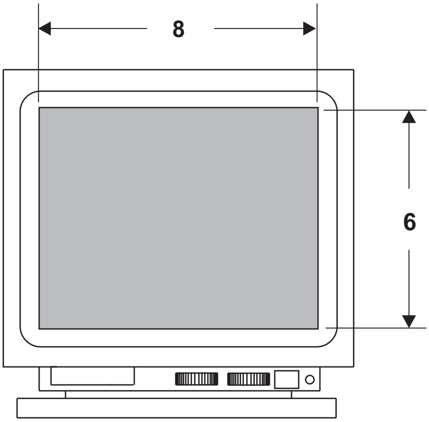


Figure 1-10 *CRT with a 4:3 Aspect Ratio*

1.6 Graphics Applications

Applications of computer graphics in general, and of 3D graphics in particular, appear to be limitless. The range of possible applications seems to relate more to economics and to technology than to intrinsic factors. It is difficult to find a sphere of computing that does not profit from graphics in one way or another. This is true of both applications and operating systems. In today's technology, graphics is the reality of computing. In PC programming graphics are no longer an option, but a standard feature that cannot be ignored.

1.6.1 Computer Games

Since the introduction of Pac Man in the mid 1980s, computer games have played an important role in personal entertainment. More recently we have seen an increase in popularity of dedicated computer-controlled systems and user-interaction devices, such as those developed by Nintendo and Sega. In the past 3 or 4 years, computer games have gone through a remarkable revival. The availability of more powerful graphics systems and of faster processors, as well as the ingenuity and talent of the developers, have brought about the increase in the popularity of this field. Computer games are one of the leading sellers in today's software marketplace, with sales supported by an extensive subculture of passionate followers. Electronic games are always at the cutting edge of computer graphics and animation. A game succeeds or fails according to its performance. It is in this field where the graphics envelope is pushed to the extreme. 3D graphics technologies relate very closely to computer games. In fact, it can be said that computer games have driven graphics technology.

1.6.2 Graphics in Science, Engineering, and Technology

Engineering encompasses many disciplines, including architecture, and mechanical, civil, and electrical, and many others. Virtually every field of engineering finds application for computer graphics and most can use 3D representations. The most generally applicable technology is *computer-aided design* (CAD), sometimes called *computer-aided drafting*. CAD systems have replaced the drafting board and the T-square in the design of components for civil, electrical, mechanical, and electronic systems. A few years ago, a CAD system required a mainframe or minicomputer with high-resolution displays and other dedicated hardware. Similar capabilities can be had today with off-the-shelf PC hardware and software. Most CAD packages now include 3D rendering capabilities.

These systems do much more than generate conventional engineering drawings. Libraries of standard objects and shapes can be stored and reused. For example, a CAD program used in mechanical engineering can store nut and bolt designs, which can be re-sized and used as needed. The same applies to other frequently used components and standard shapes. Color adds a visual dimension to computer-generated engineering drawings, a feature that is usually considered too costly and difficult to implement manually. Plotters and printers rapidly and efficiently generate high-quality hardcopy of drawings. 3D CAD systems store and manipulate solid views of graphics objects, which facilitates the production of perspective views and projections. Wire-frame and solid modeling techniques allow the visualization of real-world objects and contours. CAD systems can also have *expertise* in a particular field. This *knowledge* can be used to check the correctness and integrity of a design.

In architecture and civil engineering, graphics systems find many applications. Architects use 3D modeling for displaying the interior and exterior of buildings. A graphics technique known as *ray tracing* allows the creation of solid models that show lighting, shading, and mirroring effects.

Computer graphics are used to predict and model system behavior. Simulation techniques allow creating virtual representations of practically any engineered system, be it mechanical, electrical, or chemical. Mathematical equations are used to manipulate 3D representations and to predict behavior over a period of *simulated* time. Graphics images, usually color-coded and often in 3D, are used to display movement, and to show stress points or other dynamic features which, without this technique, would have been left to the imagination.

Geographic Information Systems (GIS) computer graphics to represent, manipulate, and store geographic, cartographic, and other social data for the analysis of phenomena where geographical location is an important factor. Usually, the amount of data manipulated in a GIS is much larger than can be handled manually. Much of this data is graphics imagery in the form of maps and charts. GIS systems display their results graphically. They find application in land use and land management, agriculture, forestry, wildlife management, archeology, and geology. Programmable satellites and instruments allow obtaining multiple images that can later be used in producing 3D images.

Remote sensing refers to collecting data at a distance, usually through satellites and other spacecraft. Most natural resource mapping done today is by this technology. As the resolution of remotely-sensed imagery increases, and their cost decreases, many more practical uses will be found for this technology.

Automation and robotics also find extensive use for computer graphics. Computer Numerical Control (CNC) and Computer Assisted Manufacturing (CAM) systems are usually implemented in a computer graphics environment. State-of-the-art programs in this field display images in 3D.

1.6.3 Art and Design

Many artists use computer graphics as a development and experimental platform, and some as a final medium. It is hotly debated whether computer-generated images can be considered fine art, but there is no doubt that graphics technology is one of the most powerful tools for commercial graphics and for product design. As CAD systems have replaced the drafting board, draw and paint programs have replaced the artist's sketch pad. The commercial artist uses a drawing program to produce any desired effect with great ease and speed, and to experiment and fine tune the design. Computer-generated images can be stretched, scaled, rotated, filled with colors, skewed, mirrored, re-sized, extruded, contoured, and manipulated in many other ways. Photo editing applications allow scanning and transforming bitmapped images, which can later be vectorized and loaded into the drawing program or incorporated into the design as bitmaps.

Digital composition and typesetting is another specialty field in which computer graphics has achieved great commercial success. Dedicated typesetting systems and desktop publishing programs allow the creation of originals for publication, from a brochure or a newsletter to a complete book. The traditional typesetting method was based on “mechanicals” on which the compositor glued strips of text and images to form pages. The pages were later photographed and the printing plates manufactured from the resulting negatives. Today, composition is done electronically. Text and images are merged in digital form. The resulting page can be transferred into a digital typesetter or used to produce the printing plates directly. The entire process is based on computer graphics.

1.6.4 Business

In recent years a data explosion has taken place. In most fields more data is being generated than there are people to process it. Imagine a day in the near future in which 15 remote sensing satellites orbit the earth, each one of them transmitting an image every 15 minutes, of an area that covers 150 square miles. The resulting acquisition rate of an image per minute is likely to create processing and storage problems, but perhaps the greatest challenge will be to find ways of using this information. How many experts will be required just to look at these images? Recently there have been just two or three remote sensing satellites acquiring earth images and it is estimated that no more than 10 percent of these images have ever been analyzed. Along this same line, businesses are discovering that they accumulate and store more data than can be used. *Data mining* and *data warehousing* are techniques developed to find some useful nugget of information in these enormous repositories of raw data.

Digital methods of data and image processing, together with computer graphics, provide our only hope of ever catching up with this mountain of unprocessed data. A business graph is used to compress and make available a large amount of information, in a form that can be used in the decision-making process. Computers are required to sort and manipulate the data and to generate these graphs. The field of image processing is providing methods for operating on image data. Technologies are being developed to allow computers to "look at" imagery and obtain useful information. If we cannot dedicate a sufficient number of human experts to look at a daily heap of satellite imagery, perhaps we will be able to train computers for this task.

Computer-based command and control systems are used in the distribution and management of electricity, water, and gas, in the scheduling of railways and aircraft, and in military applications. These systems are based on automated data processing and on graphics representations. At the factory level they are sometimes called *process controls*. In both small and large systems, graphics displays are required to help operators and experts visualize the enormous amount of information that must be considered in the decision-making process. For example, the pilot of a modern-day commercial aircraft can obtain, at a glance, considerable information about the airplane and its components as they are depicted graphically on a video display. This same information was much more difficult to grasp and mentally process when it originated in a dozen or more analog instruments.

Computer graphics also serve to enhance the presentation of statistical data for business. Graphics data rendering and computer animation serve to make the presentation more interesting; for example, the evolution of a product from raw materials to finished form, the growth of a real estate development from a few houses to a small city, or the graphic depiction of a statistical trend. Business graphics serve to make more convincing presentations of products or services offered to a client, as a training tool for company personnel, or as an alternative representation of statistical data. In sales computer graphics techniques can make a company's product or service more interesting, adding much to an otherwise dull and boring description of properties and features.

1.6.5 Simulations

Both natural and man-made objects can be represented in computer graphics. The optical planetarium is used to teach astronomy in an environment that does not require costly instruments and that is independent of the weather and other conditions. One such type of computer-assisted device, sometimes called a simulator, finds practical and economic use in experimentation and instruction. Simulators are discussed later in this book, in the context of animation programming.

1.6.6 Virtual Reality

Technological developments have made possible a new level of user interaction with a computing machine, called *virtual reality*. Virtual reality creates a digital universe in which the user is immersed. This topic is also discussed in relation to computer animation.

1.6.7 Artificial Life

Artificial life, or *ALife*, has evolved around the computer modeling of biosystems. It is based on biology, robotics, and artificial intelligence. The results are digital entities that resemble self-reproducing and self-organizing biological life forms.

1.6.8 Fractal Graphics

Natural surfaces are highly irregular. For this reason, many natural objects cannot be represented by means of polygons or smooth curves. However, it is possible to represent some types of natural objects by means of a mathematical entity called a *fractal*. The word fractal was derived from *fractional dimensions*.

1.7 State-of-the-Art in PC Graphics

During the first half of the nineties, PC graphics were mostly DOS-based. The versions of Windows and OS/2 operating systems available lacked performance and gave programmers few options and little control outside of the few and limited graphics services offered at the system level. Several major graphics applications were developed and successfully marketed during this period, including professional quality CAD, draw and paint, and digital typesetting programs for the PC. But it was not until the introduction of 32-bit Windows, and especially after the release of Windows 95, that PC graphics took off as a mainstream force.

The hegemony of Windows 95 and its successors greatly contributed to the current graphics prosperity. At the end of the decade, DOS has all but disappeared from the PC scene and graphics applications for the DOS environment have ceased to be commercially viable. By providing graphics hardware transparency Windows has made possible the proliferation of graphics coprocessors, adapters, and systems with many dissimilar functions and fields of application. At the same time, the cost of high-end graphics systems has diminished considerably.

From the software side three major forces struggle for domination of PC graphics: DirectX, OpenGL, and several proprietary game development packages, of which Glide is perhaps the best known.

1.7.1 Graphics Boards

PC graphics boards available at this time can be roughly classified by their functionality into 2D and 3D accelerators, and by their interface into *Peripheral Component Interconnect* (PCI) and *Accelerated Graphics Port* (AGP) systems. The 16-bit *Industry Standard Architecture* (ISA) expansion bus is in the process of being phased out and few new graphics cards are being made for it. Table 1-1 compares the currently available PC system buses.

Table 1-1

Specifications of PC System Buses

BUS	WIDTH	CLOCK SPEED	DATA RATE
ISA	16 bits	8 MHz	(varies)
PCI	32 bits	33 MHz	132 MBps
AGP 1X	32 bits	66 MHz	264 MBps
AGP 2X	32 bits	133 MHz	528 MBps
AGP 4X	32 bits	266 MHz	1024 MBps

The PCI bus is present in many old-style Pentium motherboards and graphics cards continue to be made for this interface. It allows full bus mastering and supports data transfer rates in burst of up to 132MBps. Some PCI buses that use older Pentium 75 to 150 run at 25 or 30MHz, but the vast majority operate at 33MHz. The 66MHz PCI is seen in specialized systems.

The AGP port is dedicated for graphics applications and quadruples PCI performance. AGP technology is based on Intel's 440LX and 440BX chipsets used in Pentium II and Pentium III motherboards and on the 440 EX chipset designed for the Intel Celeron processors. The AGP port interface is defined in Intel's AGP4x protocol. A draft version of the AGP8x Interface Specification is currently in the public review stage. This new standard provides a system-level attach point for graphics controllers and doubles the bandwidth. At the same time it remains compatible with connectors and interfaces defined in AGP4x.

The great advantage of AGP over its predecessors is that it provides the graphics coprocessors with a high bandwidth access system memory. This allows applications to store graphics data in system RAM. 3D graphics applications use this additional memory by means of a process called *direct memory execute* (DIME) or *AGP texturing* to store additional image data and to enhance rendering realism. However, since AGP systems do not require that graphics cards support texturing, this feature cannot be taken for granted in all AGP boards. In fact, few graphics programs to date actually take advantage of this feature.

1.7.2 Graphics Coprocessors

While presently it is easy to pick AGP as the best available graphics bus for the PC, selecting a graphics coprocessor is much more complicated. Several among half a dozen graphics chips share the foreground at this time. Among them are the Voodoo line from 3Dfx (Voodoo2 and Voodoo Banshee), Nvidia's RIVA and GeForce processors, MGA-G200, and S3 Savage 3D chips. All of these chips are used in top-line boards in PCI and AGP forms. Other well known graphics chips are 3D Labs Permedia, S3's Virge, Matrox's MGA-64, and Intel's i740. Recently Nvidia announced their new GeForce3 graphics processing unit with a 7.63GB/sec memory bandwith and other state-of-the-art features. Several graphics cards and on-the-motherboard graphics systems that use the GeForce3 chip are currently under development. Hercules Computer Technologies 3DProphet III is one of the graphics cards that uses Nvidia's GeForce3.

1.7.3 CPU On-Board Facilities

Graphics, especially 3D graphics, is a calculation-intensive environment. The calculations are usually simple and can be performed in integer math, but many operations are required to perform even a simple transformation. Graphics coprocessors often rely on the main CPU for performing this basic arithmetic. For this reason, graphics-rendering performance is, in part, determined by the CPU's mathematical throughput. Currently the mathematical calculating engines are the math unit and the *Multimedia Extension* (MMX). The register size of the math unit and the MMX were expanded in the Pentium 4 CPU.

In the older Intel processors the math unit (originally called the 8087 mathematical coprocessor) was either an optional attachment or an optional feature. For example, you could purchase a 486 CPU with or without a built-in math unit. The versions with the math unit were designated with the letters DX and those without it as SX. With the Pentium the math unit hardware became part of every CPU and the programmer need not be concerned about its presence. The math unit is a fast and efficient numerical calculator that finds many uses in graphics programming. Since 486-based machines can be considered obsolete at this time, our code can safely assume the presence of the Intel math unit and take advantage of its potential.

In 1997, Intel introduced a version of their Pentium processor that contained 57 new instructions and eight additional registers designed to support the mathematical calculations required in 3D graphics and multimedia applications. This additional unit was named the Multimedia Extension or MMX. The Pentium II and later processors all include MMX. MMX is based on a the *Single Instruction Multiple Data* (SIMD) technology, an implementation of parallel processing; it has a single instruction operating on multiple data elements. In the MMX the multiple data is stored in integer arrays of 64 bits. The 64 bits can divided into 8 bytes, four packed words, two doublewords, or a single quadword. The instruction set includes arithmetic operations (add, subtract, and multiply), comparisons, conversions, logical operations (AND, NOT, OR, and XOR), shifts, and data transfers. The result is a parallel, simple, and fast calculating engine quite suitable for graphics processing, especially in 3D.

1.8 3D Application Programming Interfaces

The selection of a PC graphics environment for our application is further complicated by the presence of specialized *application programming interfaces* (APIs) furnished by the various chip manufacturers. For example, 3Dfx furnishes the *Glide* API for their line of graphics coprocessors. In recent years Glide-based games and simulations have been popular within the 3D gaming community. An application designed to take full advantage of the capabilities of the 3Dfx accelerators is often coded using Glide. However, other graphics coprocessors cannot run the resulting code, which makes the boards incompatible with the software developed using Glide. Furthermore, Glide and Direct3D are mutually exclusive. When a Glide application is running, Direct3D programs cannot start and vice versa.

1.8.1 OpenGL and DirectX

One 3D graphics programming interface that has attained considerable support is *OpenGL*, developed by Silicon Graphics International (SGI). OpenGL, which stands for Open Graphics Language, originated in graphics workstations and is now part of many system platforms, including Windows 95, 98, and NT, DEC's AXP, OpenVMS, and X Windows. This led some to believe that OpenGL will be the 3D graphics standard of the future. In 1999 Microsoft and SGI joined in a venture that was, reportedly, to integrate OpenGL and DirectX. The project, code named Fahrenheit, was later cancelled.

At this time the mainstream of 3D graphics programming continues to use Microsoft's DirectX. The main advantage offered by this package is portability and universal availability on the PC. DirectX functionality is part of Windows 95, 98, and NT and Microsoft provides, free of charge, a complete development package that includes a tutorial, support code, and sample programs. Furthermore, developers are given license to provide DirectX runtime code with their products with automatic installation that can be made transparent to the user.

Chapter 2

Polygonal Modeling

Topics:

- Vector and raster images
- Coordinate systems
- Polygonal representations
- Triangles and meshes

This chapter is about how graphics objects are represented and stored in a database. The starting point of computer graphics is the representation of graphical objects. The polygon is the primitive geometrical used in graphically representing objects. The face of a newborn baby, the surface of a glass vase, or a World War II tank can all be modeled using hard-sided polygons. Here we discuss the principles of polygonal representations and modeling.

2.1 Vector and Raster Data

Computer images are classified into two general types: those defined as a pixel map and those defined as one or more vector commands. In the first case we refer to *raster graphics* and in the second case to *vector graphics*. Figure 2-1, on the following page, shows two images of a cross, first defined as a bitmap, and then as a set of vector commands.

The left-side image of Figure 2-1 shows the attribute of each pixel encoded in a bitmap. The simplest scheme consists of using a 0-bit to represent a white pixel and a 1-bit to represent a black pixel. Vector commands, on the other hand, refer to the geometrical elements in the image. The vector commands in Figure 2-1 define the image in terms of two intersecting straight lines. Each command contains the start and end points of the corresponding line in a Cartesian coordinate plane that represents the system's video display.

An image composed exclusively of geometrical elements, such as a line drawing of a building, or a machine part, can usually be defined by vector commands. On the other hand, a naturalistic representation of a landscape may best be done with a bitmap. Each method of image encoding, raster- or vector-based, has its advantages and drawbacks. One fact often claimed in favor of vector representation is the resulting memory savings. For example, in a video surface of 600-by-400 screen dots, the bitmap for representing two intersecting straight lines encodes the individual states of 240,000 pixels. If the encoding is in a two-color form, as in Figure 2-1, then 1 memory byte is required for each 8 screen pixels, requiring a 30,000-byte memory area for the entire image. This same image can be encoded in two vector commands that define the start and end points of each line. By the same token, to describe in vector commands a screen image of Leonardo's Mona Lisa would be more complicated and memory consuming than a bitmap.

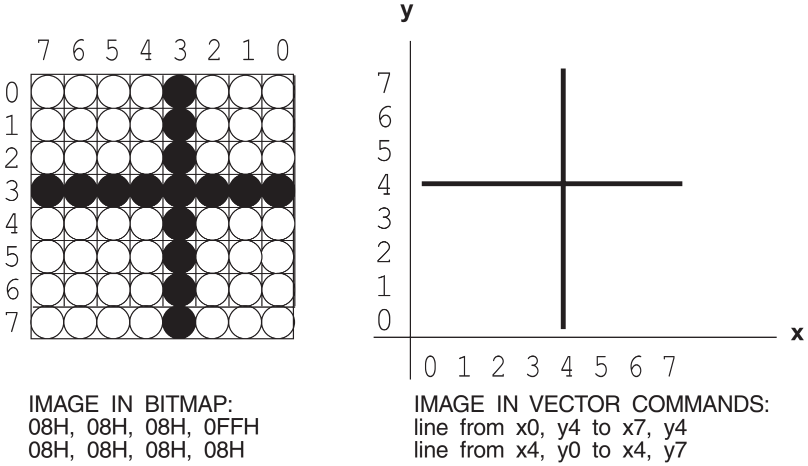


Figure 2-1 *Raster and Vector Representation of a Graphics Object*

In the 3D graphics rasterized images are mostly used as textures and backgrounds. 3D rendering is based on transformations that require graphics objects defined by their coordinate points. Software operates mathematically on these points to transform the encoded images. For example, a geometrically defined object can be moved to another screen location by adding a constant to each of its coordinate points. In Figure 2-2 the rectangle with its lower left-most vertex at coordinates $x = 1$, $y = 2$, is translated to the position $x = 12$, $y = 8$, by adding 11 units to its x coordinate and 6 units to its y coordinate.

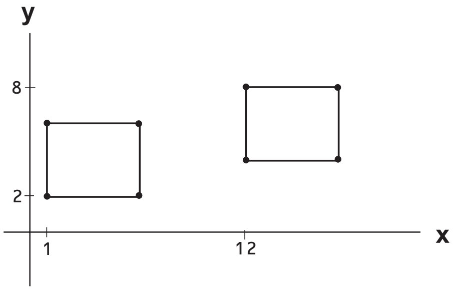


Figure 2-2 *Translating an Object by Coordinate Arithmetic*

In Chapter 3 we explore geometrical image transformations in greater detail.

2.2 Coordinate Systems

The French mathematician René Descartes (1596-1650) developed a two-dimensional grid that is often used for representing geometrical objects. In Descartes's system the plane is divided by two intersecting lines, known as the *abscissa* and the *ordinate* axis. Conventionally, the abscissa is labeled with the letter x and the ordinate with the letter y. When the axes are perpendicular, the coordinate system is said to be *rectangular*; otherwise, it is said to be *oblique*. The origin is the point of intersection of the abscissa and the ordinate axes. A point at the origin has coordinates (0,0). Coordinates in the Cartesian system are expressed in parenthesis, the first element corresponds to the x axis and the second one to the y axis. Therefore a point at (2,7) is located at coordinates $x = 2$, $y = 7$. Figure 2-3 shows the rectangular cartesian plane.

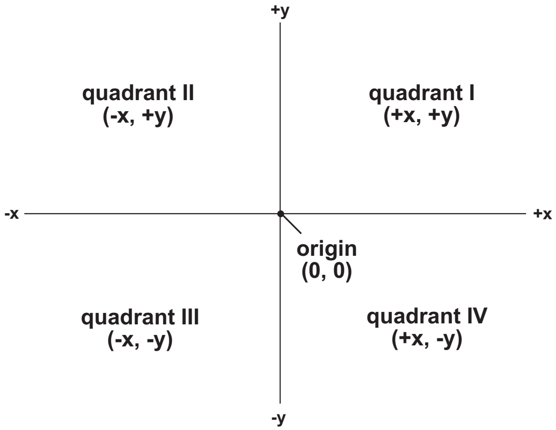


Figure 2-3 *Cartesian Coordinates*

In Figure 2-3 we observe that a point on the x-axis has coordinates (x, 0) and a point on the y-axis has coordinates (0, y). The origin is defined as the point with coordinates (0, 0). The axes divide the plane into four quadrants, usually labeled counterclockwise with Roman numerals I to IV. In the first quadrant x and y have positive values. In the second quadrant x is negative and y is positive. In the third quadrant both x and y are negative. In the fourth quadrant x is positive and y is negative.

The Cartesian coordinates plane can be extended to three-dimensional space by adding another axis, usually labeled z. A point in space is defined by a triplet that expresses its x, y, and z coordinates. Here again, a point at the origin has coordinates (0, 0, 0), while a point located on the any of the three axes has zero coordinates on the other two. In a rectangular coordinate system the axes are perpendicular. Each pair of axes determines a coordinate plane: the xy-plane, the xz-plane, and the yz-plane. The three planes are mutually perpendicular. A point in the xy-plane has coordinates (x, y, 0), a point in the xz-plane has coordinates (x,0,z), and so on. By the same token, a point not located on any particular plane has non-zero coordinates for all three axes. Figure 2-4 shows the Cartesian 3D coordinates.

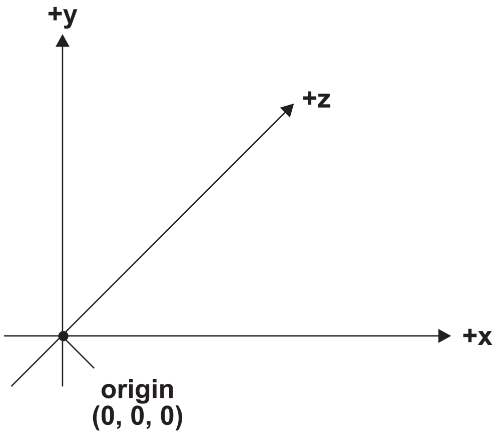


Figure 2-4 *3D Cartesian Coordinates*

The labeling of the axes in 3D space is conventional, although the most common scheme is to preserve the conventional labeling of the x and y axis in 2D space, and to add the z axis in the viewer's direction, as in Figure 2-4. However, adopting the axis labeling style in which positive x points to the right, and positive y points upward, still leaves undefined the positive direction of the z axis. For example, we could represent positive z-axis values in the direction of the viewer or in the opposite one. The case in which the positive values of the z-axis are in the direction of the viewer is called a right-handed coordinate system. The one in which the positive values of the z-axis are away from the viewer is called a left-handed system. This last system is consistent with the notion of a video system in which image depth is thought to be inside the CRT. Left- and right-handed systems are shown in Figure 2-5

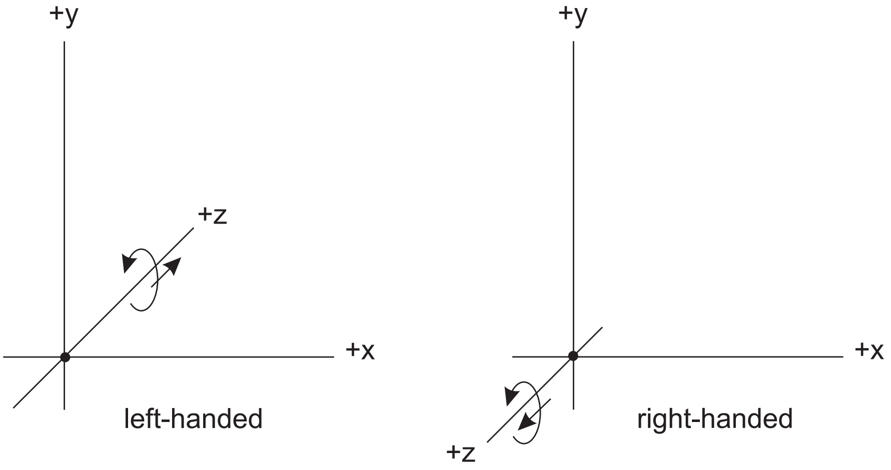
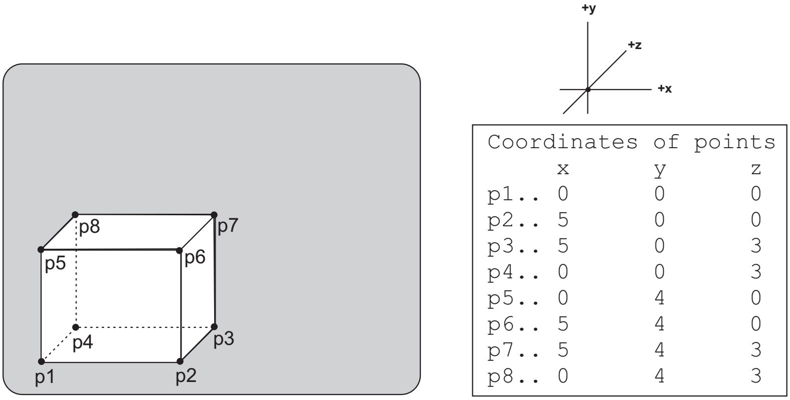


Figure 2-5 *Left- and Right-Handed Coordinates*

You can remember if a system is left- or right-handed by visualizing which hand needs to be curled over the *z*-axis so that the thumb points in the positive direction. In a left-handed system the left hand with the fingers curled on the *z*-axis has the thumb pointing away from the viewer. In a right-handed system the thumb points toward the viewer. This is shown in Figure 2-5.

3D modeling schemes do not always use the same axes labeling system. In some the *z*-axis is represented horizontally, the *y*-axis in the direction of the viewer, and the *x*-axis is represented vertically. In any case, the right- and left-handedness of a system is determined by observing the axis that lays in the viewer's direction, independently of its labeling. Image data can be easily ported between different axes' labeling styles by applying a rotation transformation, described later in this chapter. In Figure 2-6 we have used a 3D Cartesian coordinate system to model a rectangular solid with dimensions $x = 5$, $y = 4$, $z = 3$.



Coordinates of points

	x	y	z
p1..	0	0	0
p2..	5	0	0
p3..	5	0	3
p4..	0	0	3
p5..	0	4	0
p6..	5	4	0
p7..	5	4	3
p8..	0	4	3

Figure 2-6 *3D Representation of a Rectangular Solid*

The table of coordinates, on the right side of the illustration, shows the location of each vertex. Because the illustration is a 2D rendering of a 3D object, it is not possible to use a physical scale to determine coordinate values from the drawing. For example, vertices $p1$ and $p4$ have identical x and y coordinates; however, they appear at different locations on the flat surface of the drawing. In other words, the image data stores the coordinates points of each vertex; how these points are rendered on a 2D surface depends on the viewing system adopted, also called the projection transformation. Viewing systems and projections are discussed in Chapter 3.

An alternative visualization of the 3D Cartesian coordinate system is based on planes. In this model each axes pair determines a *coordinate plane*. Thus, we can refer to the *xy*-plane, the *xz*-plane, and the *yz*-plane. Like axes, the coordinate planes are mutually perpendicular. This means that the *z* coordinate of a point *p* is the value of the intersection of the *z*-axis with a plane through *p* that is parallel to the *yx*-plane. If the planes intersect the origin, then a point in the *xy*-plane has zero value for the *z* coordinate, a point in the *yz*-plane has zero value for the *x* coordinate, and a point in the *xz*-plane has zero for the *y* coordinate. Figure 2-7 shows the three planes of the 3D Cartesian coordinate system.

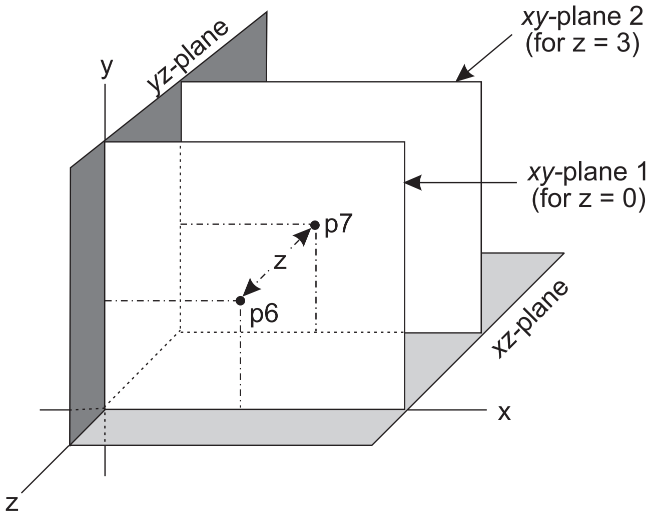


Figure 2-7 *3D Coordinate Planes*

We have transferred to Figure 2-7 points *p6* and *p7* of Figure 2-6. Point *p6* is located on *xy*-plane 1, and point *p7* in *xy*-plane 2. The plane labeled *xy*-plane 2 can be visualized as the result of sliding *xy*-plane 1 along the *z*-axis to the position *z* = 3. This explains why the *x* and *y* coordinates of points *p6* and *p7* are the same.

2.2.1 Modeling Geometrical Objects

Much of 3D graphics programming relates to representing, storing, manipulating, and rendering vector-coded geometrical objects. In this sense, the problem of representation precedes all others. Many representational forms are in use; most are related to a particular rendering algorithms associated with a graphics platform or development package. In addition, representational forms determine data structures, processing cost, final appearance, and editing ease. The following are the most frequently used:

1. *Polygonal representations* are based on reducing the object to a set of polygonal surfaces. This approach is the most popular one due to its simplicity and ease of rendering.

2. Objects can also be represented as *bicubic parameteric patch nets*. A patch net is a set of curvilinear polygons that approximate the object being modeled. Although more difficult to implement than polygonal representations, objects represented by parameteric patches are more fluid; this explains their popularity for developing CAD applications.

3. *Constructive solid geometry* (CSG) modeling is based on representing complex object by means of simpler, more elementary ones, such as cylinders, boxes, and spheres. This representation finds use in manufacturing-related applications.

4. *Space subdivision techniques* consider the whole object space and define each point accordingly. The best known application of space subdivision technique is ray tracing. With ray tracing processing is considerably simplified by avoiding brute force operations on the entire object space.

We concentrate out attention on polygonal modeling, with occasional reference to parameteric patches.

2.3 Modeling with Polygons

A simple polygon is a 2D figure formed by more than two connected and non-intersecting line segments. The connection points for the line segments are called the vertices of the polygon and the line segments are called the sides. The fundamental requirements that the line segments be connected and non-intersecting eliminates from the polygon category certain geometrical figures, as shown in Figure 2-8.

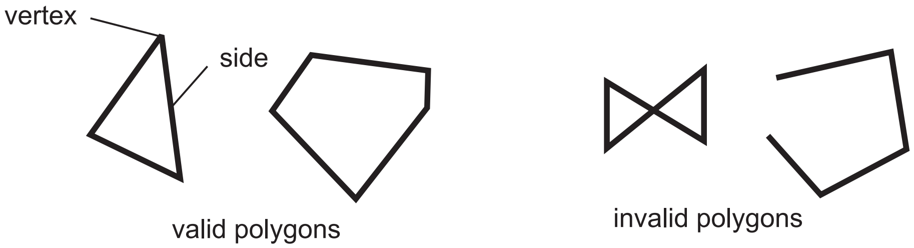


Figure 2-8 *Valid and Invalid Polygons*

Polygons are named according to their number of sides or vertices. A triangle, which is the simplest possible polygon, has three vertices. A quadrilateral has four, a pentagon has five, and so on. A polygon is said to be *equilateral* if all its sides are equal, and *equiangular* if all its angles are equal. A regular polygon is both equilateral and equiangular. Figure 2-9 shows several regular polygons.

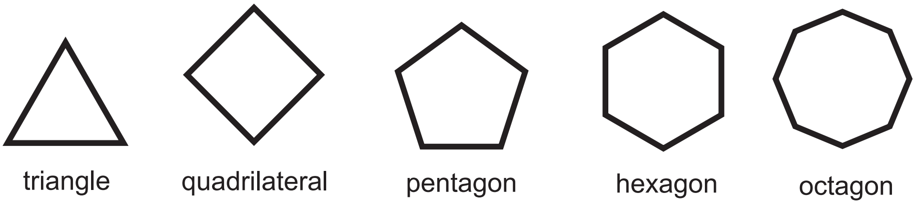


Figure 2-9 *Regular Polygons*

Polygons can be *convex* or *concave*. In a convex polygon the extension of any of its sides does not cut across the interior of the figure. We can also describe a convex polygon as one in which the extensions of the lines that form the sides never meet another side. Figure 2-10 shows a convex and a concave polygon.

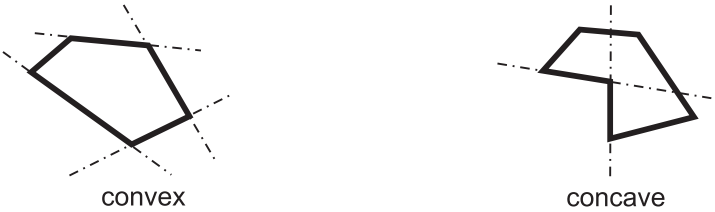


Figure 2-10 *Concave and Convex Polygons*

Specific software packages often impose additional restrictions on polygon validity in order to simplify the rendering and processing algorithms. For example, OpenGL requires that polygons be concave and that they be drawn without lifting the pen. In OpenGL, a polygon that contains a non-contiguous boundary is considered invalid.

2.3.1 The Triangle

Of all the polygons, the one most used in 3D graphics is the triangle. Not only is it the simplest of the polygons, but all the points in the surface of a triangular polygon must lie on the same plane. In other polygons this may or may not be the case. In other words, the figure defined by three vertices must always be a plane, but four or more vertices can describe a figure with more than one plane. When all the points on the figure are located on the same surface, the figure is said to be coplanar. Figure 2-11 shows coplanar and non-coplanar polygons.

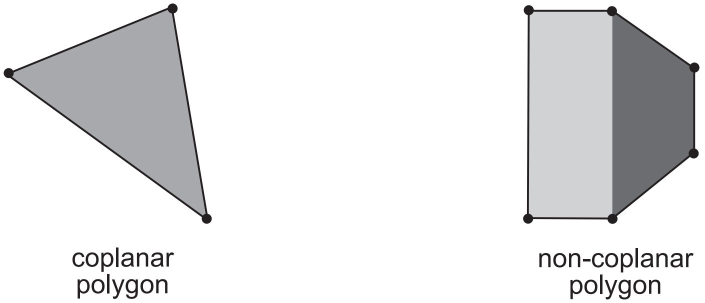


Figure 2-11 *Coplanar and Non-Coplanar Polygons*

The coplanar property of triangular polygons simplifies rendering. In addition, triangles are always convex figures. For this reason 3D software such as Microsoft's Direct3D, rely heavily on triangular polygons.

2.3.2 Polygonal Approximations

Solid objects with curved surfaces can be approximately represented by combining several polygonal faces. For example, a circle can be approximated by means of a polygon. The more vertices in the polygon, the better the approximation. Figure 2-12 shows the polygonal approximation of a circle. The first polygon has 8 vertices, while the second one has 16.

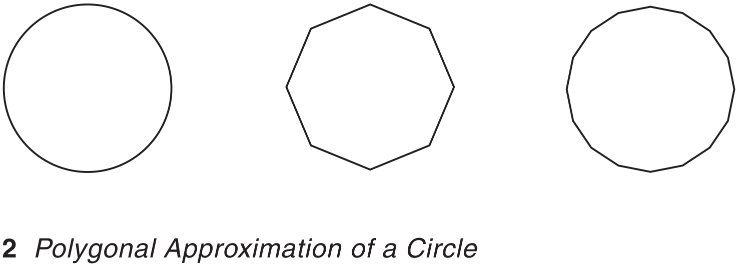


Figure 2-12 *Polygonal Approximation of a Circle*

A solid object, such as a cylinder, can be approximately represented by means of several polygonal surfaces. Here again, the greater the number of polygons, the more accurate the approximation. Figure 2-13 shows the polygonal approximation of a cylinder.

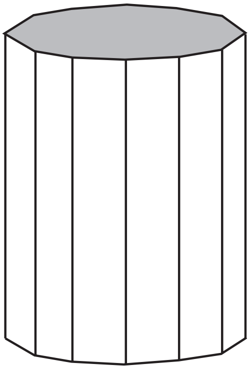


Figure 2-13 *Polygonal Approximation of a Cylinder*

2.3.3 Edges

When objects are represented by polygonal approximations, often two polygons share a common side. This connection between vertex locations that define a boundary is called an *edge*. Edge representations of polygons simplify the database by avoiding redundancy. This is particularly useful if an object shares a large number of edges. Figure 2-14 shows a figure represented by two adjacent triangular polygons that share a common edge.

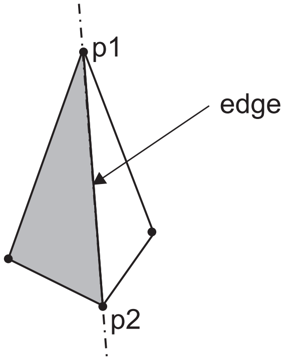


Figure 2-14 *Polygon Edge*

In an edge representation the gray triangle in Figure 2-14 is defined in terms of its three vertices, labeled $p1$, $p2$, and $p3$. The white triangle is defined in terms of its edge and point $p4$. Thus, points $p2$ and $p3$ appear but once in the database. Edge-based image databases provide a list of edges rather than of vertex locations. Figure 2-15 shows an object consisting of rectangular polygons.

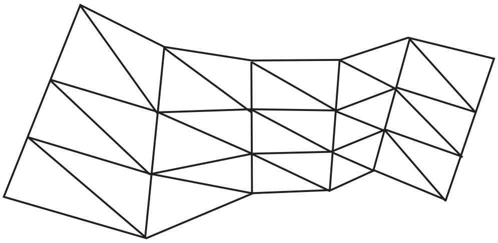


Figure 2-15 *Edge Representation of Polygons*

In Figure 2-15 each vertical panel consists of 6 triangles, for a total of 30 triangles. If each triangle were defined by its three vertices, the image database would require 90 vertices. Alternatively, the image could be defined in terms of sides and edges. There are 16 external sides which are not shared, and 32 internal sides, which are edges. Therefore, the edge-based representation could be done by defining 48 edges. The rendering system keeps track of which edges have already been drawn, avoiding duplication, processing overheads, and facilitating transparency.

2.3.4 Meshes

In 3D graphics an object can be represented as a polygon mesh. Each polygon in the mesh constitutes a *facet*. Facets are used to approximate curved surfaces; the more facets the better the approximation. Polygon-based modeling is straightforward and polygon meshes are quite suitable for using shading algorithms. In the simplest form a polygon mesh is encoded by means of the x, y, and z coordinates of each vertex. Alternatively, polygons can be represented by their edges, as previously described. In either case, each polygon is an independent entity that can be rendered as a unit. Figure 2-16 shows the polygon mesh representation of a teacup and the rendered image.

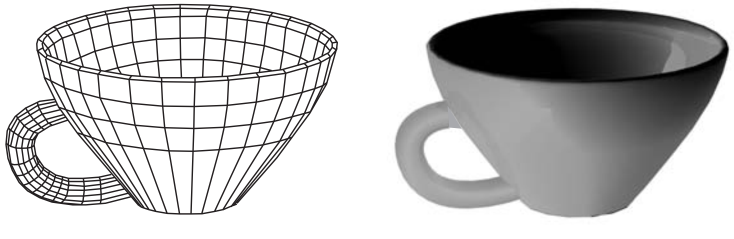


Figure 2-16 *Polygon Mesh Representation and Rendering of a Teacup*

Chapter 3

Image Transformations

Topics:

- Matrix arithmetic
- 2D transformations and homogeneous coordinates
- 3D transformations

Computer graphics rely heavily on geometrical transformations for generating and animating 2D and 3D imagery. In this chapter we introduce the essential transformation: translation, rotation, and scaling. The geometrical transformations are first presented in the context of 2D imagery, and later extended to 3D.

3.1 Matrix-Based Representations

In Chapter 2 we discussed vector images and how graphics objects are modeled by means of polygons and polygons meshes. Here we see how the coordinate points that define a polygon-based image can be manipulated in order to transform the image itself. Suppose an arrow indicating a northerly direction, which is defined by the coordinates of its start and end points. By rotating the end point 45 degree clockwise we can make the arrow point in a north-easterly direction. In general, if an image is defined as a series of points in the Cartesian plane, then the image can be rotated by a mathematical operation on the coordinates of each point. If the image is defined as one or more straight lines or simple polygons, then the transformation applied to the primitive image elements is also a transformation of the image itself.

Image transformations are simplified by storing the coordinates of each image point in a rectangular array. The mathematical notion of a matrix as a rectangular array of values turns out to be quite suitable for storing the coordinates of image points. Once the coordinates of each point that defines the image are stored in a matrix, we can use standard operations of linear algebra to perform geometrical transformations on the image. Figure 3-1 shows the approximate location of seven stars of the constellation Ursa Minor, also known as the Little Dipper. The individual stars are labeled with the letters a through g. The star labeled a corresponds to Polaris (the Pole star).

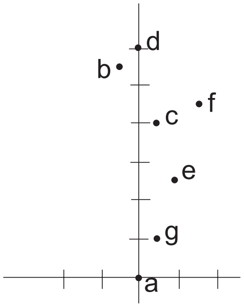


Figure 3-1 *Point Representation of the Stars In the Constellation Ursa Minor*

The coordinates of each star of the Little Dipper, in Figure 3-1, can be represented in tabular form, as follows:

Star	x	y
a	0	0
b	-1	11
c	1	8
d	0	12
e	2	5
f	3	9
g	1	2

The coordinate matrix is a sets of x,y coordinate pairs. 3D representations require an additional z coordinate that stores the depth of each point. 3D matrix representations are discussed later in this chapter.

3.1.1 Image Transformation Mathematics

An image can be changed into another one by performing mathematical operations on its coordinate points. Figure 3-2 shows the translation of a line from coordinates (2,2) and (10,14) to coordinates (10,2) and (18,14).

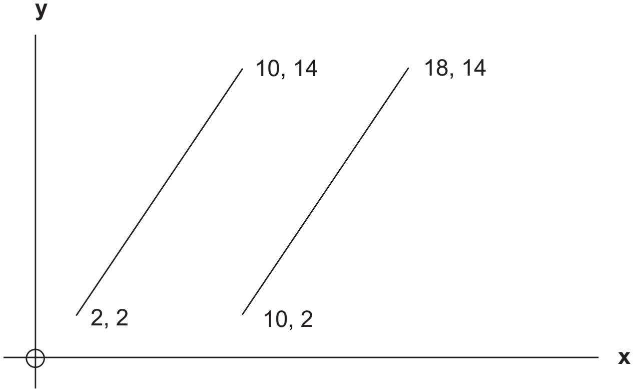


Figure 3-2 *Translation of a Straight Line*

Notice that in Figure 3-2 translation is performed by adding 8 to the start and end x coordinates of the original line. This operation on the x-axis coordinates results in a horizontal translation of the line. A vertical translation requires manipulating the y coordinate. To translate the line both horizontally and vertically we operate on both coordinate axes simultaneously.

3.2 Matrix Arithmetic

Matrices are used in many fields of mathematics. In linear algebra matrices can hold the coefficients of linear equations. Once an equation is represented in matrix form, it can be manipulated (and often solved) by performing operations on the matrix rows and columns. Here we are interested only in matrix operations that perform geometrical image transformations. The most primitive of these, translation, rotation, and scaling, are common in graphics and animation programming. Other transformations are reflection (mirroring) and shearing.

We define a matrix as a rectangular array usually containing a set of numeric values. It is customary to represent a matrix by means of a capital letter. For example, the following matrix, designated by the letter A, has three rows and two columns.

$$A = \begin{bmatrix} 10 & 22 \\ 3 & 4 \\ 7 & 1 \end{bmatrix}$$

The size of a matrix is determined by its number of rows and columns. It is common to state matrix size as a product, for example, matrix A, above, is a 3-by-2 matrix.

3.2.1 Scalar-by-Matrix Operations

A single numerical quantity is called a scalar. Scalar-by-matrix operations are the simplest procedures of matrix arithmetic. The following example shows the multiplication of matrix A by the scalar 3.

$$3A = \begin{bmatrix} 30 & 66 \\ 9 & 12 \\ 21 & 3 \end{bmatrix}$$

If a scalar is represented by the variable s, the product matrix sA is the result of multiplying each element in the matrix A by the scalar s. In the same manner, scalar addition and subtraction are performed by adding or subtracting the scalar quantity to each matrix element.

3.2.2 Matrix Addition and Subtraction

Matrix addition and subtraction are performed by adding or subtracting each element in a matrix to the corresponding element of another matrix of equal size. In the following example, matrix C is the algebraic sum of each element in matrices A and B.

$$A + B = C$$

$$\begin{bmatrix} 2 & 4 \\ 3 & 11 \\ 1 & 5 \\ 1 & -1 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 2 & 2 \\ -1 & -3 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 5 & 13 \\ 0 & 2 \\ 1 & -1 \end{bmatrix}$$

The fundamental restriction of matrix addition and subtraction is that both matrices must be of equal size, that is, they must have the same number of rows and of columns. Matrices of different sizes cannot be added or subtracted.

3.2.3 Matrix Multiplication

Matrix addition and subtraction intuitively correspond to conventional addition and subtraction. The elements of the two matrices are added or subtracted, one-to-one, to obtain the result. The fact that both matrices must be of the same size makes the operations easy to visualize. Matrix multiplication, on the other hand, is not the multiplication of the corresponding elements of two matrices, but a unique sum-of-products operation. In matrix multiplication the elements of a row in the multiplicand matrix are multiplied by the elements in a column of the multiplier matrix. These resulting products are then added to obtain the final result. The process is best explained by describing the individual steps. Consider the following matrices:

$$A = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 10 & 2 \\ 1 & 2 & 3 \\ 11 & 5 & 4 \end{bmatrix}$$

From the definition of matrix multiplication we deduce that if the columns of the first matrix are multiplied by the rows of the second matrix, then each row of the multiplier must have the same number of elements as each column of the multiplicand. Notice that the matrices A and B, in the preceding example, meet this requirement. However, observe that product B x A is not possible, since matrix B has three elements per row and matrix A has only two elements in each column. For this reason the matrix operation A x B is possible but B x A is undefined. The row by column operation in A x B is performed as follows.

$$\begin{array}{ccccccc} \text{First row of A} & & \text{Columns of B} & & \text{Products} & & \text{Sum} \\ \begin{bmatrix} 1 & 3 & 5 \\ 1 & 3 & 5 \\ 1 & 3 & 5 \end{bmatrix} & \times & \begin{bmatrix} 5 & 1 & 11 \\ 10 & 2 & 5 \\ 2 & 3 & 4 \end{bmatrix} & = & \begin{bmatrix} 5 & 3 & 55 \\ 10 & 6 & 25 \\ 2 & 9 & 20 \end{bmatrix} & = & \begin{matrix} 63 \\ 41 \\ 31 \end{matrix} \end{array}$$

$$\begin{array}{ccccccc} \text{Second row of A} & & \text{Columns of B} & & \text{Products} & & \text{Sum} \\ \begin{bmatrix}2&1&0\\2&1&0\\2&1&0\end{bmatrix} & \times & \begin{bmatrix}5&1&11\\10&2&5\\2&3&4\end{bmatrix} & = & \begin{bmatrix}10&1&0\\20&2&0\\4&3&0\end{bmatrix} & = & \begin{matrix}11\\22\\7\end{matrix}\end{array}$$

The products matrix has the same number of columns as the multiplicand matrix and the same number of rows as the multiplier matrix. In the previous example, the products matrix C has the same number of rows as A and the same number of columns as B. In other words, C is a 2 x 3 matrix. The elements obtained by the above operations appear in matrix C in the following manner:

$$C = \begin{bmatrix}63 & 41 & 31\\11 & 22 & 7\end{bmatrix}$$

Recall that in relation to matrices A and B in the previous examples, the operation A x B is possible but B x A is undefined. This fact is often described by saying that matrix multiplication is not commutative. For this reason, the product of two matrices can be different if the matrices are taken in different order. In fact, in regards to non-square matrices, if the matrix product A x B is defined, then the product B x A is undefined.

On the other hand, matrix multiplication is associative. This means that the product of three or more matrices is equal independently of the order in which they are multiplied. For example, in relation to three matrices, A, B, and C, we can state that (A x B) x C equals A x (B x C). In the coming sections you will often find use for the associative and non-commutative properties of matrix multiplication.

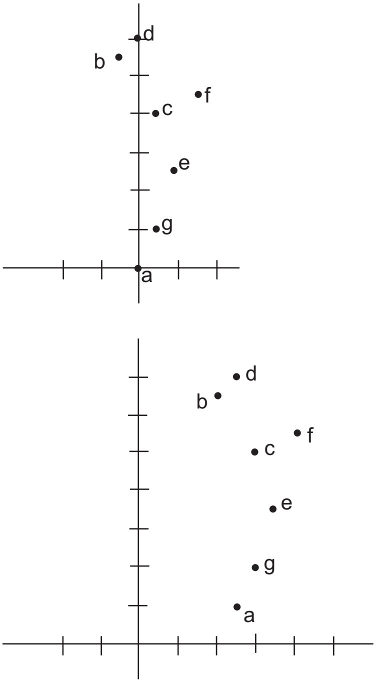
3.3 Geometrical Transformations

A geometrical transformation can be viewed as the conversion of one image onto another one by performing mathematical operations on its coordinate points. Geometrical transformations are simplified by storing the image coordinates in matrix form. In the following sections, we discuss the most common transformations: translation, scaling, and rotation. The transformations are first described in terms of matrix addition and multiplication, and later standardized so that they can all be expressed in terms only of matrix multiplication.

3.3.1 Translation Transformation

A *translation transformation* is the movement of a graphical object to a new location by adding a constant value to each coordinate point. The operation requires that the same constant be added to all the coordinates in each plane, but a different constant can be used for each plane. For example, a translation transformation takes place if the constant 5 is added to all x coordinates and the constant 2 to all y coordinates of an object represented in a two-dimensional plane.

In Figure 3-3 we see the graph and the coordinates matrix for seven stars in the Constellation Ursa Minor. A translation transformation is performed by adding 5 to the x coordinate of each star and 2 to the y coordinate. The bottom part of Figure 3-3 shows the translated image and the new coordinates.



original coordinates:

star	x	y
a	0	0
b	-1	11
c	1	8
d	0	12
e	2	5
f	3	9
g	1	2

translated coordinates (x+5, y+2):

star	x	y
a	5	2
b	4	13
c	6	10
d	5	14
e	7	7
f	8	11
g	6	4

Figure 3-3 *A Translation Transformation*

In terms of matrices, the translation transformation can be viewed as the operation:

$$A + B = C$$

where A is the matrix holding the original coordinates, B is the transformation matrix holding the values to be added to each coordinate plane, and C is the matrix of the transformed coordinated. Regarding the images in Figure 3-3 the matrix operation is as follows:

$$\begin{bmatrix} 0 & 0 \\ -1 & 11 \\ 1 & 8 \\ 0 & 12 \\ 2 & 5 \\ 3 & 9 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 5 & 2 \\ 5 & 2 \\ 5 & 2 \\ 5 & 2 \\ 5 & 2 \\ 5 & 2 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} 5 & 2 \\ 4 & 13 \\ 6 & 10 \\ 5 & 14 \\ 7 & 7 \\ 8 & 11 \\ 6 & 4 \end{bmatrix}$$

Notice that the transformation matrix holds the constants to be added to the x and y coordinates. Since, by definition of the translation transformation, the same value must be added to all the elements of a coordinate plane, it is evident that the columns of the transformation matrix always hold the same numerical value.

3.3.2 Scaling Transformation

To scale is to apply a multiplying factor to the linear dimension of an object. A *scaling transformation* is the conversion of a graphical object into another one by multiplying each coordinate point that defines the object. The operation requires that all the coordinates in each plane be multiplied by the scaling factor, although the scaling factors can be different for each plane. For example, a scaling transformation takes place when all the x coordinates of an object represented in a two-dimensional plane are multiplied by 2 and all the y coordinates of this same object are multiplied by 3. In this case the scaling transformation is said to by asymmetrical.

In comparing the definition of the scaling transformation to that of the translation transformation we notice that translation is performed by adding a constant value to the coordinates in each plane, while scaling requires multiplying these coordinates by a factor. The scaling transformation can be represented in matrix form by taking advantage of the properties of matrix multiplication. Figure 3-4 shows a scaling transformation that converts a square into a rectangle.

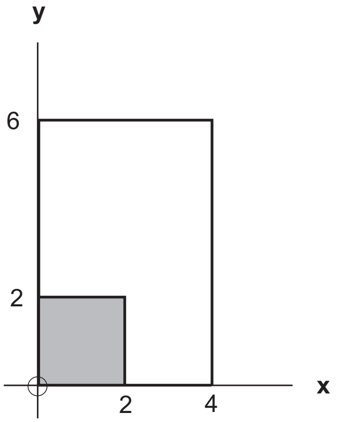


Figure 3-4 *Scaling Transformation*

The coordinates of the square in Figure 3-4 can be stored in a 4-by-2 matrix, as follows:

$$\begin{bmatrix} 0 & 0 \\ 2 & 0 \\ 2 & 2 \\ 0 & 2 \end{bmatrix}$$

In this case the transformation matrix holds the factors that must be multiplied by the x and y coordinates of each point in order to perform the scaling transformation. Using the term Sx to represent the scaling factor for the x coordinates, and the term Sy to represent the scaling factor for the y coordinates, the scaling transformation matrix is as follows:.

$$\begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix}$$

The transformation of Figure 3-4, which converts the square into a rectangle, is expressed in matrix transformation as follows:

$$\begin{bmatrix} 0 & 0 \\ 2 & 0 \\ 2 & 2 \\ 0 & 2 \end{bmatrix} \times \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 4 & 0 \\ 4 & 6 \\ 0 & 6 \end{bmatrix}$$

The intermediate steps in the matrix multiplication operation can be obtained following the rules of matrix multiplication described previously.

Figure 3-5 shows the scaling transformation of the graph of the constellation Ursa Minor. In this case, in order to produce a symmetrical scaling, the multiplying factor is the same for both axes. A symmetrical scaling operation is sometimes referred to as a *zoom*.

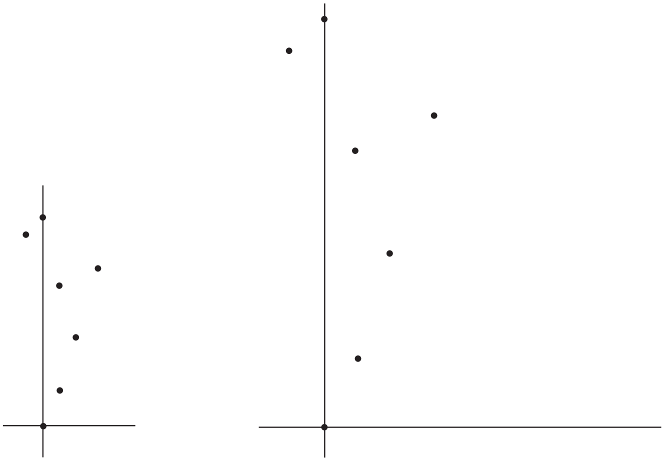


Figure 3-5 *Symmetrical Scaling (Zooming)*

3.3.3 Rotation Transformation

A *rotation transformation* is the conversion of a graphical object into another one by moving all coordinate points that define the original object, by the same angular value, along circular arcs with a common center. The angular value is called the *angle of rota-*

tion and the fixed point that is common to all the arcs is the *center of rotation*. Notice that some geometrical figures are unchanged by specific rotations. For example, a circle is unchanged by a rotation about its center, and a square is unchanged if rotated by an angle that is a multiple of 90 degrees. In the case of a square the intersection point of both diagonals is the center of rotation.

The mathematical interpretation of the rotation is based on elementary trigonometry. Figure 3-6 shows the counterclockwise rotation of points located on the coordinate axes, at unit distances from the center of rotation.

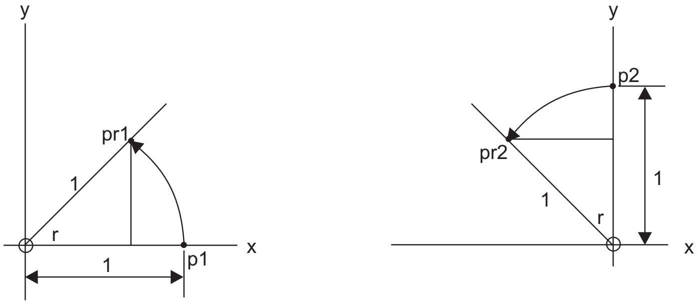


Figure 3-6 *Rotation of a Point*

The left side drawing of Figure 3-6 shows the counterclockwise rotation of point $p1$, with coordinates (1,0), through an angle r. The coordinates of the rotated point ($pr1$) can be determined by solving the triangle with vertices at O, $p1$ and $pr1$, as follows:

$$\cos r = \frac{x}{1}, x = \cos r$$

$$\sin r = \frac{y}{1}, y = \sin r$$

The coordinates of the rotated point $pr2$, shown on the right side drawing in Figure 3-6, can be determined by solving the triangle with vertices at O, $p2$ and $pr2$.

$$\sin r = \frac{-x}{1}, x = -\sin r$$

$$\cos r = \frac{y}{1}, y = \cos r$$

The coordinates of the rotated points can now be expressed as follows.

coordinates of $pr1$ = (cos r, sin r)

coordinates of $pr2$ = (–sin r, cos r)

From these equations we can derive a transformation matrix, which, through matrix multiplication, yields the new coordinates for the counterclockwise rotation through an angle A

$$\begin{bmatrix} \cos r & \sin r \\ -\sin r & \cos r \end{bmatrix}$$

We are now ready to perform a rotation transformation through matrix multiplication. Figure 3-7 shows the clockwise rotation of the stars in the constellation Ursa Minor, through an angle of 60 degrees, with center of rotation at the origin of the coordinate axes.

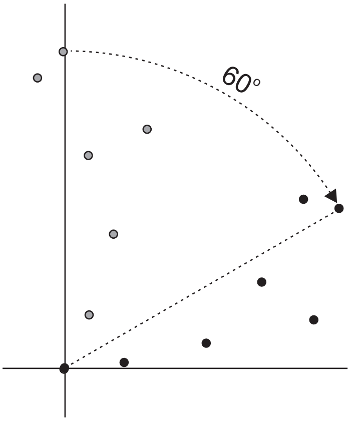


Figure 3-7 *Rotation Transformation*

Suppose that the coordinates of the four vertices of a polygon are stored in a 4-by-2 matrix as follows:

$$\begin{bmatrix} 10 & 2 \\ 12 & 0 \\ 14 & 2 \\ 12 & 4 \end{bmatrix}$$

The transformation matrix for clockwise rotation through an angle r is as follows:

$$\begin{bmatrix} \cos r & \sin r \\ -\sin r & \cos r \end{bmatrix}$$

Evaluating this matrix for 60 degrees gives the following trigonometric functions.

$$\begin{bmatrix} 0.5 & 0.867 \\ -0.867 & 0.5 \end{bmatrix}$$

Now the rotation can now be expressed as a product of two matrices, one with the coordinates of the polygon points and the other one with the trigonometric functions, as follows:

$$\begin{bmatrix} 10 & 2 \\ 12 & 0 \\ 14 & 2 \\ 12 & 4 \end{bmatrix} \times \begin{bmatrix} 0.5 & 0.867 \\ -0.867 & 0.5 \end{bmatrix} = \begin{bmatrix} 3.87 & 9.87 \\ 6 & 10.4 \\ 5.27 & 13.4 \\ 2.53 & 12.4 \end{bmatrix}$$

The resulting matrix contains the coordinates of the points rotated through and angle of 60 degrees. The intermediate steps in the matrix multiplication operation are obtained following the rules of matrix multiplication described earlier in this chapter.

3.3.4 Homogeneous Coordinates

Expressing translation, scaling, and rotation mathematically, in terms of matrix operations, allows simplifying graphical transformations. However, as previously described rotation and scaling are expressed in terms of matrix multiplication, while translation is expressed as matrix addition. It would simplify processing if all three basic transformations could be expressed in terms of the same mathematical operation. Fortunately, it is possible to represent the translation transformation as matrix multiplication. The scheme requires adding a dummy parameter to the coordinates matrices and expanding the transformation matrices to 3-by-3 elements.

If the dummy parameter, usually labeled w, is not to change the point's coordinates it must meet the following condition:

$$x = x \times w$$
$$y = y \times w$$

It follows that 1 is the only value that can be assigned to w . Using the terms Tx and Ty to represent the horizontal and vertical units of a translation, a transformation matrix for the translation operation can be expressed as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{bmatrix}$$

We test these results by performing a translation of 8 units in the horizontal direction (Tx = 8) and 0 units in the vertical direction (Ty = 0) of the point located at coordinates (5,2). In this case matrix operations are as follows:

$$\begin{bmatrix} 5 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 8 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 1 \end{bmatrix}$$

This shows the point at $x = 5$, $y = 2$ translated 8 units to the right, with destination coordinates of $x = 13$, $y = 2$. Observe that the w parameter, set to 1 in the original matrix, remains the same in the final matrix. For this reason, in actual processing the additional parameter can be ignored.

3.3.5 Concatenation

In order to take full advantage of the system of homogeneous coordinates you must express all transformations in terms of 3-by-3 matrices. As you have already seen, the translation transformation in homogeneous coordinates is expressed in the following matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{bmatrix}$$

The scaling transformation matrix is as follows:

$$\begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where Sx and Sy are the scaling factors for the x and y axes. The transformation matrix for a counterclockwise rotation through an angle r can be expressed in homogeneous coordinates as follows:

$$\begin{bmatrix} \cos r & \sin r & 0 \\ -\sin r & \cos r & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Notice that the rotation transformation assumes that the center of rotation is at the origin of the coordinate system.

Matrix multiplication is associative. This means that the product of three or more matrices is equal, no matter which two matrices are multiplied first. By virtue of this property, we are now able to express a complex transformation by combining several basic transformations. This process is generally known as *matrix concatenation*.

For example, in Figure 3-7 the image of the constellation Ursa Minor is rotated counterclockwise 60 degrees about the origin. But it is possible to perform this transformation using any arbitrary point in the coordinate system as a pivot point. For instance, to rotate the polygon about any arbitrary point *pa*, the following sequence of transformations is executed:

1. Translate the polygon so that point *pa* is at the coordinate origin.
2. Rotate the polygon.

3. Translate the polygon so that point pa returns to its original position.

In matrix form the sequence of transformations can be expressed as the following product:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -Tx & -Ty & 1 \end{bmatrix} \times \begin{bmatrix} \cos r & \sin r & 0 \\ -\sin r & \cos r & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{bmatrix}$$

Performing the indicated multiplication yields the matrix for a counterclockwise rotation, through angle r, about point pa, with coordinates (Tx,Ty).

While matrix multiplication is associative, it is not commutative. Therefore, the order in which the operations are performed can affect the results. A fact that confirms the validity of the matrix representation of graphic transformations is that, graphically, the results of performing transformations in different sequences can also yield different results. For example, the image resulting from a certain rotation, followed by a translation transformation, may not be identical to the one resulting from performing the translation first and then the rotation.

Figure 3-8 shows a case in which the order of the transformations determines a difference in the final object.

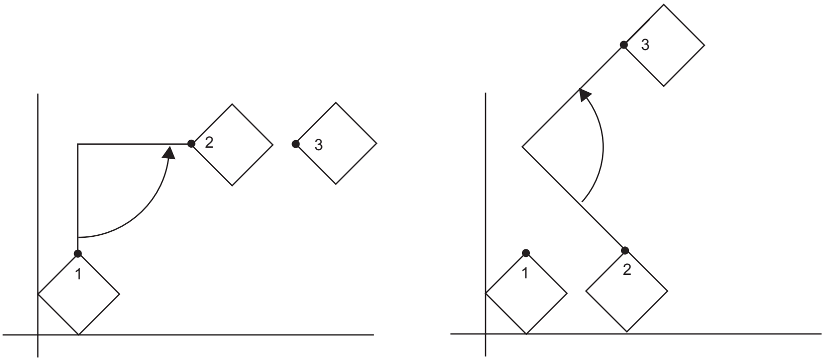


Figure 3-8 *Order of Transformations*

3.4 3D Transformations

Two-dimensional objects are defined by their coordinate pairs in 2D space. By extending this model we can represent a three-dimensional object by means of a set of coordinate triples in 3D space. Adding a z-axis that encodes the depth component of each image point produces a three-dimensional coordinate plane. The coordinates that define each image point in 3D space are a triplet of x, y, and z values. Because the three-dimensional model is an extension of the two-dimensional one, we can apply geometrical transformations in a similar manner as we did with two-dimensional objects. Figure 3-9 shows a cube in 3D space.

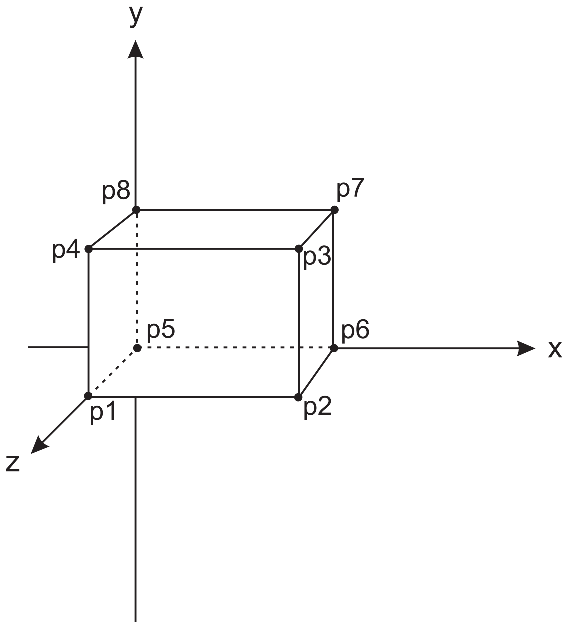


Figure 3-9 *3D Representation of a Cube.*

In Figure 3-9 the cube is defined by means of the coordinate triplets of each of its eight points, represented in the figure by the labeled black dots. In tabular form the coordinates of each point are defined as follows:

	X	Y	Z
P1	0	0	2
P2	4	0	2
P3	4	2	2
P4	0	2	2
P5	0	0	0
P6	4	0	0
P7	4	2	0
P8	0	2	0

Point p5, which is at the origin, has values of zero for all three coordinates. Point p1 is located 2 units along the z-axis, therefore its coordinates are $x = 0$, $y = 0$, $z = 2$. Notice that if we were to disregard the z-axis coordinates, then the two planes formed by points p1, p2, p3, and p4 and points p5, p6, p7, and p8 would have identical values for the x and y axis. This is consistent with the notion of a cube as a solid formed by two rectangles residing in 3D space.

3.4.1 3D Translation

In 2D representations a translation transformation is performed by adding a constant value to each coordinate point that defines the object. This continues to be true when the point's coordinates are contained in three planes. In this case the transformation constant is applied to each plane to determine the new position of each image point. Figure 3-10 shows the translation of a cube defined in 3D space by adding 2 units to the x axis coordinates, 6 units to the y axis, and -2 units to the z axis.

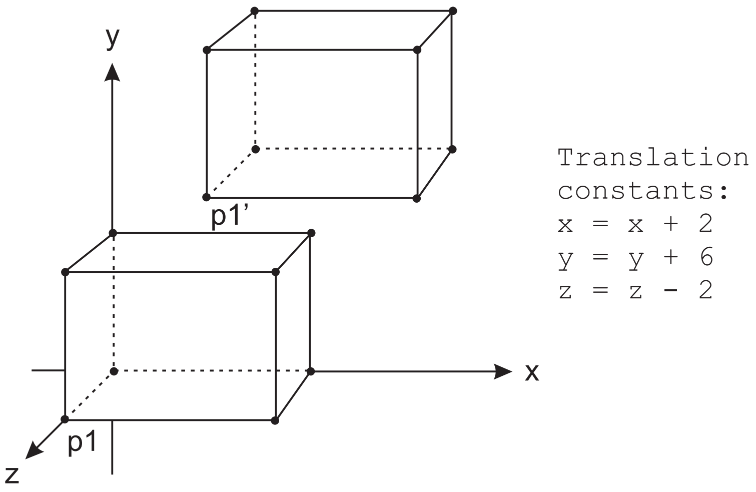


Figure 3-10 *Translation Transformation of a Cube*

If the coordinate points of the eight vertices of the cube in Figure 3-10 were represented in a 3-by-8 matrix (designated as matrix A) and the transformation constants in a second 8-by-3 matrix (designated as matrix B) then we could perform the translation transformation by means of matrix addition and store the transformed coordinates in a results matrix (designated as matrix C. The matrix operation C = A + B operation would be as follows:

$$\begin{bmatrix} 4 & 0 & 2 \\ 4 & 2 & 2 \\ 0 & 2 & 2 \\ 0 & 0 & 0 \\ 4 & 0 & 0 \\ 4 & 2 & 0 \\ 0 & 2 & 0 \end{bmatrix} + \begin{bmatrix} 2 & 6 & -2 \\ 2 & 6 & -2 \\ 2 & 6 & -2 \\ 2 & 6 & -2 \\ 2 & 6 & -2 \\ 2 & 6 & -2 \\ 2 & 6 & -2 \end{bmatrix} = \begin{bmatrix} 6 & 6 & 0 \\ 6 & 8 & 0 \\ 2 & 8 & 0 \\ 2 & 6 & -2 \\ 6 & 6 & -2 \\ 6 & 8 & -2 \\ 2 & 8 & -2 \end{bmatrix}$$

Here again, we can express the geometric transformation in terms of homogeneous coordinates. The translation transformation matrix for 3D space would be as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{bmatrix}$$

The parameters Tx, Ty, and Tz represent the translation constants for each axis. As in the case of a 2D transformation, the new coordinates are determined by adding the corresponding constant to each coordinate point of the figure to be translated. If x', y', and z' are the translated coordinates of the point at x, y, and z, the translation transformation takes place as follows:

$$x' = x + Tx$$

$$y' = y + Ty$$

$$z' = z + Tz$$

As in the case of 2D geometrical transformations, the transformed results are obtained by matrix multiplication using the matrix with the object's coordinate points as one product matrix, and the homogenous translation transformation matrix as the other one.

3.4.2 3D Scaling

A scaling transformation consists of applying a multiplying factor to each coordinate point that defines the object. A scaling transformation in 3D space is consistent with the scaling in 2D space. The only difference is that in 3D space the scaling factor is applied to each of three planes, instead of the two planes of 2D space. Here again the scaling factors can be different for each plane. If this is the case, the resulting transformation is described as an asymmetrical scaling. When the scaling factor is the same for all three axes, the scaling is described as symmetrical or uniform. Figure 3-11 shows the uniform scaling of a cube by applying a scaling factor of 2 to the coordinates of each figure vertex.

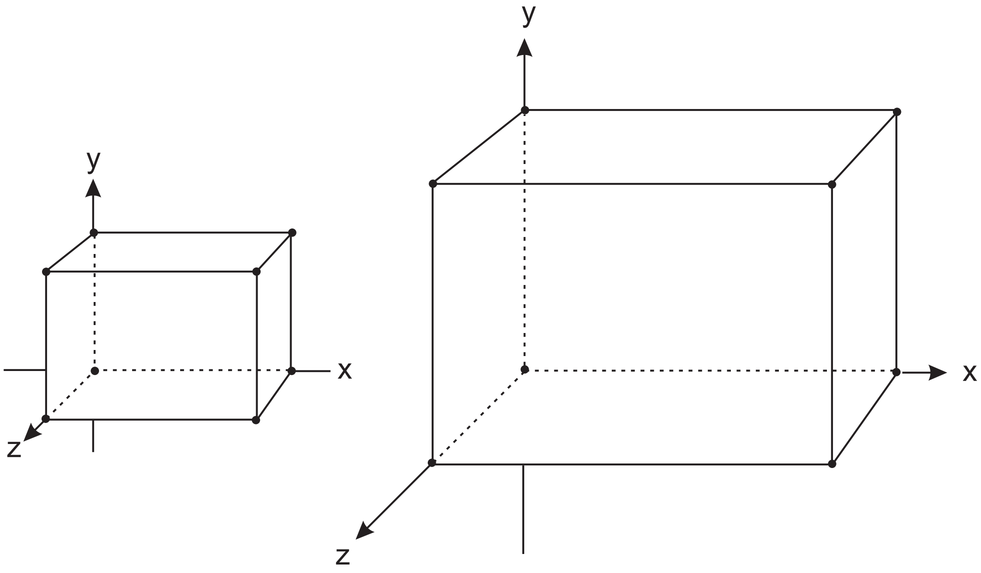


Figure 3-11 *Scaling Transformation of a Cube*

The homogeneous matrix for a 3D scaling transformation is as follows:

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The parameters Sx, Sy, and Sz represent the scaling factors for each axis. As in the case of a 2D transformation, the new coordinates are determined by multiplying the corresponding scaling factor with each coordinate point of the figure to be scaled. If x', y', *and* z' are the scaled coordinates of the point at x, y, and z, the scaling transformation takes place as follows:

$$x' = x \times Sx$$
$$y' = y \times Sy$$
$$z' = z \times Sz$$

In homogeneous terms, the transformed results are obtained by matrix multiplication using the matrix with the object's coordinate points as one product matrix, and the homogeneous scaling transformation matrix as the other one.

When the object to be scaled is not located at the origin of the coordinates axis, a scaling transformation will also result in a translation of the object to another location. This effect is shown in Figure 3-12.

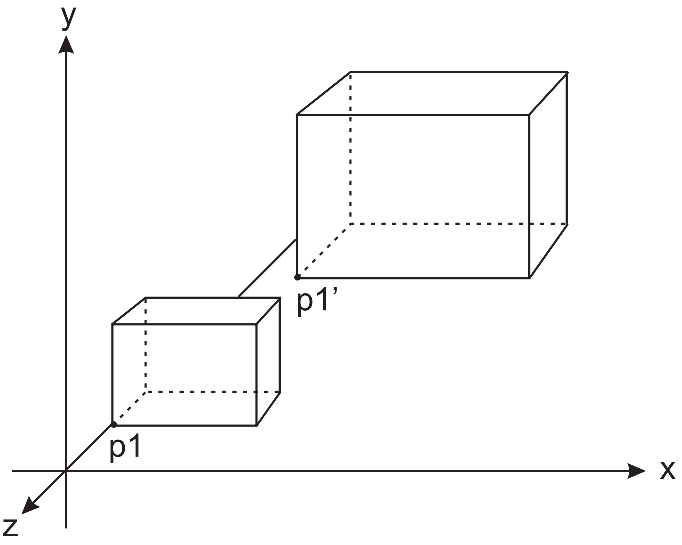


Figure 3-12 *Scaling Transformation of an Object Not at the Origin*

Assuming that point p1 in Figure 3-12 located at coordinates $x = 2$, $y = 2$, $z = -2$, and that a uniform scaling of 3 units is applied, then the coordinates of translated point p1’ are as follows:

	x	y	z
p1	2	2	-2
p1′.......	6	6	-12

The result is that not only is the cube tripled in size, it is also moved to a new position in the coordinates plane. In order to scale an image with respect to a fixed position it is necessary to first translate it to the origin, then apply the scaling factor, and finally to translate it back to its original location. The necessary manipulations are shown in Figure 3-13.

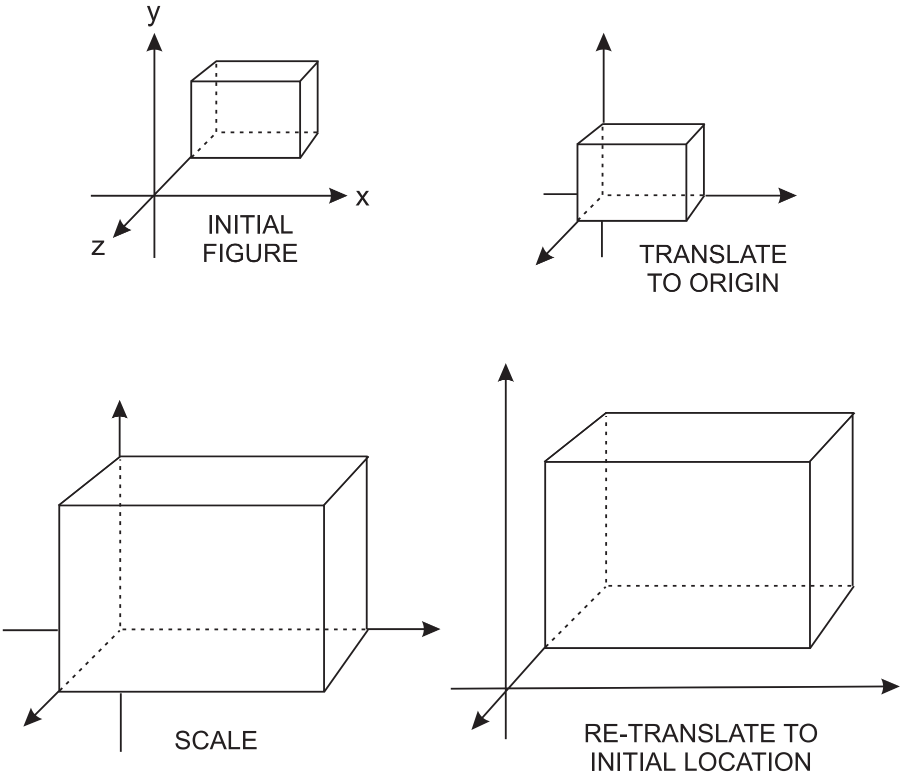


Figure 3-13 *Fixed-Point Scaling Transformation*

In terms of matrix operations a fixed-point scaling transformation consists of applying a translation transformation to move the point to the origin, then the scaling transformation, followed by another translation to return the point to its original location. If we represent the fixed position of the point as x^f, y^f, z^f, then the translation to the origin is represented by the transformation:

$$T\left(-x^{f},\ -y^{f},\ -z^{f}\right)$$

The transformation to return the point to its original location is:

$$T\left(x^{f},\ y^{f},\ z^{f}\right)$$

Therefore, the fixed-point scaling consists of:

$$T\left(-x^{f},\ -y^{f},\ -z^{f}\right)\ \times\ S(Sx,\ Sy,\ Sz)\ \times\ T(x^{f},\ y^{f},\ z^{f})$$

and the homogeneous matrix is:

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ (1-Sx)x^{f} & (1-Sy)y^{f} & (1-Sz)z^{f} & 1 \end{bmatrix}$$

where S is the scaling matrix and T the transformation matrix.

3.4.3 3D Rotation

Although 3D translation and scaling transformations are described as simple extensions of the corresponding 2D operations, the 3D rotation transformation is more complex that its 2D counterpart. The additional complications arise from the fact that in 3D space, rotation can take place in reference to any one of the three axes. Therefore an object can be rotated about the *x*, *y*, or *z* axes, as shown in Figure 3-14.

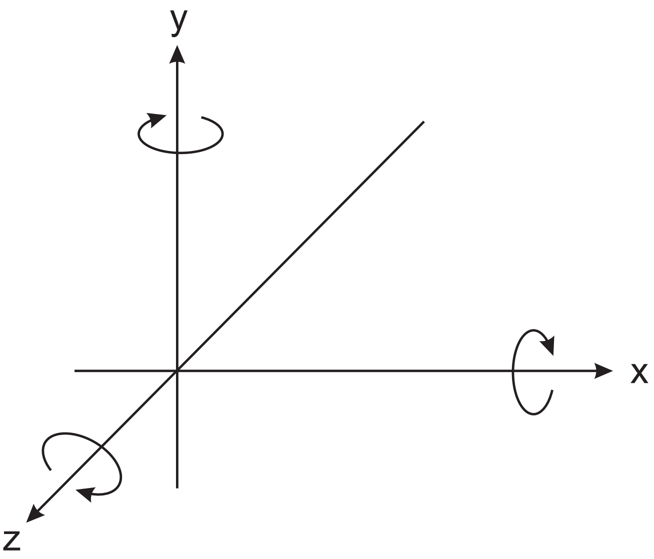


Figure 3-14 *Rotation in 3D Space*

In defining 2D rotation we adopted the convention that positive rotations produce a clockwise movement about the coordinate axes, when looking in the direction of the axis, towards the origin, as shown by the elliptical arrows in Figure 3-14. Figure 3-15 shows the positive, x-axis rotation of a cube.

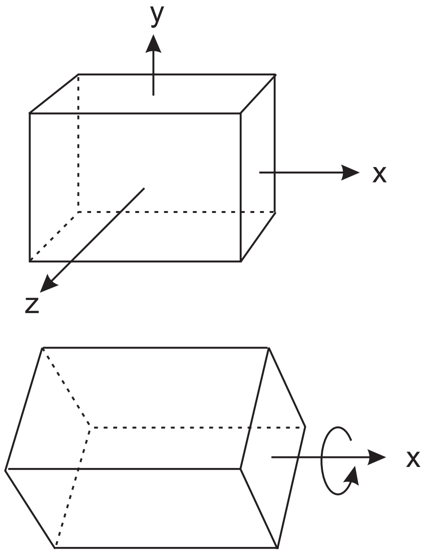


Figure 3-15 *Positive, x-axis Rotation of a Cube*

A rotation transformation leaves unchanged the coordinate values along the axis of rotation. For example, the x coordinates of the rotated cube in Figure 3-15 are the same as those of the figure at the top of the illustration. By the same token, rotating an object along the z-axis changes its y and x coordinates while the z-coordinates remain the same. Therefore, the 2D rotation transformation equations can be extended to a 3D rotation along the z-axis, as follows:

$$\begin{bmatrix} \cos r & \sin r & 0 & 0 \\ -\sin r & \cos r & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here again, r is the angle of rotation.

By performing a cyclic permutation of the coordinate parameters we can obtain the transformation matrices for rotations along the x and y axis. In homogeneous coordinates they are as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos r & \sin r & 0 \\ 0 & -\sin r & \cos r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} \cos r & 0 & -\sin r & 0 \\ 0 & 1 & 0 & 0 \\ \sin r & 0 & \cos r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.4.4 Rotation about an Arbitrary Axis

You often need to rotate an object about an axis parallel to the coordinate axis but different from the one in which the object is placed. In the case of the 2D fixed-point scaling transformation shown in Figure 3-13, we performed a translation transformation to reposition the object in the coordinates planes, then performed the scaling transformation, and concluded by re-translating the object to its initial location. Similarly, we can rotate a 3D object about an arbitrary axis by first translating it to the required position on the coordinate plane, then performing the rotation, and finally relocating the object at its original position. For example, suppose we wanted to rotate a cube, located somewhere on the coordinate plane, along its own x axis. In this case we may need to relocate the object so that the desired axis of rotation lies along the x-axis of the plane. Once in this position, we can perform the rotation applying the rotation transformation matrix for the x axis. After the rotation, the object is repositioned to its original location. The sequence of operations is shown in Figure 3-16.

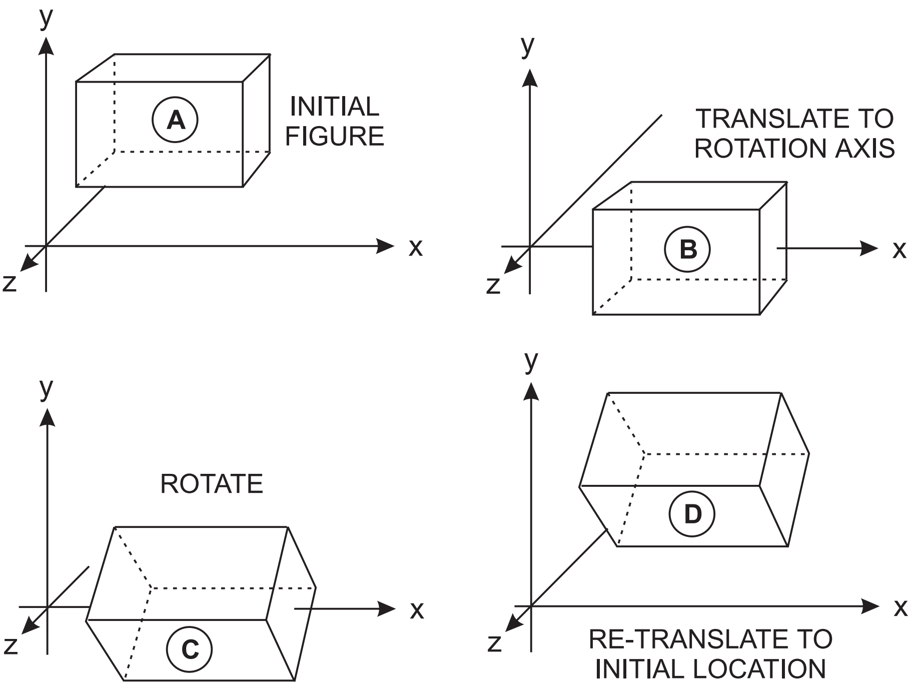


Figure 3-16 *Rotation About an Arbitrary Axis*

In this case it is possible to see one of the advantages of homogeneous coordinates. Instead of performing three different transformations, we can combine, through concatenation, the three matrices necessary for the entire transformation into a single one that performs the two translations and the rotation. Matrix concatenation was covered earlier in this chapter.

Chapter 4

Programming Matrix Transformations

Topics:

- Graphics data in matrix form
- Creating and storing matrix data
- Processing array elements
- Vector-by-scalar operations
- Matrix-by-matrix operations

The representation and manipulation of image data in matrix form is one of the most powerful tools of graphics programming in general, and of 3D graphics in particular. In this chapter we develop the logic necessary for performing matrix-based operations. Matrix-level processing operations are often used in 3D graphics programming.

4.1 Numeric Data in Matrix Form

In Chapter 3 you saw that a matrix can be visualized as a rectangular pattern of rows and columns containing numeric data. In graphics programming the data in the matrix is image related, most often consisting of the coordinate values, in 2D or 3D space, for the vertices of a polygon. In general terms, a matrix element is called an *entry*.

Matrix data is stored in computer memory as a series of ordered numeric items. Each numeric entry in the matrix takes up memory space according to the storage format. For example, if matrix data is stored as binary floating-point numbers in the FPU formats, each entry takes up the following space:

```
Single precision real ..... 4 bytes,
Double precision real ..... 8 bytes,
Extended precision real .. 10 bytes.
```

Integer matrices will vary from one high-level language to another one and even in different implementations of the same language. Microsoft Visual C++ in 32-bit versions of Windows uses the following data ranges for the integers types:

```
             char, unsigned char ....... 1 byte
             short, unsigned short ..... 2 bytes
             long, unsigned long ....... 4 bytes
```

The most common data format for matrix entries is the array. If the storage convention is based on having elements in the same matrix row stored consecutively in memory, then the matrix is in *row-major order*. However, if consecutive entries are items in the same matrix column then the matrix is in *column-major order*. C, C++, and Pascal assume row-major order, while BASIC and FORTRAN use column-major order.

4.1.1 Matrices in C and C++

In C and C++ a matrix can is usually defined as a multi-dimensional array. Like many other high-level languages, C++ implements multi-dimensional arrays as arrays of arrays. For example:

```
double matX[4][4] = {
                    {2.1, 3.0,  -1.0,  4.3},  // Array 1
                    {1.3, 0.0,   2.0, -3.2},  // Array 2
                    {1.2, 12.7, -4.0,  7.0},  // Array 3
                    {3.0, 1.0,  -1.0,  1.22}, // Array 4
                    };
```

The array matX[][] is two dimensional. In fact, it actually consists of four one-dimensional arrays. In C and C++ when a multi-dimensional array is passed as an argument to a function, the called code must be made aware of its dimensions so that it can access its elements with multiple subscripts. In the case of a two-dimensional array the function must know the number of dimensions and the number of columns in the array argument. The number of rows is not necessary since the array is already allocated in memory. The short program shows the definition of a 4-by-4 two-dimensional array and how this array is passed to a function that fills it.

```
#include <iostream.h>
#define ROWS 4
#define COLS 4

int main()
{

// Define a 4-by-4 matrix
int matrx[ROWS][COLS];

// Call function to fill matrix
FillMatrix(matrx);
// Display matrix
    for(i = 0; i < ROWS; i++)
        for(k = 0; k < COLS; k++)
             cout << matrx[i][k] << "\n";
return 0;
}

void FillMatrix(int matx[][COLS])
{
// Fill a matrix of type int
// On entry:
```

```
//       matx[][] is caller’s matrix
//       Constants ROWS and COLS define the array dimensions
int entry;

    for(int i = 0;i < ROWS; i++)
    {
        cout << “Enter row ” << i << “\n”;

       for(int j = 0; j < COLS; j++)
        {
            cout << “element ” << i << “ ” << j << “: ”;
            cin >> entry;
            matx[i][j] = entry;

       }
    }
}
```

In the preceding program the function FillMatrix() is made aware that the array passed as an argument is two-dimensional, and that each row consists of four columns. This is the information that the code requires for accessing the array with double subscript notation. The major limitation of this approach is that the function FillMatrix() requires the array size to be defined in constants. C/C++ produce an error if we attempt to define the matrix dimensions using variables. For this reason the function FillMatrix(), as coded in the preceding sample, does not work for filling a two-dimensional 5-by-5 array or, in fact, of any size other than the one for which it was originally coded. Even more complications arise if we attempt to use a template function in relation to multi-dimensional arrays.

An alternative approach for implementing matrices in C or C++ code is to define the data as a one-dimensional array and let the software handle the partitioning into columns and rows. In this manner we can avoid the drawbacks of passing multi-dimensional arrays as arguments to functions. In addition, with one-dimensional arrays it is easy to use templates in order to create generic functions that operate on arrays of different data types. The following demonstration program implements a matrix fill using one-dimensional arrays and template functions.

```
#include <iostream.h>

int main()
{

  int rows = 4;
  int cols = 4;

  // Matrix is defined as a 2D array
  matrx[16];

  // FillMatrix()
  FillMatrix(matrx, rows, cols);
    // Display matrix

    for(x = 0; x < rows; x++)
        for(y = 0; y < cols; y++)
            cout << mat2[(x * cols) + y] << “\n”;
    cout << "\n\n";

  return 0;
}

template <class A>
void FillMatrix(A *matx, int rows, int cols)
{
// Fill a matrix of type int
// On entry:
//       *mat is caller's matrix
//       parameter rows is nomber of rows in matrix
//       parameter cols is number of columns in matrix
A entry;

    for(int i = 0;i < rows; i++)
    {
       cout << "Enter row " << i << "\n";

       for(int j = 0; j < cols; j++)
        {
            cout << "element " << i << " " << j << ": ";
            cin >> entry;
            matx[(i * cols) + j] = entry;
        }
    }
}
```

Notice, in the preceding code, that the matrix is defined as a one-dimensional array and that the function FillMatrix() receives the number of rows and columns as parameters. Also that the FillMatrix() function is implemented as a template, which means that it can be used to fill a two-dimensional matrix of any size and data type.

In manipulating matrices the programmer is usually concerned with the following elements:

1. The number of rows in the matrix
2. The number of columns in the matrix
3. The memory space (number of bytes) occupied by each matrix entry

The number of rows and columns determines the dimension of the matrix. It is customary to represent matrix dimensions using the variable M for the number of rows and the variable N for the number of columns. The storage format of the entries determines the memory space occupied by each matrix entry, therefore, the number of bytes that must be skipped in order to index from entry to entry. In this sense the size of each entry is sometimes referred to as the *horizontal skip factor*. The number of entries in each matrix row must be used by the program in order to index to successive entries in the same column. This value is called the *vertical skip factor*. Low-level implementations must use the skip factors to access different matrix entries, as shown later in this chapter. High-level languages (C++ included) access matrix entries using the indices, and usually ignore the byte size of each element.

4.1.2 Finding Matrix Entries

You have seen that each matrix entry is identified by its row and column coordinates. In this context the variable i is often used to designate the entry along a matrix row and the variable j to designate the entry along a matrix column. Thus, any entry in the matrix can be identified by its $_{ij}$ coordinates. The individual matrix is usually designated with an upper case letter. We say that Matrix *A* is composed of *M* rows and *N* columns. The number of entries in the matrix (*E*) is:

$$E = M \times N$$

If each entry takes up *s* bytes of memory, the matrix memory space (S) can be expressed as follows:

$$S = M \times N \times s$$

The following diagram shows a 5-by-4 matrix.

```
                   C O L U M N S
                 0   1   2   3   4
                 |   |   |   |   |
R     0 ---   X   X   X   X   X
O     1 ---   X   X   ij  X   X
W     2 ---   X   X   X   X   X
S     3 ---   X   X   X   X   X

        M = 5 (total rows)
        N = 4 (total columns)
        i = 1 (row address of entry ij)
        j = 2 (column address of entry ij)
```

Notice that matrix dimensions are stated as the number of rows and columns. This, the dimension of the previous matrix is 4 by 5. However, the location within the rows and columns is zero-based. This scheme is consistent with array dimensioning and addressing in C and C++.

Linear systems software often has to access an individual matrix entry located at the *ith* row and the *jth* column. In high-level programming the language itself figures out the horizontal and vertical skip factors. Therefore locating a matrix entry is a simple matter of multiplying the row number by the number of columns in the matrix, then adding the offset within the row. If i designates the row, j the column, and cols is the number of columns in each row, then the offset within a matrix implemented in a one-dimensional array is given by the statement:

```
value = (M[(i*cols) + j);
```

were M is the matrix and value is a variable of the same type as the matrix. The following C++ template function returns the matrix element at row i, column j.

```
template <class A>
A Locateij(A *matx, int i, int j, int cols)
{
// Locate and return matrix entry at row i, column j
// On entry:
```

```
//       *mat is caller’s matrix
//       i = row number
//       j = column number
//       cols = number of matrix columns
 return (matx[(i * cols) + j]);
}
```

4.2 Array Processing

In the terminology of matrix mathematics, a *vector* is a matrix in which one of the elements is of the first order. In this sense you can refer to a matrix whose N dimension is 1 as a column vector. A row vector is a matrix whose M dimension is 1. In fact, a row vector is a matrix consisting of a single row, and a column vector a matrix consisting of a single column. Although, strictly speaking, a vector can be considered a one-dimensional matrix, the term matrix is more often associated with a rectangular array. Note that this use of the word *vector* is not related to the geometrical concept of a directed segment in two-dimensional or three-dimensional space, or with the physical connotation of a value specified in terms of magnitude and direction.

In order to represent individual, undirected quantities, matrix mathematics borrows from analytical geometry the notion of a *scalar*. We say that an individual constant or variable is a scalar quantity, while multi-element structures are either vectors or matrices.

Programs that perform mathematical operations on vectors and matrices are sometimes called *array processors*. In this case the word *array* refers to any multi-element structure, whether it be a matrix or a vector. Many array operations require simple arithmetic on the individual entries of the array, for example, adding, subtracting, multiplying or dividing all the entries of an array by a scalar, or finding the square root, powers, logarithmic, or trigonometric function of the individual entries. A second type of array operations refer to arithmetic between two multi-element structures, for example, the addition and multiplication of matrices, the calculation of vector products, and matrix inversion. Some matrix arithmetic operations obey rules that differ from those used in scalar operations. Finally, some array operations are oriented towards simplifying and solving systems of linear equations, for example, interchanging rows, multiplying a row by a scalar, and adding a multiple of one row to another row. Here we concentrate on array processing operations that are commonly used in graphics programming.

4.2.1 Vectors and Scalars

The word vector is used to refer to the rows and columns of a two-dimensional matrix. In this sense vector operations are those that affect the entries in a row or column, and matrix operations are those that affect all the entries in the rectangular array. Vectors constitute one-dimensional arrays of values, while matrices are a two-dimensional array. We occasionally refer to the entries in a matrix row as a row vector and the entries in a matrix column as a column vector.

Vector-by-Scalar Operations in C and C++

Graphics applications must occasionally perform operations on the individual elements of matrix rows and columns. According to the terminology presented previously, these can be designated as row and column vector operations. The functions listed in this section perform multiplication, addition, division, and subtraction of a row vector by a scalar and multiplication of a column vector by a scalar. The implementation is based on storing matrix data in one-dimensional arrays, with rows and columns handled by code. The functions are coded as templates so that they can be used with any compatible data type.

```
//****************************************************************
//              functions for vector arithmetic
//****************************************************************
//
template <class A>
void RowMulScalar(A *matx, int i, int cols, A scalar)
{
// Multiply a matrix row times a scalar
// On entry:
//       *mat is caller’s matrix
//       i is number of the row
//       cols is number of columns in the matrix
//       scalar is the value to multiply
// On exit:
//       elements in matrix row i are multiplied by scalar
int rowStart = i * cols;

for(int j = 0;j < cols ;j++)
    matx[rowStart + j] *= scalar;
}

template <class A>
void RowPlusScalar(A *matx, int i, int cols, A scalar)
{
// Add a scalar to a matrix row
// On entry:
//       *mat is caller’s matrix
//       i is number of the row
//       cols is number of columns in the matrix
//       scalar is the value to be added
// On exit:
//       Scalar is added to all elements in matrix row i
int rowStart = i * cols;

for(int j = 0;j < cols ;j++)
    matx[rowStart + j] += scalar;
}

template <class A>
void RowMinusScalar(A *matx, int i, int cols, A scalar)
{
// Subtract a scalar from each element in a matrix row
// On entry:
//       *mat is caller’s matrix
//       i is number of the row
//       cols is number of columns in the matrix
//       scalar is the value to be added
// On exit:
//       Scalar is subtracted from all elements in matrix row i
int rowStart = i * cols;

for(int j = 0;j < cols ;j++)
```

```
    matx[rowStart + j] -= scalar;
}

template <class A>
void RowDivScalar(A *matx, int i, int cols, A scalar)
{
// Divide all elements in a matrix row by a scalar
// On entry:
//       *mat is caller's matrix
//       i is number of the row
//       cols is number of columns in the matrix
//       scalar is the value
// On exit:
//       All elements in matrix row i are divided by the
//       scalar
int rowStart = i * cols;

for(int j = 0;j < cols ;j++)
    matx[rowStart + j] /= scalar;
}

template <class A>
void ColMulScalar(A *matx, int j, int rows, int cols, A scalar)
{
// Multiply a matrix column times a scalar
// On entry:
//       *mat is caller's matrix
//       j is column number
//       rows is the number of rows in the matrix
//       cols is number of columns in the matrix
//       scalar is the value to multiply
// On exit:
//       elements in matrix column j are multiplied by scalar

    for(int i = 0;i < rows ;i++)
    {
        matx[(cols * i) + j] *= scalar;

    }
}
```

Since column-level operations are not as common in array processing as row operations, we have provided a single example, which is the ColMulScalar() function. The programmer should be able to use it to develop any other column operations that may be required.

Low-Level Vector-by-Scalar Operations

Array processing are computationally intensive operations. Coding them in high-level languages is convenient and easy, but sacrifices control, performance, and possibly precision. C++ programmers can use a more efficient approach by developing the fundamental processing functions in low-level code. A C++ stub function can provide easy access to these low-level primitives.

In the code that follows, the low-level procedure receives the address of the first matrix entry, as well as the row and column parameters required for the operation. For example, to perform a row-level operation the low-level routine must know the address of the matrix, the number of elements in each column and the number of the desired row. In addition, the low-level routine must have available the horizontal skip factor. Using this information code can visit each matrix entry and perform the required operation. The code is as follows:

```
;************************************************************
;            low-level procedures for vector arithmetic
;************************************************************
;
         .CODE
_ROW_TIMES_SCALAR   PROC    USES esi edi ebx ebp
; Procedure to multiply a matrix row vector by a scalar
; On entry:
;         ST(0) = scalar multiplier
;         ESI —> matrix containing the row vector
;         EAX = number of row vector (0 based)
;         ECX = number of columns in matrix
;         EDX = horizontal skip factor
; On exit:
;         entries of row vector multiplied by ST(0)

; Formula for offset of start of vector is
;              offset = [ ((i-1) * N * s) ]
; AL holds 0-based number of the desired row vector
; CL holds the number of entries per row (N)
; DL holds skip factor
        MOV     AH,0            ; Clear high-order byte
        MUL     CL              ; AX = AL * CL
; Second multiplication assumes that product will be less than
; 65535. This assumption is reasonable since the matrix space
; assigned is 400 s
        PUSH    DX              ; Save before multiply
        MOV     DH,0            ; Clear high-order byte
        MUL     DX              ; AX = AX * DL
        POP     DX              ; Restore DX
        ADD     ESI,EAX         ; Add offset to pointer
        MOV     DH,0            ; Clear high-order byte
; At this point:
;    ESI —> first entry in the matrix row
;    ST(0) holds scalar multiplier
;    ECX = number of entries in row
;    EDX = byte length of each matrix entry
ENTRIES:
        CALL    FETCH_ENTRY
        FMUL    ST,ST(1)        ; Multiply by ST(1)
        CALL    STORE_ENTRY
        ADD     ESI,EDX         ; Index to next entry
        LOOP    ENTRIES
        RET
_ROW_TIMES_SCALAR      ENDP
;***************************************************************

_ROW_PLUS_SCALAR   PROC    USES esi edi ebx ebp
; Procedure to add a scalar to a matrix row
; On entry:
;         ST(0) = scalar multiplier
;         ESI —> matrix containing the row vector
;         EAX = number of row vector (0 based)
;         ECX = number of columns in matrix
;         EDX = horizontal skip factor
; On exit:
```

```
;         entries of row vector multiplied by ST(0)
;
; Formula for offset of start of vector is
;              offset = [ ((i-1) * N * s) ]
; AL holds 0-based number of the desired row vector
; CL holds the number of entries per row (N)
; DL holds skip factor (8 for double precision)
        MOV     AH,0            ; Clear high-order byte
        MUL     CL              ; AX = AL * CL
; Second multiplication assumes that product will be less than
; 65535. This assumption is reasonable since the matrix space
; assigned is 400 s
        PUSH    DX              ; Save before multiply
        MOV     DH,0            ; Clear high-order byte
        MUL     DX              ; AX = AX * DL
        POP     DX              ; Restore DX
        ADD     ESI,EAX         ; Add offset to pointer
        MOV     DH,0            ; Clear high-order byte
; At this point:
;    ESI —> first entry in the matrix row
;    ST(0) holds scalar multiplier
;    ECX = number of entries in row
;    EDX = byte length of each matrix entry
ENTRIES_A:
        CALL    FETCH_ENTRY
        FADD    ST,ST(1)        ; Add scalar
        CALL    STORE_ENTRY
        ADD     ESI,EDX         ; Index to next entry
        LOOP    ENTRIES_A
        RET
_ROW_PLUS_SCALAR      ENDP
;****************************************************************
;
_ROW_DIV_SCALAR          PROC
; Procedure to divide a matrix row vector by a scalar
; On entry:
;         ST(0) = scalar divisor
;         ESI —> matrix containing the row vector
;         EAX = number of row vector (0 based)
;         ECX = number of columns in matrix
;         EDX = horizontal skip factor

; On exit:
;         Entries of row vector divided by ST(0)
;         ST(0) is preserved
; Algorithm:
;          Division is performed by obtaining the reciprocal of
;          the divisor and using the multiplication routine
;                       |    ST(0)     |    ST(1)    |  ST(2)
                        ;   divisor    |      ?      |    ?
        FLD     ST(0)   ;   divisor    |   divisor   |    ?
        FLD1            ;      1       |   divisor   |  divisor
        FDIV    ST,ST(1);  1/divisor   |      1      |  divisor
        FSTP    ST(1)   ;  1/divisor   |   divisor   |    ?
        CALL    _ROW_TIMES_SCALAR
        FSTP    ST(0)   ;   divisor    |      ?      |    ?
        CLD
        RET
_ROW_DIV_SCALAR          ENDP
;***************************************************************
```

```
;
_ROW_MINUS_SCALAR        PROC
; Procedure to subtract a scalar from the entries in a matrix
; row
; On entry:
;         ST(0) = scalar to subtract
;         ESI —> matrix containing the row vector
;         EAX = number of row vector (0 based)
;         ECX = number of columns in matrix
;         EDX = horizontal skip factor
; On exit:
;         Scalar subtracted from entries of the row vector
; Algorithm:
;         Subtraction is performed by changing the sign of the
;         subtrahend and using the addition routine
;                       |    ST(0)     |    ST(1)   |  ST(2)
;                       |      #       |     ?
        FCHS           ;     -#       |     ?
        CALL    _ROW_PLUS_SCALAR
        FCHS           ;      #       |     ?
        CLD
        RET
_ROW_MINUS_SCALAR        ENDP
```

Note that in the preceding routines scalar subtraction is performed by changing the sign of the scalar addend, while division is accomplished by multiplying by the reciprocal of the divisor. Also notice that sign inversion of a row vector can be obtained by using –1 as a scalar multiplier. The row operations procedures listed previously receive the horizontal skip factor in the EDX register. The core procedures _ROW_TIMES_SCALAR and _ROW_PLUS_SCALAR then call the auxiliary procedures FETCH_ENTRY and STORE_ENTRY to access and store the matrix entries. FETCH_ENTRY and STORE_ENTRY determine the type of data access required according to the value in the EDX register. If the value in EDX is 4, then the data is encoded in single precision format. If the value is 8 then the data is in double precision. If the value is 10, then the data is in extended precision. This mechanism allows creating low-level code that can be used with any of the three floating-point types in ANSI/IEEE 754. The C++ interface routines, which are coded as template functions, use the sizeof operator on a matrix entry to determine the data type passed by the caller.

Visual C++ Version 6, in Win32 operating systems, defines the size of int, long, unsigned long, and float data types as 4 bytes. Therefore it is not possible to use the size of a data variable to determine if an argument is of integer or float type. For this reason the interface routines listed in this section can only be used with float-type arguments. Attempting to pass integer matrices or scalars will result in undetected computational errors. The C++ interface functions to the low-level row-operation procedures are as follows:

```
//****************************************************************
//   C++ interface functions to vector arithmetic primitives
//****************************************************************
template <class A>
void RowTimesScalarLL(A *matx, int i, int cols, A scalar)
{
// Multiply a matrix row times a scalar using low-level code
// in the Un32_13 module
// On entry:
// On entry:
//       *mat is caller’s matrix (floating point type)
//       i is number of the row
//       cols is number of columns in the matrix
//       scalar is the value to add (floating point type)
// Routine expects:
//       ST(0) holds scalar
//       ESI —> matrix
//       EAX = row vector number
//       ECX = number of columns in matrix
//       EDX = horizontal skip factor
// On exit:
//       elements in matrix row i are multiplied by scalar
int eSize = sizeof(matx[0]);

_asm
{

         MOV     ECX,cols          // Columns to ECX
         MOV     EAX,i             // Row number to EAX
         MOV     ESI,matx          // Address to ESI
         FLD     scalar            // Scalar to ST(0)
         MOV     EDX,eSize         // Horizontal skip
         CALL    ROW_TIMES_SCALAR
}
return;
}

template <class A>
void RowPlusScalarLL(A *matx, int i, int cols, A scalar)
{
// Multiply a matrix row times a scalar using low-level code
// in the Un32_13 module
// On entry:
//       *mat is caller’s matrix (floating point type)
//       i is number of the row
//       cols is number of columns in the matrix
//       scalar is the value to add (floating point type)
// Routine expects:
//       ST(0) holds scalar
//       ESI —> matrix
//       EAX = row vector number
//       ECX = number of columns in matrix
//       EDX = horizontal skip factor
// On exit:
//       elements in matrix row i are multiplied by scalar
int eSize = sizeof(matx[0]);

    _asm
{

         MOV     ECX,cols          // Columns to ECX
         MOV     EAX,i             // Row number to EAX
         MOV     ESI,matx          // Address to ESI
         FLD     scalar            // Scalar to ST(0)
         MOV     EDX,eSize         // Horizontal skip
         CALL    ROW_PLUS_SCALAR
}
```

```
return;
}

template <class A>
void RowDivScalarLL(A *matx, int i, int cols, A scalar)
{
// Divide a matrix row by a scalar using low-level code
// in the Un32_13 module
// On entry:
//       *mat is caller’s matrix (float type)
//       i is number of the row
//       cols is number of columns in the matrix
//       scalar is the value to add (float type)
// Routine expects:
//       ST(0) holds scalar
//       ESI —> matrix
//       EAX = row vector number
//       ECX = number of columns in matrix
//       EDX = horizontal skip factor
// On exit:
//       elements in matrix row i are multiplied by scalar
int eSize = sizeof(matx[0]);
_asm
{

         MOV     ECX,cols          // Columns to ECX
         MOV     EAX,i             // Row number to EAX
         MOV     ESI,matx          // Address to ESI
         FLD     scalar            // Scalar to ST(0)
         MOV     EDX,eSize         // Horizontal skip
         CALL    ROW_DIV_SCALAR
}
return;
}

template <class A>
void RowMinusScalarLL(A *matx, int i, int cols, A scalar)
{
// Subtract a scalar from a matrix row using low-level code
// in the Un32_13 module
// On entry:
//       *mat is caller’s matrix (float type)
//       i is number of the row
//       cols is number of columns in the matrix
//       scalar is the value to add (float type)
// Routine expects:
//       ST(0) holds scalar
//       ESI —> matrix
//       EAX = row vector number
//       ECX = number of columns in matrix
//       EDX = horizontal skip factor
// On exit:
//       elements in matrix row i are multiplicd by scalar
int eSize = sizeof(matx[0]);
_asm
{

         MOV     ECX,cols          // Columns to ECX
         MOV     EAX,i             // Row number to EAX
         MOV     ESI,matx          // Address to ESI
         FLD     scalar            // Scalar to ST(0)
         MOV     EDX,eSize         // Horizontal skip
         CALL    ROW_MINUS_SCALAR
}
return;
}
```

Matrix-by-Scalar Operations

Often we need to perform scalar operations on all entries in a matrix. In graphics programming the more useful operations are scalar multiplication, division, addition, and subtraction, in that order. In this section we present code to perform these matrix-by-scalar multiplications. Here again, because matrix-by-scalar manipulations are computationally intensive, we develop the routines in low-level code and provide C++ interface functions to the assembly language procedures. The low-level code is as follows:

```
          .CODE
_MAT_TIMES_SCALAR   PROC    USES esi edi ebx ebp
; Procedure to multiply a matrix by a scalar
; On entry:
;         ST(0) = scalar multiplier
;         ESI —> matrix containing the row vector
;         EAX = number of rows
;         ECX = number of columns
;         EDX = horizontal skip factor
; On exit:
;         entries of matrix multiplied by ST(0)
; Total number of entries is M * N
        MOV     AH,0            ; Clear high-order byte
        MUL     CL              ; AX = AL * CL
        MOV        ECX,EAX         ; Make counter in CX
; At this point:
;    ESI —> first entry in the matrix
;    ST(0) holds scalar multiplier
;    ECX = number of entries in matrix
;    EDX = byte length of each matrix entry (4, 8, or 10 bytes)
MAT_MUL:
        CALL    FETCH_ENTRY
        FMUL    ST,ST(1)        ; Multiply by ST(1)
        CALL    STORE_ENTRY
        ADD     ESI,EDX         ; Index to next entry
        LOOP    MAT_MUL
        CLD
        RET
_MAT_TIMES_SCALAR   ENDP
The C++ interface function is named MatTimesScalarLL(). The code is as
follows:
template <class A>
void MatTimesScalarLL(A *matx, int rows, int cols, A scalar)
{
// Multiply a matrix times a scalar using low-level code
// in the Un32_13 module
// On entry:
//       *mat is caller’s matrix (type double)
//       rows is number of the rows in matrix
//       cols is number of columns in the matrix
//       scalar is the value to multiply (floating point type)
// Routine expects:
```

```
//       ST(0) holds scalar
//       ESI —> matrix
//       EAX = row vector number
//       ECX = number of columns in matrix
//       EDX = horizontal skip factor
// On exit:
//       elements in matrix are multiplied by scalar
int eSize = sizeof(matx[0]);

_asm
{

         MOV     ECX,cols          // Columns to ECX
         MOV     EAX,rows          // Rows to EAX
         MOV     ESI,matx          // Address to ESI
         FLD     scalar            // Scalar to ST(0)
         MOV     EDX,eSize         // Horizontal skip
         CALL    MAT_TIMES_SCALAR
}
return;
}
```

4.2.2 Matrix-by-Matrix Operations

Two matrix-by-matrix operations are defined in linear algebra: matrix addition and multiplication. Matrix addition is the process of adding the corresponding entries of two matrices. As you saw in Chapter 3, matrix addition is defined only if the matrices are of the same size. The addition process in the case C = A + B consists of locating each corresponding entry in matrices A and B and storing their sum in matrix C.

Matrix multiplication, on the other hand, is rather counter-intuitive. Instead of multiplying the corresponding elements of two matrices, matrix multiplication consists of multiplying each of the entries in a row of matrix A, by each of the corresponding entries in a column of matrix B, and adding these products to obtain an entry of matrix C. For example

The entries in the product matrix C are obtained as follows:

First row of matrix C

$$
\begin{aligned}
C_{11} &= (A_{11}*B_{11})+(A_{12}*B_{21})+(A_{13}*B_{31}) \\
C_{12} &= (A_{11}*B_{12})+(A_{12}*B_{22})+(A_{13}*B_{32}) \\
C_{13} &= (A_{11}*B_{13})+(A_{12}*B_{23})+(A_{13}*B_{33}) \\
C_{14} &= (A_{11}*B_{14})+(A_{12}*B_{24})+(A_{13}*B_{34})
\end{aligned}
$$

Second row of matrix C

$$
\begin{aligned}
C_{21} &= (A_{21}*B_{11})+(A_{22}*B_{21})+(A_{23}*B_{31}) \\
C_{22} &= (A_{21}*B_{12})+(A_{22}*B_{22})+(A_{23}*B_{32}) \\
C_{23} &= (A_{21}*B_{13})+(A_{22}*B_{23})+(A_{23}*B_{33}) \\
C_{24} &= (A_{21}*B_{14})+(A_{22}*B_{24})+(A_{23}*B_{34})
\end{aligned}
$$

Matrix multiplication requires a series of products, which are obtained using as factors the entries in the rows of the first matrix and the entries in the columns of the second matrix. Therefore, matrix multiplication is defined only if the number of colunms of the first matrix is equal to the number of rows in the second matrix. This relationship can be visualized as follows:

```
matrix A             matrix B
  R  C                 r   c
     |______ = ________|
```

where R, C represents the rows and columns of the first matrix, and r, c represents the rows and columns of the second matrix. By the same token, the product matrix (C) will have as many rows as the first matrix (A) and as many columns as the second matrix (B). In the previous example, since matrix A is a 2-by-3 matrix, and matrix B is a 3-by-4 matrix, matrix C will be a 2-by-4 matrix.

Since matrix addition and multiplication are computationally intensive operations we implement them in low-level code and provide C++ interface routines.

Matrix Addition

The following low-level procedure performs matrix addition. The procedure requires that both matrices be of the same dimension, that is, that they have the same number of columns and rows.

```
         .486
        .MODEL flat
        .DATA
;****************************************************|
;   Data for this matrix addition and multiplication |
;****************************************************|
ELEMENT_CNT     DW      0       ; Storage for total number of
                                ; entries in matrix C
;
MAT_A_ROWS      DB      0       ; Rows in matrix A
MAT_A_COLS      DB      0       ; Columns in matrix A
MAT_B_ROWS      DB      0       ; Rows in matrix B
MAT_B_COLS      DB      0       ; Columns in matrix B
MAT_C_ROWS      DB      0       ; Rows in matrix C
MAT_C_COLS      DB      0       ; Columns in matrix C
SKIP_FACTOR     DD      0       ; Element size
;
; Control variables for matrix multiplication
PROD_COUNT      DB      0       ; Number of product in each
                                ; multiplication iteration
WORK_PRODS      DB      0       ; Working count for number of
                                ; products
WORK_ROWS       DB      0       ; Number of rows in matrices A
                                ; and C
WORK_COLS       DB      0       ; Number of columns in matrices B
                                ; and C
        .CODE
;**********************************
;      matrix addition
;**********************************
_ADD_MATRICES    PROC     USES esi edi ebx ebp
; Procedure to add all the corresponding entries of two matrices
; of the same size, as follows:
;
;    A=                 B=              C=(A+B)
;    A11  A12  A13      B11  B12  B13   A11+B11  A12+B12  A13+B13
;    A21  A22  A23      B21  B22  B23   ....
;    A31  A32  A33      B31  B32  B33                     A33+B33
;
; On entry:
;      ESI -> first matrix (A)
;      EDI -> second matrix (B)
;      EBX -> storage area for addition matrix (C)
;              Code assumes that matrix C is correctly
;              dimensioned
;      EAX = number of rows in matrix
;      ECX = number of columns in matrix
;      EDX = horizontal skip factor
;
; On exit:
;      AX = 0 if matrices are the same size, then matrix C
;      contains sum of A + B
;
;      AX = 1 if matrices are of different size and the matrix
;      sum is undefined
;
; Note: matrix addition is defined only regarding two matrices of
;       the same size. Matrices must be of type float and of the
;       same format
;
;***************************|
;   test for equal size     |
;***************************|
        CMP     AX,CX           ; Test for matrices of equal size
        JE      GOOD_SIZE       ; Go if same size
;***************************|
;       DATA ERROR          |
;***************************|
; At this point matrices cannot be added
        MOV     AX,1            ; Error code
         CLD
        RET
;***************************|
;  store matrix parameters |
;***************************|
; Calculate number of entries by multiplying matrix rows times
; matrix columns
GOOD_SIZE:
       PUSH    EDX                  ; Save register
       MUL     CX                   ; Rows times columns
       MOV     ELEMENT_CNT,AX       ; Store number of entries
        POP     EDX
; At this point:
;      ESI -> first matrix (A)
;      EDI -> second matrix (B)
;      EBX -> storage area for addition matrix (A+B)
;***************************|
;  perform matrix addition  |
;***************************|
A_PLUS_B:
; ESI -> matrix entry in matrix A
; EDX = entry size (4, 8, or 10 bytes)
        CALL    FETCH_ENTRY     ; ST(0) now holds entry of A
; Fetch entry in matrix B
        XCHG    ESI,EDI         ; ESI -> matrix B entry
        CALL    FETCH_ENTRY     ; ST(0) = matrix B entry
```

```
                                ; ST(1) = matrix A entry
        XCHG    ESI,EDI         ; Reset pointer
; Add entries
        FADD                    ; ST(0)    |  ST(1)  |  ST(2)
                                ; eA + eB | ------- |
        XCHG    EBX,ESI         ; ESI —> matrix C entry
; Store sum
        CALL    STORE_ENTRY     ; Store sum in matrix C and pop
                                ; stack
        XCHG    EBX,ESI         ; Restore pointers
; Update entries counter
        DEC     ELEMENT_CNT     ; Counter for matrix s
        JNZ     NEXT_MAT_ELE    ; Continue if not end of matrix
;***************************|
;    end of matrix addition |
;***************************|
        MOV     AX,0            ; No error flag
          CLD
        RET
;***************************|
;   index matrix pointers   |
;***************************|
; Add entry size to each matrix pointer
NEXT_MAT_ELE:
        ADD     ESI,EDX          ; Add size to pointer
        ADD     EDI,EDX
        ADD     EBX,EDX
        JMP     A_PLUS_B
;
_ADD_MATRICES    ENDP
```

The C++ interface function to the _ADD_MATRICES procedure is as follows:

```
template <class A>
void AddMatrices(A *matA, A *matB, A *matC, int rows, int cols)
{
// Perform matrix addition: C = A + B using low-level code in the
// Un32_13 module
// On entry:
//       *matA and *matB are matrices to be added
//       *matC is matrix for sums
//       rows is number of the rows in matrices
//       cols is number of columns in the matrices
// Requires:
//       All three matrices must be of the same dimensions
//       All three matrices must be of the same floating
//       point data type
// Routine expects:
//      ESI —> first matrix (A)
//      EDI —> second matrix (B)
//      EBX —> storage area for addition matrix (C)
//      EAX = number of rows in matrices
//      ECX = number of columns in matrices
//      EDX = horizontal skip factor
// On exit:
//      returns matC[] = matA[] + matB[]

int eSize = sizeof(matA[0]);

_asm
{

         MOV     ECX,cols          // Columns to ECX
         MOV     EAX,rows          // Rows to EAX
         MOV     ESI,matA          // Address to ESI
         MOV     EDI,matB
         MOV     EBX,matC
         MOV     EDX,eSize         // Horizontal skip
         CALL    ADD_MATRICES
}
return;
}
```

Matrix Multiplication

The following low-level procedure performs matrix multiplication. The procedure requires that the number of columns in the first matrix be the same as the number of rows in the second matrix. The matrix for results must be capable of storing a number of elements equal to the product of the number of rows of the first matrix by the number of columns of the second matrix. The data variables for the _MUL_MATRICES procedure were defined in the _ADD_MATRICES procedure, listed previously.

```
          .CODE
;****************************************************************
;                   matrix multiplication
;****************************************************************
_MUL_MATRICES    PROC    USES esi edi ebx ebp
; Procedure to multiply two matrices (A and B) for which a matrix
; product (A * B) is defined. Matrix multiplication requires that
; the number of columns in matrix A be equal to the number of
; rows in matrix B, as follows:
;
;            A                   B
;          R   C               r   c
;              |______ = ______|
;
; Example:
;
;    A=(2 by 3)          B=(3 by 4)
;    A11  A12  A13       B11  B12  B13  B14
;    A21  A22  A23       B21  B22  B23  B24
;                        B31  B32  B33  B34
;
; The product matrix (C) will have 2 rows and 4 columns
;   C=(2 by 4)
;   C11   C12   C13  C14
;   C21   C22   C23  C24
;
; In this case the product matrix is obtained as follows:
;   C11 = (A11*B11)+(A12*B21)+(A13*B31)
;   C12 = (A11*B12)+(A12*B22)+(A13*B32)
;   C13 = (A11*B13)+(A12*B23)+(A13*B33)
;   C14 = (A11*B14)+(A12*B24)+(A13*B34)
;
;   C21 = (A21*B11)+(A22*B21)+(A23*B31)
;   C22 = (A21*B12)+(A22*B22)+(A23*B32)
;   C23 = (A21*B13)+(A22*B23)+(A23*B33)
;   C24 = (A21*B14)+(A22*B24)+(A23*B34)
;
```

```
; On entry:
;      ESI –> first matrix (A)
;      EDI –> second matrix (B)
;      EBX –> storage area for products matrix (C)
;      AH = rows in matrix A
;      AL = columns in matrix A
;      CH = rows in matrix B
;      CL = columns in matrix B
;      EDX = number of bytes per entry
; Assumes:
;      Matrix C is dimensioned as follows:
;      Columns of C = columns of B
;      Rows of C = rows of A
; On exit:
;      Matrix C is the products matrix

; Note: the entries of matrices A, B, and C must be of type float
;       and of the same data format
;
; Store number of product in each multiplication iteration
        MOV     PROD_COUNT,AL
; At this point:
;      AH = rows in matrix A
;      AL = columns in matrix A
;      CH = rows in matrix B
;      CL = columns in matrix B
; Store matrix dimensions
       MOV      MAT_A_ROWS,AH
       MOV      MAT_A_COLS,AL
       MOV      MAT_B_ROWS,CH
       MOV      MAT_B_COLS,CL
; Store skip factor
       MOV      SKIP_FACTOR,EDX
; Calculate total entries in matrix C
; Columns in C = columns in B
;    Rows in C = rows in A
        MOV     MAT_C_COLS,CL
        MOV     MAT_C_ROWS,AH
; Calculate number of products
        MOV     AH,0            ; Clear high byte of product
        MUL     CL              ; Rows times columns
        MOV     ELEMENT_CNT,AX  ; Store  count
; At this point:
;      ESI –> first matrix (A)
;      EDI –> second matrix (B)
;      EBX –> storage area for products matrix (A*B)
       MOV     START_BMAT,EDI        ; Storage for pointer
;***************************|
; initialize row and column |
;         counters          |
;***************************|
; Set up work counter for number of rows in matrix C
; This counter will be used in determining the end of the
; matrix multiplication operation
        MOV     AL,MAT_C_ROWS          ; Rows in matrix C
        MOV     WORK_ROWS,AL           ; To working counter
; Reset counter for number of columns in matrix C
; This counter will be used in resetting the matrix pointers at
; the end of each row in the products matrix
        MOV     AL,MAT_C_COLS          ; Columns in matrix C
        MOV     WORK_COLS,AL            ; To working counter
;**************************|
;  perform multiplication  |
;**************************|
NEW_PRODUCT:
; Save pointers to matrices A and B
        PUSH    ESI             ; Pointer to A
        PUSH    EDI             ; Pointer to B
; Load 0 as first entry in sum of products
        FLDZ
                                ; ST(0)   |  ST(1)  |  ST(2)
;                               |    0    |    ?    |    ?   |
; Store number of products in work counter
        MOV     AL,PROD_COUNT           ; Get count
        MOV     WORK_PRODS,AL           ; Store in work counter
A_TIMES_B:
; Fetch entry in current row of matrix A
        MOV     EDX,SKIP_FACTOR        ;  size to DL
; ESI -> matrix entry in current row of matrix A
        CALL    FETCH_ENTRY     ; ST(0) now holds entry of A
        XCHG    ESI,EDI         ; ESI -> matrix B
        CALL    FETCH_ENTRY     ; ST(0) = matrix B
                                ; ST(1) = matrix A
        XCHG    ESI,EDI         ; Reset pointer
; Multiply s
                                ; ST(0)   |  ST(1)  |  ST(2)
        FMULP   ST(1),ST        ; eA * eB |previous | ------ |
                                ;         |  sum    | ------ |
        FADD                    ; p sum   | ------  |
; Test for last entry in product column
        DEC     WORK_PRODS      ; Is this last product
        JZ      NEXT_PRODUCT    ; Go if at end of products column
;**************************|
;      next product        |
;**************************|
; Index to next column of matrix A
        ADD     ESI,SKIP_FACTOR   ; Add size to pointer
; Index to next row in the same column in matrix B
        MOV     EAX,EDX           ; Horizontal skip factor to AL
         MUL     MAT_B_COLS        ; Times number of columns
        ADD     EDI,EAX           ; Add to pointer
        JMP     A_TIMES_B         ; Continue in same product column
;**************************|
;      store product       |
;**************************|
NEXT_PRODUCT:
; Restore pointers to start of current A row and B column
        POP     EDI              ; B matrix pointer
        POP     ESI              ; A matrix pointer
; At this point ST(0) has sum of products
; Store this sum as entry in products matrix (by DS:BX)
        XCHG    EBX,ESI          ; ESI -> matrix C
; Store sum
        MOV     EDX,SKIP_FACTOR  ; size to DL
        CALL    STORE_ENTRY     ; Store sum in matrix C and pop
                                ; stack
        XCHG    EBX,ESI         ; Restore pointers
; Index to next entry in matrix C
        ADD     EBX,SKIP_FACTOR      ; Add size to pointer
;**************************|
;  test for last column in  |
;        matrix C           |
;***************************|
; WORK_COLS keeps count of current column in matrix C
        DEC     WORK_COLS       ; Is this the last column in C
        JE      NEW_C_ROW       ; Go if last row
; Index to next column in matrix B
        ADD     EDI,SKIP_FACTOR        ; Add size to pointer
        JMP     NEW_PRODUCT
;***************************|
;     index to new row      |
;***************************|
; First test for end of processing
NEW_C_ROW:
        DEC     WORK_ROWS       ; Row counter in matrix C
        JNE     NEXT_C_ROW      ; Go if not last row of C
;***************************|
;       end of matrix       |
;       multiplication      |
;***************************|
        JMP     MULT_M_EXIT

;***************************|
;  next row of matrix C     |
;***************************|
; At the start of every new row in the products matrix, the
; matrix B pointer must be reset to the start of matrix B
; and the matrix A pointer to the start entry of the next
; row of matrix A
NEXT_C_ROW:
        MOV     EDI,START_BMAT  ; EDI —> start of B
        MOV     AH,0            ; Clear high byte of adder
; Pointer for matrix A
        MOV     EAX,SKIP_FACTOR    ; Entry size of A
        MUL     MAT_A_COLS         ; Size times columns
        ADD     ESI,EAX            ; ESI —> next row of A
; Reset counter for number of columns in matrix C
        MOV     AL,MAT_C_COLS      ; Columns in matrix C
        MOV     WORK_COLS,AL       ; To working counter
        JMP     NEW_PRODUCT        ; Continue processing
;**********************|
;         EXIT         |
;**********************|
MULT_M_EXIT:
        CLD
        RET

_MUL_MATRICES    ENDP
```

The C++ interface function to the _MUL_MATRICES procedure is as follows:

```
template <class A>
bool MulMatrices(A *matA, A *matB, A *matC,
                 int rowsA, int colsA,
                 int rowsB, int colsB)
{
// Perform matrix addition: C = A + B using low-level code in the
// Un32_13 module
// On entry:
//       *matA and *matB are matrices to be added
//       *matC is matrix for sums
//       rowsA is number of the rows in matrix A
//       colsA is number of columns in the matrix A
//       rowsB is number of the rows in matrix B
//       colsB is number of columns in the matrix B
// Requires:
//       All three matrices must be of the same dimensions
//       All three matrices must be of the same float
//       data type
// Asumes:
//       Matrix C dimensions are the product of the
//       columns of matrix B times the rows or matrix A
// Routine expects:
//      ESI —> first matrix (A)
//      EDI —> second matrix (B)
//      EBX —> storage area for addition matrix (C)
//      AH = number of rows in matrix A
//      AL = number of columns in matrix A
//      CH = number of rows in matrix B
//      CL = number of columns in matrix B
//      EDX = horizontal skip factor
// On exit:
//      returns true if matC[] = matA[] * matB[]
//      returns false if columns of matA[] not = rows
//      of matB[]. If so, matC[] is undefined

int eSize = sizeof(matA[0]);

// Test for valid matrix sizes:
//     columns of matA[] = rows of matB[]
if(colsA != rowsB)
   return false;

_asm
{
         MOV     AH,BYTE PTR rowsA
         MOV     AL,BYTE PTR colsA
         MOV     CH,BYTE PTR rowsB
         MOV     CL,BYTE PTR colsB
         MOV     ESI,matA          // Address to registers
         MOV     EDI,matB
         MOV     EBX,matC
         MOV     EDX,eSize         // Horizontal skip
         CALL    MUL_MATRICES
}
return true;
}
```

Chapter 5

Projections and Rendering

Topics:

- Perspective
- Projections
- The rendering pipeline

In order to view manipulate and view a graphics object we must find ways of storing it a computer-compatible way. In order to store an image, we must find a ways of defining and digitizing it. Considering that the state-of-the-art in computer displays is two-dimensional, the solid image must also be transformed so that it is rendered on a flat surface. The task can be broken down into three separate chores: representing, encoding, and rendering. Representing and encoding graphics images were discussed in previous chapters. Here we are concerned with rendering.

5.1 Perspective

The computer screen is a flat surface. When image data is stored in the computer it is in the form of a data structure consisting of coordinate points. You have seen in Chapters 3 and 4 how a matrix containing these image coordinate points can be translated, scaled, and rotated by means of geometrical transformations. But a data structure of image points cannot be displayed directly onto a flat computer screen. In the same way that an engineer must use a rendering scheme in order to represent a solid object onto the surface of the drawing paper, the programmer must find a way of converting a data structure of coordinates into an image on the computer monitor. You can say that both, the engineer and the programmer, have a rendering problem. Various approaches to rendering give rise to several types of projections. Figure 5-1, on the following page, shows the more common type of projections.

5.1.1 Projective Geometry

Projective geometry is the field of mathematics that studies the transformations of objects during projections. The following imaginary elements participate in every projection:

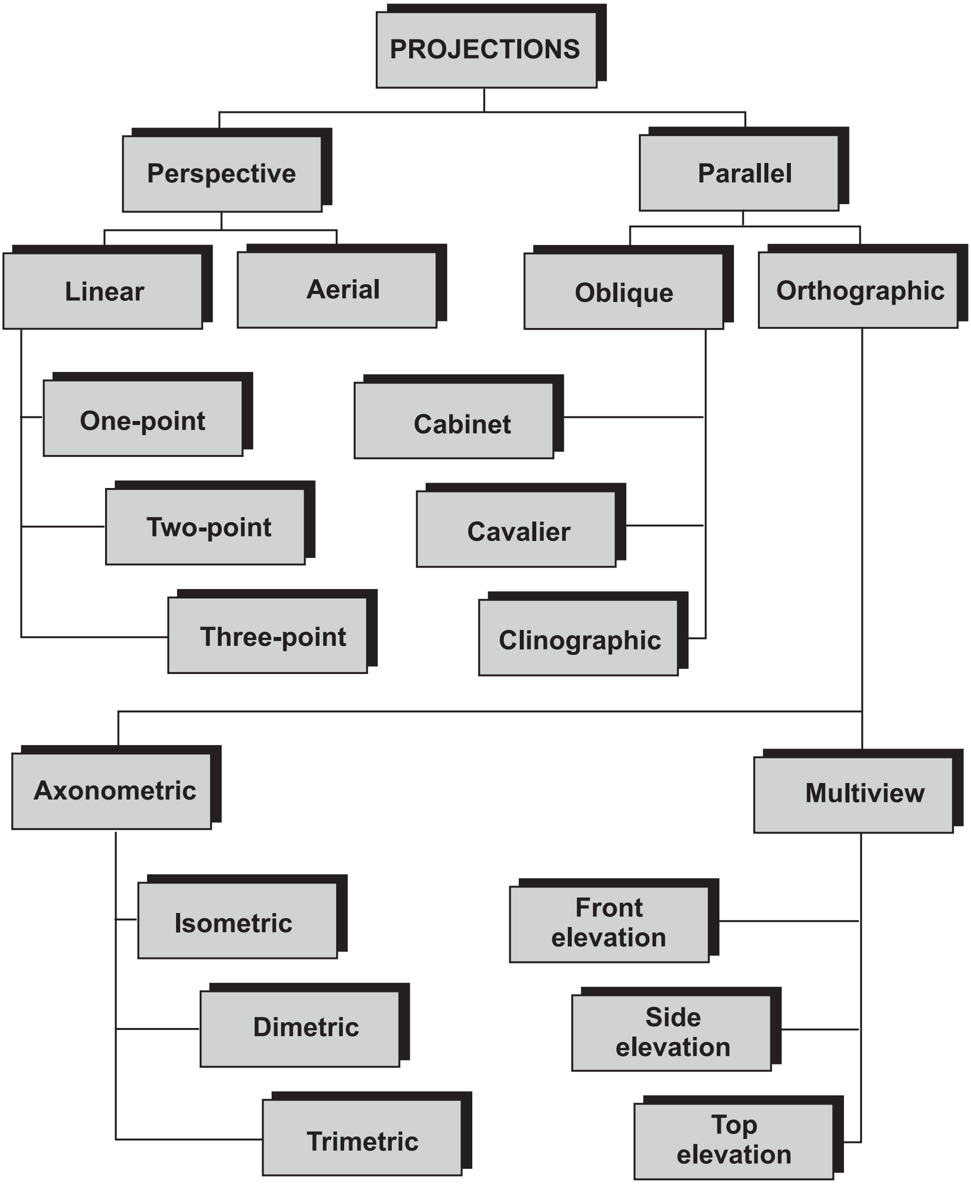


Figure 5-1 *Common Projections*

1. The observer's eye, also called the view point or center of projection.
2. The object being viewed.
3. The plane or planes of projection.
4. The visual rays that determine the line of sight, called the projectors.

Figure 5-2 shows these elements.

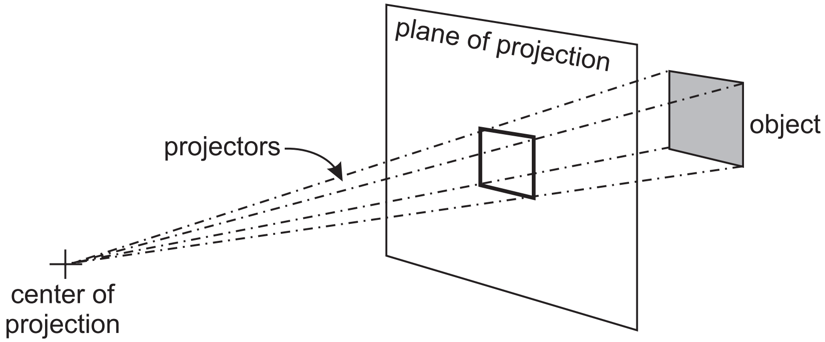


Figure 5-2 *Projection Elements*

Geometrically, the projection of a point on a plane is the point of intersection, on the plane of projection, of a line that extends from the object's point to the center of projection. This line is called the projector. Alternatively you can say that the projection of a point is the intersection between the point's projector and the plane of projection. The definition can be further refined by requiring that the center of projection not be located in the object nor in the plane of projection. This constraint makes this type of projection a central projection.

The location of the center of projection in relation to the object and the plane of projection determines the two main types of projections. When the center of projection is at a measurable distance from the plane of projection it is called a perspective projection. When the center of projection is located at infinity, the projection is called a parallel projection. Figure 5-3 shows perspective and parallel projections.

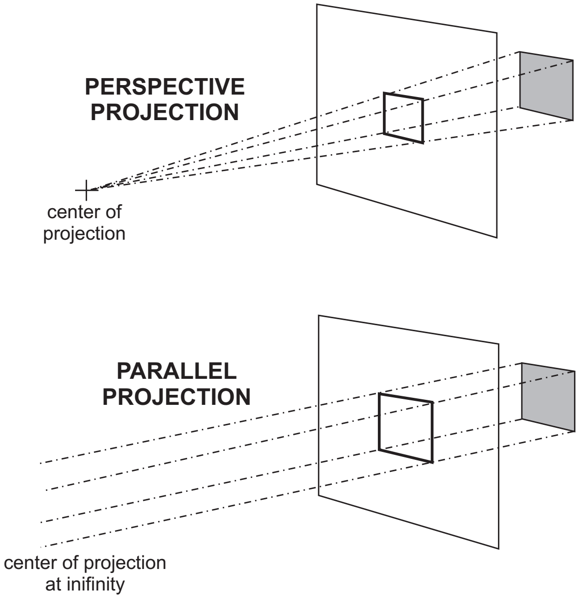


Figure 5-3 *Perspective and Parallel Projections*

In central projections the geometrical elements in the object plane are transformed into similar ones in the plane of projection. A line is projected as a line, a triangle as a triangle, and a polygon as a polygon. However, other object properties may not be preserved. For example, the length of line segments, the angular values, and the congruence of polygons can be different in the object and the projected image. Furthermore, geometrical elements that are conic sections (circle, ellipse, parabola, and hyperbola) retain the conic section property, but not necessarily their type. A circle can be projected as an ellipse, an ellipse as a parabola, and so on. Figure 5-4 shows the perspective projection of a circle as a ellipse.

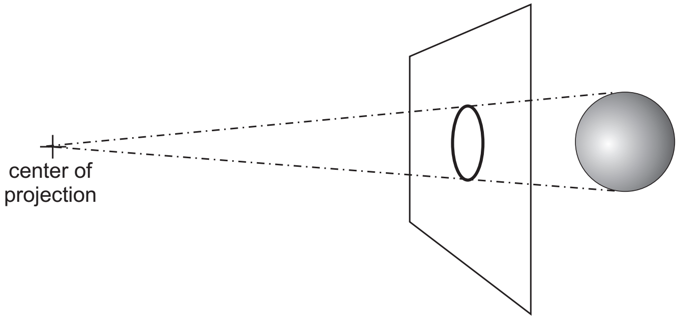


Figure 5-4 *A Circle Projected as an Ellipse*

5.1.2 Parallel Projections

Parallel projections find extensive use in drafting, engineering drawings, and architecture. They are divided into two types: oblique and orthographic. The orthographic or right-angle projection, which is the simplest of all, assumes that the planes or projection coincide with the coordinates axis. In this case the projectors are normal (perpendicular) to the plane of projection. In the oblique projection the projectors are not normal to the plane of projection.

A type of parallel projection, called a multiview projection, is often used in technical drawings. The images that result from a multiview projection are planar and true-to-scale. Therefore, the engineer or draft person can take measurements directly from a multiview projection. Figure 5-5 shows a multiview projection of an engineered object.

In Figure 5-5 the front, side, and top views are called the regular views. There are three additional views not shown in the illustration, called the bottom, right-side, and rear views. These are drawn whenever it is necessary to show details not visible in the regular views. The Cartesian interpretation of the front view is the orthographic projection of the object onto the xy-plane, the side view is the projection onto the yz-plane, and the top view is the projection onto the xz-plane. Sometimes these views are called the front-elevation, side-elevation, and top- or plan-elevation. While each multiview projection shows a single side of the object, it is often convenient to show the object pictorially. The drawing on the left-side of Figure 5-5 shows several sides of the object in a single view, thus rendering a pictorial view of the object.

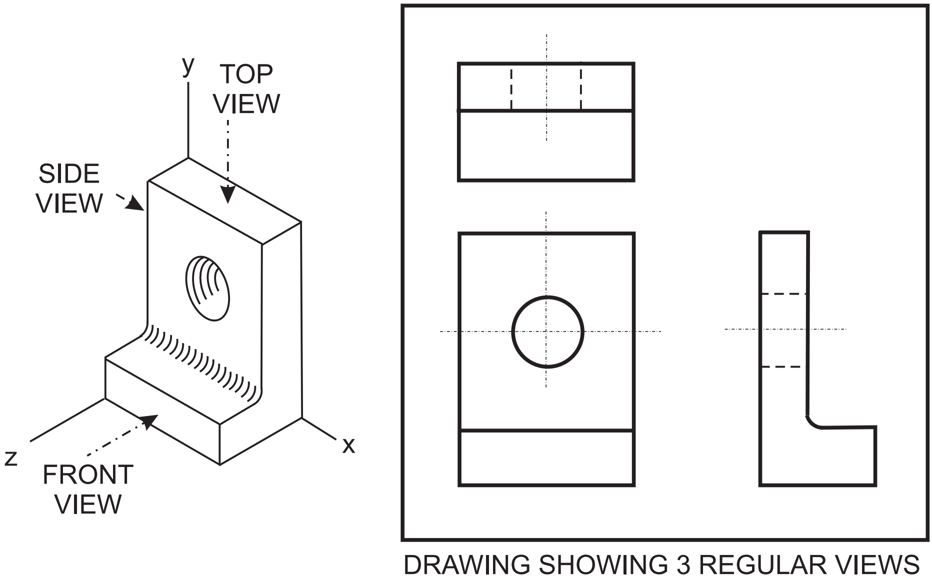


Figure 5-5 *Parallel, Orthographic, Multiview Projection*

Orthographic-axonometric projections are pictorial projections often used in technical applications. The term axonometric originates in the greek word "axon" (axis) and "metrik" (measurement). It relates to the measurements of the axes used in the projection. Notice in Figure 5-1 that the axonometric projections are further classified into isometric, dimetric, and trimetric. Isometric means "equal measure," which means that the object axes make equal angles with the plane of projection. In the dimetric projection two of the three object axes make equal angles with the plane of projection. In the trimetric, all three axes angles are different. Figure 5-6 shows the isometric, dimetric, and trimetric projections of a cube.

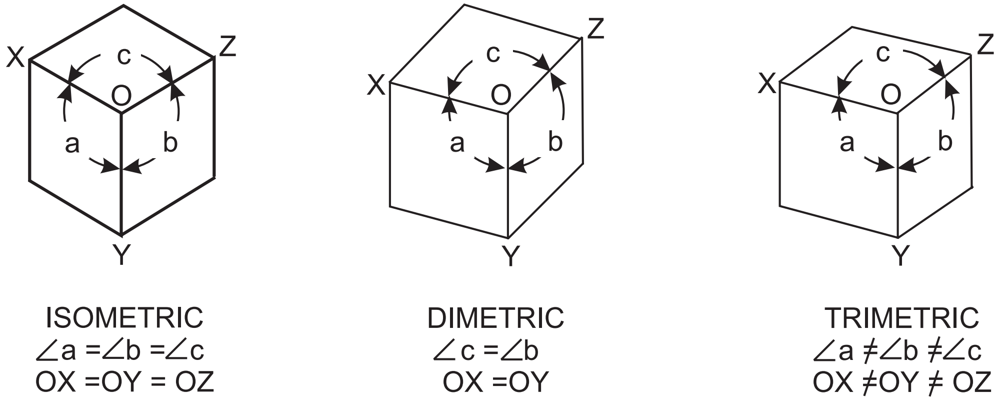


Figure 5-6 *Isometric, Dimetric, and Trimetric Projections*

5.1.3 Perspective Projections

Orthographic projections have features that make them useful in technical applications. For example, multiview projections provide dimensional information to the technician, engineer, and the architect. Axonometric projections, shown in Figure 5-6, can be mechanically generated from multiview drawings. In general, the main feature of the parallel projections is their information value.

One objection to the parallel projections is their lack of realism. Figure 5-7 shows two isometric cubes, labeled A and B, at different distances from the observer. However, both objects have projected images of the same size. This is not a realistic representation since cube B, farther away from the observer, should appear smaller than cube A.

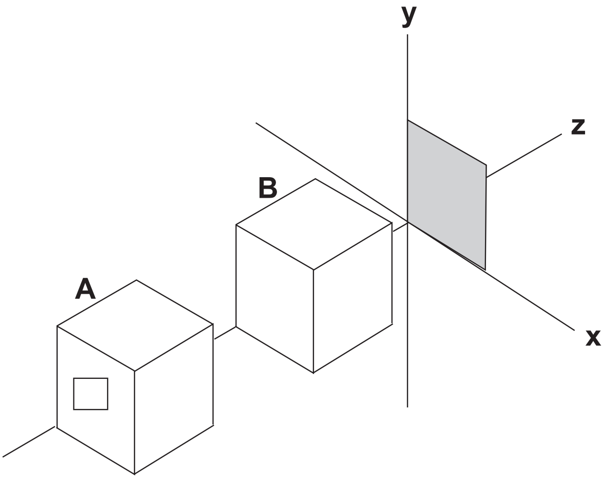


Figure 5-7 *Lack of Realism In Isometric Projection*

Perspective projection attempts to improve the realism of the image by providing depth cues that enhance relative positions, distances, and diminishing size. One of the most important depth cues is the relative size of the object at different distances from the viewing point. This effect can be achieved by means of perspective projections. The perspective projection depends on a vanishing point that is used to determine the object's relative size. Three types of perspective projections are in use, according to the number of vanishing points. They are named one-point, two-point, and three-point perspectives.

The number of vanishing points is determined by the positioning of the object in relation to the plane of projection. If a cube is placed so its front face is parallel to the plane of projection, then one set of edges converges to a single vanishing point. If the same cube is positioned so that one set of parallel edges is vertical, and the other two are not, then each of the two non-vertical edges has a vanishing point. Finally, if the cube is placed so that none of its principal edges are parallel to the plane of projection, then there are three vanishing points.

Perspective projections have some unique characteristics. In a parallel projection we take a three-dimensional object and produce a two-dimensional image. In a perspective projection we start with a three-dimensional object and produce another three-dimensional object which is modified in order to enhance its depth cues. This makes this type of projection a transformation, much like the rotation, translation, and scaling transformations discussed in Chapter 3. However, unlike rotation, translation, and scaling, a perspective transformation distorts the shape of the object transformed. After a perspective transformation, forms that were originally circles may turn into ellipses, parallelograms into trapezoids, and so forth. It is these distortions that reinforce our depth perception.

One-Point Perspective

The simplest perspective projection is based on a single vanishing point. In this projection, also called single-point perspective, the object is placed so that one of its surfaces is parallel to the plane of projection. Figure 5-8 shows a one-point perspective of a cube.

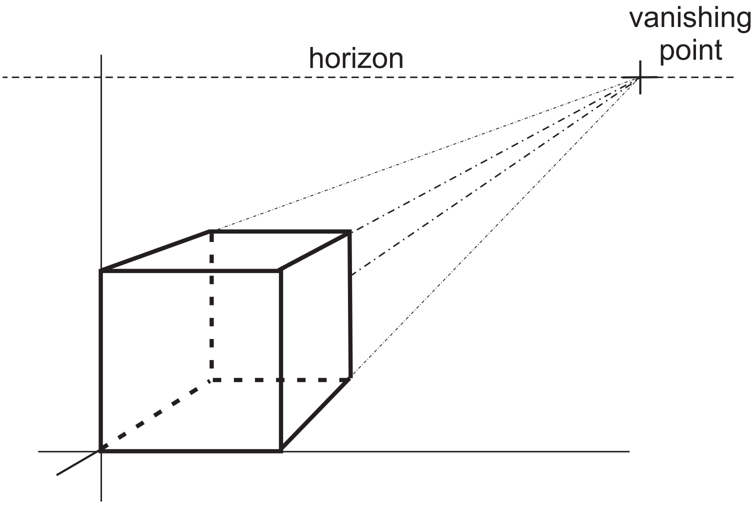


Figure 5-8 *One-Point Perspective Projection of a Cube*

One point perspective projections are simple to produce and find many practical uses in engineering, architecture, and in computer graphics. One of the features of the one-point perspective is that if an object has cylindrical or circular forms, and these are placed parallel to the plane of projection, then the forms are represented as circles or circular arcs in the perspective. This can be an advantage, considering that circles and circular arcs are easier to draw than ellipses or other conics. Figure 5-9, on the following page, is a one-point projection of a mechanical part that contains cylindrical and circular forms.

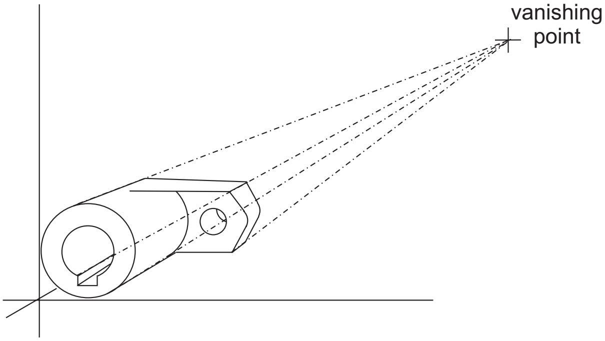


Figure 5-9 *One-Point Projection of a Mechanical Component*

A special form of the one-point perspective projection takes place when the vanishing point is placed centrally within the figure. This type of projection is called a tunnel perspective or tunnel projection. Because of the particular positioning of the object in the coordinate axes, the depth cues in a tunnel projection are not very obvious. Figure 5-10 shows the tunnel projection of a cube.

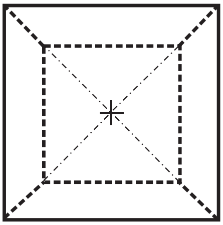


Figure 5-10 *Tunnel Projection of a Cube*

Two-Point Perspective

The depth cues in a linear perspective of a multi-faced object can be improved by rotating the object so that two of its surfaces have vanishing points. In the case of a cube this is achieved if the object is rotated along its y-axis, so that lines along that axis remain parallel to the viewing plane, but those along the two other axes have vanishing points. Figure 5-11 shows a two-point perspective of a cube.

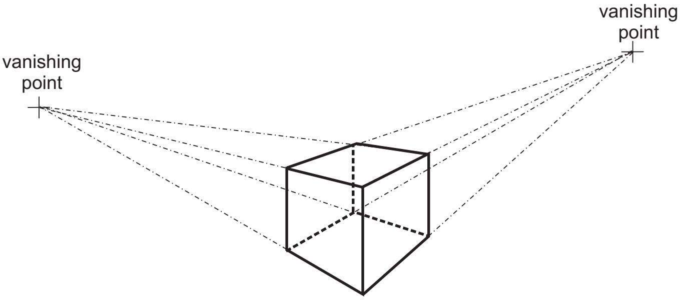


Figure 5-11 *Two-Point Perspective of a Cube*

Two-point perspective projections are realistic and easy to render. For these reasons they are frequently used in 3D graphics.

Three-Point Perspective

A three-point perspective is achieved by positioning the object so that none of its axes are parallel to the plane of projection. Although the visual depth cues in a three-point perspective are stronger than in the two-point perspective, the resulting geometrical deformations are sometimes disturbing to the viewer. Figure 5-12 is a three-point perspective projection of a cube.

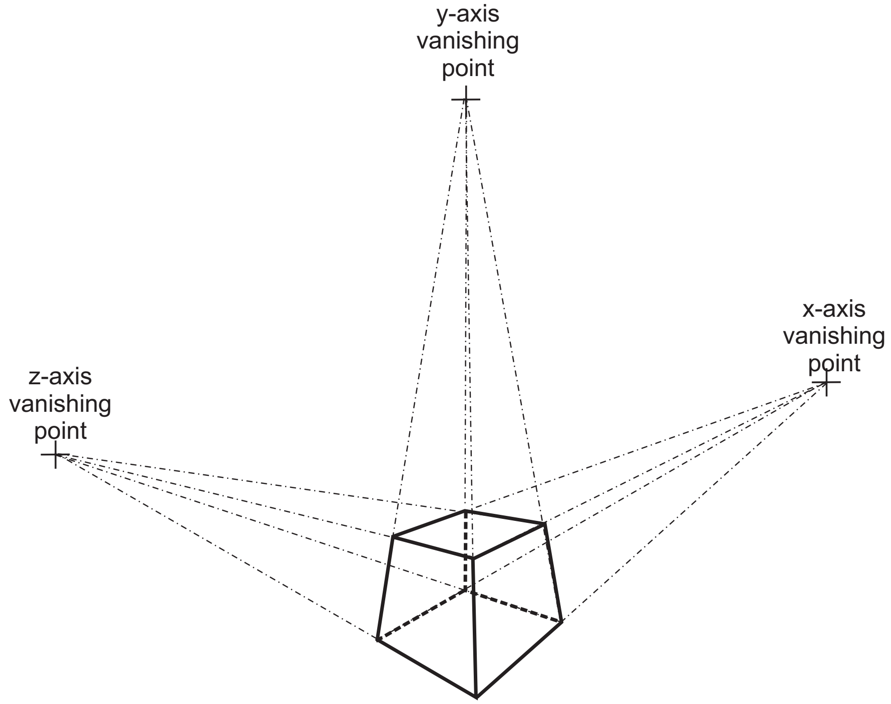


Figure 5-12 *Three-Point Perspective of a Cube*

The Perspective Projection as a Transformation

The data structure that defines the vertices of a three-dimensional object can be changed into another one that contains enhanced depth cues by performing a mathematical transformation. In other words, a perspective projection can be accomplished by means of a transformation. In calculating the projection transformation it is convenient to define a 4-by- 4 matrix so the transformation is compatible with the ones used for rotation, translation, and scaling, as described in Chapter 3. In this manner we can use matrix concatenation to create matrices that simultaneously perform one or more geometrical transformations, as well as a perspective projection.

The simplest approach for deriving the matrix for a perspective projection is to assume that the projection plane is normal to the z-axis and located at z = d. Figure 5-13 shows the variables for this case.

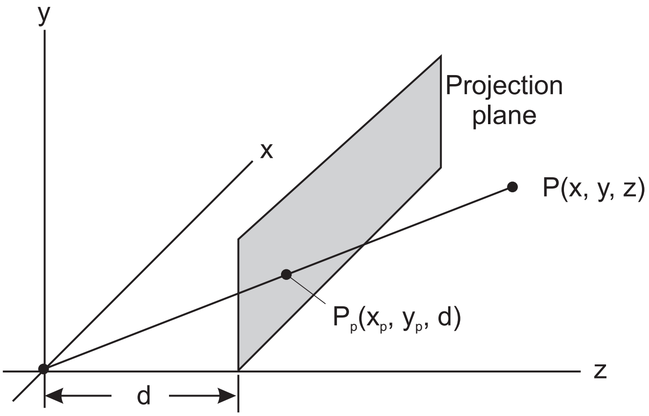


Figure 5-13 *Perspective Projection of Point P*

In Figure 5-13 point P_p represents the perspective projection of point P. According to the predefined constraints for this projection, we already know that the z coordinate of point P_p is d. To determine the formulas for calculating the x and y coordinates we can take views along either axes, and solve the resulting triangles, as shown in Figure 5-14.

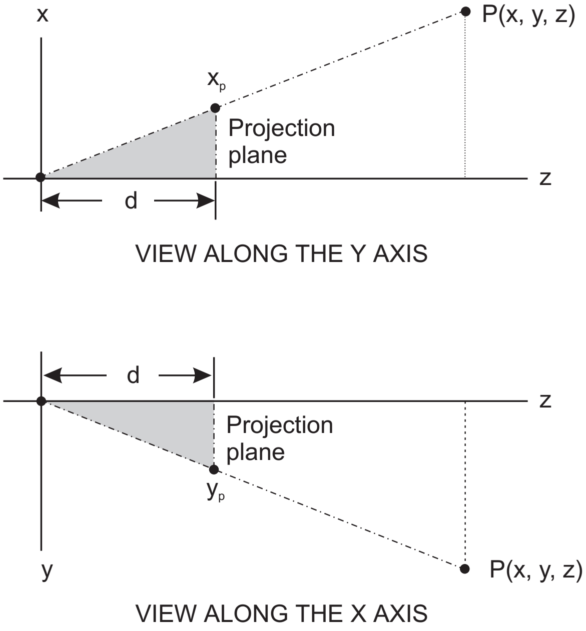


Figure 5-14 *Calculating x and y Coordinates of Point P*

Since the gray triangles in Figure 5-14 are similar, we can establish the ratios:

$$\frac{x_p}{d} = \frac{x}{z}$$

and

$$\frac{y_p}{d} = \frac{y}{z}$$

Solving for x_p and y_p produces the equations:

$$x_p = \frac{x}{z/d}, y_p = \frac{y}{z/d}$$

Since the distance d is a scaling factor in both equations, the division by z has the effect of reducing the size of more distant objects. In this case the value of z can be positive or negative, but not zero, since z = 0 defines a parallel projection. These equations can be expressed in matrix form, as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

5.2 The Rendering Pipeline

A common interpretation of the rendering process is to consider it as a series of transformations that take the object from the coordinate system in which it is encoded, into the coordinate system of the display surface. This process, sometimes referred to as the rendering pipeline, is described as a series of spaces through which the object migrates in its route from database to screen. A waterfall model of the rendering pipeline is shown in Figure 5-15.

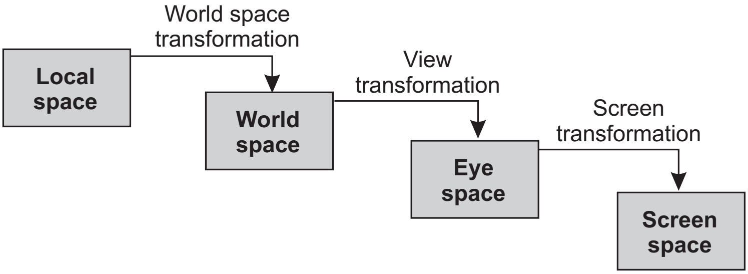
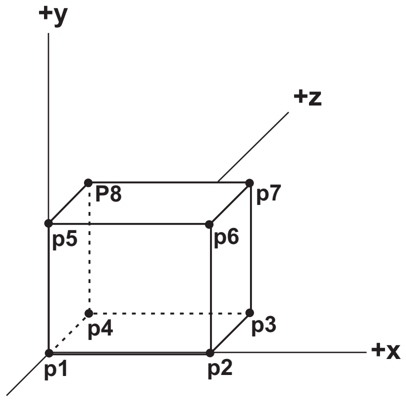


Figure 5-15 *Waterfall Model of the Rendering Pipeline*

5.2.1 Local Space

Objects are usually easier to model if they are conveniently positioned in the coordinate plane. For example, when we place the bottom-left vertex of a cube at the origin of the coordinate system, the coordinates are all positive values, as in Figure 5-16.



Local space coordinates	x	y	z
p1..	0	0	0
p2..	5	0	0
p3..	5	0	3
p4..	0	0	3
p5..	0	4	0
p6..	5	4	0
p7..	5	4	3
p8..	0	4	3

Figure 5-16 *Local Space Coordinates of a Cube with Vertex at the Origin*

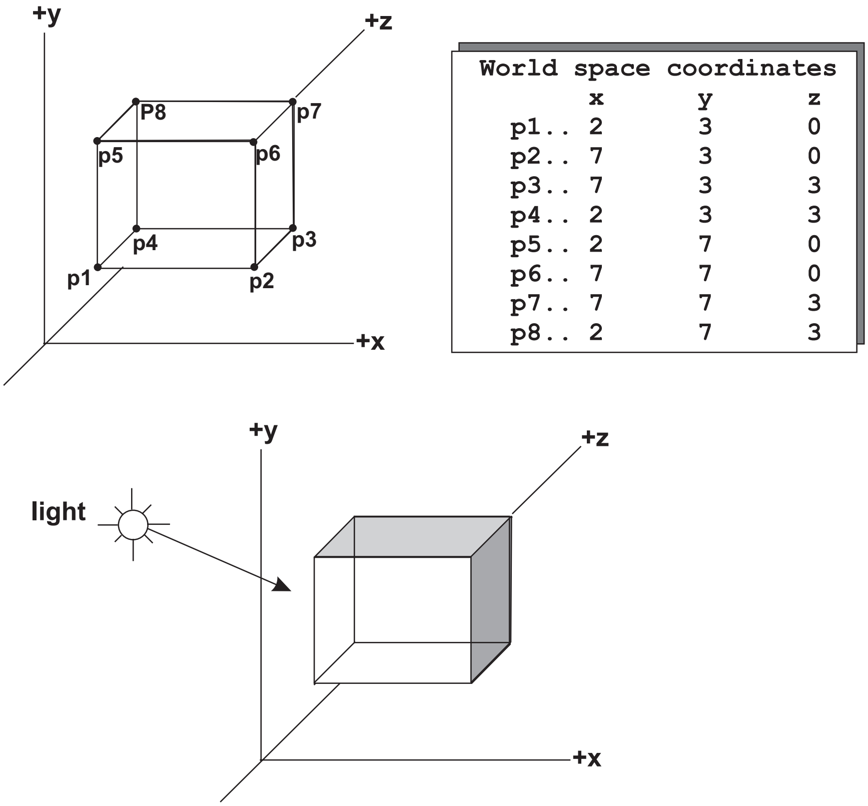
The so-called local space coordinates system facilitates numerical representation and transformations. When objects are modeled by means of polygons, the database usually includes not only the object coordinates points, but the normals to the polygon vertices and the normal to the polygon itself. This information is necessary in order to perform many of the rendering transformations.

5.2.2 World Space

The coordinate system of the scene is called the world space, or world coordinate system. Objects modeled in local space usually have to be transformed into world space at the time they are placed in a scene. For example, a particular scene may require a cube placed so that its left-bottom vertex is at coordinates x = 2, y = 3, z = 0. The process requires applying a translation transformation to the cube as it was originally defined in local space. Furthermore, lighting conditions are usually defined in world space. Once the light sources are specified and located, then shading and other rendering transformations can be applied to the polygons so as to determine how the object appears under the current illumination. Surface attributes of the object, such as texture and color, may affect the shading process. Figure 5-17 shows the world space transformation of a cube under unspecified illumination conditions and with undefined texture and color attributes.

5.2.3 Eye Space

Note in Figure 5-17 that the image is now in world space, and that some shading of the polygonal surfaces has taken place; however, the rendering is still far from complete. The first defect that is immediately evident is the lack of perspective. The second one is that all of the cube's surfaces are still visible. The eye space, or camera coordinate system, introduces the necessary transformations to improve rendering to any desired degree. Perspective transformations requires knowledge of the camera position and the projection plane. The second of these is not known until we reach the screen space phase in the rendering pipeline, therefore, it must be postponed until we reach this stage.



World space coordinates

	x	y	z
p1..	2	3	0
p2..	7	3	0
p3..	7	3	3
p4..	2	3	3
p5..	2	7	0
p6..	7	7	0
p7..	7	7	3
p8..	2	7	3

Figure 5-17 *World Space Transformation of the Cube In Figure 5-16*

The notions of eye and camera positions can be taken as equivalent, although the word "camera" is more often used in 3D graphics. The camera can be positioned anywhere in the world space and pointed in any direction. Once the camera position is determined, it is possible to eliminate those elements of the scene that are not visible. In the context of polygonal modeling this process is generically called backface elimination.

Backface Elimination or Culling

One of the most important rendering problems that must be solved at this stage of the pipeline is the elimination of the polygonal faces that are not visible from the eye position. In the simplest case, entire polygons that are not visible are removed at this time. This operation is known as culling. When dealing with a single convex object, as is a cube, culling alone solves the backface elimination problem. However, if there are multiple objects in a scene, where one object may partially obscure another one, or in the case of concave objects, then a more general backface elimination algorithm is required.

A solid object composed of polygonal surfaces that completely enclose its volume is called a polyhedron. In 3D graphics a polyhedron is usually defined so that the normals to its polygonal surfaces point away from its center. In this case, the polygons whose normals point away from the eye or camera can be assumed to be blocked by other, closer polygons, and are thus invisible. Figure 5-18 shows a cube with rods normal to each of its six polygonal surfaces. Solid arrows indicate surfaces whose normals point in the direction of the viewer. Dotted arrows indicate surfaces whose normals point away from the viewer and can, therefore, be eliminated.

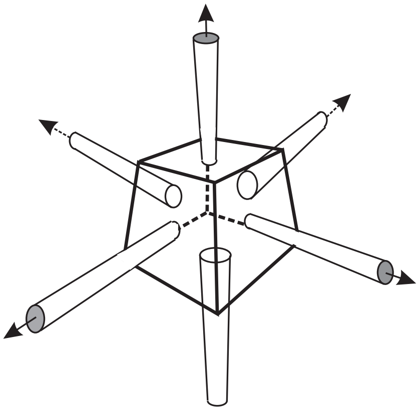


Figure 5-18 *Culling of a Polyhedron*

A single mathematical test can be used to determine if a polygonal face is visible. The geometric normal to the polygonal face is compared with a vector from the polygon to the camera or eye position. This is called the line-of-sight vector. If the resulting angle is greater than 90 degrees, then the polygonal surface faces away from the camera and can be culled. Figure 5-19 shows the use of polygonal surface and line-of-sight vectors in culling.

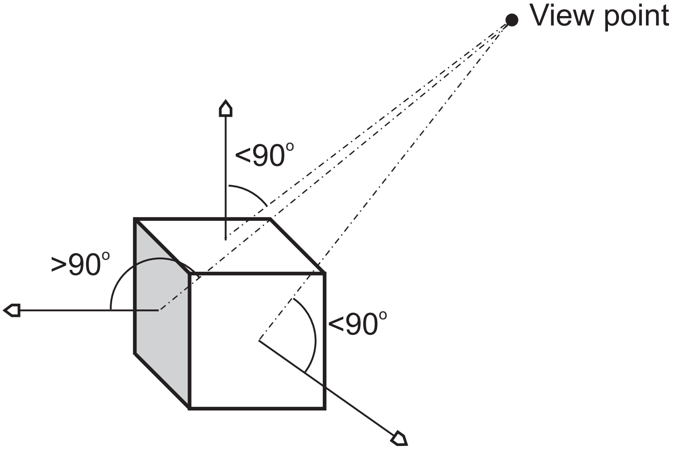


Figure 5-19 *Line-of-Sight and Surface Vectors in Culling*

Once the position of the camera is determined in the scene, it is possible to perform the backface elimination. Figure 5-20 shows the cube of Figure 5-17 after this operation.

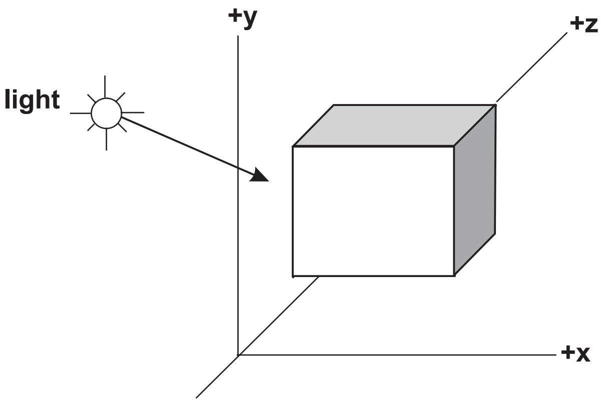


Figure 5-20 *Eye Space Transformation of the Cube In Figure 5-17*

5.2.4 Screen Space

The image, as it exists at this point of the rendering pipeline, is a numerical representation of the object. Previous illustrations, such as Figure 5-20, should not be taken literally, since the image has not yet been displayed. The last step of the rendering pipeline is the transformation onto screen space.

Changing the positioning of the camera is equivalent to rotating the object in the coordinate space. Either operation determines the type of perspective transformation: one-point, two-point, or three-point. In relation to Figure 5-17, if we position the camera so that it is normal to the face of the cube defined by points p1, p2, p6, and p5, then the result is a one-point perspective. If we position the camera so that the vertical edges of the cube remain parallel to the viewer, then the result is a two-point perspective. Similarly, we can reposition the object for a three-point perspective. In addition, the perspective transformation requires determining the distance to the plane of projection, which is known at the screen space stage of the rendering pipeline.

Screen space is defined in terms of the viewport. The final transformation in the rendering pipeline consists of eliminating those elements of the eye space that fall outside the boundaries of the screen space. This transformation is known as clipping. The perspective and clipping transformations are applied as the image reaches the last stage of the rendering pipeline. Figure 5-21, on the following page, shows the results of this stage.

5.2.5 Other Pipeline Models

The rendering pipeline model described thus far is not the only one in use. In fact, practically every 3D graphics package or development environment describes its own version of the rendering pipeline. For example, the model used in Microsoft's Direct

3D is based on a transformation sequence that starts with polygon vertices being fed into a transformations pipeline. The pipeline performs world, view, projection, and clipping transformations before data is sent to the rasterizer for display. These other versions of the rendering pipeline are discussed in the context of the particular systems to which they refer.

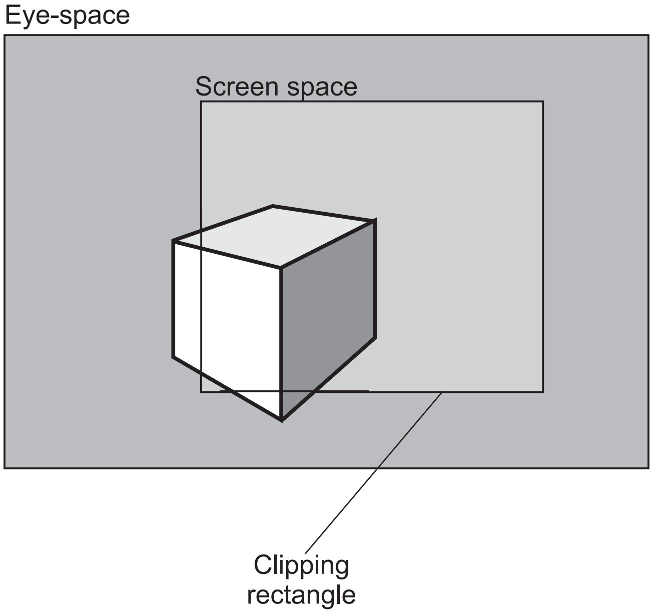


Figure 5-21 *Screen Space Transformation of the Cube in Figure 5-20*

Chapter 6

Lighting and Shading

Topics:

- Illumination models
- Reflection and shading
- Ray tracing
- Light rendering techniques

Objects are made visible by light. Our visual perception of an object is determined by the form and quality of the illumination. Lighting defines or influences color, texture, brightness, contrast, and even the mood of a scene. This chapter is an introduction to lights and shadows in 3D graphics and how lighting effects are rendered on the screen.

6.1 Lighting

To a great degree the realism of a three-dimensional object is determined by its lighting. Some solid objects are virtually impossible to represent without lighting effects. For example, a billiard ball could not be convincingly rendered as a flat disk. Figure 6-1 shows the enhanced realism that results from lighting effects on a solid object.

Figure 6-1 *Lighting Enhances Realism*

Lighting and rendering lighted objects is one of the most computationally expensive operations of 3D graphics. At this state of the technology you often have to consider not the ideal lighting effects on a scene but the minimum acceptable levels of lighting that will produce a satisfactory rendering. What is the "acceptable level" depends on the application. An interactive program that executes in real-time, such as a flight simulator or a computer game, usually places stringent limitations on lighting. For the PC animation programmer it often comes down to a tradeoff between the smoothness of the animation and the quality of the scene lighting. On the other hand, when developing applications that are not as sensitive to execution speed, or that need not execute in real-time, such as a paint program, we are able grant a greater time slice to lighting operations.

Two models are usually mentioned in the context of lighting: the reflection model and the illumination model. The reflection model describes the interaction of light within a surface. The illumination model refers to the nature of light and its intensity distribution. Both are important in developing light-rendering algorithms.

6.1.1 Illumination Models

At this point we are concerned with the light source and its characteristics; textures are considered later in this chapter. The intensity and distribution of light on the surface of an object is determined by the characteristics of the light itself, as well as by the texture of the object. A polished glass ball shows different lighting under the same illumination than a velvet-covered one.

The simplest illumination model is one in which each polygon that forms the object is displayed in a single shade of its own color. The result is a flat, monochromatic rendering in which self-luminous objects are visible by their silhouette only. One exception is if the individual polygons that form the object are assigned different colors or shades. The circular disk on the left-side of Figure 6-1 is an example of rendering without lighting effects.

There are two types of illumination — direct and indirect — which in turn, relate to two basic types of light sources — light-emitting and light reflecting. The illumination that an object receives from a light-emitting source is direct. The illumination received from a light-reflecting source is indirect. Consider a polished sphere in a room illuminated by a single light bulb. If no other opaque object is placed between the light bulb and the sphere, most of the light that falls on the sphere is direct. Indirect light, proceeding from reflection of other objects, may also take part in illuminating the sphere. If an opaque object is placed between the light bulb and the sphere, the sphere will be illuminated indirectly, which means, by reflected light only. Figure 6-2 shows a polished sphere illuminated by direct and indirect lighting, and by a combination of both.

Light sources also differ by their size. A small light source, such as the sun, is considered a point source. A rather extensive light source, such as a battery of fluorescent light, is considered an extended source. Reflected light is usually an extended source. Here again, the lighting effect of a point or extended source is modified by the object's texture. Figure 6-3 shows a polished sphere illuminated by a point and an extended source.

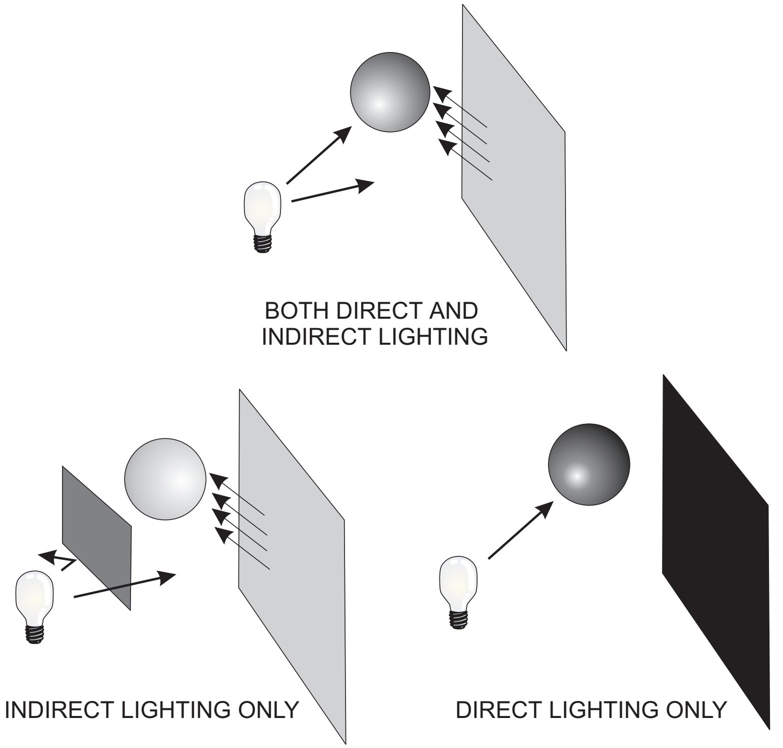


Figure 6-2 *Direct and Indirect Lighting*

6.1.2 Reflection

Except in the case of fluorescent objects, most of the lighting effects result from reflection. Ambient illumination is light that has been scattered to such a degree that it is no longer possible to determine its direction. Back lighting produces ambient illumination, as is the case in the right-hand sphere in Figure 6-3. Ambient light and matte surfaces produce diffuse reflection. Point sources and polished surfaces produce specular reflection. Variations in the light source and surface textures give rise to virtually unlimited variations between pure diffuse and pure specular reflection.

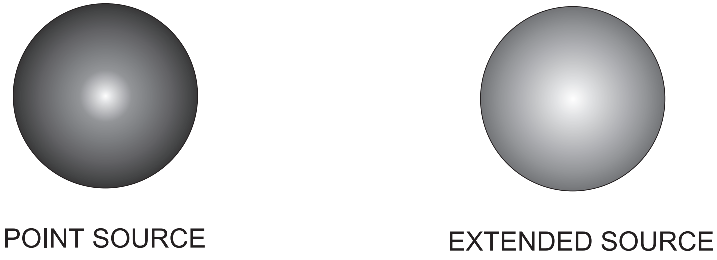


Figure 6-3 *Point and Extended Light Sources*

Diffuse Reflection

Ambient light produces a uniform illumination on the object's surface. If a surface is exposed to ambient light alone, then the intensity of reflection at any point on the surface is expressed by the formula

$$I = Ik$$

where I is the intensity of illumination and k is the ambient reflection coefficient, or reflectivity, of the surface. Notice that this coefficient is a property of the surface material. In calculations, k is assigned a constant value in the range 0 to 1. Highly reflective surfaces have values near 1. With high reflectivities light has nearly the same effects as incident light. Surfaces that absorb most of the light have a reflectivity near 0.

The second element in determining diffuse reflection is the angle of illumination, or angle of incidence. A surface perpendicular to the direction of incident light reflects more light than a surface at an angle to the incident light. The calculation of diffuse reflection can be made according to Lambert's cosine law, which states that, for a point source, the intensity of reflected light is proportional to the cosine of the angle of incidence. Figure 6-4 shows this effect.

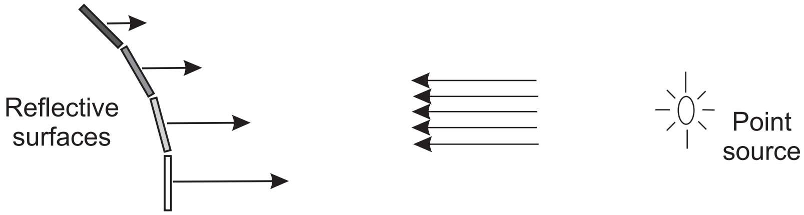


Figure 6-4 *Angle of Incidence in Reflected Light*

Diffuse reflection obeys Lambert's cosine law. Lambertian reflection is associated with matte, dull surfaces such as rubber, chalk, and cloth. The degree of diffusion depends on the material and the illumination. Given the same texture and lighting conditions, diffuse reflection is determined solely by the angle of incidence. In addition, the type of the light source and atmospheric attenuation can influence the degree of diffusion. The spheres in Figure 6-5 show various degrees of diffuse illumination.

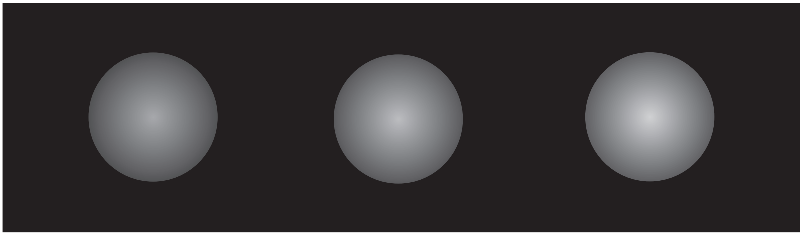


Figure 6-5 *Diffuse Reflection*

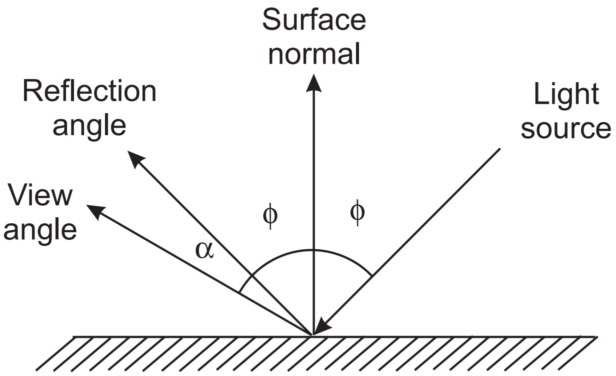


Figure 6-6 *Specular Reflection*

Specular Reflection

Specular reflection is observed in shiny or polished surfaces. Illuminating a polished sphere, such as a glass ball, with a bright white light, produces a highlight of the same color as the incident light. Specular reflection is also influenced by the angle of incidence. In a perfect reflector the angle of incidence, which is the inclination of the light source to the surface normal, is the same as the angle of reflection. Figure 6-6 shows the angles in specular reflection.

In Figure 6-6 you can notice that in specular reflection the angle of incidence (f) is the same as the angle of reflection. In a perfect reflector specular reflection is visible only when the observer is located at the angle of reflection, in other words, when $\mu = 0$. Objects that are not perfect reflectors exhibit some degree of specular reflection over a range of viewing positions located about the angle of reflection. Polished surfaces have a narrow reflection angle while dull surfaces have a wider one.

Phong's Model

In 1975 Phong Bui-Toung described a model for non-perfect reflectors. The Phong model, which is widely used in 3D graphics, assumes that specular reflectance is great in the direction of the reflection angle, and decreases as the viewing angle increases. The Phong model sets the intensity of reflection according to the function

$$I = \cos^n \infty$$

where n is called the material's specular reflection exponent. For a perfect reflector, n is infinite and the falloff is instant. In the Phong model normal values of n range from one to several hundreds, depending on the surface material. The shaded areas in Figure 6-7 show Phong reflection for a shiny and a dull surface. The larger the value of n, the faster the falloff and the smaller the angle at which specular reflection is visible. A polished surface is associated with a large value for n, while a dull surface has a small n.

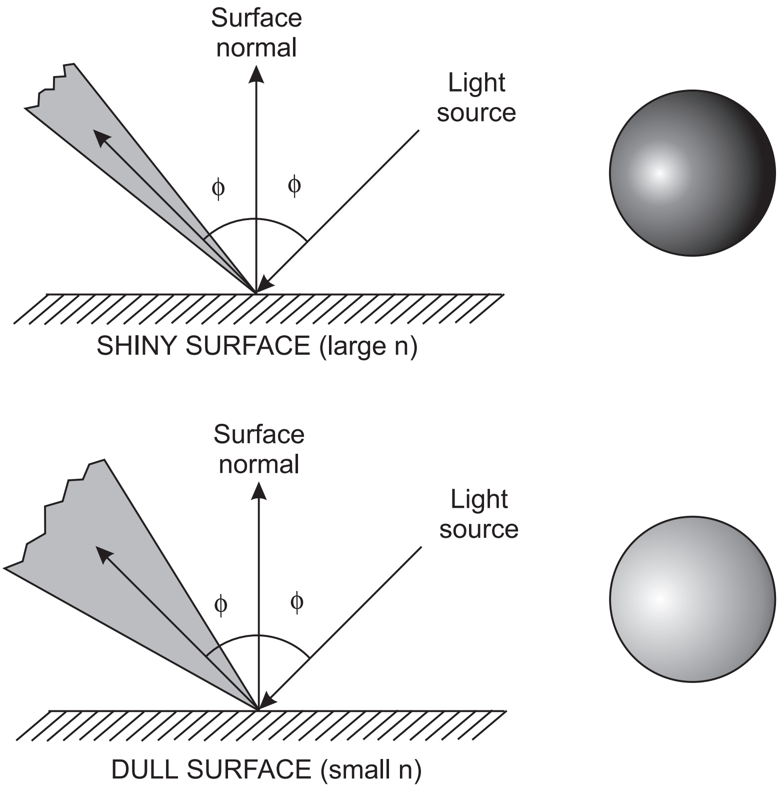


Figure 6-7 *Values of n in Phong Model of Specular Reflection*

The Phong model enjoys considerable popularity because of its simplicity, and because it provides sufficient realism for many applications. It also has some important drawbacks:

1. All light sources are assumed to be points.
2. Light sources and viewers are assumed to be at infinity.
3. Diffuse and specular reflections are modeled as local components.
4. The decrease of reflection is empirically determined around the reflection vector.
5. Regardless of the color of the surface all highlights are rendered white.

Resulting from these limitations, the following observations have been made regarding the Phong model:

1. The Phong model does not render plastics and other colored solids very well. This results from the white color or all highlights.
2. The Phong model does not generate shadows. This makes objects in a scene to appear to float in midair.
3. Object concavities are often rendered incorrectly. This is results in specular highlights in concave areas that should not have them.

6.2 Shading

Shading refers to the application of a reflection model over the surface of an object. Since graphics objects are often represented by polygons, a brute force shading method can be based on calculating the normal to each polygon surface, and then applying an illumination model, such as Phong, to that point.

6.2.1 Flat Shading

The simplest shading algorithm, called flat shading, consists of using an illumination model to determine the corresponding intensity value for the incident light, then shade the entire polygon according to this value. Flat shading is also known as constant shading or constant intensity shading. Its main advantage is that it is easy it implement. Flat shading produces satisfactory results under the following conditions:

1. The subject is illuminated by ambient light and there are no surface textures or shadows.
2. In the case of curved objects, when the surface changes gradually and the light source and viewer are far from the surface.
3. In general, when there are large numbers of plane surfaces.

Figure 6-8 shows three cases of flat shading of a conical surface. The more polygons, the better the rendering.

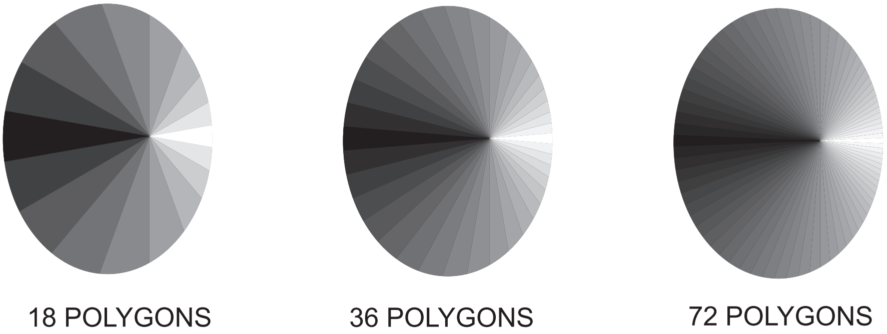


Figure 6-8 *Flat Shading*

6.2.2 Interpolative Shading

The major limitation of flat shading is that each polygon is rendered in a single color. Very often the only way of improving the rendering is by increasing the number of polygons, as shown in Figure 6-8. An alternative scheme is based on using more than one shade in each polygon, which is accomplished by interpolating the values calculated for the vertices to the polygon's interior points. This type of manipulation, called interpolative or incremental shading, under some circumstances is capable of producing a more satisfactory shade rendering with a smaller number of polygons. Two incremental shading methods, called Gouraud and Phong shading, are almost ubiquitous in 3D rendering software.

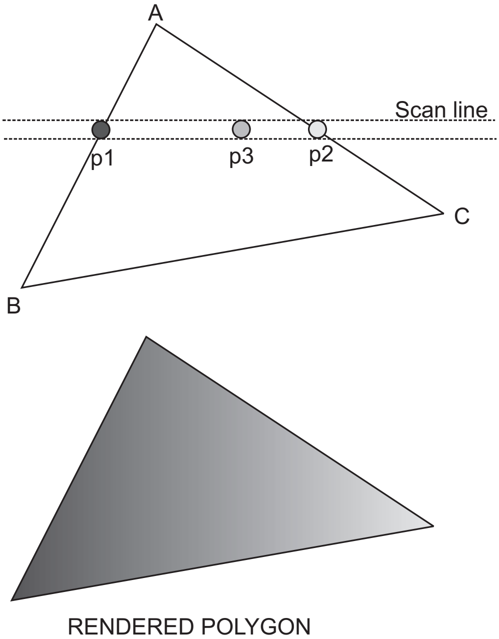


Figure 6-9 *Intensity Interpolation in Gouraud Shading*

Gouraud Shading

This shading algorithm was first described by H. Gouraud in 1971. It is also called bilinear intensity interpolation. Gouraud shading is easier to understand in the context of the scan-line algorithm used in hidden surface removal, discussed later in this chapter. For now, assume that each pixel is examined according to its horizontal (scan-line) placement, usually left to right. Figure 6-9 shows a triangular polygon with vertices at A, B, and C.

The intensity value at each of these vertices is based on the reflection model. As scan-line processing proceeds, the intensity of pixel p1 is determined by interpolating the intensities at vertices A and B, according to the formula

In the example of Figure 6-9, the intensity of p1 is closer to the intensity of vertex A than that of vertex B. The intensity of p2 is determined similarly, by interpolating

$$I_{p1} = \frac{y_{p1} - y_B}{y_A - y_B} + \frac{y_A - y_{p1}}{y_A - y_B}$$

the intensities of vertices A and C. Once the boundary intensities for the scan line are determined, any pixel along the scan line is calculated by interpolating, according to the following formula

$$I_{p3} = I_{p1} \frac{x_{p2} - x_{p3}}{x_{p2} - x_{p1}} + I_{p2} \frac{x_{p3} - x_{p1}}{x_{p2} - x_{p1}}$$

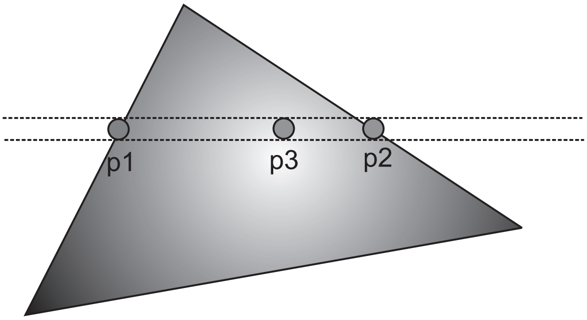


Figure 6-10 *Highlight Rendering Error in Gouraud Shading*

The process is continued for each pixel in the polygon, and for each polygon in the scene. Gouraud shading calculations are usually combined with a scan-line hidden surface removal algorithm and performed at the same time.

Gouraud shading also has limitations. One of the most important ones is the loss of highlights on surfaces and highlights that are displayed with unusual shapes. Figure 6-10 shows a polygon with an interior highlight. Since Gouraud shading is based on the intensity of the pixels located at the polygon edges, this highlight is missed. In this case pixel p3 is rendered by interpolating the values of p1 and p2, which produces a darker color than the one required.

Another error associated with Gouraud shading is the appearance of bright or dark streaks, called Mach bands.

Phong Shading

Phong shading is the most popular shading algorithm in use today. This method was developed by Phong Bui-Toung, the author of the illumination model described previously. Pong shading, also called normal-vector interpolation, is based on calculating pixel intensities by means of the approximated normal vector at the point in the polygon. Although more calculation expensive, Phong shading improves the rendering of bright points and highlights that are misrendered in Gouraud shading.

6.2.3 Ray Tracing

Other shading models find occasional use in 3D graphics. The ones discussed so far (Phong and Gouraud shading) as well as others of intermediate complexity are not based on the physics of light, but on the way that light interacts with objects. Although the notion of light intensity is used in these models, it is not formally defined. Physically-based methods, although much more expensive computationally, can produce more accurate rendering. One such method, called ray tracing, is based on backtracking the light rays from the center of projection (viewing position) to the light source.

Ray tracing originated, not in computer graphics, but in geometric optics. In 1637, René Descartes used ray tracing on a glass ball filled with water to explain rainbow formation. Around 1980, computer graphics researchers began applying ray tracing techniques in the production of very high-quality images, at a very high processing cost. Ray tracing is a versatile and powerful rendering tool. It incorporates the processing done in reflection, hidden surface removal, and shading operations. When execution time is not a factor, ray tracing produces better results, better than any other rendering scheme. This fact has led to the general judgment that ray tracing is currently the best implementation of an illumination model.

In a simple reflection model only the interaction of a surface with the light source is considered. For this reason, when a light ray reaches a surface through interaction with another surface, when it is transmitted through a partially transparent object, or by a combination of these factors, the rendering fails. Figure 6-11 shows how ray tracing captures the reflected image of a cube on the surface of a polished sphere.

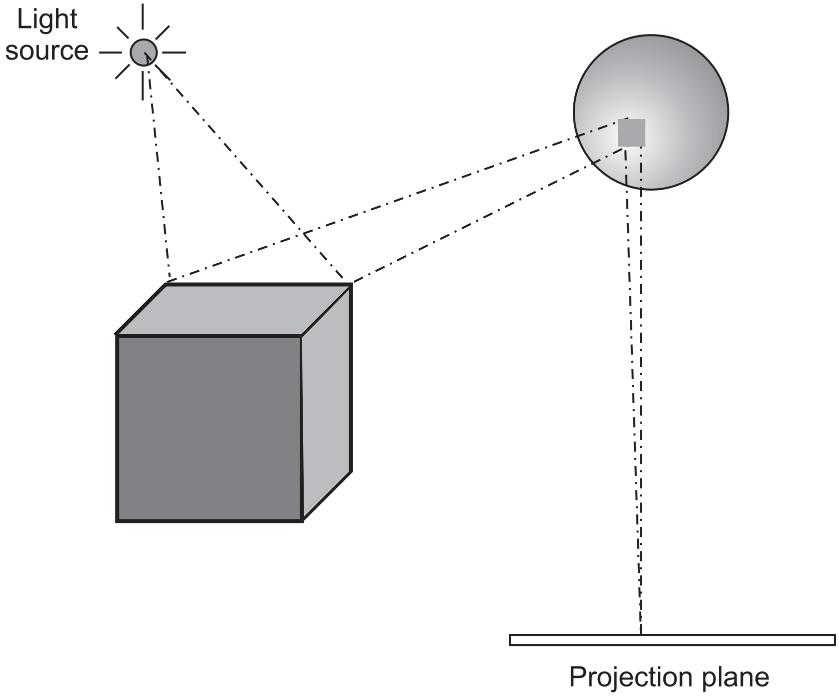


Figure 6-11 *Rendering a Reflected Image by Ray Tracing*

6.3 Other Rendering Algorithms

So far we have discussed rendering algorithms that relate to perspective, culling and hidden surface removal, illumination, and shading. In this section we look at other rendering methods that complement or support the ones already mentioned. Note that we have selected a few of the better known schemes; many others are discussed the literature.

6.3.1 Scan-Line Operations

In computer graphics the term scan-line processing or scan-line algorithms refers to a general processing method whereby each successive pixel is examined row by row, that is, in scan-line order. You already encountered scan-line processing in Gouraud shading. Scan-line methods are also used in filling the interior of polygons. In fact, most rendering engines use some form of scan-line processing. Usually several algorithms are incorporated into a scan-line routine. For example, as each pixel is examined in the scan-line order, hidden-surface removal, shading, and shadow generation logic are applied to it in order to determine how it is to be rendered. The result is a considerable saving compared to the time it would take to apply each rendering operations independently.

Hidden Surface Removal

A scan-line algorithm called the *image space method* is often used for removing hidden surfaces in a scene. This method is actually a variation of polygon filling algorithm. The processing requires that the image database contain the coordinate points for each polygon vertex. This is usually called the edge table. Figure 6-12 shows two overlapping triangles whose vertices (A, B, C, D, E, and F) are stored in the edge table.

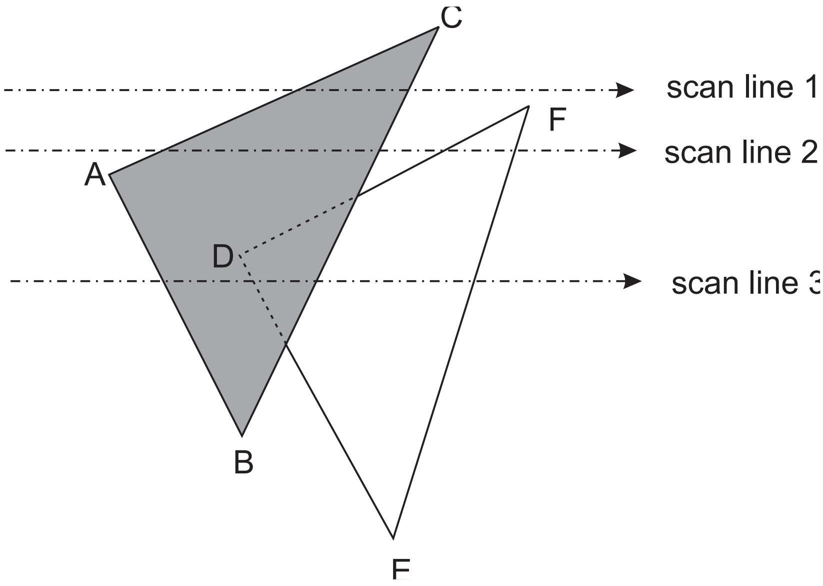


Figure 6-12 *Scan-Line Algorithm for Hidden Surface Removal*

The scan-line algorithm for hidden surface removal uses a binary flag to indicate whether a pixel is inside or outside the surface. Each surface on the scene is given one such flag. As the left-most boundary of a surface is reached, the flag is turned on. At the surface's right-most boundary the flag is turned off. When a single surface flag is on, the surface is rendered at that pixel. Scan line 1 in Figure 6-12 has some pixels in which the flag is on for triangle ABC. Scan line 2 in Figure 6-12 also poses no problem, since a single surface has its flag on at one time. In scan line 3 the flag for triangle ABC is turned on at its left-most boundary. Before the surface's right-most boundary is reached, the flag for triangle DEF is turned on. When two flags are on for a given pixel, the processing algorithm examines the database to determine the depth of each surface. The surface with less depth is rendered, and all the other ones are removed. As the scan line processing crosses the boundary defined by edge BC, the flag for triangle ABC is turned off. From that point on, the flag for triangle DEF is the only one turned on; therefore, its surface is rendered.

Shadow Projections

Ray tracing can be used to generate shadows; however, other rendering methods can also be designed to handle of shadows. For example, it is possible to add shadow processing to a scan-line routine. To illustrate this point assume an image database with a list of polygons that may mutually shadow each other. This list, called the shadow pairs, is constructed by projecting all polygons onto a sphere located at the light source. Polygon pairs that can interact are the only ones included in the shadow pairs list. The shadow pairs list saves considerable processing effort by eliminating those polygons that cannot possibly cast a shadow on each other.

The actual processing is similar to the scan-line algorithm for hidden surface removal. Figure 6-13 shows two polygons, labeled A and B. In this example we assume a single light source placed so that polygon A casts a shadow on polygon B. The shadow pairs in the database tell us that polygon B cannot shadow polygon A, but polygon A can shadow polygon B. For this reason, in scan line 1 polygon A is rendered without further query. In scan line 2 polygon B is shadowed by polygon A. Therefore, the pixels are modified appropriately. In scan line 3 polygon B is rendered.

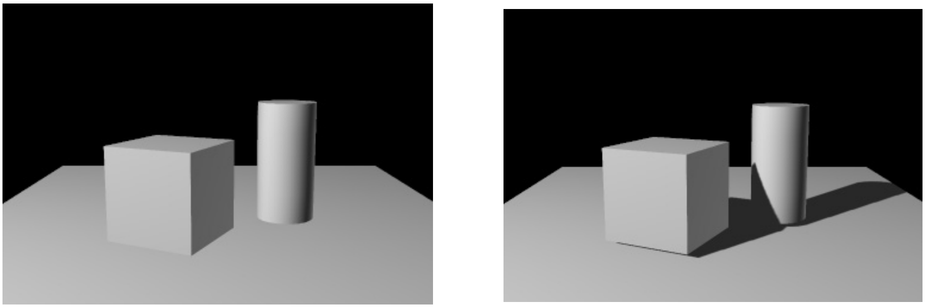


Figure 6-13 *Scan-Line Algorithm for Shadow Projection*

Figure 6-14 shows two renderings of the same scene. The one on the left-side is done without shadow projection. The one on the right is rendered using a shadow projection algorithm.

6.3.2 Z-Buffer Algorithm

Developed by Catmull in 1975, the z-buffer or depth buffer algorithm for eliminating hidden surfaces has become a staple in 3D computer graphics. The reason for its popularity is its simplicity of implementation.

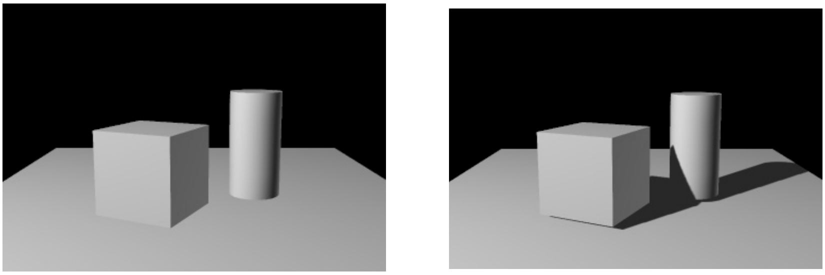


Figure 6-14 *Shadow Rendering of Multiple Objects*

The algorithm's name relates to the fact that the processing routine stores in a buffer the z-coordinates for the (x,y) points of all objects in the scene. This is the z-buffer. A second buffer, sometimes called the refresh buffer, is used to hold the intensities for each pixel. During processing, all positions in the z-buffer are first initialized to the maximum depth value, and all positions in the refresh buffer to the background attribute. At each pixel position, each polygon surface in the scene is examined for its z-coordinate value. If the z coordinate for the surface is less than the value stored in the z-buffer, then the value in the z-buffer is replaced with the one corresponding to the surface being examined. At this point the refresh buffer is also updated with the intensity value for that pixel. If the z value for the surface is greater than the value in the z-buffer, then the point is not visible and can be ignored.

Figure 6-15 shows the z-buffer algorithm in action. Three surfaces (a square, a circle, and a triangle) are located at various depths. When the z-buffer is initialized the pixel shown in the illustration is assigned the depth of the background surface, S0. The surface for the circle is examined next. Because S2 is at less depth than S0, the value S2 replaces the value S0 in the z-buffer. Now S2 is the current value in the z-buffer. Next, the value for the triangular surface S1 is examined. Since S1 has greater depth than S2 it is ignored. However, when S3 is examined it replaces S2 in the buffer, since it is at less depth.

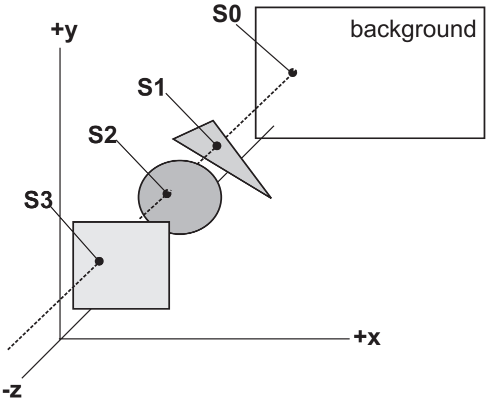


Figure 6-15 *Z-Buffer Algorithm Processing*

6.3.3 Textures

The surface composition of an object influences how it reflects light. For this reason, the reflectivity of a surface must be taken into account when calculating illumination effects. Textures were completely ignored in early 3D packages. At that time all surfaces were assumed to have identical reflection properties. The result were scenes that appeared unnatural because of their uniformity. Since then, textures have been steadily gaining popularity.

The simplest and most common implementation of textures is with bitmaps. In this case the texture refers only to the color pattern of the surface, and not to its degree of smoothness. Texture bitmaps are easy to apply to objects and are rendered as a surface attribute. In addition, texture blending and light mapping with textures provide additional enhancements to the rendering. The specifics of texture rendering are discussed in the context of 3D graphics programming.

Part II

DOS Graphics

Chapter 7

VGA Fundamentals

Topics:

- The VGA standard
- VGA components
- Alphanumeric modes
- Graphics modes
- VGA programmable components

This chapter describes the VGA video standard and its programmable elements: the CRT Controller, the Graphics Controller, the Sequencer, the Attribute Controller, and the Digital-to-Analog converter (DAC). It also describes the VGA memory structure.

7.1 The VGA Standard

In 1987 IBM introduced two video systems to be furnished as standard components for their PS/2 line. These video systems were named the MCGA (Multi-color Graphics Array) and VGA (Video Graphics Array). MCGA, an under-featured version of VGA, was furnished with the lower-end PS/2 machines Models 25 and 30. VGA was the standard video system for all other PS/2 microcomputers. Subsequently IBM extended VGA to its low-end models of the PS/2 line. Later on (August 1990) IBM announced a line of inexpensive home computers (designated as the PS/1 line) equipped with VGA graphics. Since the MCGA standard was short lived and not very popular it will not be specifically considered in this book. However, because MCGA is a sub-version of VGA, its programming is identical to VGA in those video modes that are common to both systems.

The VGA standard introduced a change from digital to analog video display driver technology. The reason for this change is that analog monitors can produce a much larger color selection than digital ones. This switch in display technology explains why the monitors of the PC line are incompatible with the VGA standard and vice versa. VGA graphics also include a digital-to-analog converter, usually called the

DAC, and 256K of video memory. The DAC outputs the red, green, and blue signals to the analog display. Video memory is divided into four 64K video maps, called the bit planes. VGA supports all the display modes in MDA, CGA, and EGA (see Table 1-1). In addition, VGA implements several new alphanumeric and graphics modes, the most notable of which are graphics mode number 18, with 640-by-480 pixel resolution in 16 colors, and graphics mode number 19, with 320-by-200 pixel resolution in 256 colors. The effective resolution of the VGA text modes is of 720-by-400 pixels. These text modes can execute in 16 colors or in monochrome. Three different fonts can be selected in the alphanumeric modes.

Access to the VGA registers and to video memory is through the system microprocessor. The microprocessor read and write operations to the video buffer are automatically synchronized by the VGA with the cathode-ray tube (CRT) controller so as to eliminate interference. This explains why VGA programs, unlike those written for the CGA, can access video memory at any time without fear of introducing screen snow or other unsightly effects.

7.1.1 Advantages and Limitations

The resolution of a graphics system is usually defined as the total number of separately addressable elements per unit area. In video display systems the individually addressable elements are the screen pixels; the resolution is measured in pixels per inch. For example, the maximum resolution of a VGA system is approximately 80 pixels per inch, both vertically and horizontally. In VGA this density is determined by a screen structure of 640 pixels per each 8-inch screen row and 480 vertical pixels per each 6-inch screen column. But not all video systems output a symmetrical pixel density. For example, the maximum resolution of the EGA standard is the same as that of the VGA on the horizontal axis (80 pixels per inch) but only of 58 pixels per inch on the vertical axis.

The asymmetrical pixel grid of the EGA and of other less refined video standards introduces programming complications. For example, in a symmetrical VGA screen a square figure can be drawn using lines of the same pixel length, but these lines would produce a rectangle in an asymmetrical system. By the same token, the pixel pattern of a circle in a symmetrical system will appear as an ellipse in an asymmetrical one, as shown in Figure 7-1.

The major limitations of the VGA system are resolution, color range, and performance. VGA density of 80 pixels per inch is a substantial improvement in relation to its predecessors, the CGA and the EGA, but still not very high when compared to the 600 dots per inch of a typical laser printer, or the 1200 and 2400 dots per inch of a high-end printer or imagesetter. The low resolution is one reason why VGA screen images are often not lifelike; bitmaps appear grainy and we can often detect that geometrical figures consist of straight-line segments. In regards to color range VGA can display up to 256 simultaneous colors; however, this color range is not available in the mode with the best resolution. In other words, the VGA programmer must chose between an 80 pixels per inch resolution in 16 colors (mode number 18) or 40 pixels per inch resolution in 256 colors (mode number 19).

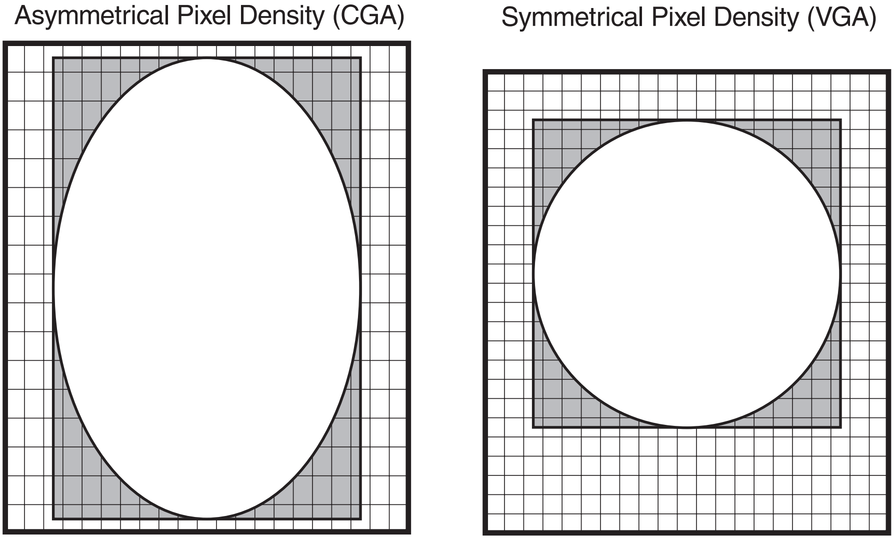

Asymmetrical Pixel Density (CGA)
Symmetrical Pixel Density (VGA)

Figure 7-1 *Symmetrical and Asymmetrical Pixel Density*

But perhaps the greatest limitation of the VGA standard is its performance. The video display update operations in VGA detract from general system efficiency, since it is the microprocessor that must execute all video read and write operations. In the second place, the video functions execute slowly when compared to dedicated graphics work stations. This slowness is particularly noticeable in the graphics modes, in which a full screen redraw can take several seconds. Most animated programs, which must update portions of the screen at a rapid rate, execute in VGA with a jolting effect that is unnatural and visually disturbing.

7.1.2 VGA Modes

The original video systems used in IBM microcomputers, such as CGA, MDA, and EGA, had monitor-specific modes. For example, the CGA turns the color burst off in modes 0, 2, and 4 and on in modes 1, 3, and 5. Mode number 7 is available in the Monochrome Display Adapter (MDA) and in an Enhanced Graphics Adapter (EGA) equipped with a monochrome display, but not in the CGA or EGA systems equipped with color monitors. In the VGA standard, on the other hand, the video modes are independent of the monitor. For example, a VGA equipped with any one of the standard direct drive color monitors can execute in monochrome mode number 7. Table 7-1, on the following page, lists the properties of the VGA video modes.

In Table 7-1 we have used decimal numbers for the video modes. Our rationale is that video modes are a conventional ordering scheme used in organizing common hardware and software characteristics of a video system, therefore we can see no reason for using hexadecimal notation in numbering these modes. Consequently, throughout the book, we have used decimal numbers for video modes, offset values, and other forms of sequential orders that do not require binary or hexadecimal notation.

Table 7-1

VGA Video Modes

MODE	COLORS	TYPE	TEXT COLS/ROWS	TEXT PIXEL BOX	SCREEN PAGES	BUFFER ADDRESS	SCREEN PIXELS
0,1	16	Alpha	40 by 25	8 x 8 8 x 14* 9 x 16+	8	B8000H	320 by 200 320 by 350 360 by 400
2,3	16	Alpha	80 by 25	8 x 8 8 x 14* 9 x 16	8	B8000H	320 by 200 320 by 350 360 by 400
4,5	4	GRA	40 by 25	8 x 8	1	A0000H	320 by 200
6	2	GRA	80 by 25	8 x 8	1	A0000H	640 by 200
7		Alpha	80 by 28	9 x 14 9 x 16	8	B0000H	720 by 350 720 by 400
13	16	GRA	40 by 25	8 x 8	8	A0000H	320 by 200
14	16	GRA	80 by 25	8 x 8	4	A0000H	640 by 200
15		GRA	80 by 25	8 x 14	2	A0000H	640 by 350
16	16	GRA	80 by 25	8 x 14	2	A0000H	640 by 350
17	2	GRA	80 by 30	8 x 16	1	A0000H	640 by 480
18	16	GRA	80 by 30	8 x 16	1	A0000H	640 by 480
19	256	GRA	40 by 25	8 x 8	1	A0000H	320 by 200

Legend:

Alpha = alphanumeric modes (text)
GRA = graphics modes
* = EGA enhanced modes
+ = VGA enhanced modes

Notice in Table 7-1 that the VGA buffer can start in any one of three possible addresses: B0000H, B8000H, and A0000H. Address B000H is used only when mode 7 is enabled, in this case VGA is emulating the Monochrome Display Adapter. In enhanced mode number 7 the VGA displays its highest horizontal resolution (720 pixels) and uses a 9 x 16 dots text font. However, in this mode the VGA is capable of text display only. Buffer address A000H is active while VGA is in a graphics mode. Also note that the video modes number 17 and 18, with 480 pixel rows, were introduced with the VGA and MCGA standards. Therefore they are not available in CGA and EGA systems. Modes 17 and 18 offer a symmetrical pixel density of 640 by 480 screen dots (see Figure 7-1). Mode number 19 has 256 simultaneous colors; the most extensive one in the VGA standard, however, its linear resolution, is half of the one in mode number 18.

7.2 VGA Components

The VGA system is divided into three identifiable components: the VGA chip, video memory, and a Digital-to-Analog Converter (DAC). Figure 7-2 shows the interconnections between the elements of the VGA system.

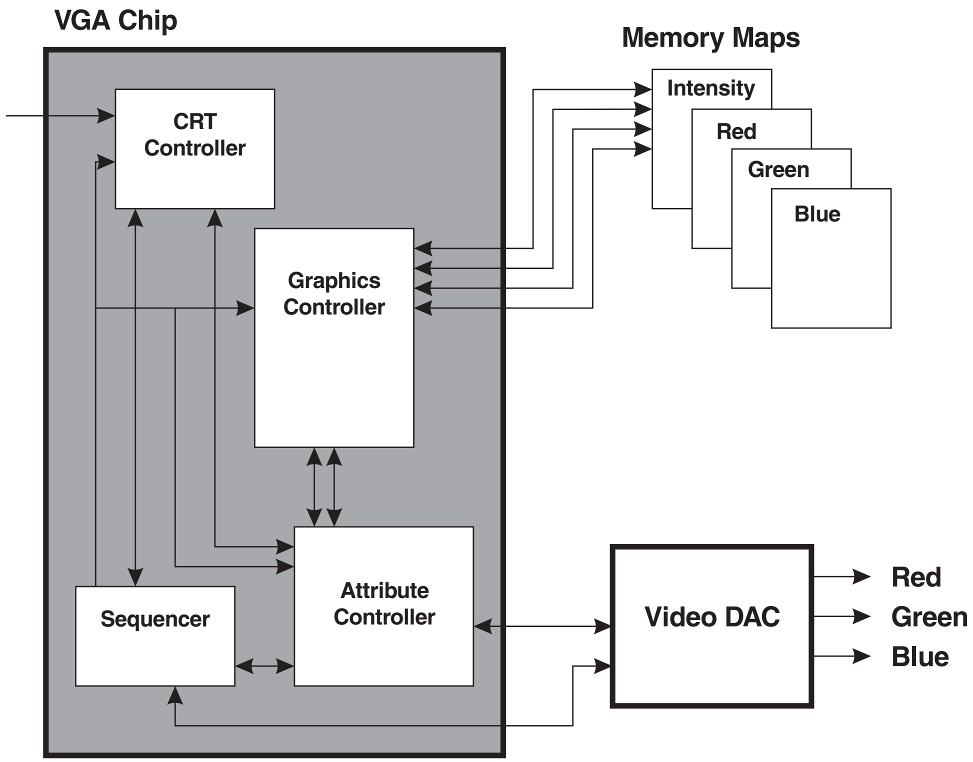


Figure 7-2 *VGA System Components*

7.2.1 Video Memory

All VGA systems contain the 256K of video memory that is part of the hardware. This memory is logically arranged in four 64K blocks that form the video maps (labeled blue, green, red, and intensity in Figure 7-2). The four maps are sometimes referred to as bit planes 0 to 3.

In EGA systems the display buffer consists of a 64K RAM chip installed in the card itself. Up to three more 64K blocks of video memory can be optionally added on the piggyback memory expansion card. The maximum memory supported by EGA is 256K divided into four 64K blocks.

Alphanumeric Modes

In the alphanumeric modes 0, 1, 2, 3, and 7 (see Table 7-1) the VGA video buffer is structured to hold character codes and attribute bytes. The organization of the video buffer in the alphanumeric modes was discussed in Part I of this book. The default functions of the bits in the attribute byte can be seen in Figures 1.11 and 1.12. However, the VGA standard allows redefining two of the attribute bits in the color alphanumeric modes: bit 7 can be redefined to control the background intensity and bit 3 can be redefined to perform a character-set select operation. Figure 7-3 shows the VGA attribute byte, including the two redefinable bits.

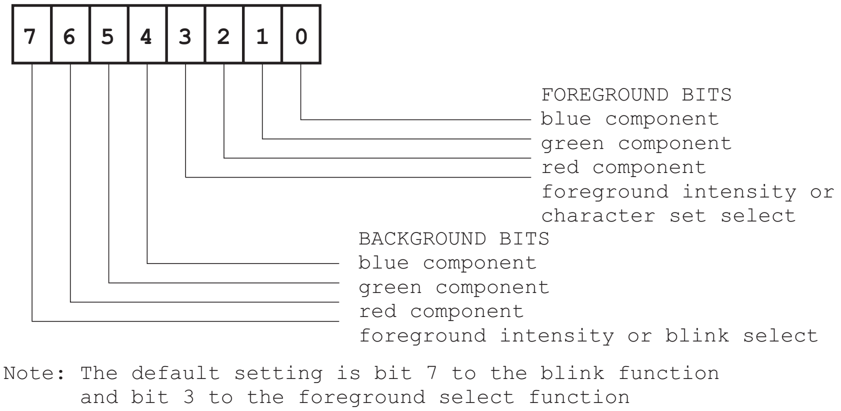


Figure 7-3 *Attribute Byte Bitmap in VGA Systems*

The programmer can toggle the functions assigned to bits 3 and 7 of the attribute byte by means of BIOS service calls or by programming the VGA registers. These operations are performed by the VGA graphics library on that is part of the book's software.

Graphics Modes

One of the problems confronted by the designers of the VGA system was the limited memory space of an IBM microcomputers under MS DOS. Recall that in VGA mode number 18 (see Table 7-1) the video screen is composed of 480 rows of 640 pixels per row, for a total of 307,200 screen pixels. If 8 pixels are encoded per memory byte, each color map would take up approximately 38K, and the four maps required to encode 16 colors available in this mode would need approximately 154K. The VGA designers were able to reduce this memory space by using a latching mechanism that maps all four color maps to the same memory area. Figure 7-4 is a diagram of the video memory structure in VGA mode number 18.

Figure 7-4 shows how the color of a single screen pixel is stored in four memory maps, located at the same physical address. Note that the color codes for the first eight screen pixels are stored in the four maps labeled Intensity, Red, Green, and Blue. In VGA mode number 18 all four maps are located at address A0000H. The first screen pixel has the intensity bit and the green bit set, therefore it appears light green. For the same reason, the second pixel, mapped to the subsequent bits in the video buffer, will be displayed as light red, since it has the red and the intensity bits set (see Figure 7-4).

VGA memory mapping changes in the different alphanumeric and graphics modes. In Figure 7-4 we see that in mode number 18 the color of each screen pixel is determined by the bit settings in four memory maps. However, in mode number 19, in which VGA can display 256 colors, each screen pixel is determined by one video buffer byte. Figure 7-5 shows the memory mapping in VGA mode number 19. In reality VGA uses all four bit planes to store video data in mode number 19, but, to the programmer, the buffer appears as a linear space starting at address A000H. The color value assigned to each pixel in the 256-color modes is explained in Chapter 8.

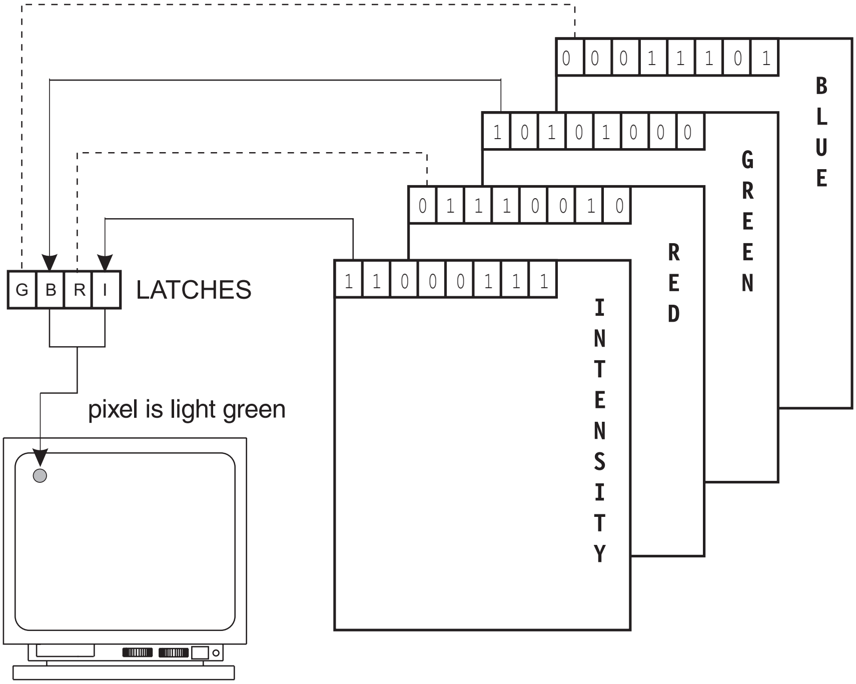


Figure 7-4 *Video Memory Mapping in VGA Mode 18*

Many VGA graphics modes were created to insure compatibility with previous video systems. Specifically, VGA graphics modes numbers 4, 5, and 6 are compatible with modes in the CGA, EGA, and PCjr; modes numbers 13, 14, 15, and 16 are compatible with EGA; and graphics mode number 17 (a two-color version of mode number 18) was created for compatibility with the MCGA standard. This leaves two proprietary VGA modes: mode number 18 with 640-by-480 pixels in 16 colors, and mode number 19, with 320-by-200 pixels in 256 colors. It is in these two most powerful VGA modes that we will concentrate our attention.

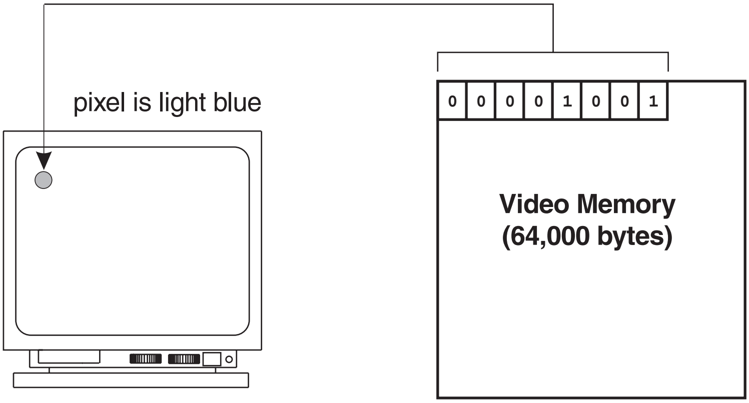


Figure 7-5 *Video Memory Mapping in VGA Mode 19*

7.3 VGA Registers

We have seen that the VGA system includes a chip containing several registers, a memory space dedicated to video functions, and a digital-to-analog converter (see Figure 7-2). The VGA registers are mapped to the system's address space and accessed by means of the central processor. The VGA programmable registers (excluding the DAC) belong to five groups (also shown in Table 7-2):

1. The General registers. This group is sometimes called the external registers due to the fact that, on the EGA, they were located outside the VLSI chip. The general registers provide miscellaneous and control functions.
2. The CRT Controller registers. This group of registers controls the timing and synchronization of the video signal. Also the cursor size and position.
3. The Sequencer registers. This group of registers controls data flow into the Attribute Controller, generates the timing pulses for the dynamic RAMs, and arbitrates memory accesses between the CPU and the video system. The Map Mask registers in the Sequencer allow the protection of entire memory maps.

Table 7-2

VGA Register Groups

		EMULATING		
REGISTER	READ/WRITE	MDA	CGA	EITHER
GENERAL REGISTERS				
1. Miscellaneous output	Write Read			03C2H 03CCH
2. Input status 0	Read			03C2H
3. Input status 1	Read	03BAH	03DAH	
4. Feature control	Write Read	03BAH	03DAH	 03CAH
5. Video Subsystem enable	R/W			03C3H
6. DAC state	Read			03C7H
CRT CONTROLLER REGISTERS				
1. Index	R/W	03B4H	03D4H	
2. Other CRT Controller	R/W	03B5H	03D5H	
SEQUENCER REGISTERS				
1. Address	R/W			03C4H
2. Other	R/W			03C5H
GRAPHICS CONTROLLER REGISTERS				
1. Address	R/W			03CEH
2. Other	R/W			03CFH
ATTRIBUTE CONTROLLER REGISTERS				
1. Address	R/W			03C0H
2. Other	Write Read			03C0H 03C1H

4. The Graphics Controller registers. This group of registers provides an interface between the system microprocessor, the Attribute Controller, and video memory, while VGA is in a graphics mode.
5. The Attribute Controller registers. This group of registers determines the characteristics of the character display in the alphanumeric modes and the pixel color in the graphics modes.

7.3.1 The General Registers

The General registers, called the External registers in EGA, are used primarily in initialization of the video system and in mode setting. Most applications let the system software handle the initialization of the video functions controlled by the General registers. For example, the easiest and most reliable way for setting a video mode is BIOS service number 0, of interrupt 10H. Figure 7-6 and Figure 7-7 show some programmable elements in the VGA General Register group.

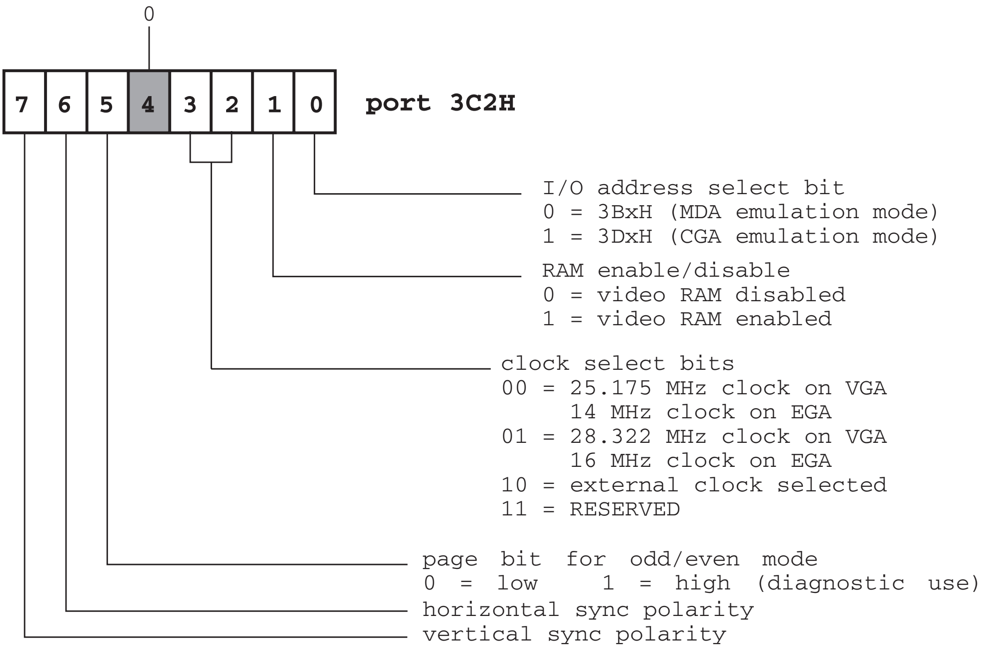
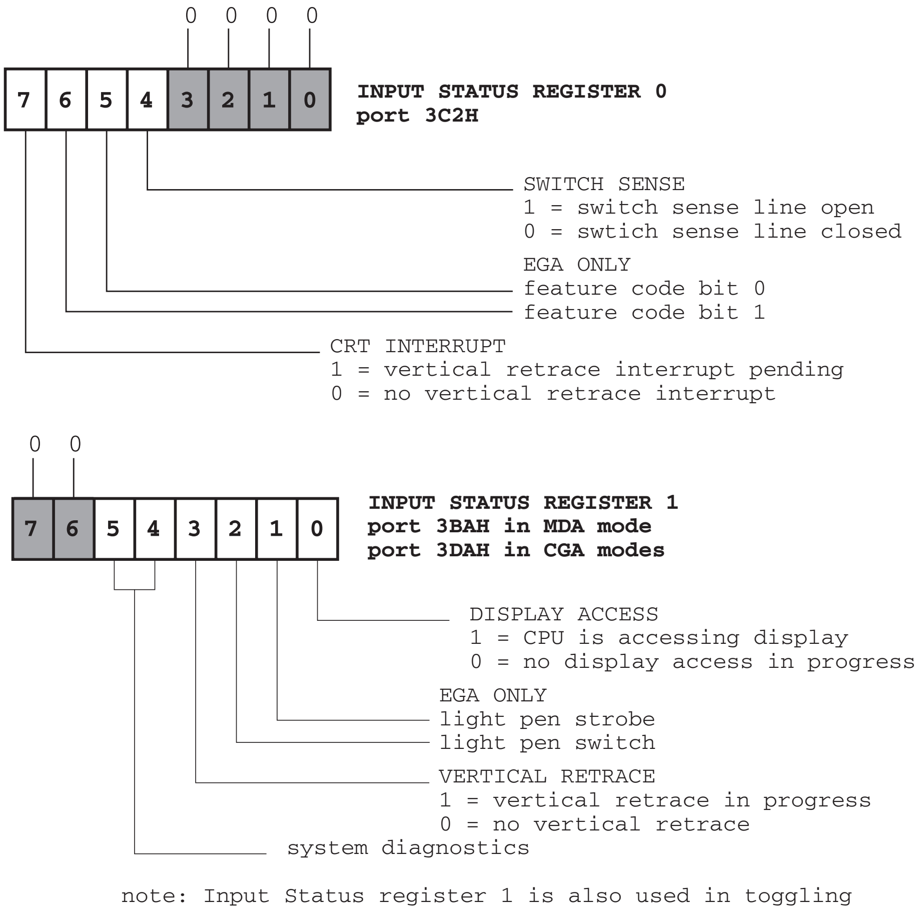


Figure 7-6 *VGA/EGA Miscellaneous Output Register*

Note that bit number 7 of Input Status Register 0, at port 3C2H (see Figure 7-7 on the following page) is used in determining the start of the vertical retrace cycle of the CRT controller. This operation is sometimes necessary to avoid interference when updating the video buffer. The procedure named TIME_VRC, in the VGA module of the GRAPHSOL library, described in Chapter 3, performs this timing operation.



note: Input Status register 1 is also used in toggling the Attribute Controller's internal flip-flop

Figure 7-7 *VGA Input Status Register*

7.3.2 The CRT Controller

The VGA CRT Controller register group is the equivalent of the Motorola 6845 CRT Controller chip of the PC line. When VGA is emulating the MDA, the port address of the CRT Controller is 3B4H; when it is emulating the CGA then the port address is 3D4H. These ports are the same as those used by the MDA and the CGA cards. Table 7-3 lists the registers in the CRT Controller group.

Most registers in the CRT Controller are modified only during mode changes. Since this operation is frequently performed by means of a BIOS service, most programs will not access the CRT Controller registers directly. The exception are the CRT Controller registers related to cursor size and position, which are occasionally programmed directly. The Cursor Size register is shown in Figure 7-8. and the Cursor Location register in Figure 7-9.

Table 7-3

VGA CRT Controller Register

PORT	OFFSET	DESCRIPTION
03x4H		Address register
03x5H	0	Total horizontal characters minus 2 (EGA) Total horizontal characters minus 5 (VGA)
	1	Horizontal display end characters minus 1
	2	Start horizontal blanking
	3	End horizontal blanking
	4	Start horizontal retrace pulse
	5	End horizontal retrace pulse
	6	Total vertical scan lines
	7	CRTC overflow
	8*	Preset row scan
	9	Maximum scan line
	10*	Scan line for cursor start
	11*	Scan line for cursor end
	12*	Video buffer start address, high byte
	13*	Video buffer start address, low byte
	14*	Cursor location, high byte
	15*	Cursor location, low byte
	16	Vertical retrace start
	17	Vertical retrace end
	18	Last scan line of vertical display
	19	Additional word offset to next logical line
	20	Scan line for underline character
	21	Scan line to start vertical blanking
	22	Scan line to end vertical blanking
	23	CRTC mode control
	24	Line compare register

Notes: Registers signaled with (*) are described separately
3x4H/3x5H = 3B4H/3B5H when emulating the MDA
3x4H/3x5H = 3D4H/3D5H when emulating the CGA

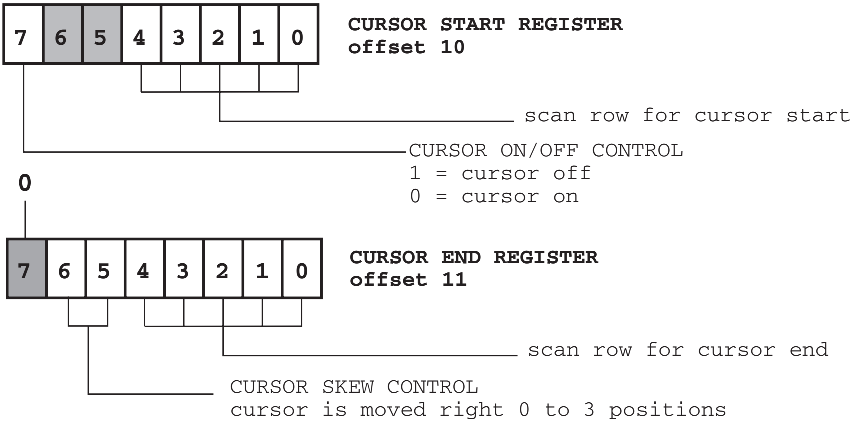


Figure 7-8 *Cursor Size Registers of the VGA CRT Controller*

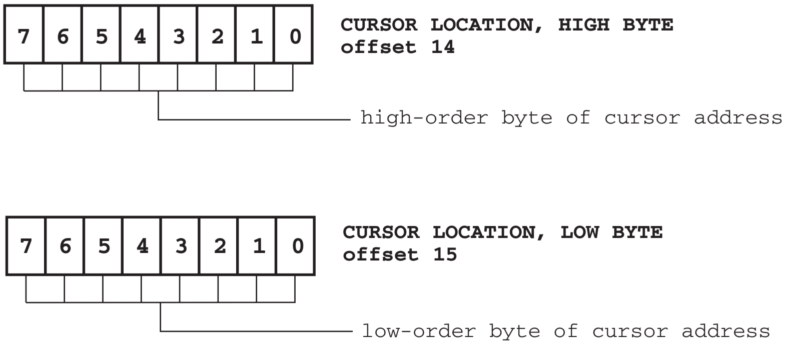

7
6
5
4
3
2
1
0
CURSOR LOCATION, HIGH BYTE
offset 14
high-order byte of cursor address
7
6
5
4
3
2
1
0
CURSOR LOCATION, LOW BYTE
offset 15
low-order byte of cursor address

Figure 7-9 *Cursor Location Registers of the VGA CRT Controller*

Figure 7-10 graphically shows the cursor scan lines and the default setting in a 8 x 14 pixel text mode (see Table 7-1).

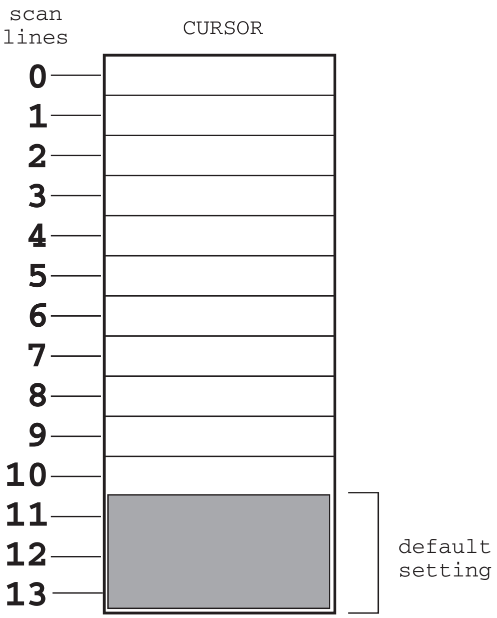


Figure 7-10 *Cursor Scan Lines in VGA Systems*

A program can change the cursor size in alphanumeric modes using service number 1 of BIOS interrupt 10H or by programming the CRT Controller cursor register directly. The use of BIOS service number 10, interrupt 10H, is discussed later in this chapter. The following code fragment shows a sequence of instructions for programming the CRT Controller cursor size registers. The action performed by the code is to change the VGA default cursor in a 8-by-14 text mode from scan lines 12 and 13 to scan lines 1 to 7.

```
        MOV     DX,3B4H         ; VGA CRTC address register
                                ; in the MDA emulation modes
        MOV     AL,10           ; Cursor start register number
        OUT     DX,AL           ; Select this register
        MOV     DX,3B5H         ; CRTC registers
        MOV     AL,1            ; Start scan line for new cursor
        OUT     DX,AL           ; Set in 6845 register
        MOV     DX,3B4H         ; Address register again
        MOV     AL,11           ; Cursor end register number
        OUT     DX,AL           ; Select this register
        MOV     DX,3B5H         ; CRTC registers
        MOV     AL,7            ; End scan line for new cursor
        OUT     DX,AL           ; Set in 6845 register
```

The cursor location on an alphanumeric mode can also be set using a BIOS service or programming the CRT Controller registers directly. BIOS service number 0, interrupt 10H, allows setting the cursor to any desired column and row address. Alternatively the cursor can be repositioned by setting the contents of the cursor address registers on the VGA CRT Controller. The cursor address registers are located at offset 14 and 15, respectively. The following code fragment will position the cursor at the start of the third screen row. The code assumes an 80 x 25 alphanumeric mode in the Monochrome Display Adapter. The offset of the second row is calculated as 80 x 2 = 160 bytes from the start of the adapter RAM. Consequently, the Cursor Address High register must be zeroed and the Cursor Address Low register set to 160.

```
        MOV     DX,3B4H         ; VGA CRTC address register
                                ; in the MDA emulation mode
        MOV     AL,14           ; Cursor Address High register
        OUT     DX,AL           ; Select this register
        MOV     DX,3B5H         ; CRTC registers
        MOV     AL,0            ; Zero high bit of address
        OUT     DX,AL           ; Set in CRTC register
        MOV     DX,3B4H         ; Address register again
        MOV     AL,15           ; Cursor Address Low register
        OUT     DX,AL           ; Select this register
        MOV     DX,3B5H         ; CRTC programmable registers
        MOV     AL,160          ; 160 bytes from adapter start
        OUT     DX,AL           ; Set in 6845 register
; Cursor now set at the start of the third screen row
```

Another group of registers within the CRT Controller that are occasionally programmed directly are those that determine the start address of the screen window in the video buffer. This manipulation is sometimes used in scrolling and panning text and graphics screens. In VGA systems the CRT Controller Start Address High and Start Address Low registers (offset 0CH and 0DH) locate the screen window within a byte offset, while the Preset Row Scan register (offset 08H) locates the window at the closest pixel row. Therefore the Preset Row Scan register is used to determine the vertical pixel offset of the screen window. The horizontal pixel offset of the screen window is programmed by changing the value stored in the Horizontal Pixel Pan register of the Attribute Controller, described later in this chapter. Figure 7-11, on the following page, shows the Start Address registers of the CRT Controller. Figure 7-12, on the following page, is a bitmap of the Preset Row Scan register.

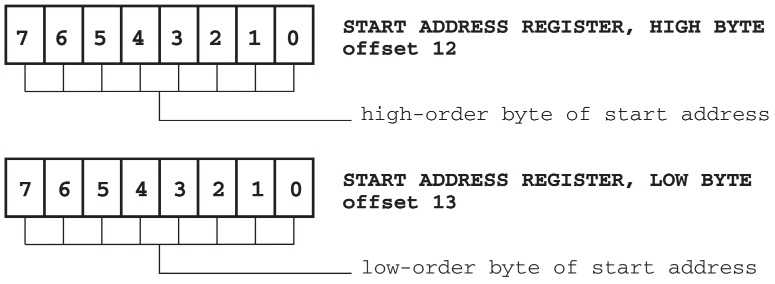

7 6 5 4 3 2 1 0
START ADDRESS REGISTER, HIGH BYTE
offset 12
high-order byte of start address
7 6 5 4 3 2 1 0
START ADDRESS REGISTER, LOW BYTE
offset 13
low-order byte of start address

Figure 7-11 *Video Start Address Register of the VGA CRT Controller*

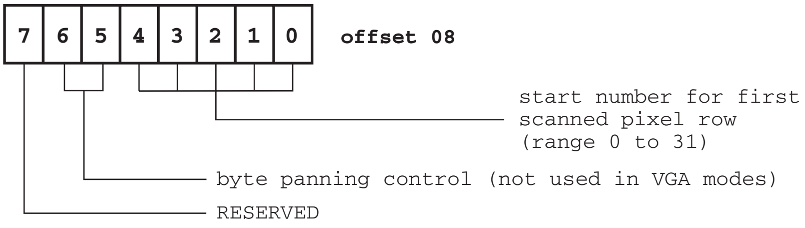


Figure 7-12 *Preset Row Scan Register of the VGA CRT Controller*

7.3.3 The Sequencer

The VGA Sequencer register group controls memory fetch operations and provides timing signals for the dynamic RAMs. This allows the microprocessor to access video memory in cycles inserted between the display memory cycles. Table 7-4 shows the registers in the VGA Sequencer.

Table 7-4

The VGA Sequencer Registers

PORT	OFFSET	DESCRIPTION
03C4H		Address register
03C5H	0	Synchronous or Asynchronous reset
	1	Clocking Mode
	2*	Map Mask
	3*	Character Map Select
	4*	Memory Mode

Note: Registers signaled with an (*) are described separately

The Map Mask register in the Sequencer group allows the protection of any specific memory map by masking it from the microprocessor and from the Character Map select register. Figure 7-13 is a bitmap of the Map Mask register.

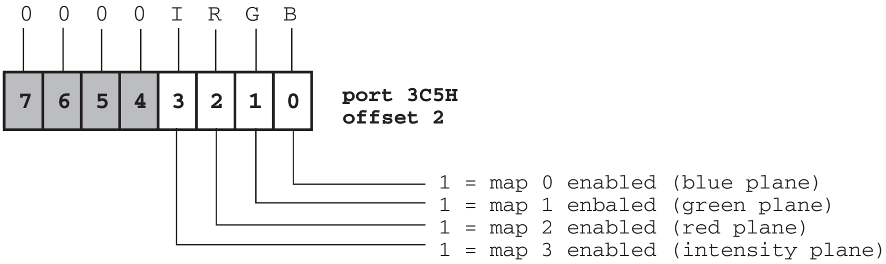
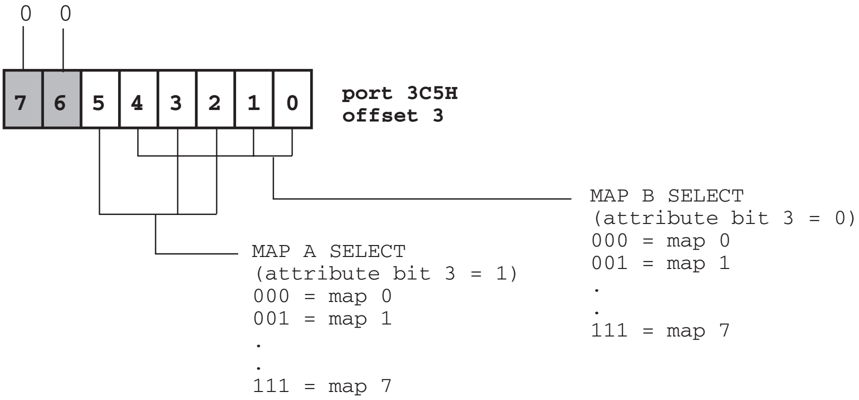

0
0
0
0
I
R
G
B
7
6
5
4
3
2
1
0
port 3C5H
offset 2
1 = map 0 enabled (blue plane)
1 = map 1 enbaled (green plane)
1 = map 2 enabled (red plane)
1 = map 3 enabled (intensity plane)

Figure 7-13 *Map Mask Register of the VGA Sequencer*

If VGA is in a color graphic mode, the Map Mask register can be used to select the color at which one or more pixels are displayed. The color is encoded in the IRGB format, as shown in Figure 7-13. To program the Map Mask register we must first load the value 2 into the address register of the Sequencer, at port 3C4H. This value corresponds with the offset of the Map Mask register (see Table 7-4). After the pixel or pixels have been set, the Map Mask register should be restored to its default value (0FH). The following code fragment shows the usual program operations.

```
; Setting 8 bright-red pixels in VGA mode number 18
; The code assumes that video mode number 18 is selected,
; that ES is set to the video segment base, and that BX points
; to the offset of the first pixel to be set
;***********************|
;    select register    |
;***********************|
        MOV     DX,3C4H         ; Address register of Sequencer
        MOV     AL,2            ; Offset of the Map Mask
        OUT     DX,AL           ; Map Mask selected
        MOV     DX,3C5H         ; Data to Map Mask
        MOV     AL,00001100B    ; Intensity and red bits set
                                ; in IRGB encoding
        OUT     DX,AL           ; Map Mask = 0000 IR00
;***********************|
;       set pixels      |
;***********************|
; Setting the pixels consists of writing a 1 bit in the
; corresponding buffer address.
        MOV     AL,ES:[BX]      ; Dummy read operation
        MOV     AL,11111111B    ; Set all bits
        MOV     ES:[BX],AL      ; Write to video buffer
;***********************|
;   restore Map Mask    |
;***********************|
; Restore the Map Mask to the default state
        MOV     DX,3C4H         ; Address register of Sequencer
        MOV     AL,02H          ; Offset of the Map Mask
        OUT     DX,AL           ; Map Mask selected
        MOV     DX,3C5H         ; Data to Map Mask
        MOV     AL,00001111B    ; Default IRGB code for Map Mask
        OUT     DX,AL           ; Map mask = 0000 IRGB
        .
        .
        .
```



LOCATION OF MAP TABLES

map No.	location	map No.	location
0	1st 8K of map 2	4	2nd 8K of map 2
1	3rd 8K of map 2	5	4th 8K of map 2
2	5th 8K of map 2	6	6th 8K of map 2
3	7th 8K of map 2	7	8th 8K of map 2

Figure 7-14 *Character Map Select Register of the VGA Sequencer*

The use of the Character Map Select register of the Sequencer is related to re-programming of bit 3 of the attribute byte (see Figure 7-3) so that it will serve to select one of two character sets. Normally the character maps, named A and B, have the same value and bit 3 of the attribute byte is used to control the bright or normal display of the character foreground. In this case only one set of 256 characters is available. However, when the Character Map Select register is programmed so that character maps A and B have different values, then bit 3 of the attribute byte is used to toggle between two sets of 256 characters each. The programming operations necessary for using multiple VGA character sets is described in Chapter 3. Figure 7-14, above, is a bitmap of the Character Map Select register.

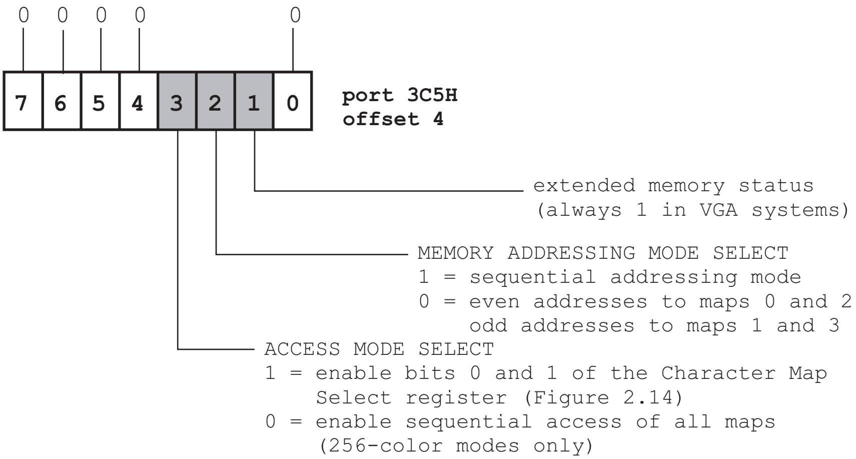

0 0 0 0 0
7 6 5 4 3 2 1 0
port 3C5H
offset 4
extended memory status
(always 1 in VGA systems)
MEMORY ADDRESSING MODE SELECT
1 = sequential addressing mode
0 = even addresses to maps 0 and 2
odd addresses to maps 1 and 3
ACCESS MODE SELECT
1 = enable bits 0 and 1 of the Character Map
Select register (Figure 2.14)
0 = enable sequential access of all maps
(256-color modes only)

Figure 7-15 *Memory Mode Register of the VGA Sequencer*

The Memory Mode register of the sequencer is related to the display modes. Most programs will leave the setting of this register to the BIOS mode select services. Figure 7-15, on the preceeding page, shows a bitmap of the Memory Mode register.

7.3.4 The Graphics Controller

The registers in the Graphics Controller group serve to interface video memory with the Attribute Controller and with the system microprocessor. The Graphic Controller is bypassed in the alphanumeric modes. Table 7-5 lists the registers in the VGA Graphics Controller group. All the registers in the Graphics Controller are of interest to the graphics applications programmer.

Table 7-5

The VGA Graphics Controller Registers

PORT	OFFSET	DESCRIPTION
03CEH		Address register
03CFH	0	Set/Reset
	1	Enable Set/Reset
	2	Color compare for read mode 1 operation
	3	Data rotate
	4	Read operation map select
	5	Select graphics mode
	6	Miscellaneous operations
	7	Read mode 1 color don't care
	8	Bit mask

The Set/Reset register of the Graphics Controller may be used to permanently set or clear a specific bit plane. This operation can be useful if the programmer desires to write a specific color to the entire screen or to disable a color map. The Set/Reset register, shown in Figure 7-16, affects only write mode 0 operations. The use of the Set/Reset register requires the use of the Enable Set/Reset register. Enable Set/Reset determines which of the maps is accessed by the Set/Reset register. This mechanism provides a double-level control over the four maps. The Enable Set/Reset register is shown in Figure 7-17, on the following page.

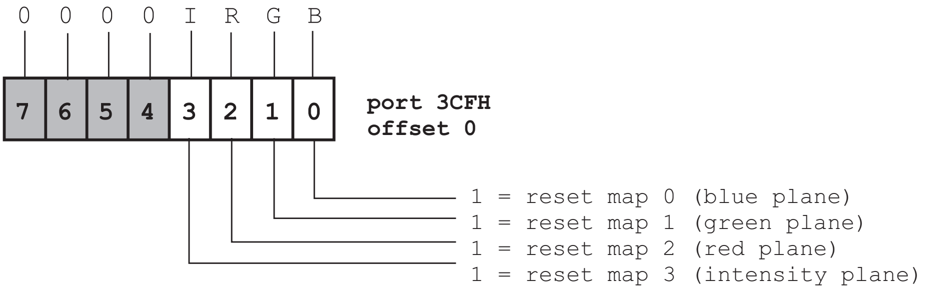
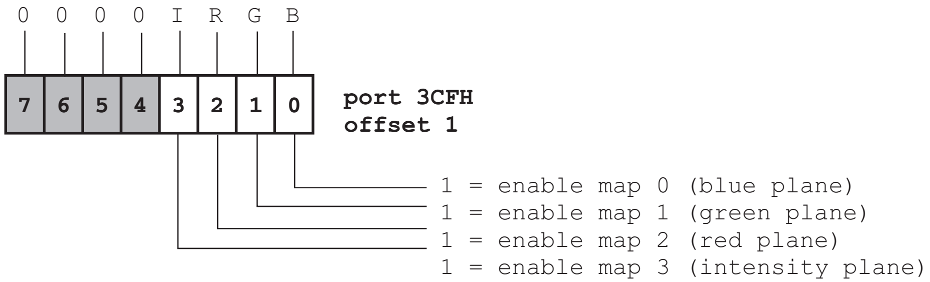

0 0 0 0 I R G B
7 6 5 4 3 2 1 0
port 3CFH
offset 0
1 = reset map 0 (blue plane)
1 = reset map 1 (green plane)
1 = reset map 2 (red plane)
1 = reset map 3 (intensity plane)

Figure 7-16 *Write Mode 0 Set/Reset Register of the VGA Graphics Controller*


0 0 0 0 I R G B
7 6 5 4 3 2 1 0
port 3CFH
offset 1
1 = enable map 0 (blue plane)
1 = enable map 1 (green plane)
1 = enable map 2 (red plane)
1 = enable map 3 (intensity plane)

Note: when set/reset is enabled for a map (bit = 0) it is written with the microprocessor data

Figure 7-17 *Enable Set/Reset Register of the VGA Graphics Controller*

The Color Compare register of the Graphics Controller group, shown in Figure 7-18, is used during read mode 1 operations to test for the presence of memory bits that match one or more color maps. For example, if a program sets bit 0 (blue) and bit 3 (intensity) of the Color Compare register, a subsequent memory read operation will show a 1-value for those pixels whose intensity and blue maps are set, while all other combinations will be reported with a zero value. One or more bit planes can be excluded from the compare by clearing (value equal zero) the corresponding bit in the Color Don't Care register. For example, if the intensity bit is zero in the Color Don't Care register, a color compare operation for the blue bitmap will be positive for all pixels in blue or bright blue color. The Color Don't Care register is shown in Figure 7-19.

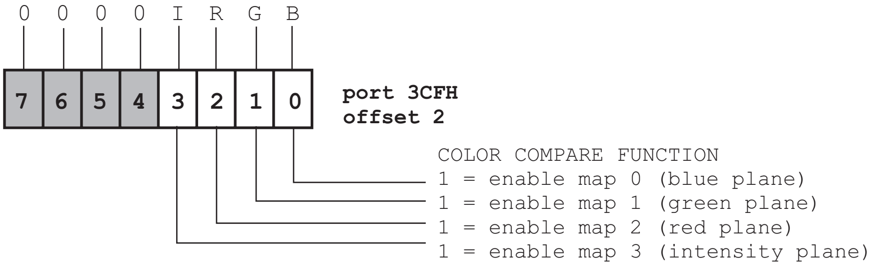

0 0 0 0 I R G B
7 6 5 4 3 2 1 0
port 3CFH
offset 2
COLOR COMPARE FUNCTION
1 = enable map 0 (blue plane)
1 = enable map 1 (green plane)
1 = enable map 2 (red plane)
1 = enable map 3 (intensity plane)

Figure 7-18 *Color Compare Register of the VGA Graphics Controller*

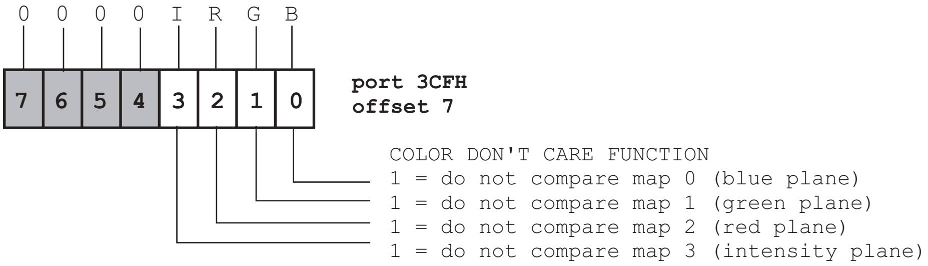

0 0 0 0 I R G B
7 6 5 4 3 2 1 0
port 3CFH
offset 7
COLOR DON'T CARE FUNCTION
1 = do not compare map 0 (blue plane)
1 = do not compare map 1 (green plane)
1 = do not compare map 2 (red plane)
1 = do not compare map 3 (intensity plane)

Figure 7-19 *Color Don't Care Register of the VGA Graphics Controller*

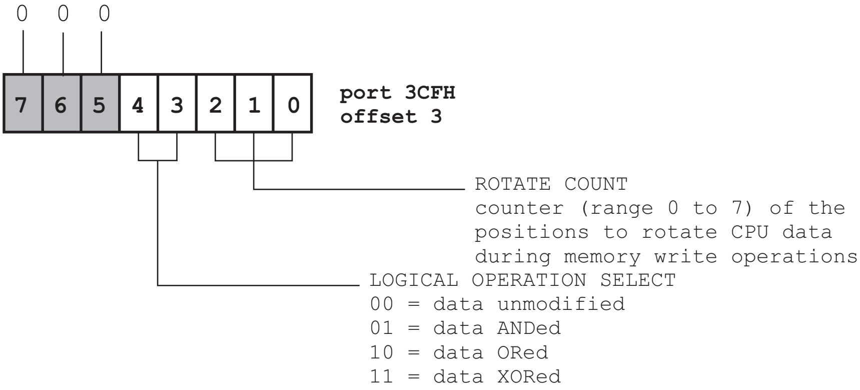


Figure 7-20 *Data Rotate Register of the VGA Graphics Controller*

The Data Rotate register of the Graphics Controller determines how data is combined with data latched in the system microprocessor registers. The possible logical operations are AND, OR, and XOR. If bits 3 and 4 are reset, data is unmodified. A second function of this register is to right-rotate data from 0 to 7 places. This function is controlled by bits 0 to 2. The Data Rotate register is shown in Figure 7-20, above.

We have seen that VGA video memory in the graphics modes is based on encoding the color of a single pixel into several memory maps. The Read Map Select register, in Figure 7-21, is used to determine which map is read by the system microprocessor.

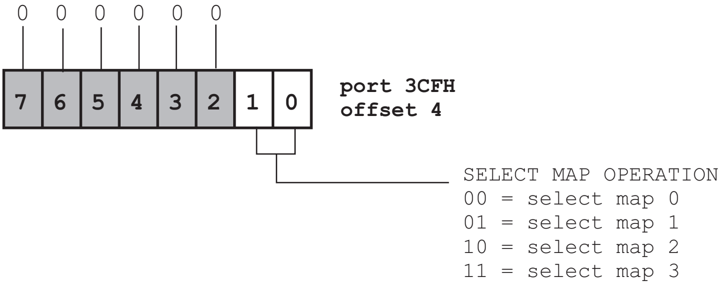


Figure 7-21 *Read Map Select Register of the VGA Graphics Controller*

The following code fragment shows the use of the Read Operation Map Select register.

```
; Code to read the contents of the 4 color maps in VGA mode 18
; Code assumes that read mode 0 has been previously set
; On entry:
;              ES = A000H
;              BX = byte offset into video map
; On exit:
;              CL = byte stored in intensity map
;              CH = byte stored in red map
;                DL = byte stored in green map
;                DH = byte stored in blue map
;
; Set counter and map selector
        MOV     CX,4         ; Counter for 4 maps to read
        MOV     DI,0         ; Map selector code
READ_IRGB:
; Select map from which to read
        MOV     DX,3CEH      ; Graphic Controller Address
                             ; register
        MOV     AL,4         ; Read Operation Map Select
        OUT     DX,AL        ; register
;
        INC     DX           ; Graphic controller at 3CFH
        MOV     AX,DI        ; AL = map selector code (in DI)
        OUT     DX,AL        ; IRGB color map selected
; Read 8 bits from selected map
        MOV     AL,ES:[BX]   ; Get byte from bit plane
        PUSH    AX           ; Store it in the stack
        INC     DI           ; Bump selector to next map
        LOOP    READ_IRGB    ; Execute loop 4 times
; 4 maps are stored in stack
; Retrieve maps into exit registers
        POP     AX           ; B map byte in AL
        MOV     DH,AL        ; Move B map byte to DH
        POP     AX           ; G map byte in AL
        MOV     DL,AL        ; Move G map byte to DL
        POP     AX           ; R map byte in AL
        MOV     CH,AL        ; Move R map byte to CH
        POP     AX           ; I map byte in AL
        MOV     CL,AL        ; Move I map byte to CL
        .
        .
        .
```

VGA systems allow several ways for performing memory read and write operations, usually known as the read and write modes. The Select Graphics Mode register of the Graphics Controller group allows the programmer to select which of two read and four write modes is presently active. The Select Graphics Mode register is shown in Figure 7-22, on the following page.

The four VGA write modes can be described as follows:

- Write mode 0 is the default write mode. In this write mode, the Map Mask register of the Sequencer group, the Bit Mask register of the Graphics Controller group, and the CPU are used to set the screen pixel to a desired color.
- In write mode 1 the contents of the latch registers are first loaded by performing a read operation, then copied directly onto the color maps by performing a write operation. This mode is often used in moving areas of memory.
- Write mode 2, a simplified version of write mode 0, also allows setting an individual pixel to any desired color. However, in write mode 2 the color code is contained in the CPU byte.

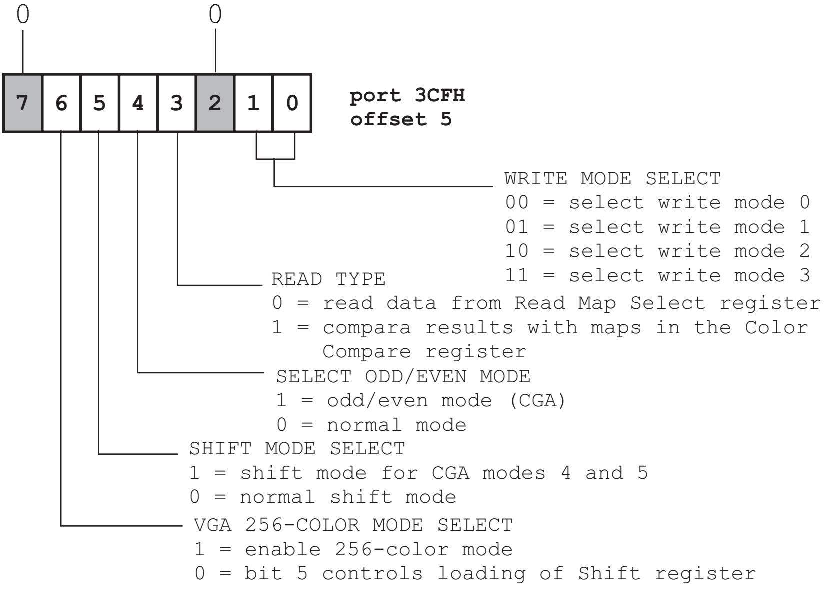


Figure 7-22 *Select Graphics Mode Register of the VGA Graphics Controller*

- In write mode 3 the byte in the CPU is ANDed with the contents of the Bit Mask register of the Graphic Controller.

The write mode is selected by setting bits 0 and 1 of the Graphic Controller's Graphic Mode register. It is a good programming practice to preserve the remaining bits in this register when modifying bits 0 and 1. This is performed by reading the Graphic Mode register, altering the write mode bits, and then re-setting the register without changing the remaining bits. The following code fragment sets a write mode in a VGA system. The remaining bits in the Select Graphics Mode register are preserved.

```
; Set the Graphics Controller's Select Graphic Mode register
; to the write mode in the AH register
          MOV     DX,3CEH          ; Graphic Controller Address
                                   ; register
          MOV     AL,5             ; Offset of the Mode register
          OUT     DX,AL            ; Select this register
          INC     DX               ; Point to Data register
          IN      AL,DX            ; Read register contents
          AND     AL,11111100B     ; Clear bits 0 and 1
          OR      AL,AH            ; Set mode in AL low bits
          MOV     DX,3CEH          ; Address register
          MOV     AL,5             ; Offset of the Mode Register
          OUT     DX,AL            ; Select again
          INC     DX               ; Point to Data register
          OUT     DX,AL            ; Output to Mode Register
; Note: the Select Mode register is read-only in EGA systems
;       therefore this code will not work correctly
```

Note that bit 6 of the Graphics Mode Register must be set for 256-color modes and cleared for the remaining ones. The SET_WRITE_256 procedure in the VGA module of the VGA graphics library (see Chapter 3) sets write mode 0 and the 256-color bit so that VGA mode number 19, in 256 colors, operates correctly.

Once a write mode is selected the program can access video memory to set the desired screen pixels, as in the following code fragment:

```
; Write mode 2 pixel setting routine
; On entry:
;               ES = A000H
;               BX = byte offset into the video buffer
;               AL = pixel color in IRGB format
;               AH = bit pattern to set (mask)
;
; Note: this procedure does not reset the default read or write
; modes or the contents of the Bit Mask register.
; The code assumes that write mode 2 has been set previously
        PUSH    AX          ; Color byte
        PUSH    AX          ; Twice
;**********************|
;      set bit mask    |
;**********************|
; Set Bit Mask register according to value in AH
        MOV     DX,3CEH     ; Graphic controller address
        MOV     AL,8        ; Offset = 8
        OUT     DX,AL       ; Select Bit Mask register
        INC     DX          ; To 3CFH
        POP     AX          ; Color code once from stack
        MOV     AL,AH       ; Bit pattern
        OUT     DX,AL       ; Load bit mask
;**********************|
;    write color       |
;**********************|
        MOV     AL,ES:[BX]  ; Dummy read to load latch
                            ; registers
        POP     AX          ; Restore color code
        MOV     ES:[BX],AL  ; Write the pixel with the
                            ; color code in AL
        .
        .
        .
```

The VGA also provides two read modes. In read mode 0, which is the default read mode, the CPU is loaded with the contents of one of the color maps. In read mode 1, the contents of the maps are compared with a predetermined value before being loaded into the CPU. The active read mode depends on the setting of bit 3 of the Graphic Mode Select register in the Graphics Controller (see Figure 7-22).

The Miscellaneous register of the Graphics Controller, in Figure 7-23, is used in conjunction with the Select Graphics Modes register to enable specific graphics function. Bits 2 and 3 of the Miscellaneous register control the mapping of the video buffer in the system's memory space. The normal mapping of each mode can be seen in the buffer address column of Table 7-1. The manipulation of the Miscellaneous register is usually left to the BIOS mode change service.

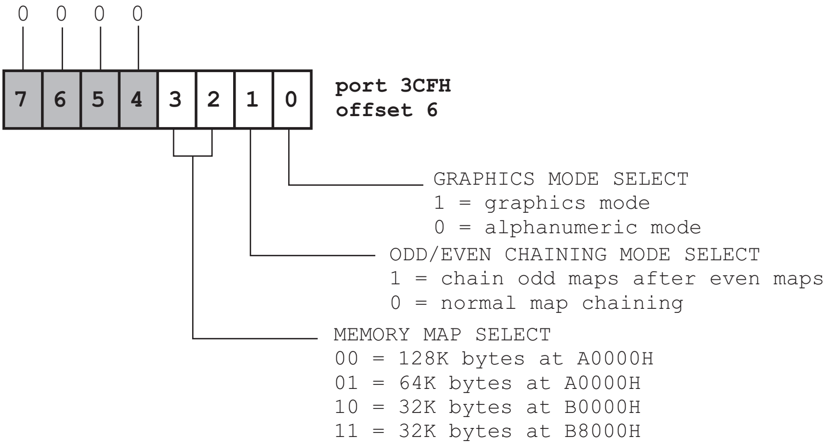


Figure 7-23 *Miscellaneous Register of the VGA Graphics Controller*

All read and write operations performed by the VGA take place at a byte level. However, in certain graphics modes, such as mode number 18, video data is stored at a bit level in four color maps. In this case, the code must mask out the undesired color maps in order to determine the state of an individual screen pixel or to set a pixel to a certain color. In 80x86 Assembly Language the TEST instruction provides a convenient way for determining an individual screen pixel following a read operation. The Bit Mask register of the Graphics Controller, in Figure 7-24, permits setting individual pixels while in write modes 0 and 2.

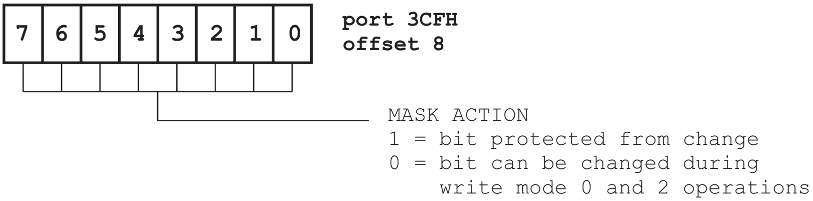

7 6 5 4 3 2 1 0
port 3CFH
offset 8
MASK ACTION
1 = bit protected from change
0 = bit can be changed during
write mode 0 and 2 operations

Figure 7-24 *Bit Mask Register of the VGA Graphics Controller*

In the execution of write operations while in VGA mode number 18, the bit mask for setting and individual screen pixel can be found from a look-up table or by right-shifting a unitary bit pattern (10000000B). The following code fragment calculates the offset into the video buffer and the bit mask required for writing an individual pixel using VGA write modes 0 or 2.

```
; Mask and offset computation from x and y pixel coordinates
; Code is for VGA mode number 18 (640 by 480 pixels)
; On entry:
;               CX = x coordinate of pixel (range 0 to 639)
;               DX = y coordinate of pixel (range 0 to 479)
; On exit:
;               BX = byte offset into video buffer
;               AH = bit mask for the write operation using
;                    write modes 0 or 2
;
;**********************|
;  calculate address   |
;**********************|
        PUSH    AX          ; Save accumulator
        PUSH    CX          ; Save x coordinate
        MOV     AX,DX       ; y coordinate to AX
        MOV     CX,80       ; Multiplier (80 bytes per row)
        MUL     CX          ; AX = y times 80
        MOV     BX,AX       ; Free AX and hold in BX
        POP     AX          ; x coordinate from stack
; Prepare for division
        MOV     CL,8        ; Load divisor
        DIV     CL          ; AX / CL = quotient in AL and
                            ; remainder in AH
; Add in quotient
        MOV     CL,AH       ; Save remainder in CL
        MOV     AH,0        ; Clear high byte
        ADD     BX,AX       ; Offset into buffer to BX
        POP     AX          ; Restore AX
; Compute bit mask from remainder
        MOV     AH,10000000B    ; Unitary mask for 0 remainder
        SHR     AH,CL       ; Shift right CL times
; The byte offset (in BX) and the pixel mask (in AH) can now
; be used to set the individual screen pixel
        .
        .
        .
```

7.3.5 The Attribute Controller

The Attribute Controller receives color data from the Graphics Controller and formats it for the video display hardware. Input to the Attribute Controller, which is in the form of attribute data in the alphanumeric modes and in the form of serialized bit plane data in the graphics modes, is converted into 8-bit digital color output to the DAC. Blinking, underlining, and cursor display logic are also controlled by this register. In VGA systems the output of the Attribute Controller goes directly to the video DAC and the CRT. Table 7-6 shows the registers in the Attribute Controller group.

Table 7-6

The VGA Attribute Controller Registers

PORT	OFFSET	DESCRIPTION
03C0H		Attribute Address and Palette Address register
03C1H		Read operations
03C0H	0 to 15	Palette registers
	16	Attribute mode control
	17	Screen border color control (overscan)
	18	Color plane enable
	19	Horizontal pixel panning
	20	Color select

Register addressing in the Attribute Controller group is performed differently than with the other VGA registers. This is due to the fact that the Attribute Controller does not have a dedicated bit to control the selection of its internal address and data registers, but uses an internal flip-flop to toggle the address and data functions. This explains why the Index and the Data registers of the Attribute Controller are both mapped to port 3C0H (see Table 7-6). Figure 7-25 shows the Attribute and Palette Address registers in the VGA Attribute Controller.

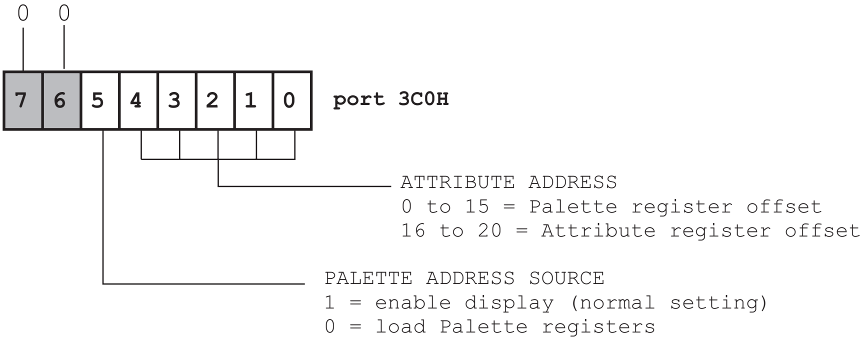

0
0
7
6
5
4
3
2
1
0
port 3C0H
ATTRIBUTE ADDRESS
0 to 15 = Palette register offset
16 to 20 = Attribute register offset
PALETTE ADDRESS SOURCE
1 = enable display (normal setting)
0 = load Palette registers

Figure 7-25 *Attribute Address and Palette Address Registers of the VGA Attribute Controller*

Programming the Attribute Controller requires accessing Input Status Register 1 of the General Register (see Figure 7-7) in order to clear the flip-flop. The address of the Status Register 1 is 3BAH in monochrome modes and 3DAH in color modes. The complete sequence of operations for writing data to the Attribute Controller is as follows:

1. Issue an IN instruction to address 3BAH (in color modes) or to address 3DAH (in monochrome modes) to clear the flip-flop and select the address function of the Attribute Controller.

2. Disable interrupts.

3. Issue an OUT instruction to the address register, at port 3C0H, with the number of the desired data register.

4. Issue another OUT instruction to this same port to load a value into the Data register.

5. Enable interrupts.

The 16 Palette registers of the Attribute Controller, at offsets 0 to 15, determine how the 16 color values in the IRGB bit planes are displayed. The default values for the Palette registers is shown in Table 7-7. The colors of the default palette can be seen by running the program named PALETTE which is part of the book's software package.

Table 7-7

Default Setting of VGA Palette Registers

REGISTER OFFSET	VALUE	BITS 0-5 R G B R G B	COLOR
0	0	0 0 0 0 0 0	Black
1	1	0 0 0 0 0 1	Blue
2	2	0 0 0 0 1 0	Green
3	3	0 0 0 0 1 1	Cyan
4	4	0 0 0 1 0 0	Red
5	5	0 0 0 1 0 1	Magenta
6	20	0 1 0 1 0 0	Brown
7	7	0 0 0 1 1 1	White
8	56	1 1 1 0 0 0	Dark grey
9	57	1 1 1 0 0 1	Light blue
10	58	1 1 1 0 1 0	Light green
11	59	1 1 1 0 1 1	Light cyan
12	60	1 1 1 1 0 0	Light red
13	61	1 1 1 1 0 1	Light magenta
14	62	1 1 1 1 1 0	Yellow
15	63	1 1 1 1 1 1	Intensified white

In VGA systems each Palette register consists of 6 bits that allow 64 color combinations in each register. The bits labeled "RGBRGB" in Table 7-7 correspond to the primary and secondary values for red, green, and blue colors. Since each color is represented by 2 bits, each one can have four possible levels of saturation; for example, the levels of saturation for the color red are:

Saturation	rgbRGB	Interpretation
0	000000	no red
1	100000	low red
2	000100	red
3	100100	high red

The Palette registers can be changed by means of BIOS service number 16, interrupt 10H, or by programming the Attribute Controller registers directly. Note that the setting of the Palette registers does not affect the color output in 256-color mode number 19, in which case the 8-bit color values in video memory are transmitted directly to the DAC. Figure 7-26, on the following page, is a bitmap of the Palette register of the Attribute Controller.

The Attribute Mode Control register of the Attribute Controller serves to select the characteristics associated with the video mode. Bit 0 selects whether the display is in an alphanumeric or in a graphics mode. Bit 1 determines if VGA operates in a monochrome or color emulation. Bit 2 is related to the handling of the ninth screen dot while displaying the graphics characters in the range C0H to DFH. If this bit is set, the graphics characters in this range generate unbroken horizontal lines. This feature refers to the MDA emulation mode only, since other character fonts do not have the ninth dot. BIOS sets this bit automatically in the modes that require it. The function of the bit fields of the Attribute Mode Control register can be seen in Figure 7-27.

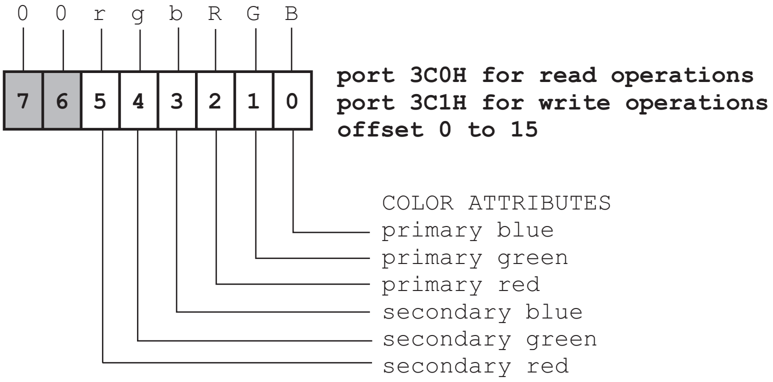


Figure 7-26 *Palette Register of the VGA Attribute Controller*

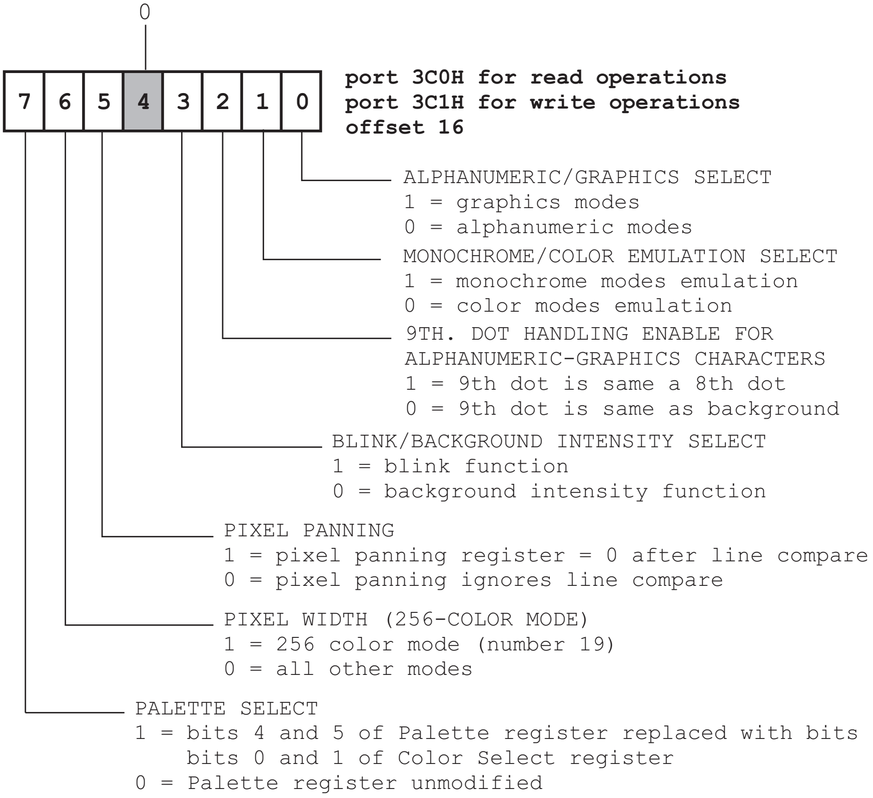

0
7 6 5 4 3 2 1 0
port 3C0H for read operations
port 3C1H for write operations
offset 16
ALPHANUMERIC/GRAPHICS SELECT
1 = graphics modes
0 = alphanumeric modes
MONOCHROME/COLOR EMULATION SELECT
1 = monochrome modes emulation
0 = color modes emulation
9TH. DOT HANDLING ENABLE FOR
ALPHANUMERIC-GRAPHICS CHARACTERS
1 = 9th dot is same a 8th dot
0 = 9th dot is same as background
BLINK/BACKGROUND INTENSITY SELECT
1 = blink function
0 = background intensity function
PIXEL PANNING
1 = pixel panning register = 0 after line compare
0 = pixel panning ignores line compare
PIXEL WIDTH (256-COLOR MODE)
1 = 256 color mode (number 19)
0 = all other modes
PALETTE SELECT
1 = bits 4 and 5 of Palette register replaced with bits
bits 0 and 1 of Color Select register
0 = Palette register unmodified

Figure 7-27 *Attribute Mode Control Register of the VGA Attribute Controller*

Bit 5 of the Attribute Mode Control register in the Attribute Controller group relates to independently panning the screen sections during split-screen operation. Split-screen programming is discussed in Chapter 3. Bit 6 of the Attribute Mode Control register is set to 1 during operation in mode number 19 (256-colors) and cleared for all other modes. Finally, bit 7 of the Attribute Mode Control register determines the source for the bits labeled r and g (numbers 4 and 5) in the Palette register. If bit 7 is set the r and g bits in the Palette register are replaced by bits 0 and 1 of the Color Select register. If bit 7 is reset then all Palette register bits are sent to the DAC.

In some alphanumeric and graphics modes the VGA display area is surrounded by a colored band. The width of this band is the same as the width of a single character (8 pixels) in the 80-column modes. The color of this border area is determined by the Overscan Color register of the Attribute Controller. Normally the screen border is not noticeable, due to the fact that the default border color is black. The border color is not available in the 40-columns alphanumeric modes or in the graphics modes with 320 pixel rows, except for VGA graphics mode number 19. The bitmap of the Overscan register is shown in Figure 7-28.

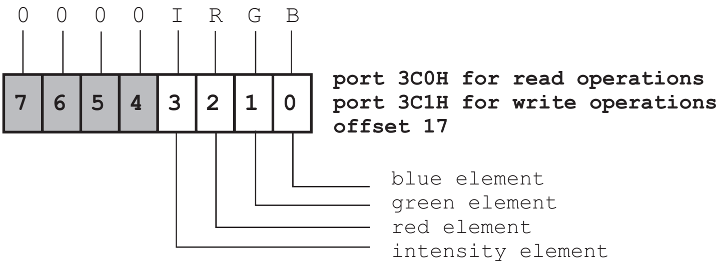


Figure 7-28 *Overscan Color Register of the VGA Attribute Controller*

The Color Plane Enable register allows excluding one or more bit planes from the color generation process. The main purpose of this function is to provide compatibility with EGA systems equipped with less than 256K of memory. Bits 4 and 5 of this register are used in system diagnostics. The bitmap of the Color Plane Enable register of the Attribute Controller group is shown in Figure 7-29.

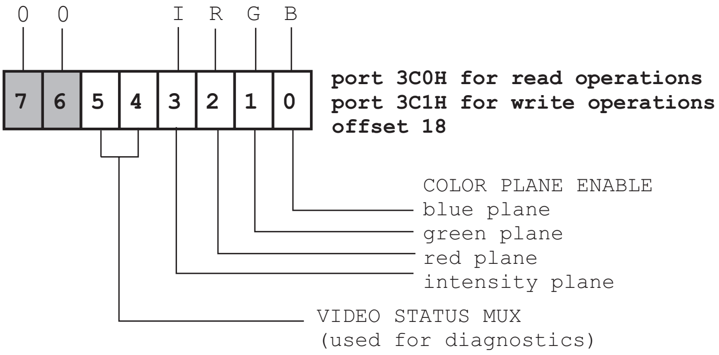


Figure 7-29 *Color Plane Enable Register of the VGA Attribute Controller*

The Horizontal Pixel Panning register of the Attribute Controller is used to shift video data horizontally to the left, pixel by pixel. This register is shown in Figure 7-30. This feature is available in the alphanumeric and graphics modes. The number of pixels that can be shifted is determined by the display mode. In the VGA 256-color graphics mode the maximum number of allowed pixels is three. In alphanumeric modes 0, 1, 2, 3, and 7, the maximum is eight pixels. In all other modes the maximum is seven pixels. The Horizontal Pixel Panning register can be programmed in conjunction with the Video Buffer Start Address registers of the CRT Controller (see Figure 7-11) to implement smooth horizontal screen scrolling in alphanumeric and in graphics modes. These manipulations are described in Chapter 8.

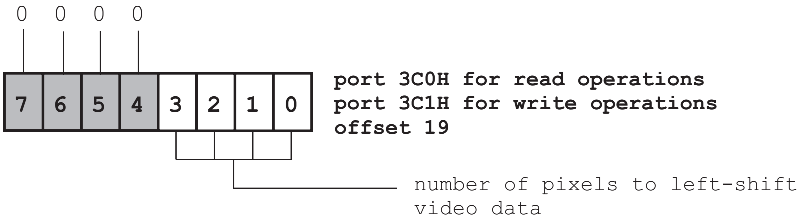

0 0 0 0
7 6 5 4 3 2 1 0
port 3C0H for read operations
port 3C1H for write operations
offset 19
number of pixels to left-shift video data

Figure 7-30 *Horizontal Pixel Panning Register of the VGA Attribute Controller*

The Color Select register of the Attribute Controller provides additional color selection flexibility to the VGA system, as well as a way for rapidly switching between sets of displayed colors. When bit 7 of the Attribute Mode Control register is clear (see Figure 7-27) the 8-bit color value sent to the DAC is formed by the 6 bits from the Palette registers and bits 2 and 3 of the Color Select register (see Figure 7-27). If bit 7 of the Attribute Mode Control register is set, then the 8-bit color value is formed with the lower four bits of the Palette register and the 4 bits of the Color Select register. Since these bits affect all Palette registers simultaneously, the program can rapidly change all colors displayed by changing the value in the Color Select register. The Color Select register is not used in the 256-color graphics mode number 19. The Color Select Register bitmap is shown in Figure 7-31.

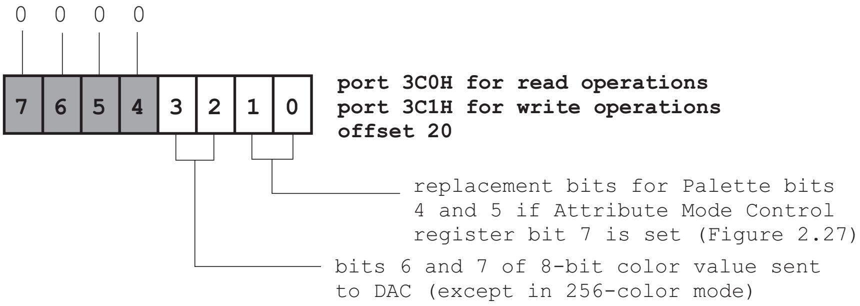

0 0 0 0
7 6 5 4 3 2 1 0
port 3C0H for read operations
port 3C1H for write operations
offset 20
replacement bits for Palette bits 4 and 5 if Attribute Mode Control register bit 7 is set (Figure 2.27)
bits 6 and 7 of 8-bit color value sent to DAC (except in 256-color mode)

Figure 7-31 *Color Select Register of the VGA Attribute Controller*

7.4 The Digital-to-Analog Converter (DAC)

The Digital-to-Analog Converter, or DAC, provides a set of 256 color registers, sometimes called the color look-up table, as well as three color drivers for an analog display. The DAC register set permits displaying 256 color combinations from a total of 262,144 possible colors. Table 7-8 shows the DAC registers.

Table 7-8

VGA Video Digital-to-Analog Converter Addresses

REGISTER	OPERATIONS	ADDRESS
Pixel address (read mode)	WRITE ONLY	03C7H
Pixel address (write mode)	READ/WRITE	03C8H
DAC State	READ ONLY	03C7H
Pixel Data	READ/WRITE	03C9H
Pixel Mask	READ/WRITE	03C6H

Note: applications must not write to the Pixel Mask register
to avoid destroying the color lookup table

Each of the DAC's 256 registers uses 6 data bits to encode the value of the primary colors red, green, and blue. The use of 6 bits per color makes each DAC register 18 bits wide. It is the possible combinations of 18 bits that allow 262,144 DAC colors. Note that the VGA color registers in the DAC duplicate the color control offered by the Palette registers of the Attribute Controller. In fact, the VGA Palette registers are provided for compatibility with the EGA card, which does not contain DAC registers. When compatibility with the EGA is not an issue, VGA programming can be simplified by ignoring the Palette registers and making all color manipulations in the DAC. Furthermore, the Palette registers are disabled when VGA is in the 256-color mode number 19, since mode number 19 has no EGA equivalent.

7.4.1 The DAC Pixel Address Register

The DAC Pixel Address register holds the number (often called the address) of one of the 256 DAC registers. Read operations to the Pixel Address register are performed to port 3C7H and write operations to port 3C8H (see Table 7-8). A write operation changes the 18-bit color stored in the register (in Red/Green/Blue format). A read operation is used to obtain the RGB value currently stored in the DAC register. Figure 7-32 is a bitmap of the DAC Pixel Address register.

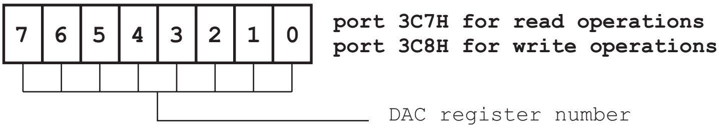


Figure 7-32 *Pixel Address Register of the VGA DAC*

7.4.2 The DAC State Register

The DAC State register encodes whether the DAC is in read or write mode. A mode change takes place when the Pixel Address register is accessed: if the Pixel Address register is set at port 3C7H (see Figure 7-32) then the DAC goes into a read mode; if it is set at port 3C8H then the DAC goes into a write mode. The DAC State register is shown in Figure 7-33. Notice that although the Pixel Address register for read operations and the DAC State register are both mapped to port 3C7H there is no occasion for conflict, since the DAC State register is read only and the Pixel Address register for read operations is write only (see Table 7-8).

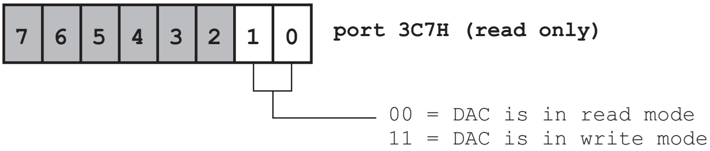


Figure 7-33 *State Register of the VGA DAC*

7.4.3 The DAC Pixel Data Register

The Pixel Data register in the DAC is used to hold three 6-bit data items representing a color value in RGB format. The Pixel Data register can be read after the program has selected the corresponding DAC register at the Pixel Address read operation port 3C7H. The Pixel Data register can be written after the program has selected the corresponding DAC register at the Pixel Address write operation port 3C8H (see Table 7-8). The current read or write state of the DAC can be determined by examining the DAC State register.

Once the DAC is in a particular mode (read or write), an application can continue accessing the color registers by performing a sequence of three operations, one for each RGB value. The read sequence consists of selecting the desired DAC register in the Pixel Address register at the read operations port (3C7H) then performing three consecutive IN instructions. The first one will load the 6-bit red value stored in the DAC register, the second one will load the green value, and the third one the blue value. The write sequence takes place in a similar fashion. This mode of operation allows rapid access to the three data items stored in each DAC register as well as to consecutive DAC registers. Because each entry in the DAC registers is 6 bits wide, the write operation is performed using the least significant 6 bits of each byte. The order of operations for the WRITE function are as follows:

1. Select the starting DAC color register number by means of a write operation to the Pixel Address write mode register at port 3C8H.
2. Disable interrupts.
3. Write the 18-bit color code in RGB encoding. The write sequence consists of 3 bytes consecutively output to the pixel data register. Only the six low-order bits in each byte are meaningful.

4. The DAC transfers the contents of the Pixel Data register to the DAC register number stored at the Pixel Address register.

5. The Pixel Address register increments automatically to point to the subsequent DAC register. Therefore, if more than one color is to be changed, the sequence of operations can be repeated from step number 3.

6. Re-enable interrupts.

Read or write operations to the video DAC must be spaced 240 nanoseconds apart. Assembly language code can meet this timing requirement by inserting a short JMP instruction between successive IN or OUT opcodes. The instruction can be conveniently coded in this manner:

```
     JMP        SHORT $+2     ; I/O delay
```

Chapter 8

VGA Device Drivers

Topics:

- VGA programming levels
- Developing VGA device driver routines
- Video memory address calculations
- Setting pixels and tiles
- Reading pixel values
- Color manipulations

This chapter describes the various levels at which the VGA system can be programmed and establishes the difference between device driver and graphics primitive routines. Section 8.2 and following refer to the design and coding of device drivers for calculating pixel address at the fine- and course-grain levels and for reading and writing pixels and pixel tiles. Section 8.3 and following discuss color operations in 16- and 256-color modes.

8.1 Levels of VGA Programming

Because the VGA system provides all the video functions in an IBM microcomputer, any display programming operations on these machines must inevitably access the VGA hardware or its memory space. However, at the higher levels of VGA programming many of the programming details are hidden by the interface software. For example, a programmer working in Microsoft QuickBASIC has available a collection of program functions that allows drawing lines, boxes, circles, and ellipses, changing palette colors, performing fill operations, and even executing some primitive animation. Therefore the QuickBASIC programmer can perform all of the above-mentioned graphics functions while ignoring the complications of VGA registers, video memory mapping, and DAC output.

The programming levels in an IBM microcomputer equipped with VGA video are as follows:

1. VGA services provided by the operating system. This includes the video services in BIOS, MS DOS, OS/2, WINDOWS, or other operating system programs or graphical environments.
2. VGA services provided by high-level languages and by programming libraries that extend the functions of high-level languages.
3. General purpose VGA libraries that can be used directly or interfaced with one or more high-level languages. The VGA graphics library furnished with this book belongs to this category.
4. Low-level routines, usually coded in 80x86 Assembly Language, that access the VGA or DAC registers or the memory space reserved for video functions.

Observe that this list refers exclusively to the VGA system. Other graphics standards, such as 8514A, XGA, and SuperVGA, include high-level functions that are furnished as a programming interface with the hardware. However, the VGA standard does not furnish higher level programming facilities. In this chapter we discuss the lowest level of VGA programming, principally at the adapter hardware level (number 4 in the previous list). These lowest level services are often called device driver routines. The VGA services in the BIOS are also mentioned occasionally. The reader wishing a greater detail in the programming descriptions should refer to the code listings (files with the extension .ASM) that are contained in the book's libraries, which describe the VGA services in the BIOS. In Chapter 9 we extend the discussion of VGA programming to higher level routines, usually called graphics primitives. The VGA services in high-level languages, in operating systems, or in graphical environments, such as WINDOWS and OS/2, are not discussed in the book.

8.1.1 Device Drivers and Primitive Routines

The term device driver is often used to denote the most elementary software elements that serve to isolate the operating system, or the high- and low-level programs, from the peculiarities of hardware devices and peripherals. It was the UNIX operating system that introduced the concept of an installable device driver. In UNIX a device driver is described as a software element that can be attached to the UNIX kernel at any time. The concept of a device driver was perpetuated by MS DOS (starting with version 2.0) and by OS/2.

A second level of graphics routines, usually more elaborate than the device drivers, is called the graphics primitives. For example, to draw a circular arc on the graphics screen of a VGA system we need to perform programming operations at two different levels. The higher level operation consists of calculating the x and y coordinates of the points that lay along the arc to be drawn. The second, and more elementary operation, is to set to a desired color the screen pixels that lay along this arc. In this case we can say that the coordinate calculations for the arc are performed in a routine called a graphics primitive, while the setting of the individual screen pixels is left to a VGA device driver.

Strictly speaking it is possible to reduce the device driver for a VGA graphic system to two routines: one to set to a predetermined color the screen pixel located at certain coordinates, and another one to read the color of a screen pixel. With the support of this simple, two-function driver, it is possible to develop VGA primitives to perform all the graphic functions of which the device is capable. Nevertheless, a system based on minimal drivers performs very poorly. For instance, a routine to fill a screen area with a certain color would have to make as many calls to the driver as there are pixels in the area to be filled. In practice, it is better to develop device drivers that perform more than minimum functions. Therefore, in addition to the pixel read and write services, it is convenient to include in the device driver category other elementary routines such as those that perform address calculations, read and write data in multi-pixel units, and manipulate the color settings at the system level.

In IBM microcomputers, under MS DOS, the VGA graphics hardware is accessed by device drivers that are not installed as part of the operating system. Several interface mechanisms are possible for these drivers. One option is to link the graphics device driver to a software interrupt. Once this driver is loaded and its vector initialized, applications can access its services by means of the INT instruction. But this type of operation, while very convenient and efficient, requires that the driver be installed as a terminate-and-stay-resident program (TSR), therefore reducing the memory available to applications. An alternative way of making the services of graphics device drivers accessible to applications is to include the drivers in one or more graphics libraries. The library routines requested in the code, which are accessed by high- and low-level programs at link time, are incorporated into the program's run file. Because of its simplicity this is the approach selected for the graphics routines provided with this book. Chapter 9 is devoted to developing the primitive routines necessary in VGA programming.

8.2 Developing the VGA Device Drivers

The VGA system can be considered as a different device in each operational mode. In fact, many VGA modes exist for no other reason than to provide compatibility with other devices. Therefore, the device drivers for VGA mode number 18, with 640-by-480 pixels in 16 colors, are unrelated and incompatible with VGA mode number 19, with 320-by-200 pixels in 256 colors. Since these two modes (numbers 18 and 19) provide the most powerful graphics functions in the VGA standard, and considering that compatibility with previous adapters is no longer a major consideration, the drivers developed for this book apply to VGA modes number 18 and 19 only.

8.2.1 VGA Mode 18 Write Pixel Routine

In VGA mode number 18 each screen pixel is mapped to four memory maps, each map encoding the colors red, green, and blue, as well as the intensity component, as shown in Figure 8-1, on the following page.

To set a screen pixel in VGA mode number 18 the program must access individual bits located in four color maps. In Figure 8-1 the screen pixel displayed corresponds to the first bit in each of the four maps. But, due to the fact that the 80x86 instruction set does not contain operations for accessing individual bits, read and write operations in 80x86 Assembly Language must take place at the byte level. Consequently, to access the individual screen pixels while in VGA mode number 18 the program has to resort to bit masking. Figure 8-2 illustrates bit-to-pixel mapping in VGA mode number 18.

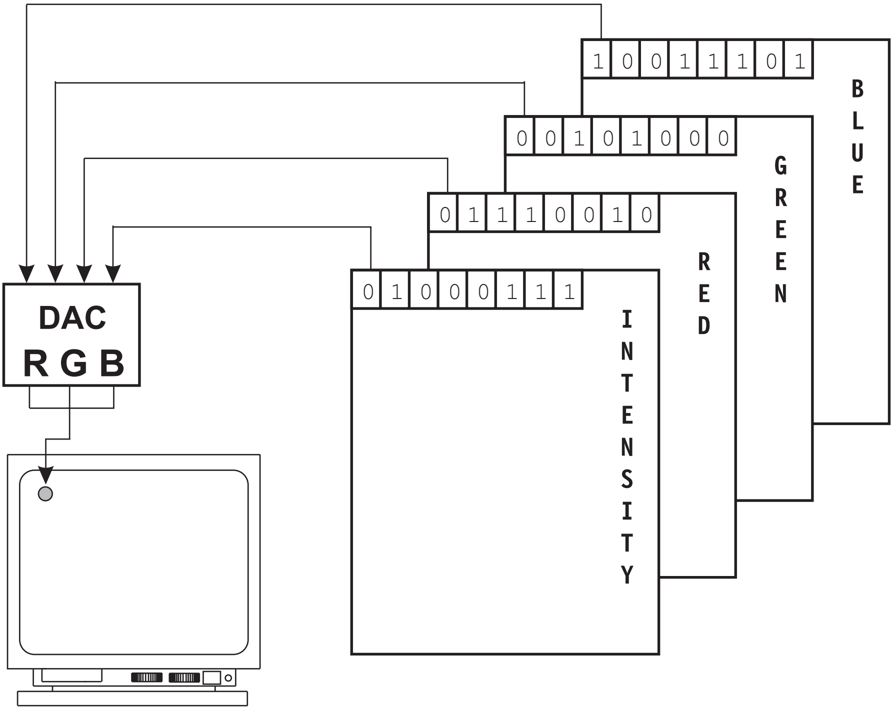


Figure 8-1 *Color Maps in VGA Mode 18*

Notice in Figure 8-2 that the eleventh screen pixel (pointed at by the arrow) corresponds to the eleventh bit in the memory map. This eleventh bit is located in the second byte.

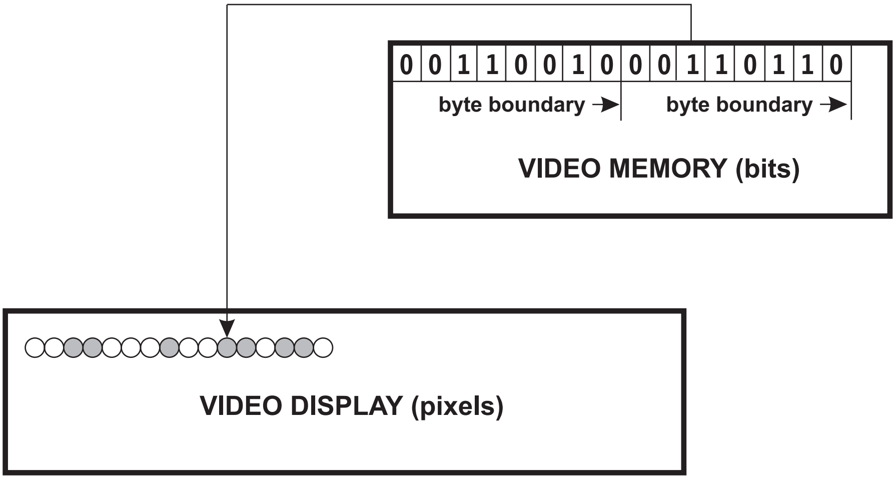


Figure 8-2 *Bit-to-Pixel Mapping Example in VGA Mode 18*

VGA write operations can take place in four different write modes, labeled 0 to 3. Also that the write mode is selected by means of bits 0 and 1 of the Select Graphics Mode register of the Graphics Controller group (see Figure 2-22). The VGA behaves as a different device in each write mode. Therefore the device driver for a pixel write operation in mode number 18 must be write-mode specific.

Each VGA write mode has its strong points but, perhaps, write mode 2 is the most direct and useful one. In write mode 2 the individual pixel within a video buffer byte is selected by entering an appropriate mask in the Bit Mask register of the Graphics Controller group. This bit mask must contain a 1 bit for the pixel or pixels to be accessed and a 0 bit for those to be ignored. For example, the bit mask 00100000B can be used to select the pixel shown in Figure 8-2.

Fine Grain Address Calculations

In the case of Figure 8-2 the code must take into account that the 11 pixel is located in the second buffer byte. In VGA mode number 18 programming this is usually accomplished by using a word-size variable, or an 80x86 machine register, as an offset pointer. Since the VGA video buffer in a graphics mode always starts at physical address A0000H, the ES register can be set to the corresponding segment base. The Assembly Language code to set the ES:BX register pair as a pointer to the second screen byte would be as follows:

```
; Code fragment to set the 11th screen pixel while in VGA mode
; number 18, write mode 2
     MOV  AX,0A000H ; Segment base for video buffer
     MOV  ES,AX          ; To ES register
; ES --> base of VGA video buffer
     MOV  BX,1      ; Offset of byte 2 to BX
; At this point ES:BX can be used to access the second byte in the
; video buffer
     .
     .
     .
```

In practice a VGA mode number 18 device driver should include a routine to calculate the pixel's byte offset and bit mask from its screen coordinates. The actual calculations are based on the geometry of the video buffer in this mode, which corresponds to 80 bytes per screen row (640 pixels) and a total of 480 rows. The following code fragment shows the necessary calculations.

```
; Address computation from x and y pixel coordinates
; On entry:
;         CX = x coordinate of pixel (range 0 to 639)
;         DX = y coordinate of pixel (range 0 to 479)
; On exit:
;         BX = byte offset into video buffer
;         AH = bit mask for the write VGA write modes 0 or 2
;         AL is preserved
; Save all entry registers
     PUSH CX
     PUSH DX
;***********************|
;   calculate address   |
;***********************|
         PUSH    AX          ; Save accumulator
         PUSH    CX          ; Save x coordinate
         MOV     AX,DX       ; y coordinate to AX
         MOV     CX,80       ; Multiplier (80 bytes per row)
         MUL     CX          ; AX = y times 80
         MOV     BX,AX       ; Free AX and hold in BX
         POP     AX          ; x coordinate from stack
; Prepare for division
         MOV     CL,8        ; Divisor
         DIV     CL          ; AX / CL = quotient in AL and
                             ; remainder in AH
; Add in quotient
         MOV     CL,AH       ; Save remainder in CL
         MOV     AH,0        ; Clear high byte
         ADD     BX,AX       ; Offset into buffer to BX
         POP     AX          ; Restore AX
;***********************|
;   calculate bit mask  |
;***********************|
; The remainder (in CL) is used to shift a unitary mask
         MOV     AH,10000000B    ; Unit mask for 0 remainder
         SHR     AH,CL       ; Shift right CL times
; Restore registers
         POP     DX
         POP     CX
     .
     .
     .
```

This address calculation routine is similar to the PIXEL_ADD_18 device driver in the VGA1 module of the graphics library furnished with this book. This library service is discussed in Section 3.3.

Setting the Pixel

Once the bit mask and byte offset into the buffer have been determined, the individual screen pixel can be set in VGA mode number 18, write mode 2. This is accomplished in two steps: first the program sets the mask in the Bit Mask register of the Graphics Controller group, then it performs a memory write operation to the address in ES:BX. The following code fragment shows both operations.

```
; VGA mode number 18 device driver for writing an individual
; pixel to the graphics screen
;
; On entry:
;               ES:BX = byte offset into the video buffer
;               AL = pixel color in IRGB format
;               AH = bit pattern to set (mask)
;
; This routine assumes VGA mode 18 and write mode 2
;
         PUSH    DX          ; Save outer loop counter
         PUSH    AX          ; Color byte
         PUSH    AX          ; Twice
;***********************|
;     first step        |
;    set bit mask       |
;***********************|
; Set Bit Mask Register according to mask in AH
```

```
         MOV     DX,3CEH      ; Graphic controller latch
         MOV     AL,8
         OUT     DX,AL        ; Select data register 8
         INC     DX           ; To 3CFH
         POP     AX           ; AX once from stack
         MOV     AL,AH        ; Bit pattern
         OUT     DX,AL        ; Load bit mask
;**********************|
;      second step:    |
;   write IRGB color   |
;**********************|
; Write color code to memory maps
         MOV     AL,ES:[BX]   ; Dummy read to load latch
                              ; registers
         POP     AX           ; Restore color code
         MOV     ES:[BX],AL   ; Write the pixel with the
                              ; color code in AL
         POP     DX           ; Restore outer loop counter
     .
     .
     .
```

The above code is similar to the one in the WRITE_PIX_18 device driver listed in the VGA1 module of the graphics library furnished with this book. The WRITE_PIX_18 routine is discussed in Section 3.3.

Coarse Grain Address Calculations

The finest possible degree of control over a video display device is obtained at the screen pixel level. However, it is often convenient to access video display device in units of several pixels. For example, when VGA mode number 18 text display operations are performed by means of the BIOS character display services, these take place on a screen divided into 80 character columns and 30 character rows (see Table 2-2). This means that each character column is 8 pixels wide (640/80 = 8) and each row is 16 pixels high (480/30 = 16). In addition, graphics software can often benefit from operations that take place at coarser-than-pixel levels. For instance, to draw a horizontal line from screen border to screen border, in mode number 18, requires 640 bit-level operations, but only 80 byte-level operations. Consequently, routines that read or write pixels in groups achieve substantially better performance than those that read or write the pixels individually.

When referring to VGA mode 18 routines that write to the video display at a byte level we use the term coarse grain, while those that output at the pixel we labeled fine grain. In order to give the coarse-grain routine a symmetrical pixel pattern, we have used 8-bit pixel groups both on the horizontal and on the vertical scale. For lack of a better word we refer to these 8-by-8 pixel units as screen tiles, or simply tiles. Coarse-grain operations, in mode number 18, see the video display as 80 columns and 60 rows of screen tiles, for a total of 4800 tiles. In this manner the programmer can envision the VGA screen in mode number 18 as consisting of 640-by-480 pixels (fine-grain visualization) or as consisting of 80-by-60 screen tiles of 8-by-8 pixels (coarse-grain visualization). Furthermore, the coarse-grain visualization can easily be adapted to text display operations on an 80-by-30 screen by grouping the 60 tile rows into pairs. The following code fragment calculates the coarse-grain offset into the video buffer from the vertical and horizontal tile count.

```
; On entry:
;         CH = horizontal tile number (range 0 to 79) = x coordinate
;         CL = vertical tile number (range 0 to 59) = y coordinate
;
; Compute coarse-grain address (in BX) as follows:
;    BX = (CL * 640) + CH
;
; On exit:
;          BX = tile offset into video buffer
;          CX is destroyed
;
     PUSH AX       ; Save accumulator
     PUSH DX       ; For word multiply
     PUSH CX       ; To save CH for addition
     MOV  AX,CX         ; Copy CX in AX
; AL = CL
     MOV  AH,0     ; Clear high byte
     MOV  CX,640        ; Multiplier
     MUL  CX       ; AX * CX results in AX
; The multiplier (640) is the product of 80 tiles columns
; times 8 vertical pixels in each tile row
     POP  CX       ; Restore CH
     POP  DX       ; and DX
     MOV  CL,CH         ; Prepare to add in CH
     MOV  CH,0
     ADD  AX,CX         ; Add
     MOV  BX,AX         ; Move sum to BX
     POP  AX       ; Restore accumulator
     .
     .
     .
```

The above code is similar to the one in the TILE_ADD_18 device driver listed in the VGA1 module of the graphics library furnished with this book.

Setting the Tile

Once the tile address has been determined, the individual tile (8-by-8 pixel groups) can be set by placing an all-ones mask in the Bit Mask register of the Graphics Controller group, and then performing write operations to 8 successive pixel rows. The following code fragment shows the setting of a screen tile.

```
; Set Bit Mask Register to all one bits
        MOV     DX,3CEH     ; Graphic controller latch
        MOV     AL,8
        OUT     DX,AL       ; Select data register 8
        INC     DX          ; To 3CFH
        MOV     AL,0FFH     ; Bit pattern of all ones
      +  OUT     DX,AL       ; Load bit mask
; Set counter for 8 pixel rows
        MOV     CX,8        ; Counter initialized
        POP     AX          ; Restore color code
;**********************|
;     set 8 pixels     |
;**********************|
SET_EIGHT:
        MOV     AH,ES:[BX]  ; Dummy read to load latch
                            ; registers
        MOV     ES:[BX],AL  ; Write the pixel with the
                            ; color code in AL
        ADD     BX,80         ; Index to next row
        LOOP    SET_EIGHT
; Tile is set
```

The above code is similar to the one in the WRITE_TILE_18 device driver listed in the VGA1 module of the graphics library furnished with this book. The WRITE_TILE_18 routine is discussed in Section 3.3.

8.2.2 VGA Mode 18 Read Pixel Routine

A program attempting to determine the state of the 11 pixel in Figure 8-2 would read the second memory byte and mask out all other bits. The mask, in this case, would have the value 00100000B. We have seen that video memory in VGA mode number 18 is divided into four memory maps, labeled I, R, G, and B for the intensity, red, green, and blue components, respectively, and that all four maps are located at the same address. For this reason, in order to read the color code for an individual pixel, the program must successively select each of the four memory maps. This is done through the Read Operation Map Select register of the Graphics Controller (see Figure 2-21). In other words, to determine the color of a single pixel in VGA mode number 18 it is necessary to perform four separate read operations, one for each of the IRGB maps.

As in the write operation, the code to read a screen pixel must calculate the address of the video buffer byte in which the bit is located and the bit mask for isolating it. This can be done by means of the code listed in Section 3.1.1 or by using the PIXEL_ADD_18 device driver in the VGA1 module of the graphics library furnished with the book (see Section 3.3). The following code fragment reads a screen pixel and returns the IRGB color value in the CL register.

```
; On entry:
;               ES:BX = byte offset into the video buffer
;               AH = bit pattern for mask
;
; On exit:
;               CL = 4 low bits hold pixel color in IRGB format
;               CH = 0
;
; The code assumes that read mode 0 is set
;
; Move bit mask to CH
         MOV     CH,AH        ; CH = bit mask for pixel
;***********************|
;   set-up read loop    |
;***********************|
         MOV     AH,3         ; Counter for 4 color maps
         MOV     CL,0         ; Clear register for pixel color
                              ; return
;***********************|
; execute 4 read cycles |
;***********************|
; AH has number for current IRGB map (range 0 to 3)
READ_MAPS:
; Select map from which to read
         MOV     DX,3CEH      ; Graphic Controller Address
                              ; register
         MOV     AL,4         ; Read Map Select register
         OUT     DX,AL        ; Activate
         INC     DX           ; Graphic Controller = 3CFH
         MOV     AL,AH        ; AL = color map number
         OUT     DX,AL        ; IRGB color map selected
;***********************|
;      read one byte    |
;***********************|
; Read 8 bits from selected map
         MOV     AL,ES:[BX]   ; Get byte from bit plane
;***********************|
; shift return register |
;***********************|
; Previous color code is in bit 0. The shift operation will free
; the low order bit and move previous bit codes to higher positions
         SHL     CL,1
;**********************|
;    mask out pixels   |
;**********************|
         AND     AL,CH        ; Pixel mask in CH
         JZ      NO_PIX_SET   ; Jump if no pixel in map
; Pixel was set in bitmap
         OR      CL,00000001B     ; Set bit 0 in pixel color
                                  ; return register
NO_PIX_SET:
         DEC     AH           ; Bump counter to next map
         JNZ     READ_MAPS    ; Continue if not last map
; 4 low bits in CL hold pixel color in IRGB format
         MOV     CH,0         ; Clear CH
     .
     .
     .
```

The above code is similar to the one in the READ_PIX_18 device driver listed in the VGA1 module of the graphics library furnished with this book.

8.2.3 VGA Mode 19 Write Pixel Routine

VGA programmers use mode number 19 when screen color range is more important than definition. In mode number 19 the VGA video display consists of 200 pixel rows of 320 pixels each. Each pixel, which can be in one of 256 colors, is determined by 1 byte in the video buffer. This scheme can be seen in Figure 8-3.

The fact that each screen pixel in mode number 19 is mapped to a video buffer byte simplifies programming by eliminating the need for a bit mask. The VGA video buffer in mode number 19 consists of 64,000 bytes. This number is the total pixel count obtained by multiplying the number of pixels per row by the number of screen rows (320 * 200 = 64,000). Although the 64,000 buffer bytes are distributed in the 4 bit planes, the VGA hardware makes it appear to the programmer as if they resided in a continuous memory area. In this manner, the top-left screen pixel is mapped to the byte at physical address A0000H, the next pixel on the top screen row is mapped to buffer address A0001H, and so forth. This byte-to-pixel mapping scheme can be seen in Figure 8-4.

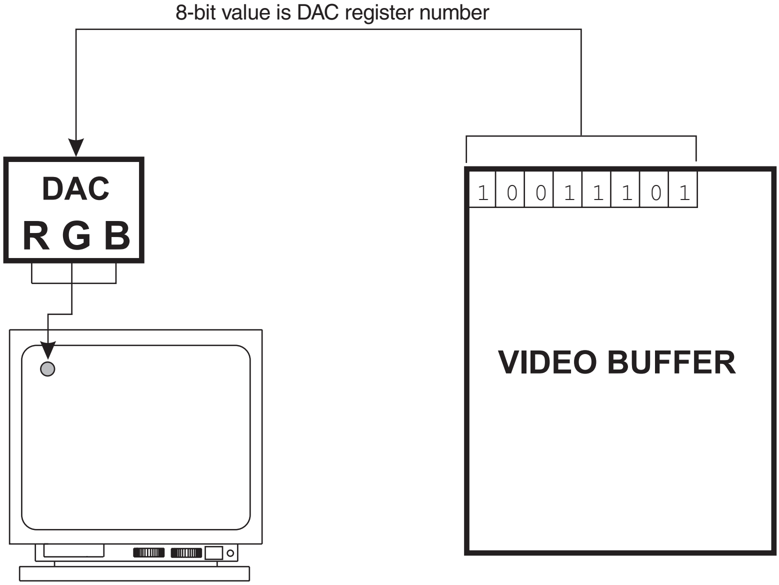


Figure 8-3 *Color Mapping in VGA Mode 19*

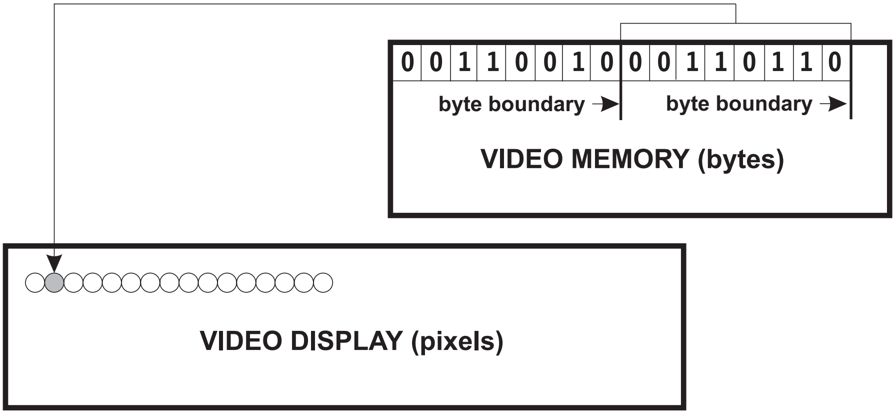


Figure 8-4 *Byte-to-Pixel Mapping Example in VGA Mode 19*

Address Calculations

Address calculations in mode number 19 are simpler than those in mode number 18. All that is necessary to obtain the offset of a pixel into the video buffer is to multiply its row address by the number of buffer bytes per pixel row (320) and then add the pixel column. The processing is shown in the following code fragment

```
; Address computation for VGA mode number 19
; On entry:
;              CX = x coordinate of pixel (range 0 to 319)
;              DX = y coordinate of pixel (range 0 to 199)
; On exit:
;               BX = offset into video buffer
;
        PUSH    CX            ; Save x coordinate
        MOV     AX,DX         ; y coordinate to AX
        MOV     CX,320        ; Multiplier is 320 bytes per row
        MUL     CX            ; AX = y times 320
        MOV     BX,AX         ; Free AX and hold in BX
        POP     AX            ; x coordinate from stack
        ADD     BX,AX         ; Add in column value
```

he above code is similar to the one in the WRITE_PIX_19 device driver listed in the VGA1 module of the graphics library furnished with this book.

Setting the Pixel

Once the segment and the offset registers are loaded, the program can set an individual screen pixel by means of a simple MOV instruction, as in the following code fragment:

```
; Write one pixel in VGA mode number 19 (256 colors)
; Code assumes that write mode 0 for 256 colors is selected
; Register setup:
;       ES = A000H (video buffer segment base)
;       BX = offset into the video buffer (range 0 to 64000)
;       AL = 8-bit color code
;
     MOV  ES:[BX],AL    ; Write pixel
```

8.2.4 VGA Mode 19 Read Pixel Routine

We have seen that in VGA mode number 19 each screen pixel is mapped to a single video buffer byte. There are 64,000 bytes in the video buffer, which is the same as the total number of screen pixels obtained by multiplying the number of pixels per row by the number of screen rows (320 * 200 = 64,000). The mapping scheme in VGA mode number 19 can be seen in Figure 8-4. The address calculations for mode number 19 were shown in Section 3.1.3. The actual read operation is performed by means of a MOV instruction, as in the following code fragment

```
; Read one pixel in VGA mode number 19 (256 colors)
; Code assumes that read mode 0 is selected
; Register setup:
;      ES = A000H (video buffer segment base)
;      BX = offset into the video buffer (range 0 to 64000)
;
     MOV  AL,BYTE PTR ES:[BX]    ; Read pixel
; AL now holds the 8-bit color code
```

8.3 Color Manipulations

The theory of additive color reproduction is based on the fact that light in the primary colors (red, green, and blue) can be used to generate all the colors of the spectrum. Red, green, and blue are called the primary colors. Technically, it is possible to create white light by blending just two colors. The color that must be blended with a primary color to form white is called the complement of the primary color, or the complementary color. Color Figure 2 shows the primary and the complementary colors. The complementary colors can also be described as white light minus a primary color. For example, white light without red, not-red, gives a shade of blue-green known as cyan; not-green gives a mixture of red and blue called magenta; and not-blue gives yellow, which is a mixture of red and green light. Video display technology is usually designed on additive color blending. Subtractive methods are based on dyes that absorb the undesirable, complementary colors. A cyan-colored filter, for example, absorbs the green and blue components of white light. Subtractive mixing is used in color photography and color printing.

In describing a color we use three characteristics that can be precisely determined: its hue, its intensity, and its saturation. A method of color measurement based on hue, intensity, and saturations (sometimes called the HIS) was developed for color television. The hue can be defined as the color of a color. Physically the hue can be measured by the color's dominant wavelength. The intensity of a color is its brightness. This brightness is measured in units of luminance or nits. The saturation of a color is its purity. If the color contains no white diluent it is said to be fully saturated.

8.3.1 256-Color Mode

While address mapping in VGA mode number 19 is simpler than in mode number 18, the pixel color encoding is considerably more complicated. This is so not only because there is a more extensive color range in mode number 19 than in mode number 18 (16 versus 256 colors) but also because the default encoding scheme is not very straightforward. This default scheme is determined by the setting of the 256 color registers in the DAC. The start-up value stored in these registers by the BIOS initialization code is designed to provide compatibility with the CGA and EGA systems. Figure 8-5 shows the default setting of the DAC Color registers in VGA mode number 19. The demonstration program named MODE19, furnished in the book's software, is an on screen display of the default setting of the DAC registers in the VGA mode number 19.

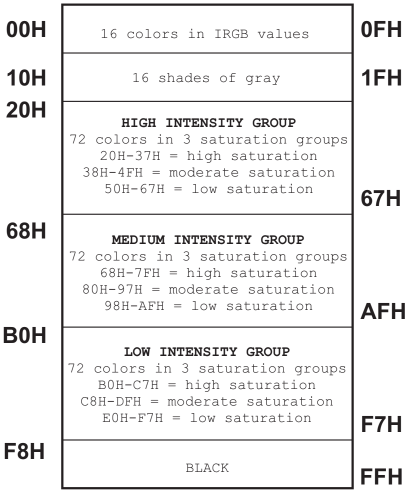

00H
16 colors in IRGB values
0FH
10H
16 shades of gray
1FH
20H
HIGH INTENSITY GROUP
72 colors in 3 saturation groups
20H-37H = high saturation
38H-4FH = moderate saturation
50H-67H = low saturation
67H
68H
MEDIUM INTENSITY GROUP
72 colors in 3 saturation groups
68H-7FH = high saturation
80H-97H = moderate saturation
98H-AFH = low saturation
AFH
B0H
LOW INTENSITY GROUP
72 colors in 3 saturation groups
B0H-C7H = high saturation
C8H-DFH = moderate saturation
E0H-F7H = low saturation
F7H
F8H
BLACK
FFH

Figure 8-5 *Default Color Register Setting in VGA Mode 19*

In Figure 8-5 the first group of default colors (range 00H to 0FH) corresponds to those in the 16-color modes. In other words, if only the 4 low-order bits of the 8-bit color code are programmed, the resulting colors in the 256-color mode are the same as those in the 16-color modes. The second group of default colors (range 10H to 1FH) corresponds to 16 shades of gray. The following group of colors (range 20H to 67H) consists of 72 colors divided into 3 sub-groups, each one representing a different level of color saturation. Each of the saturation sub-groups consists of 24 colors in a circular pattern of blue-red-green hues. Another 72-color group is used for medium intensity colors and a third one for low intensity colors.

But the programmer of VGA in 256-color mode is by no means restricted to the default values installed by the BIOS in the DAC Color registers. In fact, we can readily see that this default grouping is not convenient for many applications. Because the default tones of red, green, or blue are not mapped to adjacent bits or to manageable fields. For example, using the default setting of the DAC Color registers, the various shades of the color green are obtained with the values shown in Table 8-1.

Table 8-1

Shades of Green in VGA 256-Color Mode (default values)

VALUE/RANGE	INTENSITY	SATURATION	
02H	00000010B	medium	high
0AH	00001010B	high	high
2EH to 34H	00101110B to 00110100B	high	high
46H to 4CH	01000110B to 01001100B	high	moderate
5EH to 64H	01011110B to 01100100B	high	low
76H to 7CH	01110110B to 01111100B	medium	high
8EH to 94H	10001110B to 10010100B	medium	moderate
A6H to ACH	10100110B to 10101100B	medium	low
BEH to C4H	10111110B to 11000100B	low	high
D6H to DCH	11010110B to 11011100B	low	moderate
EEH to F4H	11101110B to 11110100B	low	low

A more rational 256-color scheme can be based on assigning 2 bits to each of the components of the familiar IRGB encoding. Figure 8-6 shows the bitmapping for this IRGB double-bit encoding.

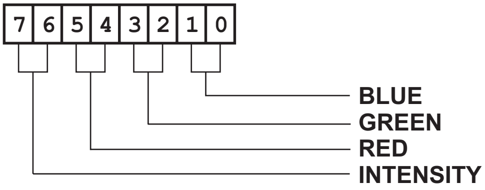


Figure 8-6 *Double-Bit Mapping for 256-Color Mode*

To enable the double-bit encoding in Figure 8-6 it is necessary to change the default setting of the DAC Color registers. The DAC Color registers consist of 18 bits, 6 bits for each color (red, green, and blue). The bitmap of the DAC Color registers is shown in Figure 8-7.

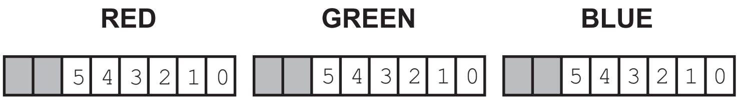


Figure 8-7 *DAC Color Register Bitmap*

To design an 8-bit encoding in a four-element (IRGB) format we have assigned 2 bits to each color and to the intensity component (see Figure 8-6). In this manner, the 2-bit values for red, green, and blue, allow four tones. Since each tone can be in four brightness levels, one for each intensity bit setting, each pure hue would have 16 saturations. In order to achieve a double-bit IRGB encoding by reprogramming the DAC Color registers (see Figure 8-7), we assign eight values to each DAC Color register, as shown in Table 8-2.

Table 8-2

DAC Register Setting for Double-Bit IRGB Encoding

NUMBER	6BIT VALUE	INTENSITY	COLOR
0	9	OFF	dark
1	18	OFF	.
2	27	OFF	.
3	36	OFF	.
4	45	ON	.
5	54	ON	.
6	63	ON	bright

The first 4 bit settings in Table 8-2 correspond to the color tones controlled by the red, green, and blue bits when the intensity bits have a value of 00B. The last three 6-bit values correspond to the three additional levels of intensity. This means that, excluding the intensity bit, the three DAC Color registers will have 64 possible combinations. Table 8-3 shows the pattern of register settings for the double-bit IRGB format.

Table 8-3

Pattern for DAC Register Settings in Double-Bit IRGB Encoding

I = 00				I = 01				I = 10				I = 11			
No.	R	G	B	No.	R	G	B	No.	R	G	B	No.	R	G	B
0	9	9	9	64	9	9	18	128	9	9	27	192	9	9	36
1	9	9	18	65	9	9	27	129	9	9	36	193	9	9	45
2	9	9	27	66	9	9	36	130	9	9	45	194	9	9	54
3	9	9	36	67	9	9	45	131	9	9	54	195	9	9	63
4	9	9	9	68	9	18	18	132	9	27	18	196	9	36	18
5	9	18	9	69	9	27	18	133	9	36	27	197	9	45	36
.				.				.				.			
.				.				.				.			
63	36	36	36	127	45	45	45	191	54	54	54	255	63	63	63

Notice in Table 8-3 that a value of 9 in the red, green, and blue color registers corresponds with the color black. It has been found that the colors generated by the low range of the DAC scale are less noticeable than those on the high range. By equating the value 9 to the color black we enhance the visible color range on a standard VGA, although in some CRTs this setting could appear as a very dark gray. The procedure named TWO_BIT_IRGB in the VGA1 module of the graphics library changes the default setting of the DAC Color registers to the values in Table 8-3. The procedure is described in Section 3.3. The program named IRGB256, furnished as part of the book's software package, shows the double-bit IRGB colors. This color pattern is displayed by the IRGB256 program.

We have seen that a double-bit IRGB setting for the DAC registers simplifies programming in the VGA 256-color mode when compared to the default setting shown in Figure 8-5. Once the DAC registers are set for the double-bit IRGB encoding the programmer can choose any one color by setting the corresponding bits in the video buffer byte mapped to the pixel. For example, the bit combinations in Table 8-4 can be used to display 16 pure tones of the complementary color named magenta (not-green). Notice that the purity of the hue is insured by the zero value in the green DAC register.

Table 8-4

16 Shades of the Color Magenta Using Double-Bit IRGB Code

NUMBER	I	R	G	B	TONE
0	00	01	00	01	darkest magenta
1	00	10	00	10	.
2	00	01	00	01	.
3	00	11	00	11	.
4	01	01	00	01	.
.		.			.
.		.			.
15	11	11	00	11	brightest magenta

But no single color encoding is ideal for all purposes. Often the programmer prefers to enhance certain portions of the color range at the expense of other portions. For example, in displaying a mountain landscape it might be preferable to extend shades of blue and green at the expense of the red. On the other hand, a volcanic explosion may require more shades of red than of green and blue. The programmer can manipulate the displayed range by choosing which set of 256 colors, from a possible total of 262,143, are installed in the DAC Color registers.

Shades of Gray

The color gray is defined as equal intensities of the primary colors, red, green, and blue. In the DAC Color registers any setting in which the three values are equal generates a shade of gray. For example, the value 20, 20, 20 for red, green, and blue, respectively, produce a 31 percent gray shade, while a value of 32, 32, 32 produce a 50 percent gray shade. Since the gray shades require that all three colors have the same value, and considering that each color register can hold 64 values, there are 64 possible shades of gray in the VGA 256-color modes. The actual setting of the VGA registers will go from 0, 0, 0, to 63, 63, 63, for red, green, and blue.

A graphics program operating in VGA 256-color mode can simultaneously use the full range of 64 gray shades, as well as 192 additional colors. This requires reprogramming the DAC Color registers. If a program were to execute in shades of gray only, then the low order 6-bits of the color encoding can be used to select the gray shades. The range would extend from a value of 0, for black, to a value of 63 for the brightest white. The setting of the DAC Color registers for a 64-step gray scale is shown in Table 8-5.

Table 8-5

Pattern for DAC Register Setting for 64 Shades of Gray

NO.	R	G	B	NO.	R	G	B	NO.	R	G	B	NO.	R	G	B
0	0	0	0	64	0	0	0	128	0	0	0	192	0	0	0
1	1	1	1	65	1	1	1	129	1	1	1	193	1	1	1
2	2	2	2	66	2	2	2	130	2	2	2	194	2	2	2
3	3	3	3	67	3	3	3	131	3	3	3	195	3	3	3
.				.				.				.			
.				.				.				.			
63	63	63	63	127	63	63	63	191	54	54	54	255	63	63	63

Notice in Table 8-5 that the gray settings are repeated four times. The effect of this repeated pattern is that the high-order bits of the color code are ignored. In other words, all possible color values will generate a gray shade, and the excess of 63 (00111111B) has no visible effect. The device driver named GRAY_256 in the VGA1 module of the graphics library changes the default setting of the DAC Color registers to the values in Table 8-5. The GRAY_256 procedure is described in detail in the discussion of the VGA1 module later in the chapter. The program named

GRAY256, furnished as part of the book's software, shows the setting of the DAC registers for 64 gray shades, repeated four times.

Summing to Gray Shades

A program can read the red, green, and blue values installed in a DAC Color register and find an equivalent gray shade with which to replace it. If this action is performed simultaneously on all 256 DAC Color registers the result will be to convert a displayed color image to monochrome. Considering that the human eye is more sensitive to certain regions of the spectrum, this conversion is usually based on assigning different weights to the red, green, and blue components. In any case, this relative color weight is used to determine the gray shade, on a scale of 0 to 63. However, as mentioned in the previous paragraph, the resulting gray scale setting must have equal proportions of the red, green, and blue elements.

BIOS Service number 16, of interrupt 10H, contains sub-service number 27, which sums all color values in the DAC registers to gray shades. The BIOS code uses a weighted sum based on the following values:

```
  red = 30%
green = 59%
 blue = 11%
-----------
total = 100%
```

The BIOS service does not preserve the original values found in the DAC registers. The primitive routine named SUM_TO_GRAY in the VGA1 module of the graphics library can be used to perform a gray scale sum based on the action of the above mentioned BIOS service (see Section 3.3).

The IBM BIOS performs several automatic operations on the VGA DAC Color registers. For example, during a mode change call (BIOS service number 0, interrupt 10H) the BIOS loads all 256 DAC Color registers with the default values. If the mode change is to a monochrome mode then a sum-to-gray operation is performed. The programmer can prevent this automatic loading of the DAC registers. BIOS service number 18, sub-service number 49, of interrupt 10H, enables and disables the default pallet loading during mode changes. Sub-service number 51 of service number 18 enables and disables the sum-to-gray function. The FREEZE_DAC and THAW_DAC device drivers in the VGA1 module of the graphics library provide a means for preventing and enabling default palette loading during BIOS mode changes. These procedures are described in Section 3.3.

8.3.2 16-Color Modes

In Table 2-2 we saw that VGA color modes can be in 2, 4, 16, and 256 colors. Since the two- and four-color modes are provided for compatibility with now mostly obsolete standards, they are of little interest to today's VGA programmer. The same can be said of the lower resolution graphics modes. This elimination leaves us with the 16-color text modes number 0 to 4 and graphics mode number 18. In the following discussion we will refer exclusively to the 16-color range in VGA graphics mode number 18.

Video memory mapping in mode number 18 can be seen in Figure 8-2; however, this illustration does not show how the color is obtained. Refer to Figure 2-4 to visualize how the pixel color in mode number 18 is determined by the values stored in four maps, usually named intensity, red, green, and blue. But this four-bit IRGB encoding is, in reality, the number of 1 of 16 palette registers located in the Attribute Controller group (see Section 2.2.5). Furthermore, the value stored in the Palette register is also an address into the corresponding DAC Color register. This dual-level color indirect addressing scheme was developed in order to provide VGA compatibility with the CGA and the EGA cards. The matter is further complicated by the fact that the DAC Color register number (an 8-bit value in the range 0 to 255) can be stored differently. If the Palette Select bit of the Attribute Mode Control register is clear, then the DAC Color register number is stored in the6 bits of the Palette register and in bits 2 and 3 of the Color Select register. While if the Palette Select bit is set, then the DAC Color register number is stored in the four low-order bits of the Palette register and in the four low-order bits of the Color Select register. The two addressing modes are shown in Figure 8-8.

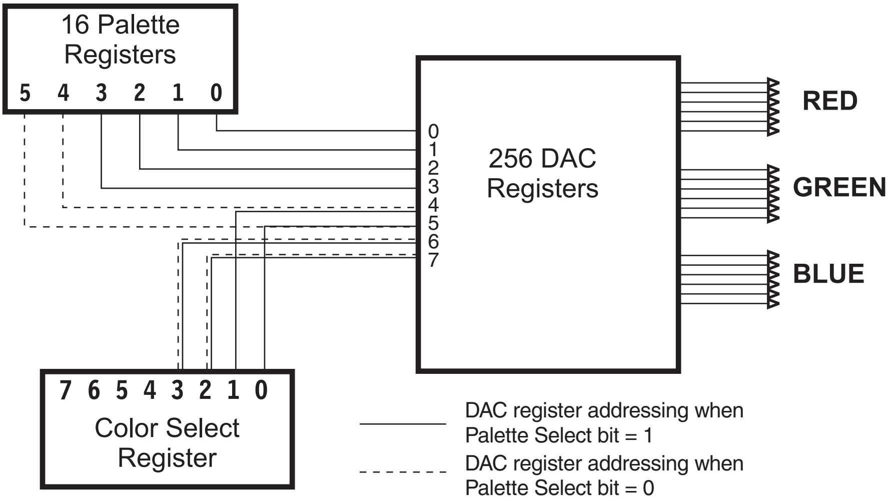


Figure 8-8 *DAC Register Selection Modes*

Notice in Figure 8-8 that when the Palette Select bit is set, bits 4 and 5 of the DAC register address are determined by bits 0 and 1 of the Color Select register, and not by bits 4 and 5 of the Palette register. This means that a program operating in this addressing mode will have to manipulate bits 4 and 5 of the desired DAC register number so that they are determined by bits 0 and 1 of the Color Select register, while bits 6 and 7 of the address are determined by bits 3 and 2 of the Color Select register.

Perhaps the simplest and most straightforward color option for VGA mode number 18 would be to set the Palette Select bit and to clear bits 0 to 3 of the Color Select register. In this manner the Palette and Color Select registers become transparent to the software, since the DAC register number is now determined by the four low bits of the Palette register, which, in turn, match the IRGB value in the bit planes. Nevertheless, this color setup would be incompatible with the one in the CGA and EGA standards, which are based on the value stored in the 16 Palette registers. The method followed by the BIOS, which is designed to achieved compatibility with the Palette registers of the CGA and EGA cards, is based on a customized set of values for the DAC Color registers which are loaded during mode 18 initialization. This set, which includes values for the first 64 DAC Color registers only, can be seen in Table 8-6.

Table 8-6

BIOS Settings for DAC Registers in Mode Number 18

NO.	R	G	B	NO.	R	G	B	NO.	R	G	B	NO.	R	G	B
0	0	0	0	16	0	21	0	32	21	0	0	48	21	21	0
1	0	0	42	17	0	21	42	33	21	0	42	49	21	21	42
2	0	42	0	18	0	63	0	34	21	42	0	50	21	63	0
3	0	42	42	19	0	63	42	35	21	42	42	51	21	63	42
4	42	0	0	20	42	21	0	36	63	0	0	52	63	21	0
5	42	0	42	21	42	21	42	37	63	0	42	53	63	21	42
6	42	42	0	22	42	63	0	38	63	42	0	54	63	63	0
7	42	42	42	23	42	63	42	39	63	42	42	55	63	63	42
8	0	0	21	24	0	21	21	40	21	0	21	56	21	21	21
9	0	0	63	25	0	21	63	41	21	0	63	57	21	21	63
10	0	42	21	26	0	63	21	42	21	42	21	58	21	63	21
11	0	42	63	27	0	63	63	43	21	42	63	59	21	63	63
12	42	0	21	28	42	21	21	44	63	0	21	60	63	21	21
13	42	0	63	29	42	21	63	45	63	0	63	61	63	21	63
14	42	42	21	30	42	63	21	46	63	42	21	62	63	63	21
15	42	42	63	31	42	63	63	47	63	42	63	63	63	63	63

We can corroborate the mapping of Palette and DAC registers in VGA mode number 18 by referring to Table 8-6. For example, the encoding for light red in Palette Register number 16 is 00111100B, which is 60 decimal. Recalling that the value in the VGA Palette register is interpreted as an index into the DAC Color register table, we can refer to Table 8-6 and observe that the setting of DAC register number 60 is 63, 21, 21 for the red, green, and blue elements, respectively. This setting corresponds to the color light red. In summary, the Palette register (in this case number 12) holds an encoding in rgbRGB format, that is also an index to the DAC Color table (in this case the rgbRGB value is equal to 60). It is the DAC Color register that holds the 18-bit RGB encoding that drives the analog color display.

Color Animation

An interesting programming technique for VGA systems is to use the bits in the Color Select register to change some or all of the displayed colors. For example, if the Palette Select bit of the Attribute Mode Control register is clear, then bits 2 and 3 of the Color Select register provide 2 high-order bits of the DAC register number (see Figure 8-8). Since two bits can encode four combinations (00, 01, 10, and 11), a program can change the value of bits 2 and 3 of the Color Select register to index into four separate areas of the DAC, each one containing 64 different color registers. By the same token, if the Palette Select bit is set, then the 4 low-order bits in the Color Select register can be used to choose one of 16 DAC areas, each one containing 16 color registers. The areas of the DAC determined through the Color Select register are sometimes referred to as color pages. Some interesting animation effects can be achieved by rapidly shifting these color pages. For example, a program can simulate an explosion by shifting the pixel colors to tints of red, orange, and yellow.

BIOS service number 16, sub-service number 19, provides a means for setting the paging mode to 4 color pages of 64 registers or to 16 color pages of 16 registers each, and also for selecting an individual color page within the DAC. In this kind of programming it is important to remember that the BIOS initialization routines for mode number 18 set color values for the first 64 DAC registers only. It is up to the software to initialize the color values in the DAC registers as necessary.

8.3.3 VGA1 Library Functions

The following are generic descriptions of the device driver routines contained in the VGA1 module of the GRAPHSOL library that is part of the book's software. The values passed and returned by the individual functions are listed in the order in which they are referenced in the code. The following listing is in the order in which the routines appear in the library source files.

ES_TO_VIDEO (Assembly Language only)

Set the ES segment register to the base address of the video buffer while in an alphanumeric mode.

```
Receives:
         Nothing
Returns:
         ES set to video buffer segment for alpha mode
Action:
         Video buffer can now be addressed in the form:
         ES:xx
```

ES_TO_APA (Assembly Language only)

Set the ES segment register to the base address of the video buffer while in a graphics mode. VGA graphics buffer is at A000H

```
Receives:
         Nothing
Returns:
         ES set to video buffer segment for graphics mode
Action:
         Video buffer can now be addressed in the form:
         ES:xx
```

PIXEL_ADD_18 (Assembly Language only)

Calculate buffer offset from pixel coordinates while in VGA mode number 18.

```
Receives:
         1. Word integer of x-axis pixel coordinate
            Range is 0 to 639
         2. Word integer of y-axis pixel coordinate
            Range is 0 to 479
Returns:
         1. Word integer of offset into video buffer
         2. Byte integer of pixel mask for write mode 0
            or 2
Action:
         Prepare for pixel read and write operations in VGA
         mode number 18.
```

WRITE_PIX_18 (Assembly Language only)

Set (write) an individual screen pixel while in VGA mode number 18, write mode 2.

```
Receives:
         1. Logical address of pixel in video buffer.
         2. Byte integer of pixel color in IRGB form
         3. Pixel mask for write mode 2
Returns:
         Nothing
Action:
         Pixel is set to one of 16 colors.
```

TILE_ADD_18 (Assembly Language only)

Calculate the coarse-grain address of an 8-by-8 pixel block (tile) while in VGA mode number 18.

```
Receives:

         1. Byte integer of x-axis tile coordinate
            Range is 0 to 79
         2. Byte integer of y-axis tile coordinate
            Range is 0 to 59
Returns:
         1. Word integer of offset into video buffer
Action:
         Prepare for tile write operation.
```

WRITE_TILE_18 (Assembly Language only)

Set (write) a screen tile (8-by-8 pixel block) while in VGA mode number 18, write mode 2.

```
Receives:
         1. Logical address of tile in video buffer
         2. Byte integer of tile color in IRGB form
Returns:
         Nothing
Action:
         Tile is set to one of 16 colors.
```

READ_PIX_18 (Assembly Language only)

Read the color code of a screen pixel in VGA mode number 18, read mode 0.

```
Receives:
         1. Logical address of pixel in video buffer
         2. Pixel mask for write mode 2
Returns:
         1. Byte integer of pixel's IRGB color code
Action:
         Pixel is read in read mode 0.
```

TWO_BIT_IRGB

Initialize DAC registers for VGA mode number 19 (256-colors) for the double bit IRGB format shown in Figure 3-6.

```
Receives:
         Nothing
Returns:
         Nothing
Action:
         DAC registers in the pattern shown in Table 3-3.
```

GRAY_256

Initialize DAC registers for VGA mode number 19 in 64 shades of gray, repeated four times.

```
Receives:
         Nothing
Returns:
         Nothing
Action:
         DAC registers in the pattern shown in Table 8-5.
```

SUM_TO_GRAY

Perform sum-to-gray function by means of BIOS service number 16, sub-service number 27, of interrupt 10H. Previous contents of DAC registers are not preserved.

```
Receives:
         Nothing
Returns:
         Nothing
Action:
         All DAC registers are converted to equivalent gray
         shades.
```

SAVE_DAC

Save current color codes in all DAC registers. Values are stored in RAM.

```
Receives:
         Nothing
Returns:
         Nothing
Action:
         The color codes in all DAC registers are stored in
         RAM.
```

RESTORE_DAC

The DAC registers are restored to the color values saved by the SAVE_DAC procedure.

```
Receives:
         Nothing
Returns:
         Nothing
Action:
         The color codes in all DAC registers are restored
         from the values saved in RAM by SAVE_DAC.
```

PIXEL_ADD_19 (Assembly Language only)

Calculate buffer offset from pixel coordinates while in VGA mode number 19.

```
Receives:
         1. Word integer of x-axis pixel coordinate
            Range is 0 to 319
         2. Word integer of y-axis pixel coordinate
            Range is 0 to 199
Returns:
         1. Word integer of offset into video buffer
Action:
         Prepare for pixel read and write operations in
         mode number 19.
```

TILE_ADD_19 (Assembly Language only)

Calculate the coarse-grain address of an 8-by-8 pixel block (tile) while in VGA mode number 19.

```
Receives:
         1. Byte integer of x axis tile coordinate
            Range is 0 to 39
         2. Byte integer of y axis tile coordinate
            Range is 0 to 25
Returns:
         1. Word integer of offset into video buffer
Action:
         Prepare for tile write operation.
```

FREEZE_DAC

Disable changes to the Palette and DAC registers during BIOS mode changes.

```
Receives:
         Nothing
Returns:
         Nothing
Action:
         The color codes in the Palette and DAC registers
         are preserved during BIOS mode changes.
```

THAW_DAC

Enable changes to the Palette and DAC registers during BIOS mode changes.

```
Receives:
         Nothing
Returns:
```

Nothing

Action:

The color codes in the Palette and DAC registers are replaced by the default values during BIOS mode changes.

Chapter 9

VGA Core Primitives

Topics:

- VGA primitives for video system setup
- VGA text display primitives
- VGA image display primitives
- VGA bit-map primitives
- VGA area fill primitives

9.1 Classification of VGA Primitives

Chapter 8 discussed the development of the most elementary and fundamental routines used in graphics programming, called the device drivers. A second level of graphics routines, usually providing higher-level functions than device drivers, are the graphics primitives. VGA primitive routines can be arbitrarily classified into the following fields:

1. Set-up, inquiry, and control primitives. This group of functions includes video mode-setting, read and write mode selection, initialization of palette and border color, inquiry of active video parameters, and other preparatory and initialization functions.
2. Text primitive routines. This group includes the selection of fonts and character attributes and the display of text characters in graphics modes.
3. Bit-block and area fill primitive routines. This group includes routines to manipulate bitmapped images in video or RAM memory.
4. Raster graphics primitive routines. This group includes object-oriented routines to draw the most common geometrical figures, to fill screen areas with colors or attributes, and to transform figures stored in the video buffer or in data files.

The primitive routines in the GRAPHSOL VGA library furnished with this book are organized in the listed fields. In the present chapter we will discuss the primitive routines in the first three groups. Because of their complexity, Chapter 10 is devoted to VGA raster graphics.

9.2 VGA Primitives for Set-Up, Control, and Query

The VGA graphics programmer must perform operations that are preparatory, controlling, or inquisitory. For example, an application using VGA graphics could start its execution by setting the desired video mode and the read and write modes, initializing a segment register to the base address of the video buffer, and installing a set of color values in the pallet and border color registers. These preparations could also require investigating the present state of the video system in order to restore it at the conclusion of the application.

Many VGA preparatory and initialization operations can be performed by means of services in the BIOS interrupt 10H. For example, a graphics program that uses a standard video mode will usually let the BIOS handle the complications of initializing the VGA registers that control display characteristics. Since mode setting usually takes place once or twice during the execution of an application, the slowness usually associated with BIOS services can be disregarded for this purpose. The same applies to many other initialization and set-up operations, which can be conveniently executed through the BIOS, and which seldom appear in the code. Such is the case with operations to set and read the Palette, Overscan, and DAC Color registers, to select the color paging mode, to sum DAC output to gray shades, and to obtain VGA system data.

On the other hand, some initialization operations are conspicuously missing from the services offered by BIOS interrupt 10H. For example, there are no BIOS services to set the VGA read and write modes. This is particularly noticeable when operating in mode number 19 (256 colors) which requires setting bit 6 of the Graphics Controller Graphics Mode Register (see Figure 2-22). Furthermore, other BIOS graphics services, such as those to set and read an individual screen pixel, perform so poorly that they are practically useless.

In summary, while most applications can benefit from BIOS VGA initialization and setup services, very few graphics programs could execute satisfactorily if they were limited to these BIOS services.

9.2.1 Selecting the VGA Write Mode

To make the VGA more useful and flexible its designers implemented several ways in which to write data to the video display. These are known as the write modes. VGA allows four different write modes, which are selected by means of bits 0 and 1 of the Graphics Mode register of the Graphics Controller (see Figure 2-22). The fundamental functions of the various write modes are as follows:

Write mode 0 is the default mode. In write mode 0 the CPU, Map Mask register of the Sequencer (Figure 2-13), and the Bit Mask register of the Graphics Controller (Figure 2-24) are used to set a screen pixel to any desired color. Other VGA registers are also used for specific effects. For example, the Data Rotate register of the Graphics Controller (Figure 2-20) has two fields which are significant during write mode 0 operations. The data rotate field (bits 0 to 3) determines how many positions to rotate the CPU data to the right before performing the write operation. The logical operation select field (bits 3 and 4) determines how the data stored in video memory is logically combined with the CPU data. The options are to write the CPU data unmodified or to AND, OR, or XOR it with the latched data.

In write mode 1 the contents of the latch registers, previously loaded by a read operation, are copied directly onto the color maps. Write mode 1, which is perhaps the simplest one, is often used in moving one area of video memory into another one. This write mode is particularly useful when the software takes advantage of the unused portions of video RAM. The location and amount of this unused memory varies in the different video modes. For example, in VGA graphics mode 18 the total pixel count is 38,400 pixels (640 pixels per row times 480 rows). Since the video buffer maps are 64K bytes, in each map there are 27,135 unused buffer bytes available to the programmer. This space can be used for storing images or data. On the other hand, video mode number 19 consists of one byte per pixel and there are 320 by 200 screen pixels, totaling 64,000 bytes. Since the readily addressable area of the video buffer is limited to 65,536 bytes, the programmer has available only 1,536 bytes for image manipulations.

Write mode 2 is a simplified version of write mode 0. Like mode 0, it allows setting an individual pixel to any desired color. However, in write mode 2 the data rotate function (Data Rotate register) and the set-reset function (Set/Reset register) are not available. One advantage of write mode 2 over write mode 0 is its higher execution speed. Another difference between these write modes is that in write mode 2 the pixel color is determined by the contents of the CPU, and not by the setting of the Map Mask register or the Enable Set-Reset and Set-Reset registers. This characteristic simplifies coding and is one of the factors that determines the better performance of write mode 2. The WRITE_PIX_18 device driver routine developed in Chapter 7 uses write mode 2.

In write mode 3 the Data Rotate register of the Graphics Controller (Figure 2-20) operates in the same manner as in write mode 0. The CPU data is ANDed with the Bit Mask register. The resulting bit pattern performs the same function as the Bit Mask register in write modes 0 and 2. The Set/Reset register also performs the same function as in write mode 0. However, the Enable Set/Reset register is not used. Therefore, the pixel color can be determined by programming either the Set/Reset register or the Map Mask register. The Map Mask register can also be programmed to selectively enable or disable the individual maps.

An application can use several read and write modes without fear of interference or conflict, since a change in the read or write mode does not affect the displayed image. On the other hand, a change in the video mode will normally clear the screen and reset all VGA registers. The code for changing the write mode, which is quite simple and straightforward, is shown in the following fragment:

```
; Set the Graphics Controller's Graphic Mode Register to the
; write mode in the AL register
        PUSH    AX          ; Save mode
        MOV     DX,3CEH     ; Graphic Controller Address
                            ; register
        MOV     AL,5        ; Offset of the Mode register
        OUT     DX,AL       ; Select this register
        INC     DX          ; Point to Data register
        POP     AX           ; Recover mode in AL
        OUT     DX,AL        ; Selected
```

The VGA graphics programmer must be aware that certain BIOS services reset the write mode. For example, BIOS service number 9, of interrupt 10H, often used to display text messages in an APA mode, sets write mode number 0 every time it executes. For this reason graphics software must often reset the write mode after executing a BIOS service. The procedure named SET_WRITE_MODE in the VGA1 module of the GRAPHSOL library sets the video mode in a similar manner as the previous fragment. In addition, SET_WRITE_MODE resets the Bit Mask register to its default value.

Writing Data in the 256-Color Modes

Writing a pixel in VGA mode number 19 (256 colors) requires that bit 6 of the Graphics Controller Graphics Mode register be set. Therefore a set write mode routine for VGA 256-color mode operation takes this into account. The following code fragment shows the required processing.

```
; Set the Graphics Controller's Graphic Mode Register to the
; write mode in the AL register, for 256 colors
        PUSH    AX           ; Save mode
        MOV     DX,3CEH      ; Graphic Controller Address
                             ; register
        MOV     AL,5         ; Offset of the Mode register
        OUT     DX,AL        ; Select this register
        INC     DX           ; Point to Data register
        POP     AX           ; Recover mode in AL
; Set bit 6 to enable 256 colors
        OR      AL,01000000B    ; Mask for bit 6
        OUT     DX,AL        ; Selected
```

The procedure named SET_WRITE_256 in the VGA1 module of the GRAPHSOL library sets the video mode in a similar manner as the previous fragment. In addition, SET_WRITE_256 resets the Bit Mask register to its default value.

9.2.2 Selecting the Read Mode

The VGA standard provides two different read modes. Read Mode 0, which is the default, loads the CPU with the contents of one of the bitmaps. In mode number 18 we conventionally designate the color maps with the letters I, R, G, and B, to represent the intensity, red, green, and blue elements. In this mode, which map is read into the CPU depends on the current setting of bits 0 and 1 of the Read Operation Map Select register of the Graphics Controller (see Figure 2-21). Sometimes we say that the selected read map is latched onto the CPU. In order to read the contents of all four maps, the program must execute four read operations to the same video buffer address; this latching is usually preceded by code to set the Read Operations Map Select register.

Read Mode 0 is useful in obtaining the contents of one or more video maps, while Read Mode 1 is more convenient when the programmer wishes to test for the presence of pixels that are set to a specific color or color pattern. In Read Mode 1 the contents of all four maps are compared with a predetermined mask. This mask must have been stored beforehand in the Color Compare register of the Graphics Controller (see Figure 7-18). For example, to test for the presence of bright blue pixels, the

IRGB bit pattern 1001B is stored in the Color Compare register. Thereafter, a read operation appears to execute four successive logical ANDs with this mask. If a bit in any of the four maps matches the bit mask in the Color Compare register, it will be set in the CPU; otherwise it will be clear.

The read mode is determined by bit 3 of the Select Graphics Mode register of the Graphics Controller (see Figure 7-22). The code to set the read mode is shown in the following fragment:

```
; Set the Graphics Controller Graphic Mode Select register to read
; mode 0 or 1, according to the value in AL
        CMP     AL,1            ; If entry value is not 1
        JNE     OK_BIT3         ; read mode 0 is forced
        MOV     AL,08H          ; 00001000B to set bit 3
OK_BIT3:
        PUSH    AX              ; Save mode
        MOV     DX,3CEH         ; Graphic controller address
                                ; register
        MOV     AL,5            ; Offset of the mode register
        OUT     DX,AL           ; Select this register
        INC     DX              ; Point to data register
        POP     AX              ; Recover mode in AL
        OUT     DX,AL           ; Selected
```

The procedure named SET_READ_MODE in the VGA1 module of the GRAPHSOL library sets the read mode in a similar manner as the previous fragment. The procedure named READ_MAPS_18, also in the VGA1 module, reads the contents of all four maps while in mode number 18 and returns the result in machine registers. This operation is performed by successively selecting the I, R, G, and B maps by means of the Read Map Select register of the Graphics Controller.

9.2.3 Selecting Logical Operation

In Chapter 7 you saw that the Data Rotate register of the Graphics Controller determines how data is combined with data latched in the system microprocessor registers. The programmer can select the AND, OR, and XOR logical operations by changing the value of bits 3 and 4 .

Although all three logical operation modes find occasional use in VGA graphics programming, the XOR mode is particularly useful. In animation routines the XOR mode provides a convenient way of drawing and erasing a screen object. The advantages of the XOR method are simpler and faster execution, and an easier way for restoring the original screen image. This is a convenient programming technique when more than one moving object can coincide on the same screen position.

One disadvantage of the XOR method is that the object's color depends on the color of the background over which it is displayed. If a graphics object is moved over different backgrounds, its color will change. The reader can observe that the cross-hair symbol of the MATCH program appears in different colors when overlaid over the running boar than when over the gray background. In this case the effect is not objectionable, but in other applications it could make the XOR technique unsuitable.

The programmer should note that some BIOS services set the Data Rotate register of the Graphics Controller to the normal mode. For example, if BIOS service number 9 of interrupt 10H is used to display text messages in a graphics application, when execution returns the logical mode is set to normal operation. Therefore, a program that uses the XOR, AND, or OR logical modes must reset the Data Rotate register after using this BIOS service.

XOR Operations in Animation Routines

The illusion of movement of a screen object is often produced by means of geometrical transformations. The simple transformations are named translation, rotation, and scaling. Complex transformations consist of combining two or more of simple transformations; for instance, a screen object moves across the screen while becoming progressively larger. The combined transformations generate the feeling that a three-dimensional object is diagonally approximating the viewer.

Geometrical transformations are usually performed by replacing the previous image of the object with a new image. In lateral translation an object appears to move across the screen by progressively redrawing it at slightly different horizontal coordinates. The boar symbol in the MATCH program is translated in this manner. Note that the graphics software must not only draw a series of consecutive images, but also erase the previous images from the screen. Otherwise, the animated object leaves a visible track of illuminated screen pixels. Although this effect could be occasionally desirable, frequently this is not the case. Also note that erasing the screen object is at least as time consuming as drawing it, since each pixel in the object must be changed to its previous state.

Erasing and redrawing of the screen object can be performed in several ways. One method is to save that portion of the screen image that is to be replaced by the object. The object can then be erased by redisplaying the original image. This method adds an additional burden to the graphics routine, which must also read and store every screen pixel that will be occupied by the object, but in many situations it is the only satisfactory solution. We have mentioned that another method of erasing the screen image is based on performing a logical XOR operation. The effect of the XOR is that a bit in the result is set if both operands contain opposite values. Consequently, XORing the same value twice restores the original contents, as in the following example:

```
          10000001B
     XOR  10110011B
         ----------
          00110010B

     XOR  10110011B
         ----------
          10000001B
```

An application that has set the Data Rotate register to the XOR mode can successively display and erase a screen object by XORing its bitmap. The effect can be used to animate the screen object by progressively changing its screen coordinates. The MATCH program, which is furnished on the book's software package as an illustration of VGA programming techniques, uses the XOR mode to display and erase two animated objects: one represents the outline of a running boar target and the other one the cross-hair of a rifle scope. The procedure named XOR_XHAIR in the MATCHD.ASM source file and the procedures XOR_RBOAR and XOR_LBOAR in the MATCHC.ASM source file, perform the draw/erase operations. Both procedures assume that the logical mode for XOR operation has been previously set.

9.2.4 System Status Operations

In contrast with its predecessors (EGA and CGA) all VGA registers that hold relevant system data can be read by the processor. This allows a program to investigate the video status by performing a read operation to the relevant register. In addition, BIOS service number 27 and number 28 provide means for obtaining VGA data and for saving and restoring the video state.

A function that is conspicuously missing in the BIOS is one to save the setting in the 256 VGA DAC color registers. For this reason, a program that uses BIOS sum-to-gray-shades function (service number 16, sub-service 27, of interrupt 10H) has no way of restoring the original DAC colors. The procedure named SAVE_DAC, in the VGA1 module of the GRAPHSOL library, provides a way for saving the state of the DAC registers. The procedure RESTORE_DAC can be used to restore the DAC register setting saved with SAVE_DAC.

9.2.5 Vertical Retrace Timing

Raster scan displays operate by projecting an electron beam on each horizontal row of screen pixels. Pixel scanning proceeds, row by row, from the top left screen corner to the bottom right. To avoid visible interference, the electron beam is turned off during the period in which the gun is re-aimed back to the start of the next pixel row (horizontal retrace). The beam is also turned off while it is re-aimed from the last pixel on the bottom right corner of the screen to the first pixel at the top left corner (vertical retrace). Because of the distance and directions involved, the vertical retrace period takes much longer than the horizontal retrace one.

In the CGA card it was the programmer's responsibility to time each access to the video buffer with the vertical retrace cycle of the CRT controller. Otherwise the result would be a visible interference, usually called snow. The VGA was designed to avoid this form of interference when using conventional display programming methods. However, animation techniques, which must flash consecutive screen images at a rapid rate, are not free from interference. Therefore, in this case the program must time the buffer accesses with the vertical retrace cycle of the CRT controller.

This timing requirement introduces an additional burden on animated graphics software. For example, the screen refresh periods in VGA graphics modes take place at an approximate rate of 70 times per second. An animated program that flashes images on the screen at a minimum rate of 20 per second must take into account that each display operation has to be timed with a vertical retrace cycle that takes place 70 times per second. This synchronization delay must be added to the processing time in order to maintain an interference-free image-flashing rate.

The start of the vertical retrace cycle can be determined by reading bit 7 of the VGA Input Status register 0 in the General register group. This bit is set if a vertical retrace is in progress. But in order to maximize the interference-free time available during a vertical retrace, the code must wait for the start of a vertical retrace cycle. This requires first waiting for a vertical retrace cycle to end, if one is in progress, and then detecting the start of a new cycle. The programming is shown in the following code fragment:

```
; Test for start of the vertical retrace cycle
; Bit 7 of the Input Status register 0 is set if a vertical cycle
; is in progress
        MOV     DX,3C2H         ; Input status register 0
                                ; In VGA color modes
VRC_CLEAR:
        IN      AL,DX           ; Read byte at port
        JMP     SHORT $+2       ; I/O delay
        TEST    AL,10000000B    ; Is bit 7 set?
        JNZ     VRC_CLEAR       ; Wait until bit clear
; At this point the vertical retrace ended. Wait for it to
; restart
VRC_START:
        IN      AL,DX           ; Read byte at port
        JMP     SHORT $+2       ; I/O delay
        TEST    AL,10000000B    ; Is bit 7 set?
        JZ      VRC_START       ; Wait until bit set
; Vertical retrace has now started
```

The procedure named TIME_VRC, in the VGA1 module of the GRAPHLIB library, detects the start of the CRT vertical retrace cycle so that video access operations can be synchronized.

9.3 VGA Text Display Primitives

Very few graphics applications execute without some form of text display. If the text display functions in an application take place in separate screens from the graphics operations, the programmer has the convenient option of selecting a text mode and either using text output keywords in a high-level language or one of the text display functions available in the BIOS. However, if a graphics program must combine text and graphics on the same screen, the text display functions available to the programmer are more limited.

9.3.1 BIOS Text Display Functions

In any mode, alphanumeric or graphics, BIOS service number 9, INT 10H, can be used to display a character at the current cursor position. Note that this is the only BIOS character display service that can be used in a graphics mode, but that several other services can be used in alphanumeric modes. Service number 2, INT 10H, to set the cursor position, can also be used in conjunction with service number 9. Note that there is no physical cursor in VGA graphics modes, and that the action of service number 2, interrupt 10H, is simply to fix a position for the text display operation that will follow. This invisible cursor is sometimes called a virtual cursor. The procedure named SET_CURSOR, in the ALFA modules of the GRAPHSOL library, uses service number 2, interrupt 10H, to set the cursor. Once the virtual cursor is positioned at the desired screen location, the program can display characters on the graphics screen by means of service number 9, interrupt 10H.

Text Block Display

But VGA programs that have frequent need to display text while in a graphics mode often need a more convenient method than setting a virtual cursor and calling BIOS service number 9. One option is a routine capable of displaying any number of text lines, starting at any screen position, and using any desired color available in the active mode. A convenient way of storing the display parameters for the text message is in a header block preceding the message itself. The GRAPHIC_TEXT procedure in the VGA2 module of the GRAPHSOL library displays a text message with embedded parameters. In this case the first byte in the header encodes the screen row at which the message is to be displayed, the second byte encodes the screen column, and the third one the color code. Since the procedure operates in any text of graphics mode, the range and encodings for these parameters depend on the active mode.

BIOS Character Sets

The BIOS stores several sets of text characters encoded in bitmap form (see Figure 1-10). VGA systems contain three complete character fonts and two supplemental fonts. The characteristics of these fonts are shown in Table 9-1.

Table 9-1

VGA BIOS Character Sets

CHARACTER BOX SIZE	MODE
8 by 8	0, 1, 2, 3, 4, 5, 13, 14, and 19
8 by 14	0, 1, 2, 3, 15, and 16
8 by 16	17, and 18
9 by 14*	7
9 by 16*	0, 1, and 7

Legend:
* = supplemental sets

The supplemental character sets (Table 9-1) do not contain all of the 256 character maps of the full sets, but only those character maps that are different in the 9-bit wide fonts. In the process of loading a 9-bit character set the BIOS first loads the corresponding set of 8-bit character maps and then overwrites the ones that need correction and that appear in the supplemental set. This mechanism is usually transparent to the programmer, who sees a full set of 9 by 14 or 9 by 16 characters.

9.3.2 A Character Generator

VGA graphics programs can perform simple character display operations by means of the BIOS functions, but for many purposes these functions are too limiting. Perhaps the most obvious limitation of character display by means of BIOS services is that the text characters must conform to a grid of columns and rows determined by the active character font and video mode. For example, a graphics program executing in mode number 18 uses BIOS service number 9, interrupt 10H, to display screen text using the

8 by 16 character font. This program will be constrained to a text screen composed of 80 character columns by 30 rows and will not be able to locate text outside this imaginary grid.

Moving a BIOS Font to RAM

A program can obtain considerable control in text display functions by operating its own character generator, in other words, by manipulating the text character maps as if they were a regular bitmap. The process can often be simplified by using existing character maps. In VGA systems the most easily available character maps are the BIOS character sets (see Table 9-1). The software can gain the necessary information regarding the location of any one of the BIOS character maps by means of service number 17, sub-service number 48, of interrupt 10H. Once the address of the character table is known, the code can move all or part of this table to its own address space, where it becomes readily accessible. The procedure named FONT_TO_RAM in the VGA2 module of the GRAPHSOL library can be used to load any one of the three full VGA character sets into a buffer furnished by the caller.

In loading a BIOS character font to RAM memory so that the font can be used with the display procedures in the GRAPHSOL library the caller must precede the font storage area with two data bytes that encode the font's dimensions. For example, the storage area for the BIOS 8 by 8 font can be formatted as follows:

```
;**********************|
;   storage for BIOS   |
;   symmetrical font   |
;**********************|
; RAM storage for symmetrical font table from BIOS character maps
; Each font table is preceded by two bytes that determine its
; dimensions, as follows:
;   Byte at font table minus 1 = number of pixel rows
;   Byte at font table minus 2 = number of horizontal bytes
;
; 1 x 8 built in ROM font
                DB      1       ; bitmap x dimension, in bytes
                DB      8       ; bitmap y dimension, in bytes
FONT_1X8        DB      2048 DUP (00H)
```

Note that 2,048 bytes are reserved for the 8 by 8 BIOS font, which contains 256 character maps of 8 bytes each (256 * 8 = 2048). By the same token, the 1-by-16 character font would require 4,096 bytes of storage.

Once the BIOS font table is resident in the caller's memory space it can be treated as a collection of bitmaps, one for each character in the set. In this manner the programmer is able to control, at the pixel level, the screen position of each character. Consequently, the spacing between characters, which is necessary in line justification, also comes under software control. Also the spacing between text lines and even the display of text messages at screen angles becomes possible.

The VGA2 module of the GRAPHSOL library contains three display procedures for displaying text messages using a BIOS character set resident in the program's memory space. The procedure named COARSE_TEXT provides a control similar to the one that can be obtained using BIOS service number 9, interrupt 10H, that is, text is displayed at column and row locations. Its operation is also similar to the GRAPHIC_TEXT procedure previously described. The procedure named FINE_TEXT allows the display of a single text line starting at any desired pixel location and using any desired spacing between characters on the horizontal and the vertical axes. This means that if the vertical spacing byte is set to zero in the text header block all the characters will be displayed on a straight line in the horizontal plane. However, by assigning a positive or negative value to this parameter, the programmer using this procedure can display a text message skewed at any screen angle. Finally, the procedure named MULTI_TEXT in the VGA2 module of the GRAPHSOL library makes possible the display of a text message consisting of multiple lines, starting at any desired pixel location. When using the MULTI_TEXT procedure the programmer has two header parameters to control character and row spacing, but the skewing option is not available.

The program named TEXTDEMO, furnished in the book's software package, contains a demonstration of the use of the text display procedures contained in the VGA2 library.

Display Type

The use of character generator software and BIOS character tables, as described in the previous paragraphs, considerably expands the programmer's control over text display on the VGA graphics modes. However, the BIOS character sets consist of relatively small symbols. Many graphics applications require larger characters (sometimes called display type) for use in logos, titles, headings, or other special effects. Since the largest character sets available in BIOS are the 8 by 16 and 9 by 16 fonts, the programmer is left to his or her own resources in this matter.

The programmer has many ways of creating or obtaining display type screen fonts. These include the use of scalable character sets, the design of customized screen font tables, the adaptation of printer fonts to screen display, the enlargement of existing screen fonts, and even the artistic design of special letters and logos. Which method is suitable depends on programming needs and availability of resources. Ideally, the display programmer would have available scalable text fonts in many different typefaces and styles. In fact, some sophisticated graphics programs and programming environments furnish screen versions of the Postscript language, which comes close to achieving this optimum level of text display control.

In the development of text-intensive applications, such as desktop publishing and graphics design software, the programmer should aim at the most sophisticated levels of text display technology. On the other hand, this absolute control over text display operations is often not necessary. In the MATCH program, which is provided in the book's software package as a demonstration of VGA programming techniques, we can see the use of two methods for creating display type. The first method was used for the program logo; in this case a large rendering of the word "Match" was created in the AutoCAD program, then output to a pen plotter, scanned, edited, and saved as a disk file image in TIFF format. The second method was to use a Hewlett-Packard style printer font (also called a PCL format) as a screen display type. The text in the first MATCH screen: "GRAPHICS SOLUTIONS - VGA Demo Press any Key to Start Match" is displayed using a PCL printer font. We have used a

PCL font in the programming demonstrations because they provide acceptable display quality and are often available to the programmer.

Using a PCL Font

One noticeable difference between the BIOS screen fonts and the printer fonts in PCL format is that the former have a symmetrical pattern for all the text characters, that is, all character maps occupy the same memory space. For example, in a BIOS 8 by 16 font each character map takes up 16 bytes of storage. In this case the software can reach any character map by adding a multiple of 16 to the address that marks the start of the font table. In other words, the offset of any desired character map is the product of its ASCII code by the number of bytes in each character map.

However, the optimization methods followed in the creation of PCL printer fonts determine that all character maps are not of identical size. Therefore, in a typical PCL font the character map for the letter "M" is larger than the character map for the letter "i". This characteristic complicates the problem of finding the start of a desired character map in the font table and in obtaining its specific horizontal and vertical dimensions. The procedure named INDEX_HP in the VGA2 module of the GRAPHSOL library is an auxiliary routine to find the offset and the character dimensions of any character in a PCL font. The details of the PCL encoding can be found in the source code of the INDEX_HP procedure which is located in the VGA2.ASM module in the book's software package.

The use of a PCL font in screen display operation requires loading the font's character maps into the program's address space. This operation is similar to loading a BIOS font as performed by the FONT_TO_RAM procedure. One difference is that the BIOS font resides in system memory while the PCL font is stored in a disk file. The procedure READ_HPFONT in the VGA2 module loads a printer font in PCL format into a RAM buffer provided by the caller. In this case the caller must provide a pointer to a buffer that holds an ASCIIZ string with the filename of the PCL disk file as well as a pointer to another buffer that will hold the loaded character set. Note that an ASCIIZ string is an MS DOS structure that holds a pathname followed by a zero byte. An example of the necessary structures and operations for loading a PCL font can be found in the TEXTDEMO program contained in the book's software.

Once the PCL font is resident in the program's memory space, its characters can be conveniently displayed on the screen by means of a character generator routine. The FINE_TEXTHP procedure in the VGA2 module of the GRAPHSOL library is a character generator routine for use with PCL format character maps. This routine provides, for PCL fonts, the text control features provided by the FINE_TEXT procedure for BIOS character maps.

Note that PCL font sizes are scaled to the standard density of a Hewlett-Packard laser printer, which is of 300 dots per inch. Since the pixel density in mode number 18 is 75 pixels per inch, the displayed characters appear four times larger than printed ones. In other words, an 8-point PCL font will be displayed as 32-point characters.

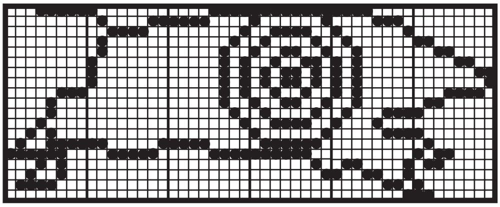
9.4 Bit-Block and Fill Primitives

Computer graphics images are roughly classified into two types: bitmapped and object-oriented. A bitmap is a data structure that serves to encode image elements into memory units. The character maps discussed in the previous section are bitmaps. In VGA systems the structure of a bitmap depends on the video mode. For example, in mode number 18, in which each screen pixel can be in one of sixteen colors (IRGB format) a full bitmap requires four bits per pixel. Figure 3-1 shows how the screen pixels (in mode number 18) are mapped to the VGA memory planes. However, a RAM bitmap for a mode 18 graphics image does not necessarily have to encode data in all four color planes. For example, a monochrome image can be encoded in a single map, while its color code is stored in a separate variable.

9.4.1 Mode 18 Bitmap Primitives

The most convenient bitmap format depends on the characteristic of the image, the video hardware, and the computer system. In the present section we discuss the VGA primitive routines to display the images encoded in bitmaps that have been customized for a specific VGA mode.

Figure 9-1 is a bitmap of the running boar target using in the MATCH demonstration program furnished in the book's software package. Also in Figure 9-1 is the bitmap that encodes in one-bits the screen pixels that are set in the running boar image. Because the bitmap is on a bit-to-pixel ratio it is quite suited to VGA mode number 18.



```
1.   1FH  80H  0FH  FFH  F0H  00H
2.   00H  43H  F0H  81H  0EH  00H
3.   00H  3CH  01H  3CH  81H  00H
4.   00H  40H  02H  42H  40H  C0H
5.   00H  40H  04H  99H  20H  30H
6.   00H  80H  05H  24H  A0H  0CH
7.   00H  80H  05H  5AH  A0H  03H
8.   00H  80H  05H  5AH  A0H  01H
9.   07H  00H  05H  24H  A0H  1EH
10.  08H  00H  04H  99H  20H  60H
11.  08H  00H  02H  42H  47H  80H
12.  10H  00H  01H  3CH  88H  00H
13.  28H  00H  00H  81H  07H  80H
14.  5FH  C1H  F0H  3FH  00H  40H
15.  FCH  3EH  0FH  FCH  00H  B0H
16.  14H  00H  00H  02H  61H  60H
17.  24H  00H  00H  01H  99H  00H
18.  78H  00H  00H  00H  06H  80H
19.  00H  00H  00H  00H  01H  C0H
```

Figure 9-1 *Pixel Image and Bitmap of a Graphics Object*

A VGA mode number 18 graphics routine to display a bitmapped image as the one shown in Figure 9-1 will need to know the screen coordinates at which the image is to be displayed, the dimensions of the bitmap, and the color code for the total image, or for each pixel or group of pixels. Two procedures in the VGA2 library can be used to display a bit map in mode number 18. The procedure MONO_MAP_18 displays an image in single color while the procedure COLOR_MAP_18 can be used to display an image in which each pixel is encoded in a different color. In the MONO_MAP_18 procedure the color is stored in a single IRGB byte that is used to display all pixels in the map.

In the COLOR_MAP_18 procedure the color is passed as a pointer to an array of color codes stored in a byte-per-pixel table. This scheme, although simple and fast, is not the most memory-efficient one, since in mode number 18 the 4-bit color code can be represented in one nibble (4 bits). However, the masking and indexing operations required in a nibble-per-pixel encoding would considerably slow down execution. An alternative and quite feasible bitmap scheme for VGA mode number 18 can be based on the video system's color map structure (see 1). In this design the image is stored in four RAM bitmaps, each map representing an element in the IRGB format. While this encoding requires less than half the storage than the one used by the COLOR_MAP_18 procedure, it requires almost four times more space than a single monochrome code, as the one in the MONO_MAP_18 procedure. Another advantage of the design adopted in the bitmap display procedures in the VGA2 module is that either routine (MONO_MAP_18 and COLOR_MAP_18) can be used with the same image map by changing the color table pointer.

9.4.2 Mode 19 Bitmap Primitive

We have seen that in mode number 19 each screen pixel is mapped to a memory byte which encodes its color. The procedure named COLOR_MAP_19, in the VGA2 module of the GRAPHSOL library, displays a bitmap in VGA mode number 19. The code assumes that the bitmap is preceded by a header that holds the screen coordinates for the graphics image and the dimensions of the pixel map. Following this header is the byte-to-pixel map of the graphics image.

Fill Primitives

Primitives to perform fill operations are used to clear or initialize the screen, to set a geometrical area to a color or pattern, or to fill a closed boundary figure. The VGA2 module of the GRAPHSOL library contains fill routines to clear the video screen and to initialize a rectangular pixel area. Geometrical fill routines are developed in Chapter 10.

9.5 Primitive Routines in the VGA1 and VGA2 Modules

The library module named VGA1 of the GRAPHSOL library that is part of the book's software contains the VGA device drivers routines as well as the setup, inquiry, and control primitives mentioned in the present chapter. The VGA2 module contains the text display primitives and the bitmap display and rectangular fill primitives.

9.5.1 Primitive Routines in the VGA1 Module

The following are generic descriptions of the setup, inquiry, and control primitive routines contained in the VGA1 libraries. The values passed and returned by the individual functions are listed in the order in which they are referenced in the code.

SET_MODE

Sets the BIOS video display mode using service number 0 of interrupt 10H.

```
Receives:
         1. Byte integer of desired video mode
Returns:
         Nothing
Action:
         New video mode is enabled.
         Screen is cleared.
```

GET_MODE

Obtains the current BIOS video mode using service number 15 of interrupt 10H.

```
Receives:
         Nothing
Returns:
        1. Byte integer of number of character columns
           Valid values are 40 and 80
        2. Byte integer of active video mode
        3. Byte integer of active display page
```

TIME_VRC

Test for start of the vertical retrace cycle of the VGA CRT controller.

```
Receives:
         Nothing
Returns:
         Nothing
Action:
         Execution returns to caller at start of vertical
         retrace cycle
```

SET_WRITE_MODE

Set the Graphics Controller Write Mode register in VGA less-than-256-color modes.

```
Receives:
         1. Byte integer of desired write mode
Returns:
         Nothing
```

SET_WRITE_256

Set the Graphics Controller Write Mode register in VGA 256-color mode.

```
Receives:
         1. Byte integer of desired write mode
Returns:
         Nothing
```

SET_READ_MODE

Set the Graphics Controller Mode Select register to read mode 0 or 1.

```
Receives:
         1. Byte integer of desired read mode
Returns:
         Nothing
```

LOGICAL_MODE

Set the Graphics Controller Data Rotate register to XOR, OR, AND, or NORMAL mode.

```
Receives:
         1. Byte integer encoding desired logical mode
Returns:
         Nothing
```

READ_MAPS_18

Read contents of four color maps in VGA mode number 18.

```
Receives:
         1. Logical address of video buffer byte to read
Returns:
         1. Byte integer of intensity map
         2. Byte integer of red map
         3. Byte integer of green map
         4. Byte integer of blue map
Action:
         Routine assumes that read mode 0 is active
Assumes:
         ES --> video buffer base address
```

9.5.2 Primitive Routines in the VGA2 Module

The following are generic descriptions of the text display, bitmap display, and rectangular fill primitives contained in the VGA2 libraries. The values passed and returned by the individual functions are listed in the order in which they are referenced in the code. The following listing is in the order in which the routines appear in the library source files.

GRAPHIC_TEXT

Display a formatted text message using BIOS service number 9, interrupt 10H. This procedure can be used in VGA modes number 18 and 19.

```
Receives:
         1. Offset pointer to message text (DS assumed)
Returns:
         Nothing
Message format:
OFFSET  STORAGE UNIT    CONTENTS
  0     Byte integer    Screen row for start of display
  1     Byte integer    Screen column for start of display
  2     Byte integer    Color code
Control codes:
CODE         ACTION
 00H         End of message
 FFH         End of text line
```

FINE_TEXT

Display a single-line text message using a RAM font table in which all bitmaps have the same dimensions (symmetrical font). Display position is defined at a pixel boundary. Mode number 18 only.

```
Receives:
         1. Offset pointer to message text (DS assumed)
         2. Offset pointer to RAM font table (DS assumed)
```

```
Returns:
         Nothing
Message format:
OFFSET  STORAGE UNIT    CONTENTS
  0     Word integer    Pixel row for start of display
  2     Word integer    Pixel column for start of display
  4     Word integer    Character spacing on x axis
  6     Word integer    Character spacing on y axis
  8     Byte integer    Color code in IRGB format
Control codes:
CODE         ACTION
 00H         End of message
Assumes:
         ES --> video buffer base address
```

MULTI_TEXT

Display a multiple-line text message using a RAM font table in which all bitmaps have the same dimensions (symmetrical font). Display position is defined at a pixel boundary. Mode number 18 only.

```
Receives:
         1. Offset pointer to message text (DS assumed)
         2. Offset pointer to RAM font table (DS assumed)
Returns:
         Nothing
Message format:
OFFSET  STORAGE UNIT   CONTENTS
  0     Word integer   Pixel row for start of display
  2     Word integer   Pixel column for start of display
  4     Word integer   Character spacing (x axis)
  6     Word integer   Line spacing (y axis)
  8     Byte integer   Color code in IRGB format
Control codes:
CODE         ACTION
 00H         End of message
 FFH         End of text line
Assumes:
         ES --> video buffer base address
```

FINE_TEXTHP

Display a single-line text message using a RAM font table in PCL format (asymmetrical font). Display position is defined at a pixel boundary. Mode number 18 only.

```
Receives:
         1. Offset pointer to message text (DS assumed)
         2. Offset pointer to RAM font table (DS assumed)
Returns:
         Nothing
Message format:
OFFSET  STORAGE UNIT   CONTENTS
  0     Word integer   Pixel row for start of display
  2     Word integer   Pixel column for start of display
  4     Word integer   Character spacing on x axis
  6     Word integer   Character spacing on y axis
  8     Byte integer   Color code in IRGB format
Control codes:
CODE         ACTION
 00H         End of message
Assumes:
```

```
          ES --> video buffer base address
```

READ_HPFONT

Read into RAM a PCL format printer font stored in a disk file

```
Receives:
         1. Offset pointer to ASCIIZ filename for PCL soft
            font located in current path (DS assumed)
         2. Offset pointer to RAM storage area (DS assumed)
Returns:
         Carry clear if no error
         Carry set if file not found or disk error
```

FONT_TO_RAM

Read a BIOS character map into RAM

```
Receives:
         1. Byte integer encoding BIOS font desired
             8 = 8 by 8 font
            14 = 8 by 14 font
            16 = 8 by 16 font
         2. Offset pointer to RAM storage area (DS assumed)
Returns:
         Nothing
```

MONO_MAP_18

Display a single-color, bitmapped image stored in the caller's memory space, while in VGA mode 18.

```
Receives:
         1. Offset pointer to bitmap (DS assumed)
         2. Offset pointer to color code (DS assumed)
Returns:
         Nothing
Bitmap format:
OFFSET  STORAGE UNIT   CONTENTS
  0     Word integer   Pixel row for start of display
  2     Word integer   Pixel column for start of display
  4     Byte integer   Number of rows in bitmap
  5     Byte integer   Bytes per row in bitmap
  6                    Start of bitmapped image
Assumes:
         ES --> video buffer base address
```

COLOR_MAP_18

Display a multi-color, bitmapped image stored in the caller's memory space, while in VGA mode 18.

```
Receives:
         1. Offset pointer to bitmap (DS assumed)
         2. Offset pointer to color table (DS assumed)
Returns:
         Nothing
Bitmap format:
OFFSET  STORAGE UNIT   CONTENTS
  0     Word integer   Pixel row for start of display
  2     Word integer   Pixel column for start of display
  4     Byte integer   Number of rows in bitmap
  5     Byte integer   Bytes per row in bitmap
  6                     Start of bitmapped image
Color table format:
One color byte per image pixel
Assumes:
         ES --> video buffer base address
```

COLOR_MAP_19

Display a multi-color, byte-mapped image stored in the caller's memory space, while in VGA mode 19. One byte encodes each image pixel for display in 256 color mode.

```
Receives:
         1. Offset pointer to header data of color byte map
            (DS assumed)
Returns:
         Nothing
Bitmap format:
OFFSET  STORAGE UNIT   CONTENTS
  0     Word integer   Pixel row for start of display
  2     Word integer   Pixel column for start of display
  4     Byte integer   Number of rows in bitmap
  5     Byte integer   Bytes per row in bitmap
  6                    Start of color byte-mapped image
Color table format:
One color byte per image pixel
Assumes:
         ES --> video buffer base address
```

CLS_18

Clear screen using IRGB color code while in VGA mode number 18.

```
Receives:
         1. Byte integer of IRGB color code
Returns:
         Nothing
Action:
         Entire 640 by 480 pixel screen area is initialized
         to the color passed by the caller.
```

CLS_19

Clear screen using IRGB color code while in VGA mode number 19. Encoding depends on setting of DAC registers.

```
Receives:
         1. Byte integer of IRGB color code
Returns:
         Nothing
Action:
         Entire 320 by 200 pixel screen area is initialized
         to the color passed by the caller.
```

TILE_FILL_18

Initialize a rectangular screen area, at the tile level, to a passed color code while in mode 18.

```
Receives:
         1. Byte integer of x axis start tile
         2. Byte integer of y axis start tile
         3. Byte integer of horizontal tiles in rectangle
```

```
        4. Byte integer of vertical tiles in rectangle
        5. Byte integer of color code in IRGB format
Returns:
        Nothing
Assumes:
        ES --> video buffer base address
```

TILE_FILL_19

Initialize a rectangular screen area, at the tile level, to a passed color code while in mode 19.

```
Receives:
        1. Byte integer of x axis start tile
        2. Byte integer of y axis start tile
        3. Byte integer of horizontal tiles in rectangle
        4. Byte integer of vertical tiles in rectangle
        5. Byte integer of color code (format depends
           on DAC color register settings)
Returns:
        Nothing
Assumes:
        ES --> video buffer base address
```

Chapter 10

VGA Geometrical Primitives

Topics:

- Geometrical graphics objects
- Plotting straight lines
- Plotting the conic curves
- Normalization and transformations
- Region fills

This chapter describes vector graphics in relation to the calculation and display of geometrical figures that can be expressed in a mathematical formula. Geometrical primitives are developed for calculating and displaying straight lines, circles, ellipses, parabolas, and hyperbolas, and also for performing rotation and clipping transformations and for filling the inside of a geometrical figure.

10.1 Geometrical Graphics Objects

Bitmapped graphics are used to encode and display pictorial objects, such as the running boar target in Figure 9-1. However, graphics applications often also deal with geometrical objects, that is, graphical objects that can be represented by means of algebraic equations; such is the case with straight lines, parallelograms, circles, ellipses, and other geometrical figures. The terms vector graphics, raster graphics, and object-oriented graphics are often used, somewhat imprecisely, when referring to computer graphics operations on geometrical objects.

In VGA graphics any image, including geometrical objects, can be encoded in a bitmap and displayed using the bitmap routines developed in Chapter 3. However, objects that can be represented mathematically can be treated by the graphics software in mathematical form. For example, it is often more compact and convenient to encode a screen circle by means of the coordinates of its origin and the magnitude of its radius than by representing all its adjacent points in a bitmap. The same applies to other geometrical figures, even to complex figures if they can be broken down into individual geometric elements.

10.1.1 Pixel-Path Calculations

In previous chapters we saw that the VGA graphics screen appears to the programmer as a two-dimensional pixel grid. Geometrical images on VGA can be visualized as points in this two-axes coordinate system, equated to x and y axes of the Cartesian plane. In dealing with geometrical figures the graphics programmer can use the equation of the particular curve to determine the pixel path that will represent it on the video screen or other device. In the VGA video display this process involves the calculation of the pixel addresses that lie along the path of the desired curve.

In high-level language graphics the software can use the language's mathematical functions. However, mathematical operations in high-level languages are generally considered too slow for screen graphics processing. Since performance is so important in video graphics programming, the preferred method of geometrical pixel plotting is usually the fastest one. In IBM microcomputers this often means low-level mathematics.

10.1.2 Graphical Coprocessors

One approach to performing the required pixel path calculations in the manipulation of geometrical images is the use of graphical coprocessor hardware. Several such chips have been implemented in silicon. For example, the XGA video graphics system, discussed in Chapter 6, contains a graphical coprocessor chip that assists in performing block fills, line drawings, logical mixing, masking, scissoring, and other graphics functions. Unfortunately, the VGA system does not contain a graphical coprocessor chip.

The 80x87 as a Graphical Coprocessor

Since no graphical coprocessor is included in VGA systems the programmer is often forced to use the central processor to perform geometrical and other calculations necessary in graphics software. But 80x86 mathematics are slow, cumbersome, and limited. However, most IBM microcomputers can be equipped with an optional mathematical coprocessor chip of the Intel 80x87 family. The power of the math coprocessor can be a valuable asset in performing the pixel path calculation required in the drawing of geometrical figures. For example, suppose that a graphics program that must plot a circular arc with a radius of z pixels, start coordinates at x1, y1 and end coordinates at x2, y2. One approach to this pixel-plotting problem is to calculate the set of x and y coordinates, starting at point x1, y1, and ending at x2, y2. The computations can be based on the Pythagorean expression

$$y = \quad r - x$$

where x and y are the Cartesian coordinates of the point and r is the radius of the circle. The software can assign consecutive values to the x variable, starting at x1, and calculate the corresponding values of the y variable that lie along the arc's path. The pixels can be immediately plotted on the video display or the coordinates can be stored in a memory table and displayed later.

It is in performing such calculations that the mathematical coprocessor can be of considerable assistance. One advantage is that many mathematical operations are directly available, for example, the FSQRT instruction can be used to calculate the square root of a number. On the other hand, the main processor is capable only of integer arithmetic. Therefore the calculation of powers, roots, exponential, and trigonometric functions on the 80x86 must be implemented in software. A second and perhaps more important factor is the speed at which the calculations are performed with the coprocessor, estimated at 30 to 50 times faster than with the CPU. Convenience and speed make the 80x87 a powerful programming tool for VGA geometrical graphics.

By limiting the calculations to integer values, the VGA programmer can simplify pixel plotting using the 80x87. We have seen that in VGA mode number 18 the y coordinate of a screen point can be represented by an integer in the range 0 to 479, and the x coordinate, an integer in the range 0 to 639. Since the 80x87 word integer format is a 16-bit value, with an identical numerical range as a main processor register, a graphical program can transfer integer data from processor to coprocessor, and vice versa. These manipulations are illustrated in the examples contained in the VGA3 module of the GRAPHSOL library. The details of programming the 80x87 coprocessor, which is beyond the scope of this book, can be found in several titles listed in the Bibliography.

Emulating the 80x87

One practical consideration is that the 80x87 mathematical coprocessor is an optional device in IBM microcomputers; the exceptions are the machines equipped with the 486 chip, in which coprocessor functions are built-in. This optional nature of the coprocessor determines that applications that assume the presence of this chip will not execute in machines not equipped with an 80x87. This could create a serious problem for graphics code that relies on the coprocessor for performing pixel plotting calculations. Fortunately there is a solution to this problem: a software emulation of the coprocessor.

An 80x87 emulator is a program that simulates, in software, the operations performed by the 80x87 chip. Once the emulator software is installed, machines not equipped with an 80x87 are able to execute 80x87 instructions, although at a substantial performance penalty. Ideally, a program that uses 80x87 code could test for the presence of an 80x87 hardware component; if one is found, the chip is used in the calculations, if not, its operation is simulated by an emulator program. In reality this ideal solution is possible only in machines that are equipped with the 80286 or 80386 CPU. The reason is that in 8086 and 8088 machines not equipped with an 8087 chip the presence of a coprocessor instruction hangs up the system in a wait forever loop.

This problem was solved in the design of the 80286 CPU by means of 2 bits in the Machine Status Word register. These bits, named Math Present (MP) and Emulate (EM), allow encoding the presence, absence, or availability of the 80287 component. If the EM bit is set, then the execution of a coprocessor opcode will automatically generate an interrupt 7 exception. In this case a handler vectored at this interrupt can select an emulated version of the opcode, and the machine will not hang up. A similar mechanism is used in the 80386 CPU, but not in the 8086 or the 8088 processors.

Therefore, 8086/8088 software that alternatively uses the real 8087 if one is present or the emulator if no chip is installed in the system must contain both real and emulated code. In this case a routine can be devised to test for the presence of the hardware component; if one is found, execution is directed to routines that use real 8087 code, if no 8087 is detected, the emulator software is initialized and execution is directed to routines that contain calls to the emulator package. Since this method works in any IBM microcomputer, it is the one adopted in the GRAPHSOL graphics library furnished with this book. The test for the presence of the 80x87 chip, the installation of the emulator software, and the setting of the code selection switch is performed in the INIT_X87 procedure in the VGA3 module of the GRAPHSOL library, which also contains the routines that use 80x87 hardware instructions. The emulated code for the geometrical calculation routines is found in the VGA3_EM module of the aforementioned library.

Over the years emulator programs have been made available by Intel, Ingenierburo Franke, and other sources.

10.2 Plotting a Straight Line

Geometrical figures can be drawn on the video display by mathematical pixel plotting when the pattern of screen pixels lies along a path that can be expressed in a mathematical equation. In this case the graphical software calculates successive coordinate pairs and sets the pixels that lie along the curve's path. We have mentioned that the 80x87 mathematical coprocessor is a valuable tool for performing these calculations rapidly and precisely. Figure 10-1 shows a pixel representation of three straight lines.

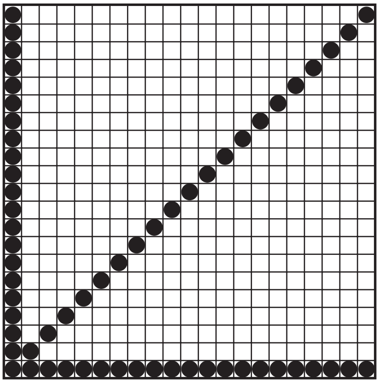


Figure 10-1 *Pixel Plots for Straight Lines*

In Figure 10-1 we see that horizontal and vertical lines are displayed on the screen by setting adjacent pixels. A line at a 45 degree angle can also be rendered accurately by diagonally adjacent pixels, although pixel to pixel distance will be greater in a diagonal line than in a horizontal or vertical one. But the pixel plot of a straight line that is not in one of these three cases cannot be exactly represented on a symmetrical grid, whether it be a pixel grid, or a quadrille drawing paper. Figure 10-2 shows the staircase effect that results from displaying an angled straight line on a symmetrical grid.

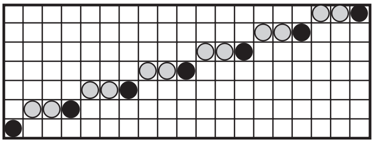


Figure 10-2 *Non-Adjacent Pixel Plot of a Straight Line*

Notice that the black-colored pixels in Figure 10-2 represent the coordinates that result from calculating successive unit increments along the vertical axis. If only the black colored dots were used to represent the straight line in Figure 10-2, the graph would be discontinuous and the representation not very accurate. An extreme case of this discontinuity would be a straight line at such a small angle that it would be defined by two isolated pixels, one at each screen border. In conclusion, if no corrective measures are used, the screen drawing of a line or curve by simple computation of pixel coordinates can produce unsatisfactory results. The non-adjacency problem can be corrected by filling in the intermediate pixels. This correction is shown in gray-colored pixels in Figure 10-2.

10.2.1 Insuring Pixel Adjacency

Notice that the pixel plotting routines in the VGA3 module of the GRAPHSOL library store in memory the coordinate pairs found during the calculations phase, rather than immediately setting the screen pixels. Due to this mode of operation, the program must establish the necessary structures for holding the data. The following code fragment shows several storage assignations used in the routines contained in the VGA3 module.

```
; Scratch-pad for temporary data
THIS_X   DW      0        ; Normalized coordinate of x
THIS_Y   DW      0        ; Normalized coordinate of y
LAST_Y   DW      0        ; Internal control for
                         ; noncontinuous y-axis points
; Buffers for storing 1K of x and y coordinates of the first
; quadrant and a counter for the number of points computed
Y_BUFFER        DB      2048 DUP (00H)
X_BUFFER        DB      2048 DUP (00H)
POINTS_CNT      DW      0        ; Number of entries in buffers
```

Programmers have devised many computational strategies for generating the coordinate pairs to plot geometrical curves. Consider, at this point, that the straight line is geometrically defined as a curve. One of the processing operations performed by pixel-plotting routines is to fill in the spaces left empty by the mathematical computations (see Figure 10-1), thus insuring that the screen pixels representing the curve are adjacent to each other. The following code fragment corrects nonadjacent plots generated by any pixel-plotting primitive that calculates y as a function of x. The routine also assumes that x is assigned consecutive values by decrementing or incrementing the previous x value by one.

```
;***********************|
;   Test for adjacent   |
;     y coordinates     |
;***********************|
; On entry:
;          CS:SI --> buffer holding x coordinates of curve
;          CS:DI --> buffer holding y coordinates of curve
; Adjacency correction is required if the previous y coordinate
; is not adjacent (one less) to the present y coordinate. Code
; assumes that the data variables are located in the code segment
        MOV     DX,CS:THIS_Y
        MOV     CX,CS:THIS_X
TEST_ADJACENT:
; Is this y < last y minus 1
        MOV     BX,CS:LAST_Y
        DEC     BX                      ; Last y minus 1
        CMP     DX,BX                   ; Compare to this y
        JL      FILL_IN_PIXEL
; Is this y > last y plus 1
        MOV     BX,CS:LAST_Y
        INC     BX                      ; Last y plus 1
        CMP     DX,BX                   ; Compare to this y
        JG      FILL_IN_PIXEL
        JMP     STORE_PIX_XYS
;***********************|
; correct non-adjacency |
;***********************|
; BX = last y coordinate minus 1
FILL_IN_PIXEL:
        MOV     CS:[SI],BX             ; Store y coordinate adjacent
                                       ; to previous point
        MOV     CS:[DI],CX             ; Store this x coordinate
        ADD     SI,2                   ; Bump pointers
        ADD     DI,2
        INC     CS:POINTS_CNT          ; Bump points counter
        MOV     CS:LAST_Y,BX           ; Update to this point
        JMP     TEST_ADJACENT
;***********************|
;   store coordinates   |
;***********************|
STORE_PIX_XYS:
        MOV     CS:[SI],DX             ; Store normalized
                                       ; y coordinate
        MOV     CS:[DI],CX             ; Store normalized x
                                       ; coordinate
; Bump both buffer pointers
        ADD     SI,2
        ADD     DI,2
        INC     CS:POINTS_CNT          ; Bump points counter
        MOV     CS:LAST_Y,DX           ; Update LAST_Y variable
        .
        .
        .
```

The auxiliary procedure named ADJACENT in the VGA3 module of the GRAPHSOL library uses similar logic as the above fragment.

10.2.2 Calculating Straight Lines Coordinates

A straight line in the Cartesian plane can be defined in several ways. One common mathematical expression consists of defining the line by means of the coordinates of its two end points. In this manner we can refer to a line with start coordinates at x1, y1 and end coordinates at x2, y2. An alternative way of defining a straight line is by means of the coordinates of its start point, its angle, and one of its end point coordinates. In this manner we can refer to a straight line from x1, y1, with a slope of 60 degrees, to a point at x2. Both expressions are useful to the graphics programmer.

Bresenham's Algorithm

One of the original algorithms for plotting a straight line between two points was developed by J. E. Bresenham and first published in the IBM Systems Journal in 1965. Bresenham's method consists of obtaining two pixel coordinates for the next y value and then selecting the one that lies closer to the path of an ideal straight line. The following code fragment shows the plotting of a straight line in VGA mode number 18 using Bresenham's method.

```
; Routine to draw a straight line with starting coordinates
; stored at CS:ORIGIN_X and CS:ORIGIN_Y and end coordinates at
; CS:END_X  and CS:END_Y
;
; Set unit increments for pixel-to-pixel operation
        MOV     CX,1
        MOV     DX,1
; Determine negative or positive slope from difference between
; the y and x coordinates of the start and end points of the line
; This difference is also the length of the line
        MOV     DI,CS:END_Y
        SUB     DI,CS:ORIGIN_Y  ; Length
        JGE     POS_VERTICAL    ; Vertical length is positive
        NEG     DX              ; DX = -1 in 2's complement form
        NEG     DI              ; Make distance positive
POS_VERTICAL:
        MOV     CS:INCR_FOR_Y,DX ; Increments on the y-axis will
                                ; be positive or negative
; Calculate horizontal distance
        MOV     SI,CS:END_X
        SUB     SI,CS:ORIGIN_X
        JGE     POS_HORZ        ; Horizontal length is positive
        NEG     CX              ; CX = -1 in 2's complement form
        NEG     SI              ; Distance has to be positive
POS_HORZ:
        MOV     CS:INCR_FOR_X,CX ; Increments on the x axis can
                                 ; also be positive or negative
; Compare the horizontal and vertical lengths of the line to
; determine if the straight segments will be horizontal (if
; this length is greater) or vertical (otherwise)
        CMP     SI,DI            ; SI = horizontal length
                                 ; DI = vertical length
        JGE     HORZ_SEGMENTS
; Vertical length is greater, straight segments are vertical
        MOV     CX,0             ; No horizontal segments
        XCHG    SI,DI            ; Invert lengths
        JMP     SET_CONTROLS
HORZ_SEGMENTS:
        MOV     DX,0             ; No vertical segments
```

```
SET_CONTROLS:
        MOV     CS:STRT_HSEGS,CX ; Will be 1 or 0
        MOV     CS:STRT_VSEGS,DX ; Also 1 or 0
; Calculate adjustment factor
        MOV     AX,DI            ; Smaller direction component
        ADD     AX,AX            ; Double the shorter distance
        MOV     CS:STRT_TOTAL,AX ; Straight component total
                                 ; pixels
        SUB     AX,SI            ; Subtract larger direction
                                 ; component
        MOV     BX,AX            ; General component counter
        SUB     AX,SI            ; Calculate
        MOV     CS:DIAG_TOTAL,AX ; Diagonal component total
                                 ; pixels
; Prepare to draw line
        MOV     CX,CS:ORIGIN_X
        MOV     DX,CS:ORIGIN_Y
; SI = the length of the line along the longer axis
        INC     SI
        MOV     AL,CS:LINE_COLOR ; Color code for line
;*********************|
;   draw line points  |
;*********************|
LINE_POINTS:
        DEC     SI               ; Counter for total pixels
        JNZ     PIX_DRAW
        JMP     END_OF_LINE      ; Line is finished
;*********************|
;   display pixel     |
;*********************|
PIX_DRAW:
        CALL    PIXEL_WRITE_18  ; Routine to set pixel in mode 18
        CMP     X,0             ; If BX < 0 then straight segment
                                ; diagonal segment otherwise
        JGE        DIAGONAL
; Draw straight line segments
        ADD     CX,CS:STRT_HSEGS ; Increment CX if horizontal
        ADD     DX,CS:STRT_VSEGS ; Increment DX if vertical
        ADD     BX,CS:STRT_TOTAL ; Counter plus adjustment
        JMP     LINE_POINTS
; Draw diagonal segment
DIAGONAL:
        ADD     CX,CS:INCR_FOR_X       ; X direction
        ADD     DX,CS:INCR_FOR_Y       ; Y direction
        ADD     BX,CS:DIAG_TOTAL       ; Adjust counter
        JMP     LINE_POINTS
END_OF_LINE:
        .
        .
        .
```

The procedure named BRESENHAM in the VGA3 module of the GRAPHSOL library is based on this algorithm.

An Alternative to Bresenham

A program can use the 80x87 code to calculate the coordinates of a line defined by means of its end points. In a machine equipped with the coprocessor hardware this method performs better than the Bresenham routine listed above. On the other hand, if the 80x87 code is to be emulated in software, then Bresenham's algorithm executes faster. The 80x87 calculations can be based on the differential equation for the slope of a straight line:

```
if
Dy/Dx = constant
then
Dy/Dx = (y2 - y1) / (x2 - x1)
therefore, the slope of the line is expressed:
m = Dy/Dx
```

The actual calculations are as follows:

```
; Memory variables stored in the code segment:
;          CS:X1 = x coordinate of leftmost point
;          CS:Y1 = y coordinate of leftmost point
;          CS:X2 = x coordinate of second point
;          CS:Y2 = y coordinate of second point
; During computations:
;          x coordinate ......... CS:THIS_X ......... Word
;          y coordinate ......... CS:THIS_Y ......... Word
; On exit:
;       CS:BUFFER_X holds the set of x coordinates for the line
;       CS:BUFFER_Y holds the set of y coordinates
;       CS:POINTS_CNT is a counter for the number of x,y pairs
;                     stored in BUFFER_X and BUFFER_Y
;*********************|
;     preparations    |
;*********************|
; Set registers and variables to coordinates
        LEA     SI,CS:Y_BUFFER        ; y buffer pointer
        LEA     DI,CS:X_BUFFER        ; x buffer pointer
        MOV     CS:LAST_Y,0           ; First iteration
        MOV     CS:POINTS_CNT,0       ; Reset points counter
;
; Calculate Dy/Dx (slope m)
;                            |  ST(0)   |   ST(1)  |   ST(2)  |
        FILD    CS:X1        ;   x1     |
        FILD    CS:X2        ;   x2     |     x1   |
        FSUB    ST,ST(1)     ; x2 - x1  |     x1   |
        FSTP    ST(1)        ; x2 - x1  |  empty   |
; Store in variable for the normalized x coordinate of
; start point
        FIST    CS:THIS_X
        FILD    CS:Y1        ;   y1     |  x2 - x1 |
        FILD    CS:Y2        ;   y2     |     y1   | x2 - x1  |
        FSUB    ST,ST(1)     ; y2 - y1  |     y1   | x2 - x1  |
        FSTP    ST(1)        ; y2 - y1  |  x2 - x1 |  empty   |
        FDIV    ST,ST(1)     ; Dy/Dx    |  x2 - x1 |
        FSTP    ST(1)        ; Dy/Dx    |  empty   |
;*********************|
;   y coordinate      |
;   calculations      |
;*********************|
Y_POINT:
        FILD    CS:THIS_X    ;    x     |  Dy/Dx   |         |
; Solve y = x * Dy/Dx
        FMUL    ST,ST(1)     ; x*Dy/Dx  |  Dy/Dx   |
; Store in variable for normalized y coordinate of this point
        FISTP   CS:THIS_Y    ; Dy/Dx    |  empty   |
;********************|
;  test for adjacent  |
;      y values       |
;********************|
        CALL    ADJACENT     ; Adjacency procedure
;********************|
; test for last pixel |
;********************|
        CMP     CS:THIS_X,0    ; x = 0 must be calculated
        JE      EXIT_POINTS
        DEC     CS:THIS_X
        JMP     Y_POINT
; Adjust 80x87 stack pointer
EXIT_POINTS:
        FSTP    ST(0)
        .
        .
        .
```

A Line by its Slope

We saw, in the previous example, that a straight line can be defined by its slope. The mathematical expression for this line, called the point-slope form, in which the y coordinate is a function of the x coordinate can be expressed in the following equation:

$$y = mx$$

where x and y are the coordinate pairs and m is the slope. The slope is the difference between the y coordinates divided by the difference between the x coordinates of any two points in the line, expressed as follows:

$$m = (y2 - y1) / (x2 - x1).$$

Notice that y2 - y1 can have a positive or negative value, therefore m can be positive or negative. The following code fragment calculates the y coordinates for successive x coordinates using the point-slope equation for a straight line. The calculations, which use 80x87 code, assume that a real or emulated 80x87 coprocessor is available.

```
; Routine to plot and store the pixel coordinates of a straight
; line of slope s, located in the fourth quadrant
; The slope is in degrees and must be in the range 0 < s > 90
;
; On entry:
;            CS:X1    = x coordinate of origin
;            CS:Y1    = y coordinate of origin
;            CS:X2    = x coordinate of end point
;            CS:SLOPE = slope in degrees
; During computations:
;         X coordinate .......... CS:THIS_X ......... word
;         Y coordinate .......... CS:THIS_Y ......... word
;
; Formula:
;         y = x Tan s
;********************|
;     preparations    |
;********************|
; Set registers and variables to coordinates
        LEA     SI,CS:Y_BUFFER  ; y buffer pointer
        LEA     DI,CS:X_BUFFER  ; x buffer pointer
        MOV     CS:LAST_Y,0     ; First iteration
        MOV     CS:POINTS_CNT,0 ; Reset points counter
; Calculate the normalized x coordinate for the rightmost point
;                              |  ST(0)   |   ST(1)  |   ST(2)  |
        FILD    CS:X1          ;   x1     |
        FILD    CS:X2          ;   x2     |     x1   |
        FSUB    ST,ST(1)       ; x2 - x1  |     x1   |
        FSTP    ST(1)          ; x2 - x1  |  empty   |
; Store in variable for the normalized x coordinate of
; rightmost point
        FIST    THIS_X
        FSTP    ST(0)         ; empty    |
; Obtain and store tangent of slope
;                             |  ST(0)   |   ST(1)  |   ST(2)  |
        FILD    CS:SLOPE      ; s(deg)   |
        CALL    DEG_2_RADS    ; s(rads)  |
        CALL    TANGENT       ; tan s    |
Y_BY_SLOPE:
;                             |  tan s   |
        FILD    CS:THIS_X     ;     x    |  tan s  |
        FMUL    ST,ST(1)      ;     y    |  tan s  |
; Store in variable for normalized y coordinate of this point
        FISTP   CS:THIS_Y     ;  tan s   |
;*********************|
;  test for adjacent  |
;      y values       |
;*********************|
        CALL    ADJACENT      ; Adjacency test procedure
;*********************|
; test for last pixel |
;*********************|
        CMP     CS:THIS_X,0     ; x = 0 must be calculated
        JE      EXIT_SLOPE
        DEC     CS:THIS_X
        JMP     Y_BY_SLOPE
; Adjust 8087 stack registers
EXIT_SLOPE:
        FSTP    ST(0)
        .
        .
        .
```

Notice that the graphic primitives named BRESENHAM and LINE_BY_SLOPE, in the VGA3 module of the GRAPHSOL library, share several code segment variables. Also that the procedure named ADJACENT is called by the LINE_BY_SLOPE primitive to correct nonadjacent pixel conditions that can arise during the plotting calculations. The VGA3 module includes a local procedure, named TANGENT, that performs the calculations for the tangent function required in the line-by-slope formula. Since the calculations performed by the TANGENT procedure use the radian measure of the angle, the auxiliary procedure named DEG_2_RADS in the VGA3 module is used to convert from degrees to radian.

Displaying the Straight Line

The LINE_BY_SLOPE procedure in the VGA3 module of the GRAPHSOL library is limited to calculating and storing the pixel coordinates of the straight line defined by the caller. This mode of operation makes the routine more device independent and also makes possible certain manipulations of the stored data. However, most applications will, sooner or later, need to draw the line on the screen. The following code fragment shows the necessary operations.

```
; Display coordinates stored in CS:X_BUFFER and CS:Y_BUFFER
; Total number of coordinates is stored in CS:POINTS_CNT
; Setup pointers and counter
        LEA     SI,CS:Y_BUFFER       ; y coordinates
        LEA     DI,CS:X_BUFFER       ; x coordinates
        MOV     CX,CS:POINTS_CNT
        MOV     CS:OPS_CNT,CX        ; Operational counter
DISP_1:
        MOV     CX,CS:X1             ; x coordinate of origin
        MOV     DX,CS:Y1             ; y coordinate of origin
; Add stored values to origin
        ADD     CX,WORD PTR CS:[DI]
        SUB     DX,WORD PTR CS:[SI]
; CS:CX = x coordinate, CS:DX = y coordinate of point
        PUSH    AX                 ; Save color code
        CALL    PIXEL_ADD_18       ; Procedures in VGA1 module
        CALL    WRITE_PIX_18
        POP     AX                 ; Restore color code
        ADD     CS:SI,2            ; Bump coordinates pointers
        ADD     CS:DI,2
        DEC     CS:OPS_CNT         ; Operation points counter
        JNZ     DISP_1
        .
        .
        .
```

The procedure named DISPLAY_LINE in the VGA3 module of the GRAPHSOL library can be used to display a straight line plotted by means of the LINE_BY_POINTS procedure. Notice that the procedure named BRESENHAM displays the pixels as the coordinates are calculated.

10.3 Plotting Conic Curves

By intersecting a right circular cone at different planes it is possible to generate several geometrical curves. These curves, or conic sections, are the circle, the ellipse, the parabola, and the hyperbola. A VGA graphics program can plot the coordinates of the conic curves employing similar methods as the ones developed for plotting straight lines (see Section 10.2).

10.3.1 The Circle

A circle in the Cartesian plane can be described by the coordinates of its origin, and by its radius. As far as the calculation of the coordinate points only the radius parameter is necessary, although the origin coordinates is required to position the circle in the viewport. To calculate the pixel coordinates of a circle described by its radius we can use the Pythagorean formula, which allows us to obtain the corresponding values of y for each x. The curve and formula can be seen in Figure 10-3.

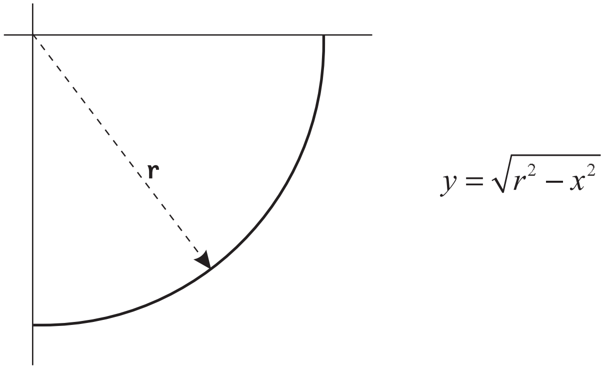


Figure 10-3 *Plot and Formula for a Circle*

The following code fragment shows the calculations necessary for plotting the coordinates of a circular arc in the fourth quadrant. The calculations are performed by means of the 80x87 mathematical coprocessor.

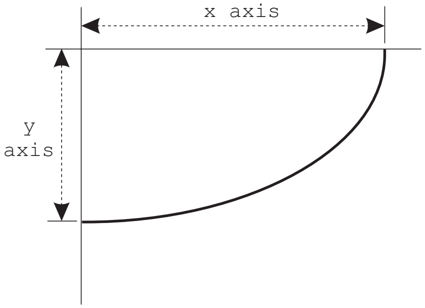
```
; Routine to plot and store the pixel coordinates of a circular
; arc in the fourth quadrant
;
; On entry:
;         Radius: ........... CS:R ......... word
;
; During computations:
;         x coordinate ... CS:THIS_X ........ word
;         y coordinate ... CS:THIS_Y ........ word
;********************|
;    preparations    |
;********************|
; Reset counters and controls
        MOV     CS:THIS_X,0           ; Start values for x
        MOV     CS:LAST_Y,0           ; and LAST_Y
; Buffer pointers:
;       SI --> Y values buffer
;       DI --> X values buffer
        LEA     SI,CS:Y_BUFFER
        LEA     DI,CS:X_BUFFER
        MOV     CS:POINTS_CNT,0        ; Reset counter
;********************|
; calculate y values |
;********************|
CIRCLE_Y:
;                               |  ST(0)    |   ST(1)  |
        FILD    CS:THIS_X      ;    x      |
        FMUL    ST,ST(0)       ;    x^2    |
        FILD    CS:R           ;    r      |    x^2
        FMUL    ST,ST(0)       ;    r^2    |    x^2
        FSUB    ST,ST(1)       ; r^2 - x^2 |    x^2
        FSQRT                  ;Rt(r^2-x^2)|    x^2
        FISTP   CS:THIS_Y      ;    x^2    |
        FSTP    ST(0)          ;  EMPTY    |
; Test adjacency condition
        CALL    ADJACENT       ; Library procedure
        INC     CS:THIS_X      ; x increments in a circle's
```

```
                              ; fourth quadrant
        CMP     CS:THIS_Y,0    ; Test for end of execution
        JNE     CIRCLE_Y
; At this point all coordinates have been plotted
        .
        .
        .
```

The procedure named CIRCLE in the VGA3 module of the GRAPHSOL library can be used to plot the coordinates of a circular arc in the fourth quadrant. The code used by this procedure is similar to the one in the preceding listing.

10.3.2 The Ellipse

An ellipse in the Cartesian plane can be described by the coordinates of its origin, and by its major and minor semi-axes. As far as the calculation of the coordinate points only the axes parameters are necessary, although the origin coordinates will be required to position the ellipse in the viewport. The curve and formula can be seen in Figure 10-4.



$$y = \sqrt{m^2\left(1 - \frac{x^2}{M^2}\right)}$$

Figure 10-4 *Plot and Formula for Ellipse*

In Figure 10-4, the variable M represents the major semi-axis of the ellipse and the variable m, the minor semi-axis. The following code fragment shows the calculations necessary for plotting the coordinates of an elliptical curve in the fourth quadrant.

```
; Routine to plot and store the pixel coordinates of an
; elliptical curve in the fourth quadrant
;
; On entry:
;       x semi-axis (M) .... CS:X_AXIS ........ word
;       y semi-axis (m) .... CS:Y_AXIS ........ word
;
; During computations:
;          x coordinate ... CS:THIS_X ........ word
;          y coordinate ... CS:THIS_Y ........ word
;
; Variables:
;      CS:m = minor axis (X_AXIS or Y_AXIS variables)
;      CS:M = major axis (X_AXIS or Y_AXIS variables)
;
;*********************|
;     preparations    |
;*********************|
; Reset counters and controls
        MOV     CS:THIS_X,0         ; Start value for x
        MOV     CS:LAST_Y,0         ; and for LAST_Y
;
; Buffer pointers:
;       SI --> Y values buffer
;       DI --> X values buffer
        LEA     SI,CS:Y_BUFFER
        LEA     DI,CS:X_BUFFER
        MOV     CS:POINTS_CNT,0     ; Reset counter
;
ELLIPSE_Y:
; Calculate primitive coordinate of y
; First solve x^2 / M^2
;                               |  ST(0)     |   ST(1)  |
        FILD    CS:X_AXIS       ;    M       |
        FMUL    ST,ST(0)        ;    M^2     |
        FILD    CS:THIS_X       ;    x       |     M^2   |
        FMUL    ST,ST(0)        ;    x^2     |     M^2   |
        FDIV    ST,ST(1)        ; x^2/M^2    |     M^2   |
; Solve 1 - (x^2 / M^2)
        FLD1                    ;    1       | x^2/M^2   | ? |
        FSUB    ST,ST(1)        ;1-(x^2/M^2) | x^2/M^2   | ? |
; Solve  m^2 * [1-(x^2/M^2)]
        FILD    CS:Y_AXIS       ;    m       |1-(x^2/M^2)| ? | ? |
        FMUL    ST,ST(0)        ;    m^2     |1-(x^2/M^2)| ? | ? |
        FMUL    ST,ST(1)        ;m^2 * [1-(x^2/M^2)]|  ? | ? | ? |
; Find square root
        FSQRT                   ;    y       | ? | ? | ? |
        FISTP   CS:THIS_Y       ; Store y in memory
; Adjust stack
        FSTP    ST(0)           ; ? | ? |
        FSTP    ST(0)           ; ? |
        FSTP    ST(0)           ; Stack is empty
; Insure pixel adjacency condition
        CALL    ADJACENT        ; Library procedure
        INC     CS:THIS_X       ; x increments in a clockwise plot
                                ; of the first quadrant
        CMP     CS:THIS_Y,0     ; Test for end of processing
        JNE     ELLIPSE_Y
; At this point all coordinates have been plotted
        .
        .
        .
```

The procedure named ELLIPSE in the VGA3 module of the GRAPHSOL library can be used to plot the coordinates of an elliptical curve in the fourth quadrant. The code used by this procedure is similar to the one in the preceding listings.

10.3.3 The Parabola

A parabola in the Cartesian plane can be described by the coordinates of its origin and by its focus. The curve and formula can be seen in Figure 10-5.

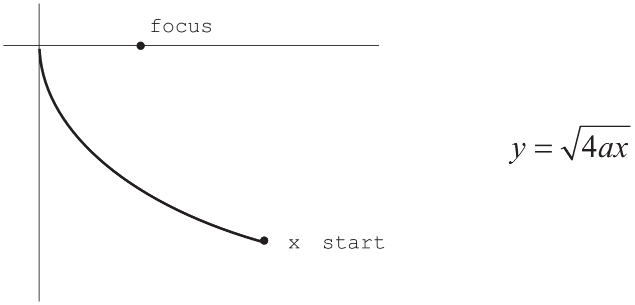


Figure 10-5 *Plot and Formula for Parabola*

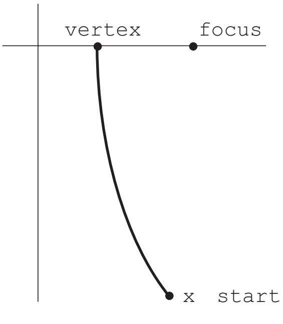
In order to plot and store the coordinates of a parabolic curve two input parameters are required: the focus of the parabola and the start value for the x coordinate. Notice that no initial x value is required in the circle and the ellipse plotting routines, while the routines for plotting a parabola and a hyperbola both require an initial value for the x coordinate. The reason for this difference is that the circle and the ellipse are closed curves, therefore their fourth quadrant plot extends from axis to axis. On the other hand, the parabola and the hyperbola are open curves, therefore a start value for the x coordinate is required to define the curve. The following code fragment shows the calculations necessary for plotting the coordinates of a parabolic curve in the fourth quadrant.

```
; Routine to plot and store the pixel coordinates of a parabolic curve
; in the fourth quadrant
;
; On entry:
;    focus of parabola  .......... CS:FOCUS  ......... word
;    start x coordinate .......... CS:X_START ........ word
;
; During computations:
;          x coordinate .......... CS:THIS_X ......... word
;          y coordinate .......... CS:THIS_Y ......... word
; Formula:
;       y = SQR. ROOT (4ax)
;       Y_ABS = SQR. ROOT (4 * FOCUS * X_ABS)
;
;*********************|
;     preparations    |
;*********************|
; Reset counters and controls
        MOV     AX,CS:X_START         ; Start value for X
        MOV     CS:THIS_X,AX
        MOV     CS:LAST_Y,0           ; Reset LAST_Y
; Buffer pointers:
;       SI --> Y values buffer
;       DI --> X values buffer
        LEA     SI,CS:Y_BUFFER
        LEA     DI,CS:X_BUFFER
        MOV     CS:POINTS_CNT,0       ; Reset counter
PARA_Y:
; Calculate primitive coordinate of y
;       y = SQR. ROOT (4ax)
;       THIS_Y = SQR. ROOT (4 * FOCUS * THIS_X)
;                              |   ST(0)    |   ST(1)   |   |   |
        FILD    CS:THIS_X      ;     x      |
        FILD    CS:FOCUS       ;     a      |      x    |
        FMUL    ST,ST(1)       ;    ax      | ? |
        FLD1                   ;    1       |     ax    | ? |
        FADD    ST,ST(0)       ;    2       |     ax    | ? |
        FADD    ST,ST(0)       ;    4       |     ax    | ? |
        FMUL    ST,ST(1)       ;   4ax      | ? | ? |
        FSQRT                  ;    y       | ? | ? |
        FISTP   CS:THIS_Y      ; Store y in memory
; Adjust stack
        FSTP    ST(0)          ; ? |
        FSTP    ST(0)          ; Stack is empty
; Insure pixel adjacency conditions
        CALL    ADJACENT       ; Library procedure
        DEC     CS:THIS_X
        CMP     CS:THIS_Y,0    ; Test for end of processing
        JNE     PARA_Y
; At this point all coordinates have been plotted
        .
        .
        .
```

The procedure named PARABOLA in the VGA3 module of the GRAPHSOL library can be used to plot the coordinates of a parabolic curve in the fourth quadrant. The code used by this procedure is similar to the one in the preceding listings.

10.3.4 The Hyperbola

A hyperbola in the Cartesian plane can be described by its focus, vertex, and by the coordinates of its start point. The curve and formula can be seen in Figure 10-6.



$$y = \frac{\sqrt{c^2 - a^2}}{a\sqrt{x^2 - a^2}}$$

Figure 10-6 *Plot and Formula for Hyperbola*

In order to plot and store the coordinates of a hyperbolic curve the routine requires the focus and vertex parameters, as well as the start value for the x coordinate. The following code fragment shows the calculations necessary for plotting the coordinates of a hyperbolic curve in the fourth quadrant.

```
; Routine to plot and store the pixel coordinates of a hyperbolic
; curve in the fourth quadrant
;
```

```
; On entry:
;    focus of hyperbola .......... CS:FOCUS  ......... word
;    vertex of hyperbola ......... CS:VERTEX ......... word
;    start x coordinate .......... CS:X_START ........ word
;
; During computations:
;          X coordinate .......... CS:THIS_X ......... word
;          Y coordinate .......... CS:THIS_Y ......... word
;
; Scratch-pad variables:
;          Numerator radix ..... CS:B_PARAM .......... word
;          Vertex squared ...... CS:VERTEX2 .......... word
;
;*********************|
;     preparations    |
;*********************|
; Reset counters and controls
        MOV     AX,CS:X_START        ; Start value for X
        MOV     CS:THIS_X,AX
        MOV     CS:LAST_Y,0          ; Reset LAST_Y
; Buffer pointers:
;       SI --> Y values buffer
;       DI --> X values buffer
        LEA     SI,CS:Y_BUFFER
        LEA     DI,CS:X_BUFFER
        MOV     CS:POINTS_CNT,0      ; Reset counter
; Compute numerator radical from VERTEX and FOCUS
; Solve: B_PARAM = SQR. ROOT (FOCUS^2 - VERTEX^2)
;                             |   ST(0)    |   ST(1)   |   |   |
        FILD    CS:VERTEX     ;     a      |
        FMUL    ST,ST(0)      ;     a^2    |
        FILD    CS:FOCUS      ;     c      |    a^2    |
        FMUL    ST,ST(0)      ;     c^2    |    a^2    |
        FSUB    ST,ST(1)      ;  c^2 - a^2 |    a^2    |
        FSQRT                 ;     b      |    a^2    |
; Store b
        FISTP   CS:B_PARAM    ;    a^2     |
; Store VERTEX^2 for calculations
        FISTP   CS:VERTEX2    ; Stack is empty
HYPER_Y:
; Calculate primitive coordinate of y
;  y = b / a * SQR ROOT (x^2 - a^2)
;  or:
;  Y_ABS = B_PARAM / VERTEX * SQR ROOT (X_ABS^2 - VERTEX2)
;                             |   ST(0)    |   ST(1)   |   |   |
        FILD    CS:VERTEX2    ;   a^2      |
        FILD    CS:THIS_X     ;    x       |    a^2    |
        FMUL    ST,ST(0)      ;    x^2     |    a^2    |
        FSUB    ST,ST(1)      ; x^2-a^2    | ? |
        FSQRT                 ; SR(x^2-a^2)| ? |
        FILD    CS:B_PARAM    ;     b      | # | ? |
        FILD    CS:VERTEX     ;     a      |     b     | # | ? |
        FDIV    ST(1),ST      ;     b      |    b/a    | # | ? |
        FSTP    ST(0)         ;    b/a     | # | ? |
        FMUL    ST,ST(1)      ;    y       | ? | ? |
        FISTP   CS:THIS_Y     ; Store y in memory
; Adjust stack
        FSTP    ST(0)         ; ? |
        FSTP    ST(0)         ; Stack is empty
; Insure pixel adjacency condition
```

```
        CALL    ADJACENT      ; Library procedure
        DEC     CS:THIS_X
        CMP     CS:LAST_Y,0   ; Test for end of processing
        JNE     HYPER_Y
; At this point all coordinates have been plotted
        .
        .
        .
```

The procedure named HYPERBOLA in the VGA3 module of the GRAPHSOL library can be used to plot the coordinates of a hyperbolic curve in the fourth quadrant. The code used by this procedure is similar to the one in the preceding listings.

10.3.5 Displaying the Conic Curve

The procedure DISPLAY_LINE, developed previously, outputs to the CRT display, in VGA mode number 18, the pixel patterns stored by the line plotting routine. The DISPLAY_LINE procedure assigns a positive value to all the coordinates stored in X_BUFFER and Y_BUFFER. This determines that the displayed curve is always located in the fourth quadrant.

Notice that the routines for plotting and storing the coordinates of the four conic curves (circle, ellipse, parabola, and hyperbola), described in the previous sections, assume that the curve is located in the fourth Cartesian quadrant. In other words, the plotted curves are normalized to the signs of x and y in this quadrant. However, at display time, it is possible to change the sign of the coordinates so that the curve can be located in any one of the four quadrants.

The VGA3 module of the GRAPHSOL library, furnished with the book, includes four procedures to display the conic curves in any one of the four quadrants. These primitives are named QUAD_I, QUAD_II, QUAD_III, and QUAD_IV. The procedure named DO_4_QUADS can be used to display the curve in all four Cartesian quadrants.

10.4 Geometrical Operations

The design of program structures to be used in storing graphics image data is one of the most challenging tasks of designing a graphic system or application. The details of the storage format depend on several factors:

1. The programming language or languages that manipulate the stored data.
2. The available storage resources.
3. The transformations applied to the stored images.

In the manipulation of graphical data it is usually preferable to design independent procedures to interface with the data structures. An advantage of this approach is that the routines that perform the graphics transformations are isolated from the complexities of the storage scheme. Principles of memory economy usually advise that each data item be encoded in the most compact format that allows representing the full range of allowed values. Also that a data structure should not be of a predetermined size, but that its size be dynamically determined according to the number of parameters to be stored.

In implementing these rules the more elaborate graphics systems or applications create a hierarchy of image files, display files, and image segments of varying degrees of complexity. The entire structure is designed to facilitate image transformation by manipulating the stored data. For example:

1. An image can be mirrored to the other Cartesian quadrants by changing the sign of its coordinates.
2. An image can be translated (moved) by performing signed addition on its coordinates.
3. An image can be rotated by moving its coordinates along a circular arc. The rotation formulas are obtained from elementary trigonometry.
4. An image can be scaled by multiplying its coordinates by a scaling factor.
5. An image can be clipped by eliminating all the points that fall outside a certain boundary.

At the lowest level, the ideal storage structure for image coordinates is in a matrix form. A matrix is a mathematical concept in which a set of values is arranged in a rectangular array. Each value in the array is called an element of the matrix. In the context of graphics programming, matrices are used to hold the coordinate points of graphical figures. This form of storing graphical data allows the use of the laws of linear algebra to perform geometrical transformations by performing mathematical operations on the matrix.

In the VGA3 module we have used a very simple storage scheme in which the image coordinate points are placed in two rectangular matrices: X_BUFFER holds the x coordinates and Y_BUFFER the y coordinates. Although each matrix is stored linearly, the programmer can visualize it as a two-dimensional array by screen columns and rows. The geometrical routines operate on normalized coordinates. In other words, the code calculates the pixel pattern for a line or a conic curve independently of the screen position at which the curve is displayed. In this manner, once the basic coordinates for a given curve have been calculated and stored, the software can display as many curves as necessary in any screen position. Furthermore, since the conic curves are symmetrical in all four quadrants, only the coordinates of one quadrant need to be calculated. The images in the other quadrants are obtained by operating on the stored data.

10.4.1 Screen Normalization of Coordinates

To further simplify calculations for VGA mode number 18, the origin of the coordinate system is relocated on the Cartesian plane so that the screen map of 640 by 480 pixels lies entirely in one quadrant. Also, the values of the y coordinate are made to grow downward, as in the conventional representation of the video screen. This concept is shown in Figure 10-7.

The use of only positive values for representing the x and y coordinate points simplifies image calculations and manipulations.

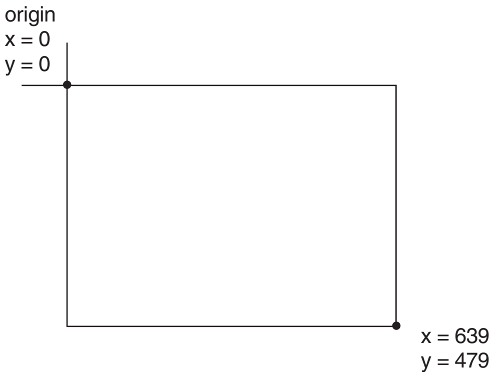


Figure 10-7 *Normalization of Coordinates in VGA Mode 18*

10.4.2 Performing the Transformations

The routines named QUAD_I, QUAD_II, QUAD_III, and QUAD_IV, in the VGA3 module of the GRAPHSOL library, display at any desired screen position the pixel coordinate pairs stored in X_BUFFER and Y_BUFFER. Since the coordinates are stored in screen-normalized form (see Section 10.4.1), the display routines must make the corresponding sign correction at the time of translating the image map to the specific screen position. For example, to display an image in the first quadrant the QUAD_I routine adds the pixel column at which the image is to be displayed to each of the coordinates in the matrix named X_BUFFER, and subtracts the pixel row from each coordinate in Y_BUFFER. Table 10-1 shows the operations performed on the screen-normalized coordinate pairs according to the quadrant.

Table 10-1

Transformation of Normalized Coordinates by Quadrant in VGA

QUAADRANT I		QUADRANT II		QUADRANT III		QUADRANT IV	
x	y	x	y	x	y	x	y
+	−	+	−	+	−	+	−

Translation

Translation is the movement of a graphical object to a new location by adding a constant value to each coordinate point. The operation requires that a constant be added to all the coordinates, but the constants can be different for each plane. In other words, a two-dimensional graphical object can be translated to any desired screen position by adding or subtracting values from the set of x and y coordinates that define the object. Notice that display routines QUAD_I, QUAD_II, QUAD_III, and QUAD_IV in fact perform an image translation from the screen top left corner to the screen position requested by the caller. The VGA3 module also contains a routine named DO_4_QUADS that displays an image in all four Cartesian quadrants.

Scaling

In graphical terms, to scale an image is to apply a multiplying factor to its linear dimensions. Thus, a scaling transformation is the conversion of a graphical object into another one by multiplying each coordinate point that defines the object. The operation requires that all the coordinates in each plane be multiplied by the same scaling factor, although the scaling factors can be different for each plane. For example, a three-to-four scaling transformation takes place when the x coordinates of a two-dimensional object are multiplied by a factor of two and the y coordinates are multiplied by a factor of four.

The fundamental problem of scaling a pixel map is that the resulting image can be discontinuous. This determines that it is often easier for the software to calculate the parameters that define the image, rather than to scale those of an existing one. For this reason we have not provided a scaling routine in the VGA3 library.

Rotation

Rotation is the conversion of a graphical object into another one by moving, by the same angular value, all coordinate points that define the original object along circular arcs with a common center. The angular value is called the angle of rotation, and the fixed point common to all the arcs is called the center of rotation. Some geometrical figures are unchanged by some rotations. For example, a circle is unchanged by a rotation about its center, and a square is unchanged if it is rotated by an angle that is a multiple of 90 degrees and using the intersection point of both diagonals as a center of rotation.

To perform a rotation transformation each coordinate that defines the object is moved along a circular arc. The effect of a 30 degree counterclockwise rotation of a polygon can be seen in Figure 10-8.

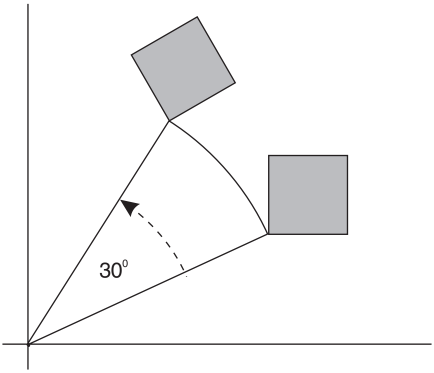


Figure 10-8 *Rotation Transformation of a Polygon*

The rotation formulas, which can be derived using elementary trigonometry, are:

```
x' = x cos @ - y sin @
y' = y cos @ + x sin @
```

where x',y' are the rotated coordinates of the point x,y and @ is the angle of rotation in clockwise direction. Since the rotation calculations require the sine and cosine functions, the VGA3 module includes the procedures SINE and COSINE that calculate these trigonometric functions. Notice that the calculations performed by the SINE and COSINE procedures use the radian measure of the angle. The auxiliary procedure named DEG_2_RADS, in the VGA3 module, perform the conversion from degrees to radians. Rotation is performed by the procedures named ROTATE_ON and ROTATE_OFF. The actual rotation calculations are performed by the local procedure named ROTATE.

Clipping

The graphical concept of clipping is related to that of a clipping window. In general, a graphics window can be defined as a rectangular area that delimits the computer screen, also called the viewport. Clipping a graphical object is excluding the parts of this object that lie outside a defined clipping window. Figure 10-9 shows the clipping transformation of an ellipse.

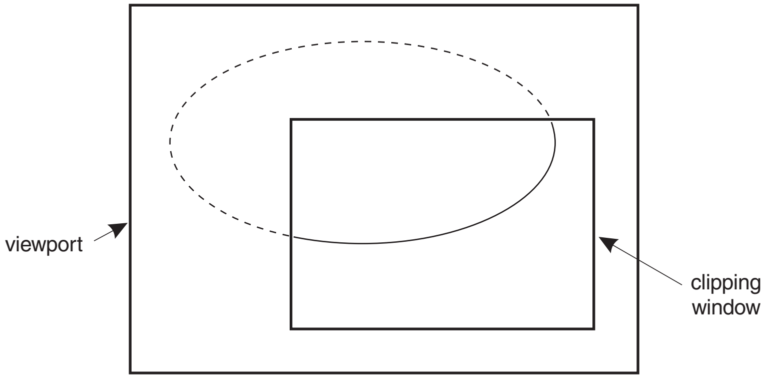


Figure 10-9 *Clipping Transformation of an Ellipse*

In Figure 10-9 the dotted portion of the ellipse, which lies outside of the clipping window, is eliminated from the final image, while the part shown in a continuous line is preserved. In the VGA3 library clipping is performed by the procedures named CLIP_ON and CLIP_OFF. The actual clipping calculations are done by the local procedure named CLIP.

Notice that in the VGA3 module the actual translation, rotation, and clipping transformations are done at display time, with no change to the stored image. In this manner, a program to display the clipped ellipse in Figure 10-9 would first call the ELLIPSE procedure, which calculates and stores the coordinates of the curve to screen-normalized parameters. Then the program calls the CLIP_ON procedure and defines the clipping window. Finally the DO_4_QUADS routine can be used to translate the ellipse to the actual screen position and display those portions of the curve that lie inside the clipping rectangle. If a rotation transformation is to be used, it must be executed before the clipping takes place.

Clipping transformations can also be used to make sure that the coordinate points of a geometrical image are within the physical limits of the graphics device. For example, an application working in VGA mode number 18, with a screen definition of 640 pixel columns by 480 pixel rows, can set the clipping rectangle to the dimensions of this viewport to make sure that the display routines do not exceed the physical screen area. In this manner the clipping routine serves as an error trap for the display function.

10.5 Region Fills

The graphics routines described in the previous sections of this chapter were designed to display the outline of a geometrical figure in the form of a continuous pixel line. But often a graphics application needs to display geometrical images filled with a uniform color or with a monochrome pattern. If the geometrical figure delimits a closed screen area, it is possible to use a fill operation to set all the pixels within the enclosed area to a specific color or pattern. This enclosed area is sometimes called a region.

10.5.1 Screen Painting

The name screen painting is usually given to routines that perform a general region fill in which all closed screen areas are colored with the value of its border pixels. The border pixels serve as a boundary for the fill operation. The logic of many screen painting routines is based on alternating between a searching and a coloring mode. One variation is to define a background color and then to scan the entire screen, pixel by pixel, searching for pixels that do not match the background. These non-matching pixels are said to define a boundary. When a boundary pixel is encountered, the searching mode is changed to the coloring mode, and each successive pixel is changed to the color of the boundary pixel. When another boundary pixel is encountered, the mode is toggled back to searching.

In screen painting algorithms the scanning usually starts at the top-left screen corner. The mode is changed to searching at the start of each new pixel row. The algorithms must take into account conditions that require special handling, for example, how to proceed if there is a single boundary pixel on a scan line, several adjoining boundary pixels, an odd number of boundaries, or if a vertex is encountered.

10.5.2 Geometrical Fills

The geometrical fill is a special case of the fill algorithms that is suited to filling closed geometrical figures with a given color or pattern. The geometrical fill is different from a general painting case in that in the geometrical fill the caller must define a pixel location inside the figure. The simplest case is based on the following assumptions:

1. That the starting location, sometimes called the seed point, is inside a closed-boundary figure within the viewport.
2. That there are no other figures or lines within the boundary of the figure to be filled.
3. That all consecutive points within the same horizontal line are adjacent.

Figure 10-10 shows two classes of geometrical shapes in regards to a region fill operation.

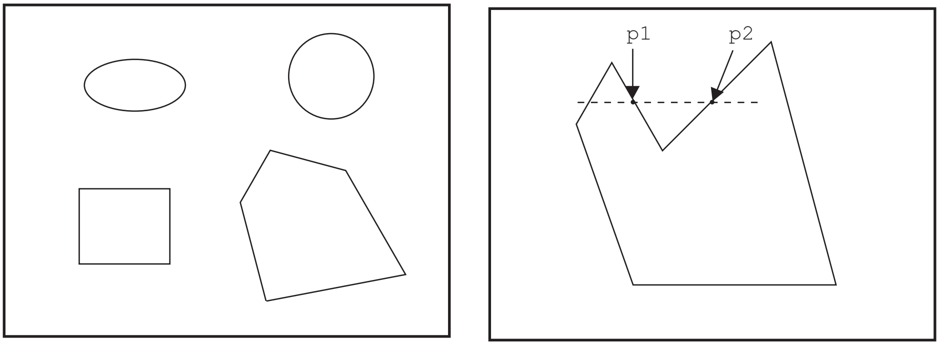


Figure 10-10 *Geometrical Interpretation of a Region Fills*

The geometrical shapes in Figure 10-10a meet the constraints defined above, while the polygon in Figure 10-10b does not. In Figure 10-10-b, consecutive points p1 and p2, located on the same horizontal line, are not adjacent. The simplest fill algorithm, based on a line-by-line scan for a single boundary pixel, works only with geometric figures similar to those in Figure 10-10-a. The logic requires a preliminary search for the figure's low and high points. This precursory operation simplifies the actual fill by insuring that the scan does not exceed the figure's boundaries at any time, therefore avoiding the tests for vertices and for external boundaries. Figure 10-11 is a flowchart of a region fill algorithm for a figure that meets the three constraints mentioned above.

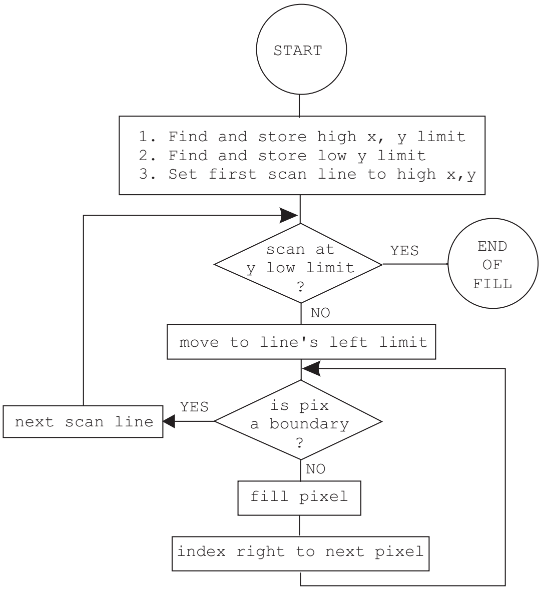

START
1. Find and store high x, y limit
2. Find and store low y limit
3. Set first scan line to high x,y
scan at y low limit ?
YES
END OF FILL
NO
move to line's left limit
is pix a boundary ?
YES
next scan line
NO
fill pixel
index right to next pixel

Figure 10-11 *Region Fill Flowchart*

The procedure named REGION_FILL in the VGA3 module of the GRAPHSOL library furnished with this book performs a region fill operation on geometrical shapes of the type shown in Figure 10-10a. The logic of this routine is based on the flowchart in Figure 10-11.

An algorithm like the one illustrated in the flowchart of Figure 10-11 is sometimes classified as a line-adjacency iteration. In VGA mode number 18 the performance of the line-adjacency method can be considerably improved by pre-scanning a group of 8 horizontal pixels for a boundary. If no boundary is found, all 8 pixels are set at once. Pixel-by-pixel scanning takes place only if the pre-scan detects a boundary pixel.

An alternative algorithm for a region fill operation consists of scanning the pixels that form the outside border of the figure and storing their x, y coordinates in a data structure. After the border of the figure is defined, the code scans the interior of the figure for holes. Once the exterior boundary and the holes are known, the fill operation can be reduced to displaying line segments that go from one exterior boundary to another, or from an exterior boundary to a hole boundary. This algorithm, sometimes called a border fill method, is relatively efficient and can be used to fill more complex shapes than those in Figure 10-10a.

10.6 Primitive Routines in the VGA3 Module

The library module named VGA3 of the GRAPHSOL library, furnished with the book's software package, contains several VGA mode 18 geometric primitives. The following are generic descriptions of the geometrical primitive routines contained in the VGA3 libraries. The values passed and returned by the individual functions are listed in the order in which they are referenced in the code. The following listing is in the order in which the routines appear in the library source files.

BRESENHAM

Draw a straight line using Bresenham's algorithm

```
Receives:
         1. Byte integer of color of line
         2. Word integer of start point of x coordinate
         3. Word integer of start point of y coordinate
         4. Word integer of end point of x coordinate
         5. Word integer of end point of y coordinate
Returns:
         Nothing
Action:
         Straight line is displayed
```

LINE BY SLOPE

Plot and store the pixel coordinates of a straight line of slope s, located in the fourth quadrant. The slope must be in the range 0 < s > 90 degrees.

```
Receives:
         1. Word integer of start point of x coordinate
         2. Word integer of start point of y coordinate
         3. Word integer of end point of x coordinate
         4. Word integer of slope
```

```
      Returns:
               Nothing
      Action:
               Straight line is calculated and stored
```

CIRCLE

Plot and store the pixel coordinates of a circular arc in the fourth quadrant.

```
      Receives:
               1. Word integer of radius of circle
      Returns:
               Nothing
      Action:
               Circular arc is calculated and stored
```

ELLIPSE

Plot and store the pixel coordinates of an ellipse in the fourth quadrant.

```
      Receives:
               1. Word integer of x semi-axis of ellipse
               2. Word integer of y semi-axis of ellipse
      Returns:
               Nothing
      Action:
               Elliptical arc is calculated and stored
```

PARABOLA

Plot and store the pixel coordinates of a parabola in the fourth quadrant.

```
      Receives:
               1. Word integer of x focus of parabola
               2. Word integer of start x coordinate
      Returns:
               Nothing
      Action:
               Parabolic arc is calculated and stored
```

HYPERBOLA

Plot and store the pixel coordinates of a hyperbola in the fourth quadrant.

```
      Receives:
               1. Word integer of x focus of hyperbola
               2. Word integer of vertex of hyperbola
               3. Word integer of start x coordinate
      Returns:
               Nothing
      Action:
               Hyperbolic arc is calculated and stored
```

QUAD_I

Display a geometrical curve in the first quadrant, while in VGA mode number 18, using its stored coordinates.

```
      Receives:
               1. Byte integer of IRGB color code
               2. Word integer of x coordinate of origin
               3. Word integer of y coordinate of origin
      Returns:
               Nothing
      Action:
               Curve is displayed
```

QUAD_II

Display a geometrical curve in the second quadrant, while in VGA mode number 18, using its stored coordinates.

```
      Receives:
               1. Byte integer of IRGB color code
               2. Word integer of x coordinate of origin
               3. Word integer of y coordinate of origin
      Returns:
               Nothing
      Action:
               Curve is displayed
```

QUAD_III

Display a geometrical curve in the third quadrant, while in VGA mode number 18, using its stored coordinates.

```
      Receives:
               1. Byte integer of IRGB color code
               2. Word integer of x coordinate of origin
               3. Word integer of y coordinate of origin
      Returns:
               Nothing
      Action:
               Curve is displayed
```

QUAD_IV

Display a geometrical curve in the fourth quadrant, while in VGA mode number 18, using its stored coordinates.

```
      Receives:
               1. Byte integer of IRGB color code
               2. Word integer of x coordinate of origin
               3. Word integer of y coordinate of origin
      Returns:
               Nothing
      Action:
               Curve is displayed
```

DO_4_QUADS

Display all four quadrants by calling the procedures QUAD_I, QUAD_II, QUAD_III, and QUAD_IV.

```
      Receives:
               Nothing
      Returns:
               Nothing
      Action:
               Curve is displayed in all four quadrants
```

ROTATE_ON

Activate the rotate operation during display.

```
      Receives:
               1. Word integer of clockwise angle of rotation
```

```
            in the range 0 to 90 degrees
Returns:
         Nothing
Action:
         Rotation angle is stored and rotation is enabled during
         display operations
```

ROTATE_OFF

De-activate the rotate operation during display.

```
Receives:
         Nothing
Returns:
         Nothing
Action:
         Rotation is disabled during display operations
```

CLIP_ON

Activate clipping operation during display.

```
Receives:
        1. Word integer of left corner of clipping window
        2. Word integer of top corner of clipping window
        3. Word integer of right corner of clipping window
        4. Word integer of bottom corner of clipping window
Returns:
         Nothing
Action:
         Clipping values are stored and clipping is enabled during
         display operations
```

CLIP_OFF

De-activate clipping during display.

```
Receives:
         Nothing
Returns:
         Nothing
Action:
         Clipping is disabled during display operations
```

INIT_X87

Initialize 80x87 hardware or emulator and set rounding control to even.

```
Receives:
         Nothing
Returns:
         Nothing
Action:
         If 80x87 hardware is detected an internal switch is set
         so that the coprocessor will be used during geometrical
         calculations. Otherwise the switch will direct execution to
         emulated code. In both cases the control word is set to round
         to even numbers.
```

REGION_FILL

Fill a closed geometrical surface, with no internal holes, composed of unbroken horizontal lines. Uses VGA mode number 18.

```
Receives:
        1. Byte integer of IRGB color code
        2. Word integer of x coordinate of seed point
        3. Word integer of y coordinate of seed point
Returns:
         Nothing
Action:
        Figure is filled
```

Chapter 11

XGA and 8514/A Adapter Interface

Topics:

- XGA and 8514/A Adapter Interface
- The Adapter Interface software
- AI Communications and concepts
- AI programming fundamentals

Describes the XGA and 8514/A video systems and their architecture. Also of programming XGA and 8514/A by means of the IBM Adapter Interface (AI) software package. The chapter includes programming examples in assembly language.

11.1 8514/A and XGA

In 1987 IBM introduced a high-end video graphics system intended for applications that demand high-quality graphics, such as CAD, desktop publishing, graphical user interfaces to operating systems, image editing, and graphics art software. The best graphics mode available in a fully equipped 8514/A system is of 1,024 by 768 pixels in 256 colors. Compared to VGA mode number 18 (640 by 480 pixels in 16 colors) this 8514/A graphics mode offers 2.5 times the number of screen pixels and 16 times as many colors. The major features of the 8514/A standard are the following:

1. 8514/A is furnished as an add-on card for PS/2 Micro Channel microcomputers with VGA systems on the motherboard. The 8514/A board is installed in a slot with a special connector that allows a VGA signal to pass through.
2. Memory architecture follows a planar scheme similar to the one used by the CGA, EGA, and VGA systems. The card is furnished in two versions, one with 512K of on-board VRAM and another one with 1,024K. The maximum resolution of 1,024 by 768 pixels in 256 colors is available only in the board equipped with 1,024K of video RAM.
3. 8514/A is furnished with three character fonts. The character sizes are of 12 by 20, 8 by 14, and 7 by 15 pixels for the 1,024 by 768 resolution mode. The 8-by-14 pixel character size is the only one available in the 640-by-480 pixel mode (see Table 11-11 later in this

chapter). The character fonts are stored as disk files in the diskette supplied with the adapter.

4. The adapter contains ROM code that is used by the BIOS Power-on Self Test (POST) to initialize the hardware, but no BIOS programmer services are included.
5. Programming the 8514/A adapter is by means of an Adapter Interface (AI) software. The software is in the form of a TSR program. The TSR installation routine is an executable program named HDILOAD.EXE.
6. The 8514/A AI contains services to control the adapter hardware, to draw lines, rectangles, and small bitmaps (markers), to fill enclosed figures, to manipulate the color palette, to perform bit block transfers (bitBLTs), to change the current drawing position, line type, width, and display color, to select among 16 logical and 14 arithmetic mix modes, and to display text strings and individual characters.
7. The color palette consists of 262,144 possible colors of which 256 can be displayed simultaneously. The gray scale is of 64 shades.

The internal architecture of the 8514/A consists of three central components: a drawing engine, a display processor, and the on-board video RAM. In addition, the board contains a color look-up table (LUT), a digital-to-analog converter (DAC), and associated registers, as well as a small amount of initialization code in ROM. Figure 11-1 is a diagram of the components in the 8514/A system.

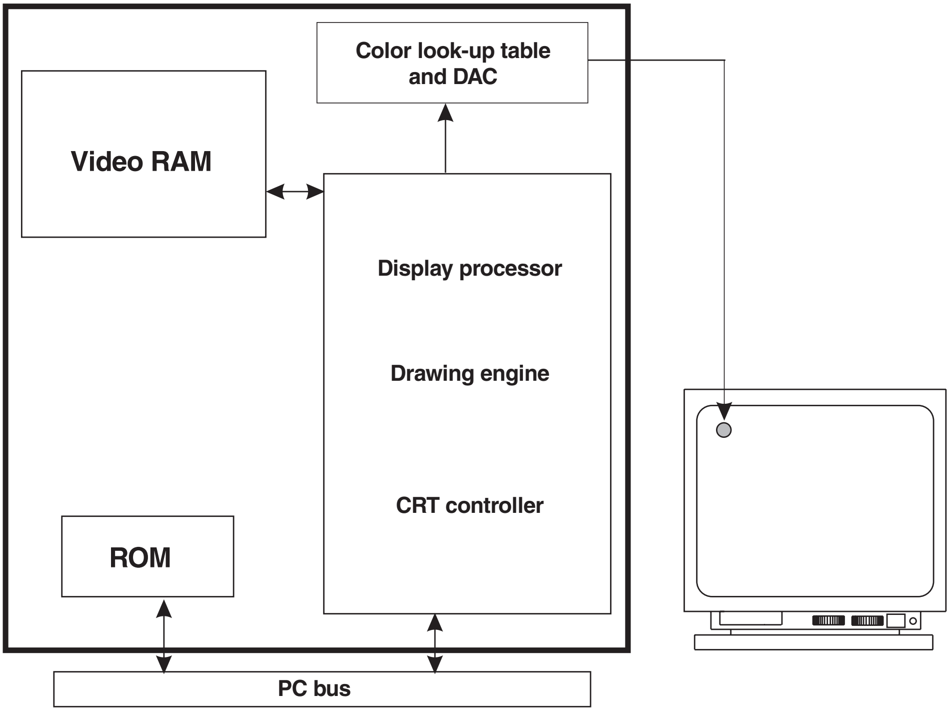


Figure 11-1 *8514/A Component Diagram*

The 8514/A adapter, in spite of the substantial improvements that it brought to PC video graphics, enjoyed only limited success. The following limitations of the 8514/A adapter have been noted:

1. 8514/A requires a Micro Channel bus. This makes the card unusable in many IBM-compatible computers.
2. The AI interface offers limited graphics services, for example, no curve drawing functions are available, nor are there direct services for reading or setting an individual screen pixel.
3. Video memory operations must take place through a single transfer register. The absence of DMA slows down image transfer procedures.
4. 8514/A requires the presence of a VGA system on the motherboard. This duplication of video systems often constitutes an unnecessary expense.
5. Register information regarding the 8514/A was published by IBM only after considerable pressure from software developers. For several years there was no other way for programming the system than using the AI services.
6. 8514/A supports only interlaced displays. This determines that applications that generate single-pixel horizontal lines (such as CAD programs) are afflicted with flicker. Notice that some clone 8514/A cards offer non-interlaced display.
7. IBM documentation for programming 8514/A refers almost exclusively to C language. Programmers working in assembler or in high-level languages other than C were left to their own resources.

In September 1990 IBM disclosed preliminary information on a new graphics standard designated as the Extended Graphics Array, or XGA. Two configurations of the XGA standard have since been implemented: as an adapter card and as part of the motherboard. The XGA adapter is compatible with PS/2 Micro Channel machines equipped with the 80386 or 486 CPU. The XGA system is integrated in the motherboard of the IBM Model 95 XP 486. Figure 11-2, on the following page, is a diagram of the XGA system.

Several features of the XGA system are similar to those of the 8514/A:

1. The maximum resolution is of 1,024 by 768 pixels in 256 colors.
2. The XGA system is compatible with the 8514/A Adapter Interface software.
3. The display driver is interlaced at 1,024 by 768 pixel resolution.
4. The XGA digital-to-analog converter (DAC) and color look-up table (LUT) operate identically to those in the 8414/A. This means that palette operations are compatible in both systems.
5. The adapter version of XGA is furnished with either 512K or 1,204K of on-board video RAM.

However, there are several differences between the two systems, such as:

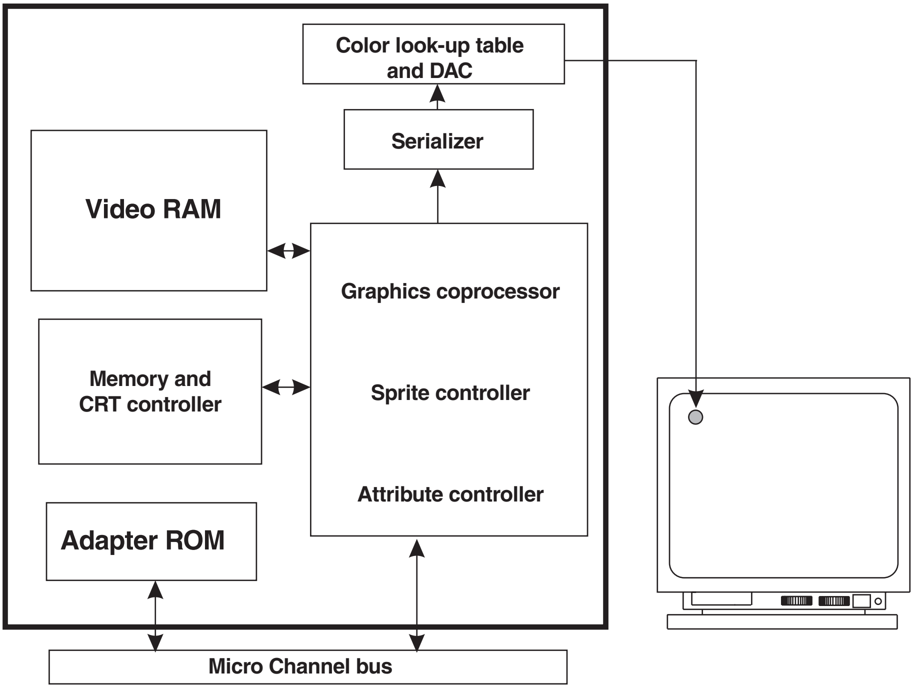


Figure 11-2 *XGA Component Diagram*

1. The XGA is compatible with the VGA standard at the register level. This makes possible the use of XGA in the motherboard while still maintaining VGA compatibility. This is the way in which it is implemented in the IBM Model 95 XP 486 microcomputer.
2. XGA includes two display modes that do not exist in 8514/A: a 132-column text mode, and a direct color graphics mode with 640-by-480 pixel resolution in 64K colors. Notice that this graphics mode is available only in cards with 1,024K video RAM installed.
3. XGA requires a machine equipped with a 80386 or 486 CPU while 8514/A can run in machines with the 80286 chip.
4. XGA implements a three-dimensional, user-definable drawing space, called a bitmap. XGA bitmaps can reside anywhere in the system's memory space. The application can define a bitmap in the program's data space and the XGA uses this area directly for drawing, reading, and writing operations.
5. XGA is equipped with a hardware controlled cursor, called the sprite. It maximum size is 64 by 64 pixels and it can be positioned anywhere on the screen without affecting the image stored in video memory.
6. The XGA Adapter Interface is implemented as a .SYS device driver while the driver for the 8514/A is in the form of a TST program. The module name for the XGA driver is XGAAIDOS.SYS. The XGA AI adds 17 new services to those available in 8514/A.
7. The XGA was designed taking into consideration the problems of managing the video image in a multitasking environment. Therefore it contains facilities for saving and restoring the state of the video hardware at any time.

8. The XGA hardware can act as a bus master and access system memory directly. This bus-mastering capability frees the CPU for other tasks while the XGA processor is manipulating memory.
9. IBM has provided register-level documentation for the XGA system. This will facilitate cloning and development of high-performance software.

Some of the objections raised for the 8514/A still apply to the XGA, for instance, the Micro Channel requirement, the limitations of the AI services, and the interlaced display technology. On the other hand, the XGA offers several major improvements in relation to the 8514/A.

11.2 Adapter Interface Software

The Adapter Interface (AI) is a software package furnished with 8514/A and XGA systems that provides a series of low-level services to the graphics programmer. In the 8514/A the AI software is in the form of a Terminate and Stay Resident (TSR) program while in the XGA the AI is a .SYS driver. The respective module and directory names are shown in Table 11-1.

Table 11-1

Module and Directory Names for the Adapter Interface Software

8514/A		XGA	
FORM	PATHNAME	FORM	PATHNAME
TSR	HDIPCDOS\HDILOAD.EXE	.SYS	XGAPCDOS\XGAAIDOS.SYS

The AI was originally documented by IBM in the IBM Personal System/2 Display Adapter 8514/A Technical Reference (document number S68X-2248-0) published in April 1987. IBM has also published a document named the IBM Personal System/2 Display Adapter 8514/A Adapter Interface Programmer's Guide (document number 00F8952). This product includes a diskette containing a demo program, a collection of font files, and several programmer utilities. The corresponding IBM document for XGA AI is called the IBM Personal System/2 XGA Adapter Interface Technical Reference (document number S-15F-2154-0). All of the above documents are available from IBM Technical Directory (1-800-426-7282). Other IBM documents regarding XGA hardware are mentioned in Chapter 7.

11.2.1 Software Installation

The AI driver software must be installed in the machine before its services become available to the system. In the case of the 8514/A the AI driver is in the form of a TSR program, while in the XGA it is furnished as a .SYS file. Installation instructions for the AI software are part of the adapter package. In the case of the XGA AI several versions of the AI are furnished by IBM: one for MS-DOS, another one for Windows, and a third one for the OS/2 operating system.

In the MS-DOS environment the installation routine, for either the 8514/A or XGA, creates a dedicated directory (see Table 11-1), selects the appropriate driver software, and optionally includes an automatic setup line. In the 8514/A the automatic setup line is added to the user's AUTOEXEC.BAT file and in the XGA to the

CONFIG.SYS file. This insures that the driver software is made resident every time the system is booted.

The 8514/A installation process makes the AI functions available, but does not automatically switch video functions to the 8514/A display system. Notice that, since 8514/A does not include VGA, a typical 8514/A configuration is a machine with two display adapters, one attached to the motherboard VGA and the other one to the 8514/A card. With the XGA, which includes VGA functions, it is possible to configure a machine with a single display attached either to a motherboard XGA or to an adapter version of the XGA. Alternatively, the adapter version of the XGA can be configured with two or more displays. For example, a machine with VGA on the motherboard can be furnished with an XGA card and monitor. In this case, the XGA resembles the typical 8514/A arrangement described above.

11.2.2 XGA Multi-Display Systems

If and when XGA becomes the video standard for IBM microcomputers a typical machine will probably be equipped with a single display attached to XGA hardware on the motherboard. This is already the case in the IBM Model 95 XP 486 microcomputer. However, most present day implementations of XGA consist of PS/2 machines, originally equipped with VGA on the motherboard, and which have been supplemented with an XGA adapter card. Since XGA includes VGA, this upgrade version can be configured with a single monitor attached to the XGA video output connector. An alternative setup uses two monitors: one attached to the VGA connector on the motherboard and one to the XGA card.

A multi-display XGA system setup offers some interesting possibilities, for example, in graphics applications it s possible for the XGA to display the graphics image while the VGA on the motherboard is used in interactive debugging operations. XGA systems can have up to six adapters operating simultaneously, although in most machines the number of possible XGA adapters is limited by the number of available slots. This is not the case with 8514/A, which cannot have more than two displays per system.

The possibility of multi-display XGA systems creates new potentials in applications and systems programming. For example, by manipulating the XGA address decoding mechanism an application can display different data on multiple XGA screens. In this manner it is possible to conceive an XGA multitasking program with several display systems. One feasible setup is to use the first monitor to show output of a word processing program, the second monitor a database, and the third one a spreadsheet. The user could switch rapidly between applications while the data displayed remains on each screen. Another sample use of a multi-display system is an airport software package that would show arrival schedules on one screen, and departures on another one, while a third monitor is attached to the reservations desk. Finally, in a graphics applications environment, we can envision a desktop publishing system in which the central monitor would display the typesetting software, the monitor on one side would be attached to a graphics illustration program, and the one on the other side to a text editor.

11.2.3 Operating Modes

Both 8514/A and XGA systems can operate in one of two modes: the VGA mode or the advanced functions mode. The operating mode is selected by the software. In the VGA mode the graphics system is a full-featured VGA (see Table 2-2). The advanced function mode refers to the Adapter Interface software. Table 11-2 shows the characteristics of the display modes available in the AI.

Table 11-2

XGA and 8514/A Advanced Function Modes

	LOW RESOLUTION MODE	HIGH RESOLUTION MODE
RAM installed	512K	1,024K
Interlaced	NO	YES
pixel columns	640	1,024
pixel rows	480	768
number of colors	16	256
Palette	256K	256K

11.2.4 The XGA and 8514/A Palette

8514/A and XGA video memory is organized in bit planes. Each bit plane encodes the color for a rectangular array of 1,024 by 1,024 pixels. In practice, since the highest available resolution is of 1,024 by 768 pixels, there are 256 unused bits in each plane. This unassigned area is used by AI software as a scratchpad during area fills and in marker manipulations, as well as for storing bitmaps for the character sets. When the graphics system is in the low resolution mode video memory consists of eight 1,024 by 512 bit planes. However, the 8 bit planes are divided into two separate groups of 4 bit planes each. These 2 bit planes can be simultaneously addressed. In low resolution mode the color range is limited to 16 simultaneous colors. In the high resolution mode (see Table 11-2) video memory consists of 8 bit planes of 1,024 by 1,024 pixels. In this mode the number of simultaneous colors is 256. Figure 11-3 shows the bit-plane mapping in XGA and 8514/A high resolution modes.

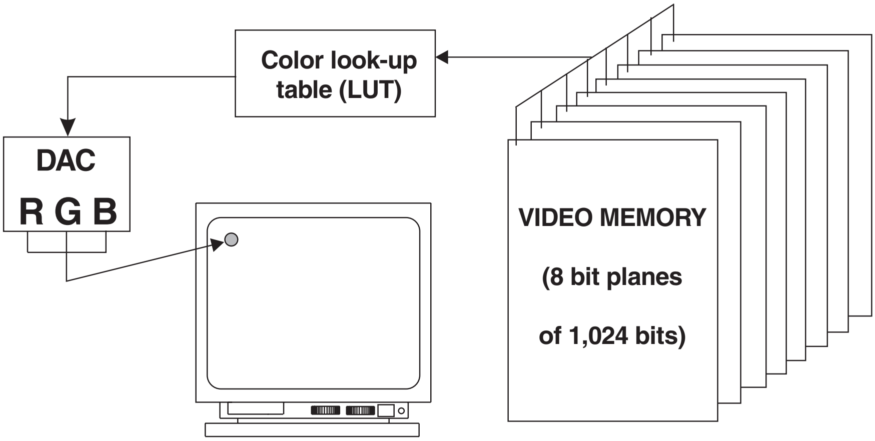


Figure 11-3 *Bit Planes in XGA and 8514/A High-Resolution Modes*

Color selection is performed by means of a color look-up table (LUT) associated with the DAC. The selection mechanism is similar to the one used in VGA mode number 19, described previously. This means that the 8-bit color code stored in XGA and 8514/A video memory serves as an index into the color look-up table (see Figure 11-3). For example, the color value 12 in video memory selects LUT register number 12, which in the default setting stores the encoding for bright red. The default setting of the LUT registers can be seen in Table 11-3.

Table 11-3

Default Setting of LUT Registers in XGA and 8514/A

REGISTER NUMBER	6-BIT COLOR (HEX VALUE) R	G	B	COLOR
0	00	00	00	Black
1	00	00	2A	Dark blue
2	00	2A	00	Dark green
3	00	2A	2A	Dark cyan
4	2A	00	00	Dark red
5	2A	00	2A	Dark magenta
6	2A	15	00	Brown
7	2A	2A	2A	Gray
8	15	15	15	Dark gray
9	15	15	3F	Light blue
10	15	3F	15	Light green
11	15	3F	3F	Light cyan
12	3F	15	15	Light red
13	3F	15	3F	Light magenta
14	3F	3F	15	Yellow
15	3F	3F	3F	Bright white
16 to 31	00	00	2A	Dark blue
32 to 47	00	2A	00	Dark green
48 to 63	00	2A	2A	Dark cyan
64 to 79	2A	00	00	Dark red
80 to 95	2A	00	2A	Dark magenta
96 to 111	2A	15	00	Brown
112 to 127	2A	2A	2A	Gray
128 to 143	15	15	15	Dark gray
144 to 159	15	15	3F	Light blue
160 to 175	15	3F	15	Light green
176 to 191	15	3F	3F	Light cyan
192 to 207	3F	15	15	Light red
208 to 223	3F	15	3F	Light magenta
224 to 239	3F	3F	15	Yellow
240 to 255	3F	3F	3F	Bright white

XGALUT program, provided in the book's software package, displays the color in the XGA palette. The colors displayed by the program match those in Table 11-3. Notice that the default setting for the XGA and 8514/A LUT registers represent only 16 color values, which correspond to registers 0 to 15 in Table 11-3. The default colors encoded in LUT registers 16 to 255 are but a repetition, in groups of 15 registers, of the encodings in the first 16 LUT registers. Consequently, software products that intend to use the full color range of XGA and 8514/A systems must reset the LUT registers.

In the documentation for Display Adapter 8514/A IBM recommends an 8-bit color coding scheme in which 4 bits are assigned to the green color and 2 bits to the red and blue colors respectively. This scheme is related to the physiology of the human eye, which is more sensitive to the green area of the spectrum than to the red or blue areas. One possible mapping, which conforms with the XGA direct color mode, is to devote bits 0 and 1 to the blue range, bits 2 to 5 to the green range, and bits 6 and 7 to the red range. This bitmapping is shown in Figure 11-4.

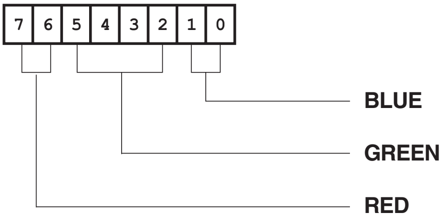


Figure 11-4 *XGA/8514/A Bit-to-Color Mapping*

An alternative mapping scheme can be based on assigning 2 bits to the intensity, red, green, and blue elements, respectively. A similar double-bit IRGB encoding was developed in Section 8.3.1 and in Table 8-3 for VGA 256-color mode number 19. The XGA and 8514/A color registers (color look-up table) consist of 18 bits, 6 bits for each color (red, green, and blue). The bitmap of the LUT registers is shown in Figure 11-5.

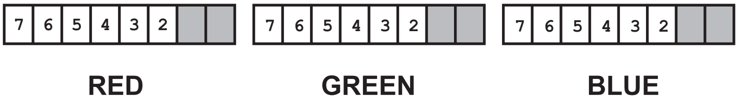


Figure 11-5 *Bitmap of XGA and 8514/A Color Registers*

Notice that the XGA bitmap for the LUT register uses the six high-order bits while the VGA bitmap uses the 6 low-order bits (see Figure 3.7). As a result of this difference the values for a VGA palette must be shifted left 2 bits (multiplied by 4) in order to convert them to the XGA bit range.

11.2.5 Alphanumeric Support

The XGA and 8514/A Adapter Interface provides services for the display of text strings and of individual characters. The string-oriented services are designated as text functions in the AI documentation while the character-oriented services are called alphanumeric functions. The AI text and character display services are necessary since BIOS and DOS functions for displaying text do not operate on the XGA and the 8514/A video systems.

Both text and alphanumeric functions in the AI require the use of character fonts, several of which are part of the XGA and 8514/A software package. These character fonts are stored in disk files located in the adapter's support diskette. During installation the font files are moved to a special directory in the user's hard disk drive. The 8514/A adapter is furnished with three standard fonts while there are four standard fonts in the XGA diskette. In addition, the XGA diskette contains four supplementary fonts that have been optimized for XGA hardware. Finally, the diskette furnished with the IBM Personal System/2 Display Adapter 8514/A Adapter Interface Programmer's Guide (see Section 6.1) contains 22 additional fonts, which are also compatible with the XGA system.

Fonts for the AI software can be in three different formats: short stroke vector, single-plane bitmaps, and multiplane bitmaps. The fonts furnished with 8514/A are of short stroke vector type. The supplementary fonts furnished with the XGA diskette are in single-plane bitmap format. The fonts furnished with the 8514/A Programmer's Guide diskette are also in the single-plane bitmap format. Multiplane bitmapped fonts, although documented in the Display Adapter 8514/A Technical Reference, have not been furnished by IBM for either 8514/A or XGA systems. In the XGA diskette it is possible to identify the fonts in short stroke vector format by the extension .SSV, while the single-plane bitmap fonts have the extension .IMG. However, the 8514/A short stroke vector fonts have the extension .FNT. An additional complication is that the XGA installation routine changes the extension .SSV for .FNT. For these reasons it is not always possible to identify the font format by means of the extension to the filename.

Font File Structure

All font files compatible with the AI software must conform to a specific format and structure. Each of the standard fonts supplied in the Adapter Interface diskette contains five different character sets, named code pages in the IBM documentation. The code page codes and corresponding alphabets can be seen in Table 11-4.

Table 11-4

IBM Code Pages

CODE	DESIGNATION
437	US/English alphabet
850	Multilingual alphabet
860	Portuguese alphabet
863	Canadian/French alphabet
865	Nordic alphabet

At the start of each font file is a font file header that contains general information about the number of code pages, the default code pages, and the offset of each character set within the disk file. The font file header can be seen in Table 11-5.

Each code page (character set) in a font file is preceded by a header block that contains the necessary data for displaying the encoded characters. The character set header is called the character set definition block in IBM documentation. The offset of the character set headers can be obtained from the corresponding entry in the font file header (see Table 11-5). In this manner, a program can locate the header block for the first code page (US/English alphabet) by adding the word value at offset 10 of the font file header (see Table 11-5) to the offset of the start of the disk file.

Table 11-5

Adapter Interface Font File Header

OFFSET	UNIT	CONTENTS
0	word	Number of code pages in the font file
2	word	Number of the default code page (range 0 to 4)
4	word	Number of alternate default code page (range 0 to 4)
6	doubleword	4-character id string for the first code page ('437'0)
10	word	Offset within the disk file of the first code page
12	doubleword	4-character id string for the second code page ('850'0)
16	word	Offset within the disk file of the second code page
18	doubleword	4-character id string for the third code page ('860'0)
22	word	Offset within the disk file of the third code page
24	doubleword	4-character id string for the fourth code page ('863'0)
28	word	Offset within the disk file of the fourth code page
30	doubleword	4-character id string for the fifth code page ('865'0)
34	word	Offset within the disk file of the fifth code page

Table 11-6, on the following page, shows the data encoded in the character set header. Notice that the byte at offset 1 of the character set header encodes the image format as bitmapped (value 0) or as short stroke vector type (value 1). If the image is in bitmapped format, then bit 14 of the word at offset 12 determines if the image is single or multiplane. The byte at offset 7 of the character set header measures the number of horizontal pixels in the character cell while the byte at offset 8 measures its vertical dimension. The cell size, which is stored at the word at offset 10, represents the number of bytes used in storing each character encoded in bitmap format. This value is obtained by multiplying the pixel width (offset 7) by the pixel height (offset 8) and dividing the product by 8.

The index table, which can be located by means of the address stored at offset 14 of the character set header, contains the offset of the character definitions for each individual character. For single-plane fonts the start location of the character definition table can be found from the address stored at offset 24. Therefore, a program can locate the bitmap for a particular character by adding its offset in the table, obtained from the index table, to the offset of the start of the character definition table. The code for first and last characters, at offsets 22 and 23 of the character set header, serves to delimit the character range of the font. For example, if a font does not start with character code 1, the value at offset 22 in the character set header must be used to scale the character codes into the index table.

Multiplane fonts consist of three monochrome images, whose bitmaps can be located by means of the addresses stored at offsets 24, 30, and 36 of the character set header (see Table 11-6). To the present date, multiplane image fonts have not been furnished by IBM. Single-plane image fonts are encoded in a single bitmap, which is located at the address stored at offset 24 of the character set header (see Table 11-6). The character's image is encoded in a bit-to-pixel scheme. The character's foreground and background colors are determined by means of foreground color and background color settings described later in this chapter.

Table 11-6

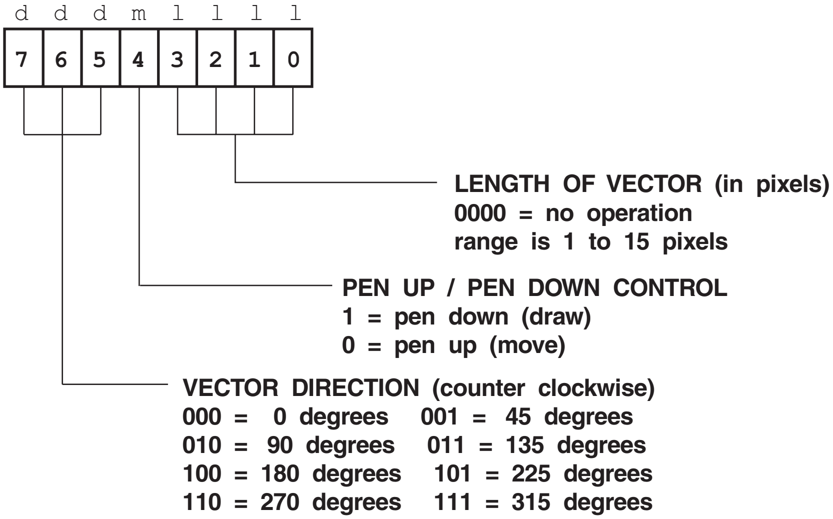
Adapter Interface Character Set Header

OFFSET	UNIT	CONTENTS
0	byte	Reserved
1	byte	Image formated as follows: 0 = single or multiplane image 3 = short stroke vector image
26		Reserved
7	byte	Pixel width of character cell
8	byte	Pixel height of character cell
9	byte	Reserved
10-11	word	Cell size (in bytes per character)
12-13	word	Character image format: Bit 14: 0 = single plane image 1 = multiplane image Bit 13: 0 = not proportionally spaced 1 = proportionally spaced All other bits are reserved (0)
14-17	doubleword	Offset:segment of index table
18-21	doubleword	Offset:segment of porportional spacing table
22	byte	Code for first character
23	byte	Code for last character
24-27	doubleword	Offset:segment of first characterdefinition table (all font types)
28-29		Reserved
30-33	doubleword	Offset:segment of second character definition table (multiplane fonts)
34-35		Reserved
36-39	doubleword	Offset:segment of third character definition table (multiplane fonts)

The location of the character definition table for short stroke vector fonts is the same as for single stroke, bitmapped fonts. However, short stroke vector characters are encoded in the form of drawing orders, each of which is represented in a 1-byte command. The character drawings are made up of a series of straight lines (vectors) that can be no longer than 15 pixels. Each vector must be drawn at an angle that is a multiple of 45 degrees. Therefore the lines must be either vertical, horizontal, or diagonal. Figure 11-6 shows the bitmap of the short stroke vector commands.

The vector direction field, marked with the letters d in Figure 11-6, determines the direction and angle of each vector. The reference point is at the origin of the Cartesian plane and the angle is measured in a counterclockwise direction. In this manner the value 010 corresponds with a vector drawn in the vertical direction, downward from the start point. The field marked with the letter m in Figure 11-6 determines if the vector is a draw or move operation. We have used the plotter terminology of pen up and pen down to illustrate this function. If a vector is defined as a pen up vector the current position is changed but no drawing takes place. If the m bit is set (pen down), then the vector command draws a line on the video screen. The length of the vector is determined by the 4 bits in the field marked with the letters l in Figure 11-6. A 0000 value in this field is interpreted as no operation. The maximum length of a vector corresponds with the field value of 1111, which is equivalent to 15 pixels. The current drawing position is moved one pixel further than the value encoded in the l field.

Figure 11-6 *Bitmap of the Short Stroke Vector Command*


d d d m l l l l
7 6 5 4 3 2 1 0
LENGTH OF VECTOR (in pixels)
0000 = no operation
range is 1 to 15 pixels
PEN UP / PEN DOWN CONTROL
1 = pen down (draw)
0 = pen up (move)
VECTOR DIRECTION (counter clockwise)
000 = 0 degrees 001 = 45 degrees
010 = 90 degrees 011 = 135 degrees
100 = 180 degrees 101 = 225 degrees
110 = 270 degrees 111 = 315 degrees

11.3 Communicating with the AI

The Adapter Interface software was conceived as a layer of software services for initializing, configuring, and programming the 8514/A graphics system. XGA is furnished with a compatible set of services, which are a superset of those furnished for 8514/A. In both cases, 8514/A and XGA, the programming interface documentation assumes that programming is in C language. Access methods from other languages have not been described to this date. One difference between the AI software, as furnished for 8514/A and XGA, is that the former is a Terminate and Stay Resident (TSR) program while the latter is an MS-DOS device driver of the .SYS file type.

The AI installation selects one of two versions of the software according to the amount of memory in the graphics system. Once installed, the address of the AI handler is stored at interrupt vector 7FH. The AI services are accessed by means of an INT 7FH instruction or by a far call to the address of the service routine.

11.3.1 Interfacing with the AI

Before an application can start using the AI services it must first certify that the software is correctly installed and obtain the address of the service routine. Since interrupt 7FH has been documented as a reserved vector in IBM literature, the application can assume, with relative certainty, that the value stored at this vector is zero if no AI has been installed. However, this assumption risks that a non-conforming program has improperly used the vector for its own purposes. In which case the vector could store a non-zero value, while no AI is present.

The documented access mechanism for the AI services is by means of a far call. It appears that the AI is preceded by a jump table to each of its service routines and that each address in the jump table is a 4-byte far pointer. Therefore the calling program must multiply the AI service request by four to obtain the offset in the jump table. This jump table offset is placed in the SI register, the offset element of the address of the AI service routine is in BX, and its segment in ES. Once these registers are set up, the far call to a particular AI service can be performed by means of the instruction

```
        CALL    DWORD PTR ES:[BX+SI]
```

Notice that the offset element of the address is determined by the sum of the pointer register (BX) and the offset of the service routine in the jump table (SI).

C Language Support

Two support files and a demonstration program for the AI are included in both the 8514/A and the XGA diskettes furnished with the adapters. The C language header files are named AFIDATA.H and IBMAFI.H. In addition, the assembly language source file named CALLAFI.ASM contains three public procedures for initializing and calling the AI. The object file CALLAFI.OBJ must be linked with the application's C language modules in order to access the AI. The header files and the object module CALLAFI.OBJ provide a convenient interface with the AI for C language applications.

AI Entry Points

We saw that an application accesses the AI services by means of a jump table of service numbers. The C language support software provided with XGA and 8514/A contains an ordered list of the code names of the services and their associated entry points. In this manner an application coded in C language need only reference the service name and the support software will calculate the routine's entry point from the furnished table. Table 11-7 lists the service routine code names and entry point numbers for the AI services available in both 8514/A and XGA systems.

Table 11-7

8514/A and XGA Adapter Interface Services

NAME	ENTRY POINT NUMBER	DESCRIPTION
HLINE	0	Draw line
HCLINE	1	Draw line at current point
HRLINE	2	Draw line from start point
HCRLINE	3	Draw line from start point
HSCP	4	Set current point
HBAR	5	Begin area for fill operation
HEAR	6	End area for fill operation
HSCOL	7	Set current color
HSOPEN	8	Open adapter for AI operations
HSMX	9	Set mix
HSBCOL	10	Set background color

(continues)

Table 11-7

8514/A and XGA Adapter Interface Services ***(continued)***

NAME	ENTRY POINT NUMBER	DESCRIPTION
HSLT	11	Set line type
HSLW	12	Set line width
HEGS	13	Erase graphics screen
HSGQ	14	Set graphics quality
HSCMP	15	Set color compare register
HINT	16	Synchronize with vertical retrace
HSPATTO	17	Set pattern reference
HSPATT	18	Set pattern shape
HLDPAL	19	Load palette
HSHS	20	Set scissor
HBBW	21	Write bit block image data
HCBBW	22	Write bit block at current point
HBBR	23	Read bit block
HBBCHN	24	Chain bit block data
HBBC	25	Copy bit block
HSCOORD	26	Set coordinate type
HQCOORD	27	Query coordinate type
HSMODE	28	Set adapter mode
HQMODE	29	Query adapter mode
HQMODES	30	Query adapter modes
HQDPS	31	Query drawing process state
HRECT	32	Fill rectangle
HSBP	33	Set bit plane controls
HCLOSE	34	Close adapter
HESC	35	Escape (terminate processing)
HXLATE	36	Assign multiplane color tables
HSCS	37	Select character set
HCHST	38	Display character string
HCCHSET	39	Display string at current point
ABLOCKMFI	40	Display character block (MFI mode)
ABLOCKCGA	41	Display character block (CGA mode)
AERASE	42	Erase character rectangle
ASCROLL	43	Scroll character rectangle
ACURSOR	44	Set current cursor position
ASCUR	45	Set cursor shape
ASFONT	46	Select character set
AXLATE	47	Assign color index
HINIT	48	Initialize adapter state
HSYNC	49	Synchronize adapter with task
HMRK	50	Display marker
HCMRK	51	Display marker at current point
HSMARK	52	Set marker shape
HSLPC	53	Save linepattern count
HRLPC	54	Restore saved linepattern count
HQCP	55	Query current point
HQDFPAL	56	Query default palette
HSPAL	57	Save palette
HRPAL	58	Restore pallete
HSAFP	59	Set area fill plane
ASCELL	60	Set cell size

The XGA adapter contains 18 additional AI services that are not available in 8514/A. These XGA proprietary services are listed in Table 11-8.

Table 11-8

XGA Adapter Interface Services

NAME	ENTRY POINT NUMBER	DESCRIPTION
ASGO	61	Set alpha grid origin
HDLINE	62	Disjoint line at point
----------	63	
HPEL	64	Write pixel string
HRPEL	65	Read pixel string
HPSTEP	66	Plot and step
HCPSTEP	67	Plot and step at current position
HRSTEP	68	Read and step
HSBMAP	69	Set bitmap attributes
HQBMAP	70	Query bitmap attributes
HBMC	71	Bitmap copy
HSDW	72	Set display window
HSPRITE	73	Sprite at given position
HSSPRITE	74	Set sprite shape
HRWVEC	75	Read/write vector
-----------	76	
-----------	77	
HSFPAL	78	Save full palette
HRFPAL	79	Restore full palette
HQDEVICE	80	Query device specific (no action)

Obtaining the AI Address

The following procedure can be used to test the AI initialization and, if the service software is installed, to acquire the address of the AI service routines.

```
AI_VECTOR       PROC    FAR
; Procedure to obtain the address of the XGA and 8514/A Adapter
; Interface. This procedure must be called before calls are made
; to the Adapter Interface services
;
; On entry:
;          nothing
; On exit:
;          carry set if no AI installed
;          carry clear if AI present
;             CX => segment of AI link table
;             DX => offset of AI link table
;
;**********************|
;    get vector 7FH    |
;**********************|
; Use MS DOS service number 53, interrupt 21H, to obtain the
; vector for the XGA and 8514-A AI interrupt (7FH)
        MOV     AH,53           ; MS DOS service number
        MOV     AL,7FH          ; AI interrupt
        INT     21H             ; MS DOS interrupt
; ES => segment of interrupt handler
; BX => offset of handler
;**********************|
;    test for no AI    |
;**********************|
; The code assumes that the vector at INT 7FH will be 0000:0000
; if the AI is not initialized
        MOV     AX,ES           ; Segment to AX
        OR      AX,BX           ; OR segment and offset
        JNZ     OK_AI           ; Go if address not 0000:0000
;**********************|
;    ERROR - no AI      |
;**********************|
NO_AI:
        STC                     ; Error return
        RET
;**********************|
;  get AI address       |
;**********************|
; Service number 0105H, interrupt 7FH, returns the address of the
; XGA/8514-A entry point
OK_AI:
        MOV     AX,0105H        ; Service request number
        INT     7FH             ; in XGA AI interrupt
        JNC     OK_AI           ; Go if no error code returned
        JMP     NO_AI           ; Take error exit
; At this point CX:DX holds the address of the XGA and 8514/A
; Adapter Interface handler (in segment:offset form)
        CLC                     ; No error
        RET
AI_VECTOR       ENDP
```

Typically, the application calling the AI_VECTOR procedure will store the address of the service routine in its own data space. For example, a doubleword storage can be reserved for the logical address of the service routine, in this manner:

```
AI_ADD          DD      0       ; Doubleword storage for address
                                ; of Adapter Interface services
```

After a call to the AI_VECTOR procedure the code can proceed as follows:

```
;**********************|
;    get AI address     |
;**********************|
; The procedure AI_VECTOR obtains the segment:offset address of
; the AI handler
        CALL    AI_VECTOR       ; Local procedure
        JNC     OK_VECTOR       ; Go if no carry
;
; If execution reaches this point there is no valid AI installed
; and an error exit should take place
        .
        .
        .
OK_VECTOR:
; Store segment and offset of AI handler
        MOV     WORD PTR AI_ADD,DX      ; Store offset of address
        MOV     WORD PTR AI_ADD+2,CX    ; and segment
; AI entry point is now stored in a DS variable
```

Using the AI Call Mechanism

Once the application has stored the address of the AI service routine in a data variable, it can access any of its services. The access mechanism requires the entry point number (see Table 11-7 and 6.8) for the desired service as well as a pointer to a parameter block containing the data received and passed by the service routine. Notice that a few AI services do not require or return user data and, in these cases, the parameter block is a dummy value. The following procedure, named AI_SERVICE, performs the arithmetic operations required to obtain the offset of the desired routine in the AI jump table, sets up the registers for the far call to the service routine, and performs some housekeeping operations.

```
AI_SERVICE      PROC    NEAR
; Procedure to access the services in the XGA and 8514/A Adapter
; Interface software
;
; On entry:
;         AX = service number
;         DS:BX = address of parameter block
;
        PUSH    BP               ; Save base pointer
        MOV     BP,SP            ; Set BP to stack
; Push address of caller's parameter block
        PUSH    DS
        PUSH    BX               ; the offset
; Multiply by 4 to form offset as required by AI
        SHL     AX,1             ; AX times 2
        SHL     AX,1             ; again
        MOV     SI,AX            ; Offset to SI
        LES     BX,AI_ADD        ; Entry block address (ES:BX)
        CALL    DWORD PTR ES:[BX+SI]    ; Call AI service
        POP     BP               ; Restore caller's BP
        RET
AI_SERVICE      ENDP
```

The parameter block passed by the caller to the AI service is a data structure whose size and contents vary in each service. One common element in all parameter blocks is that the first byte serves to determine the size of the block. In this manner the word at offset 0 of the parameter block indicates the byte size of the remainder of the block. Table 11-9 shows the structure of the AI parameter block.

Table 11-9

Structure of the Adapter Interface Parameter Block

OFFSET	DATA SIZE	CONTENTS
0	word	Byte length of parameter block
2 . .	byte, word, doubleword, or string	First data item
length 2		Last data item

AI Initialization Operations

Before the general AI services can be used by an application the adapter must be initialized by presetting it to a known state. Two AI services, named HOPEN and HINIT, are provided for this purpose. The HOPEN service (entry point number 8 in Table 11-7) presets the adapter's control flags and selects an extended function mode. If the adapter is successfully opened, the AI call clears a field in the parameter block. A non-zero value in this field indicates that a hardware mismatch is detected. The following code fragment shows the data segment setup of the parameter block of the HOPEN service as well as a call to this AI service.

```
DATA    SEGMENT
        .
        .
        .
HOPEN_DATA      DW      3       ; Length of data block
INIT_FLAGS      DB      0       ; 7 6 5 4 3 2 1 0 <= flags
                                ; | | ___________
                                ; | |       |_______ Reserved
                                ; | |_ Do not load default
                                ;      palette
                                ; |___ Do not clear bit planes
AF_MODE         DB      0       ; Advanced function mode
                                ; No.   Pixels     Text
                                ; 00    1024x768   85x38
                                ; 01    640x480    80x34
                                ; 02    1024x768   128x54
                                ; 03    1024x768   146x51
RET_STATUS      DB      0       ; Status returned by AI call
                                ; 0 if initialization successful
                                ; Not 0 if initialization failed
        .
        .
        .
DATA    ENDS

CODE    SEGMENT
        .
        .
;**********************|
;    initialize AI     |
;**********************|
; Call HOPEN service (enable adapter)
        MOV     INIT_FLAGS,0    ; Set initialization flags
                                ; to clear memory and load
; default palette
        MOV     AF_MODE,0       ; Set 1024x768 mode number 0
        MOV     AX,8            ; Code number for this service
        LEA     BX,HOPEN_DATA   ; Pointer to parameter block
        CALL    AI_SERVICE      ; Procedure to perform AI call
; The RET_STATUS field is filled by the service call
; This field is not zero if an error was detected
        CMP     RET_STATUS,0    ; Not zero if open error
        JE      OK_OPEN         ; Go if no error
; At this point an error was detected during HOPEN function
        .
        .
        .
; At this point adapter was successfully opened
OK_OPEN:
        .
        .
        .
CODE    ENDS
```

Once the adapter has been successfully opened the program must inform the AI of the location (in the application's memory space) of a special task state buffer. The main purpose of the task state buffer is to assist multitasking by providing a record of the adapter's state for each concurrent task. When a task is restored to the foreground, the task state buffer provides to the AI software all the necessary information for restoring the adapter to its previous state. Although DOS programs have absolute control of the machine's hardware, they must also allocate a task state buffer before beginning AI operations. Table 11-10 lists the data items stored in the task state buffer as well as their initial settings.

Table 11-10

Task State Buffer Data after Initialization

ITEM	VALUE
Current point	Coordinates 0,0
Foreground color	White (all bits are 1)
Background color	Black (all bits are 0)
Foreground mix	Destination = source (overpaint mode)
Background mix	Leave alone
Comparison color	Not initialized
Comparison logic	False
Line type	Solid
User line	Not initialized
Line width	1 pixel
Line pattern	Position not initialized
Saved line pattern	Position not initialized
Area pattern	Solid
Pattern origin	Coordinates 0,0
Text control	Block pointer not initialized
Marker shape	Not intialized
Scissors	Clipping to full screen
Graphics quality	High precision
Plane mask	All planes enabled
Color index table	8 entries set linearly (0 to 7)
Alphanumeric cursor	Top left of screen (0,0)
Cursor definition	Invisible
Translate table	16 values for foreground and background
Character set	Not selected

In order to allocate space for the task state buffer an application must know its size, but the length of the task state buffer is not hard-coded in the adapter's software. However, an application can use the HQDPS function (listed in Table 11-7 and described later in the chapter) in order to determine the memory space required for this data structure. Once the size of the task state buffer is known, the code can dynamically allocate sufficient memory for it. An alternative, although not as elegant, method is to assume that the task state buffer for DOS is 360 bytes and allocate this amount of space. In fact, the task state buffer for XGA systems is 341 bytes, so assigning 360 bytes leaves a 19-byte safety margin.

Space for the task state buffer is allocated and its values initialized by means of the HINIT adapter function. The call requires the segment address of the task state buffer, while it assumes that the buffer is at offset 0000 in this segment. This characteristic of the HINIT service suggests that the task state buffer be placed in a separate segment. This assignation has the added advantage of not using the application's data space for this purpose. In DOS the assignment of buffer space and the HINIT call can be performed as in the following code fragment

```
;****************************************************************
;                  segment for task state data
;****************************************************************
TASK_STATE    SEGMENT
```

```
;*********************|
;  AI state buffer     |
;*********************|
STATE_BUF       DB      360 DUP (00H)
;
TASK_STATE    ENDS

;****************************************************************
;                          data segment
;****************************************************************
DATA    SEGMENT
        .
        .
        .
;
HINIT_DATA      DW      2       ; Length of data block
BUF_SEG         DW      0       ; Segment of task state buffer
        .
        .
        .
DATA    ENDS

;****************************************************************
;                          code segment
;****************************************************************
CODE    SEGMENT
        .
        .
        .

; Call HINIT (Initialize adapter state)
        MOV     AX,TASK_STATE   ; Segment for task state buffer
        MOV     BUF_SEG,AX      ; Store segment in parameter
                                ; block
        MOV     AX,48           ; Code number for this service
        LEA     BX,HINIT_DATA   ; Pointer to data block
        CALL    AI_SERVICE      ; Procedure to perform AI call
; No information is returned by HINIT. Software must assume that
; task state buffer was successfully allocated and initialized
        .
        .
        .
```

The program named AI_DEMO.ASM, furnished in the book's software package, is a demonstration of some elementary AI functions. The code performs AI initialization and setup following a method similar to the one described in the present section. The source file named AI_INIT.ASM is an initialization template that performs the conventional AI operations usually required to start programming XGA or 8514/A systems. The programmer can use AI_INIT.ASM as a coding template for programs that use AI operations.

11.3.2 AI Data Conventions

Many Adapter Interface functions operate on data passed by the caller while some functions return information. In the previous section we discussed (see Table 11-9) the structure of the parameter block whose address is passed to the AI by the calling program. The calling program uses this parameter block to transfer data to and from the AI. However, notice that not all AI functions operate on data items. Some functions (such as HEGS and HCLOSE) require no parameters and return no data to the calling program.

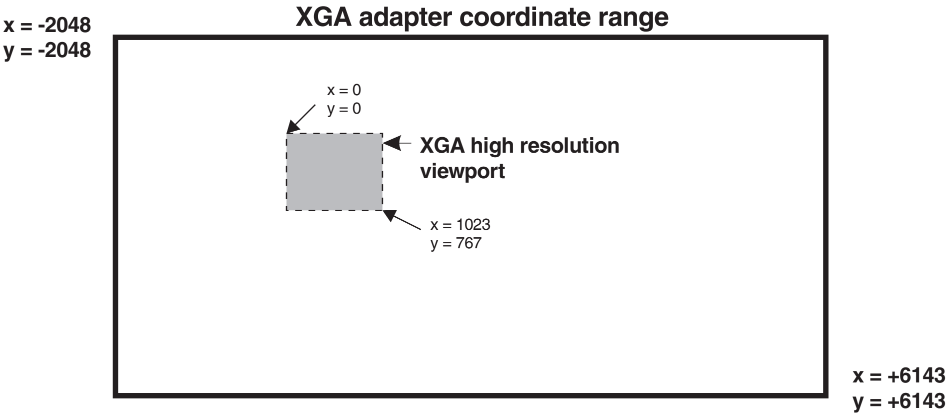
The data items operated on by the AI can be classified into three general groups: numeric data, screen data, and address data.

8514/A numeric data is defined in three integer formats: byte, word, and doubleword. The IBM XGA documentation adds quadword to this list. Byte ordering of numeric data is according to the Intel convention; that is, the least significant byte is located at the lowest numbered memory address. Usually, the programmer need not be concerned with this matter since the assembler or compiler will handle multi-byte ordering automatically. Bit numbering is also in the conventional format, that is the least-significant-bit is assigned the number 0.

Screen data refers to coordinates and to dimensions. Absolute coordinates are stored in a word field, in two's complement binary format. Relative coordinates are stored in byte fields, also in two's complement binary form. Screen dimensions are defined in the Cartesian plane: the x coordinate represents the horizontal value and the y coordinate the vertical value. The origin is located at the top-left screen corner. In the 8514/A the valid coordinate range is from -512 to +1535 in the x and y planes, respectively, while in XGA it is from -2048 to +6145 for both Cartesian coordinates. The viewport (video buffer) is in one of two modes in both systems: in low resolution mode the x coordinate is in the range 0 to 639 and the y coordinate in the range 0 to 479. In high-resolution mode the x coordinate is in the range 0 to 1023 and the y coordinate in the range 0 to 767. The image buffer and viewports for XGA systems are shown in Figure 11-7.

Figure 11-7 *XGA System Coordinate Range and Viewport*

Address data is in conventional Intel logical address format, that is, in segment:offset form. If offset and segment are stored separately in word-size data items, the offset element precedes the segment element, as in the following parameter block for the HSCS (select character set) command:



```
HSCS_DATA        DW       4        ; Length of data block
```

```
FONT_OFF        DW      0       ; Offset of loaded font
FONT_SEG        DW      0       ; Segment of loaded font
```

Address data does not always require a logical address. For example, in the parameter block for the HINIT function call only the segment element of the address is required, as shown in the following code fragment:

```
HINIT_DATA      DW      2       ; Length of data block
BUF_SEG         DW      0       ; Segment of task state buffer
```

11.4 AI Concepts

Before venturing into the details of AI programming it is convenient to gain familiarity with some graphics concepts often mentioned in the adapter's literature. Most of these concepts are taken from the general terminology of computer graphics, although, in a few cases, IBM documentation varies from the more generally accepted terms.

11.4.1 Pixel Attributes

A pixel's color is primarily determined by the value stored in the memory maps and by the setting of the LUT registers, as shown in Figure 11-3 and discussed in section 11.2.4. By means of the AI services an application can access the color value stored in the bit planes through the HSCOL (set current color) and HSBCOL (set background color) commands. Generally, a 1-bit in a draw order is displayed using the current foreground color while a 0-bit is displayed using the current background color. In text operations the background color refers to the rectangular pixel block on which text characters are drawn, while the foreground color refers to the text characters themselves.

Mixes

XGA and 8514/A system provide a second level of control over pixel display by means of a mechanism called mixes. Mixes are logical or mathematical operations performed between a new color value and the one already stored in display memory. The mix mode is selected independently for the foreground and background colors.

Color Compares

The color compare mechanism in the XGA and 8514/A AI provides a means by which the programmer can exclude specific bit planes from graphics operations. Comparison logic allows operations of equal-to, less-than, greater-than, greater-than-or-equal-to, and less-than-or-equal-to. When the comparison evaluates to TRUE the bit plane data is unmodified. When the comparison evaluates to FALSE, then the active mix operation is allowed to take place. The color compare function is selected by means of the HSCMP (set color compare register). Notice that the color compare function is not active during the AI alphanumeric services.

Bit Plane Masking

In addition to the controls offered by foreground and background colors, mix mode, and the color compare setting, an application can use masking to selectively enable and disable individual bit planes. The bit plane masking function allows separate control for graphics and alphanumeric operations. The masking function takes place before compares and mixes are applied; therefore the mask can be used to exclude compare and mix operations. Bit plane masking is performed by means of the HSBP (set bit plane control) function.

11.4.2 Scissoring

The AI software provides a function by which an application can limit graphics operations to a rectangular area within the viewport. This function, called scissoring in the IBM documentation, is useful in developing programs that use screen windows, since it inhibits operations outside a predefined screen rectangle. During adapter initializing the scissoring rectangle is set to the size of the viewport, but an application can redefine it by means of the HSHS (set scissor) function.

11.4.3 Absolute and Current Screen Positions

Several AI graphics and text functions are based on absolute screen locations. For example, the HLINE function (see Section 11.5.2) can be used to draw one or more straight lines starting at a given screen coordinate point. On the other hand, other AI graphics and text functions operate from a current screen position which is maintained by the adapter. For example, the HCLINE function can be used to draw one or more straight line segments starting at the current position. In this function the current screen position is automatically updated to the end point of the last line segment. The current screen position can be set by means of the HSCP (set current position) function, described in Section 11.5.2.

11.4.4 Polymarkers

A marker, in the context of the XGA and 8514/A AI programming, is a bitmapped object that can be as large as 255 by 255 pixels. The AI software allows displaying one or more markers at the predefined absolute coordinates or at the current display position. Since more than one marker can be displayed by the same command, the AI function should be classified as a polymarker operation.

The marker image is a rectangular, unpadded bitmap. If defined as a monochrome marker it is displayed using the current foreground color and according to the selected mix. If the marker is defined as a multicolor one, it is displayed using a color table supplied by the caller.

In 8514/A the multicolor table consists of a 1-byte color code for each bit in the marker bitmap. In XGA the program can select a color table in byte-per-pixel mode (compatible with 8514/A) or in packed format. In the packed format the mapping of the color table depends on the system's resolution. For example, if the pixel color is determined by 4 video memory bits, then the color table consists of a series of packed, 4-bit color codes. Notice that the packed format is not supported in the 8514/A.

The current marker is defined by means of the HSMARK (set marker shape) function. One or more markers are displayed at absolute screen positions by means of the HMRK (display marker) function. The HCMRK (marker at current point) function is used to display one or more markers at the current position. These functions are described in Section 11.5.4.

11.4.5 Line Widths and Types

The XGA and 8514/A AI allow selecting the line width and type to be used in line drawing operations. Line width options are of one or three pixels. Three-pixel-wide lines are drawn as three separate lines, one pixel apart. There are eight built-in line types in the AI software: dotted, short dashed, dash-dot, double dot, long dashed, dash-double-dot, solid, and invisible lines. In addition, the XGA AI offers a second dotted line type not available in 8514/A. An application can also define its own customized line type.

Each line type consists of a repeating pattern of dots and dashes. While drawing a non-continuous line, the AI software keeps track of the current position in the line pattern. Although most line drawing functions reset the pattern counter at the start of a line, an application can override this mode of operation by saving and restoring the current position in the line pattern. The AI function named HSLPC (save line pattern count) and HRLPC (restore line pattern count) are used for this purpose. These functions are particularly useful when a non-continuous line must straddle a scissor boundary.

The line type selection option in the AI simplifies considerably the development of drafting and computer-assisted design software. On the other hand, the line width selection option is often considered too limited to be of practical use. Line width selection is performed by means of the HSLW (select line width) function while line type is chosen by the HSLT (select line type) function.

11.4.6 Bit Block Operations

Graphics programs often operate on rectangular blocks of bitmapped data called bit blocks. The manipulations of these blocks are called bit block transfers; the expression is often shortened to bitBLTS (pronounced bit blits). BitBLT operations often refer to a source block, a destination block, and to the logical operation to be performed in combining them into a result block. In the AI the logical operation is selected by means of the mix (see Section 11.4.1).

BitBLTs are one of the most powerful graphics tools in the AI. The bit block transfer operations can take place from the application's memory space to video memory, from video memory to the application's memory space, and from video memory to video memory. When the bitmapped image stored by the applications is transferred to the adapter's video memory we speak of a bitBLT write. When the data stored in the adapter's video RAM is moved to the application's memory we speak of a bitBLT read. Operations by which data are moved within the application's video space are called a bitBLT copy.

BitBLTs operate on a rectangular area. They proceed from the top-left corner of the rectangle, left-to-right and top-to-bottom. Due to this mode of operations they are sometimes called raster functions.

BitBLT Copy

An AI bitBLT copy operation produces a second screen image based on the pixel data stored in a screen rectangle defined by the caller. The second image is displayed according to the current mix and comparison and clipped according to the scissoring. If the two images overlap, the AI correctly places the new image overlapping the existing one. The copy operation can be performed in one of two modes. In the single-plane mode the application selects a single image plane which is copied by the AI service. In the multiplane mode the entire image is copied to the new position.

The AI function for performing a bitBLT copy operation is named HBBC (bitBLT copy). In this function the caller must provide a parameter block containing the desired mode (single-plane or multiplane), the dimensions of the bitBLT rectangle, the selected bit plane if the single-plane mode is active, and the coordinates of the source and destination areas.

BitBLT Write

An application can display a bitmapped image stored in its own memory space by performing a bitBLT write operation. The screen image is displayed according to the current mix and comparison values and is clipped according to the scissoring. In XGA and 8514/A systems the write operation can take place in one of two modes. If the monochrome mode is selected, the image bitmap is displayed using the current foreground color for the 1-bits and the current background color for the 0-bits. In this case the bitmap is assumed to be encoded in a 1-bit per pixel format.

If the color mode is selected then the AI assumes that the image is encoded in a byte-per-pixel format. In other words, the caller provides an image map in which each screen pixel is represented by the color code stored in 1 data byte. The actual color displayed depends on the present setting of the LUT registers and the number of active bit planes. In addition to the monochrome and color modes, the XGA AI offers an additional packed bits mode. In the packed mode the number of bits per pixel depends on the current display mode. For example, if the adapter is in a 4-bit plane display mode, then the AI assumes that the caller's image data is encoded in a one-nibble-per-pixel format. The packed mode is not available in 8514/A systems.

Three different AI functions are related to bitBLT write operations. The function named HBBW (bitBLT write) is used to transfer image data to a screen location specified by the caller. HCBBW (bitBLT write at current position) transfers the image data to the current position. Both of these functions are of preparatory nature. The actual display of the bit block requires the use of an AI service named HBBCHN (bitBLT chain). This command includes the address of the bitmap in the application's memory space as well as its dimensions. The use of HBBW, HCBBW, and HBBCHN commands is illustrated in Section 11.5.4.

BitBLT Read

An application can also use the AI bitBLT services to move a video image to its own memory space. In this type of operation, called a bitBLT read, the application defines the coordinates of a screen rectangle, as well as the location, in its application's memory space, of a buffer for storing the video data. The AI then makes a copy of the screen image in the application's RAM. The size of the image rectangle can be as small as a single pixel or as large as the entire screen.

As is the case in the bitBLT write operation, XGA and 8514/A systems allow bitBLT reads in one of two modes. If the monochrome mode is selected, the image is read from the bit plane specified by the caller. In this case the application must provide a storage space of one bit per screen pixel. If the color mode is selected the AI will read all 8 bit planes and store a byte-per-pixel color code in the buffer provided by the caller. In addition to the monochrome and color modes, the XGA AI offers an additional packed bits mode, similar to the one described for the bitBLT write operation. The packed mode is not available in 8514/A systems.

Two AI functions are related to bitBLT read operations. The function named HBBR (bitBLT read) is used to transfer video image data to a buffer supplied by the caller. This AI function is of preparatory nature. The actual storage of bit block data requires the use of the HBBCHN (bitBLT chain) AI service. The HBBCHN command provides the address of the storage buffer in the application's memory space as well as its dimensions.

11.5 Details of AI Programming

In the present section we offer examples of AI programming. The examples are presented in the form of assembly language code fragments with the corresponding comments and explanations. We have mentioned that the IBM AI documentation uses C language almost exclusively. In our examples we have selected assembly language instead in order to provide an alternative programming medium, and also because we feel that examples in assembly language provide clearer illustration of data structure and of the machine hardware operations than do examples in high level languages. Once a reader understands the fundamental programming elements in an AI function, this knowledge can be easily applied in using the function from any particular programming language.

We remind the reader that the documentation published by IBM for XGA and 8514/A (see Section 11.2) contains descriptions, examples, and utility programs that are practically indispensable to the AI programmer. The book by Ritcher and Smith, titled *Graphics Programming for the 8514/A* (see Bibliography) will also be useful. In addition, the programs named AI_DEMO and AI_LUT included in the software furnished with this book include demonstration of AI programming examples.

11.5.1 Initialization and Control Functions

The fundamental initialization operations for the AI as well as the access mechanism for using the AI commands were described in Section 11.3. The following code fragment shows the typical sequence of AI commands that an application would execute in order to establish communications with the adapter software. In this example we assume that the access mechanism is by the procedure named AI_SERVICE described in Section 11.3.1. The code is virtually identical to the one in the AI_INIT.ASM template furnished in the book's software package.

```
;****************************************************************
;                               stack segment
;****************************************************************
STACK   SEGMENT stack
```

```
        DB      0400H DUP ('?')    ; Default stack is 1K
;
STACK   ENDS
;
;****************************************************************
;                segment for task state data
;****************************************************************
TASK    SEGMENT
;**********************|
;  AI state buffer     |
;**********************|
STATE_BUF       DB      360 DUP (00H)
;
TASK    ENDS
;
;****************************************************************
;                          data segment
;****************************************************************
DATA    SEGMENT
;**********************|
;   AI list address    |
;**********************|
AI_ADD          DD      0       ; Doubleword storage for address
                                ; of Adapter Interface services
;
;
HQDPS_DATA      DW      6       ; Length of data block
BUF_SIZE        DW      0       ; Buffer size
STK_SIZE        DW      0       ; Stack usage, in bytes
PAL_SIZE        DW      0       ; Palette buffer size, in bytes
;
HOPEN_DATA      DW      3       ; Length of data block
INIT_FLAGS      DB      0       ; 7 6 5 4 3 2 1 0 <= flags
                                ; | | ___________
                                ; | |       |_______ Reserved
                                ; | |_ Do not load palette
                                ; |___ Do not clear bit planes
AF_MODE         DB      0       ; Advanced function mode
                                ; No.   Pixels     Text
                                ; 00    1024x768   85x38
                                ; 01    640x480    80x34
                                ; 02    1024x768   128x54
                                ; 03    1024x768   146x51
RET_FLAGS       DB      0       ; Status
                                ; 0 if initialization successful
                                ; Not 0 if initialization failed
;
HINIT_DATA      DW      2       ; Length of data block
BUF_SEG         DW      0       ; Segment of AI buffer
HCLOSE_DATA     DW      0       ; Length field is zero for HCLOSE
HEGS_DATA       DW      0       ; Length field is zero for HEGS
DUMMY           DW      0       ; Dummy data area
        .
        .
        .
DATA    ENDS

;****************************************************************
;                          code segment
;****************************************************************
```

```
;
CODE    SEGMENT
        ASSUME  CS:CODE
;
START:
; Establish data and extra segment addressability
        MOV     AX,DATA         ; Address of DATA to AX
        MOV     DS,AX           ; and to DS
        ASSUME  DS:DATA         ; Assume from here on
;*********************|
;  get adapter address |
;*********************|
; The local procedure AI_VECTOR obtains the segment:offset
; address of the adapter handler
        CALL    AI_VECTOR       ; Local procedure
        JNC     OK_VECTOR       ; Go if no carry
;*********************|
;     error exit      |
;*********************|
AI_ERROR:
; HEGS (erase graphics screen)
        MOV     AX,13           ; Code number for this service
        LEA     BX,HEGS_DATA    ; Pointer to dummy data block
        CALL    AI_SERVICE
;*********************|
;     exit to DOS     |
;*********************|
DOS_EXIT:
        MOV     AH,4CH          ; DOS service request code
        MOV     AL,0            ; No error code returned
        INT     21H             ; TO DOS
;*********************|
;    AI installed     |
;*********************|
OK_VECTOR:
; Store segment and offset of AI handler
        MOV     WORD PTR AI_ADD,DX      ; Store offset of address
        MOV     WORD PTR AI_ADD+2,CX    ; and segment
; Entry point for AI services is now stored in a DS variable
;*********************|
;    initialize AI    |
;*********************|
; Call HQDPS service (query drawing process state)
        MOV     AX,31           ; Code number for this service
        LEA     BX,HQDPS_DATA   ; Pointer to data block
        CALL    AI_SERVICE
; The following information is stored by the query drawing
; process command
; 1. size of task state buffer
; 2. stack usage, in bytes
; 3. size of palette buffer
; This information may later be required by the application
;
; Call HOPEN service (enable adapter)
        MOV     INIT_FLAGS,0    ; Set initialization flags
                                ; to clear memory and load
                                ; default palette
        MOV     AF_MODE,0       ; Set 1024x768 mode number 0
        MOV     AX,8            ; Code number for this service
        LEA     BX,HOPEN_DATA   ; Pointer to data block
        CALL    AI_SERVICE
; The HOPEN command returns system information in the RET_FLAGS
; field of the parameter block.
        MOV     AL,RET_FLAGS    ; Not zero if open error
        CMP     AL,0            ; Test for no error
        JZ      OK_OPEN         ; Go if no error
        JMP     AI_ERROR        ; Error exit
;
; Call HINIT (Initialize adapter state)
OK_OPEN:
        MOV     AX,TASK         ; Segment for task state
        MOV     BUF_SEG,AX      ; Store segment of adapter state
                                ; buffer
        MOV     AX,48           ; Code number for this service
        LEA     BX,HINIT_DATA   ; Pointer to data block
        CALL    AI_SERVICE
; At this point the AI is initialized and ready for use
;****************************************************************
;                       application's code
;***************************************************************
        .
        .
        .
;****************************************************************
;                           procedures
;***************************************************************
AI_VECTOR       PROC    NEAR
; Procedure to obtain the address vector to the XGA/8514/A
; AI. This procedure must be called before calls are made
; to the Adapter Interface services (by means of the AI_SERVICE
; procedure)
;
; On entry:
;          nothing
; On exit:
;          carry set if no AI installed
;          carry clear if AI present
;             CX => segment of AI link table
;             DX => offset of AI link table
;
;*********************|
;    get vector 7FH    |
;*********************|
; Use MS DOS service number 53, interrupt 21H, to obtain the
; vector for the XGA/8514-A AI interrupt (7FH)
        MOV     AH,53           ; MS DOS service number
        MOV     AL,7FH          ; AI interrupt
        INT     21H             ; MS DOS interrupt
; ES => segment of interrupt handler
; BX => offset of handler
        MOV     AX,ES           ; Segment to AX
        OR      AX,BX           ; OR segment and offset
        JNZ     OK_AI           ; Go if address not 0000:0000
;*********************|
;  ERROR - no AI       |
;*********************|
NO_AI:
        STC                     ; Error return
        RET
;*********************|
;  get AI address       |
;*********************|
; Service number 0105H, interrupt 7FH, returns the address of the
; XGA and 8514/A jump table
OK_AI:
        MOV     AX,0105H        ; Service request number
        INT     7FH             ; in XGA AI interrupt
        JC      NO_AI           ; Go if error code returned
; At this point CX:DX holds the address of the XGA/8514-A entry
; point (in segment:offset form)
        CLC                     ; No error code
        RET
AI_VECTOR       ENDP
;****************************************************************
;
AI_SERVICE      PROC    NEAR
; Procedure to access the services in the XGA and 8514/A Adapter
; Interface
;
; On entry:
;         AX = service number
;         DS:BX = address of parameter block
;
        PUSH    BP              ; Save base pointer
        MOV     BP,SP           ; Set BP to stack
; Push address of caller's parameter block
        PUSH    DS
        PUSH    BX              ; the offset
; Multiply by 4 to form offset as required by AI
        SHL     AX,1            ; AX time 2
        SHL     AX,1            ; again
        MOV     SI,AX           ; Offset to SI
        LES     BX,AI_ADD       ; Entry block address (ES:BX)
        CALL    DWORD PTR ES:[BX][SI]    ; Call AI service
        POP     BP              ; Restore caller's BP
        RET
AI_SERVICE      ENDP
;****************************************************************
CODE    ENDS
        END     START
```

11.5.2 Setting the Color Palette

The structure of the XGA and 8514/A color look-up table (LUT) and the digital-to-analog converter is discussed in Section 11.2.4. The actual manipulation of the XGA and 8514/A DAC registers is by means of three palette commands: HSPAL (save palette), HLDPAL (load palette registers), and HRPAL (restore palette). The following code fragment shows the use of the AI palette commands.

```
;****************************************************************
;                           data segment
;****************************************************************
DATA    SEGMENT
        .
        .
        .
;*********************|
;    palette data      |
;*********************|
; Data area for HLDPAL (load palette) function
HLDPAL_DATA     DW      10      ; Length of data block
LOAD_CODE       DB      0       ; Palette code
                                ; 0 = load user pallete
                                ; 1 = load default pallete
                DB      0       ; Reserved
                DW      0       ; Number of first entry
                DW      256     ; Number of entries to load
PAL_OFF         DW      0       ; Offset of user palette
PAL_SEG         DW      0       ; Segment of user palette
;
; Data area for HSPAL (save palette data)
; and HRPAL (restore palette)
HSPAL_DATA      DW      769     ; Length of palette
                DW      769 DUP (00H)   ; Storage for palette
;
; Double-bit IRGB palette in the following format
;            7 6 5 4 3 2 1 0  <= Bits
;            | | | | | | | |
;            | | | | | | |_|_______ Blue
;            | | | | |_|___________ Green
;            | | |_|_______________ Red
;            |_|___________________ Intensity

;
; First group of 64 registers
; Notice that the DAC color registers are in the order
; Red-Blue-Green
;                      | R   B   G       R   B   G     |
IRGB_SHADES     DB      000,000,000,000,036,072,036,000 ; 1
                DB      036,108,036,000,036,144,036,000 ; 3
                DB      036,036,072,000,036,072,072,000 ; 5
                DB      036,108,072,000,036,144,072,000 ; 7
                DB      036,036,108,000,036,072,108,000 ; 9
                DB      036,108,108,000,036,144,108,000 ; 11
                DB      036,036,144,000,036,072,144,000 ; 13
                DB      036,108,144,000,036,144,144,000 ; 15
                DB      072,036,036,000,072,072,036,000 ; 17
                DB      072,108,036,000,072,144,036,000 ; 19
                DB      072,036,072,000,072,072,072,000 ; 21
                DB      072,108,072,000,072,144,072,000 ; 23
                DB      072,036,108,000,072,072,108,000 ; 25
                DB      072,108,108,000,072,144,108,000 ; 27
                DB      072,036,144,000,072,072,144,000 ; 29
                DB      072,108,144,000,072,144,144,000 ; 31
                DB      108,036,036,000,108,071,036,000 ; 33
                DB      108,108,036,000,108,144,036,000 ; 35
                DB      108,036,072,000,108,072,072,000 ; 37
                DB      108,108,072,000,108,144,072,000 ; 39
                DB      108,036,108,000,108,072,108,000 ; 41
                DB      108,108,108,000,108,144,108,000 ; 43
                DB      036,036,144,000,108,072,144,000 ; 45
                DB      108,108,144,000,108,144,144,000 ; 47
                DB      144,036,036,000,144,072,036,000 ; 49
                DB      144,108,036,000,144,144,036,000 ; 51
                DB      144,036,072,000,144,072,072,000 ; 53
                DB      144,108,072,000,144,144,072,000 ; 55
                DB      144,036,108,000,144,072,108,000 ; 57
                DB      144,108,108,000,144,144,108,000 ; 59
                DB      144,036,144,000,144,072,144,000 ; 61
```

```
                    DB      144,108,144,000,144,144,144,000 ; 63
    ; Second register group
                    DB      072,072,072,000,072,108,072,000 ; 1
                    DB      072,144,072,000,072,180,072,000 ; 3
                    DB      072,072,108,000,072,108,108,000 ; 5
                    DB      072,144,108,000,072,180,108,000 ; 7
                    DB      072,072,144,000,072,108,144,000 ; 9
                    DB      072,144,144,000,072,180,144,000 ; 11
                    DB      072,072,180,000,072,108,180,000 ; 13
                    DB      072,144,180,000,072,180,180,000 ; 15
                    DB      108,072,072,000,108,108,072,000 ; 17
                    DB      108,144,072,000,108,180,072,000 ; 19
                    DB      108,072,108,000,108,108,108,000 ; 21
                    DB      108,144,108,000,108,180,108,000 ; 23
                    DB      108,072,144,000,108,108,144,000 ; 25
                    DB      108,144,144,000,108,180,144,000 ; 27
                    DB      108,072,180,000,108,108,180,000 ; 29
                    DB      108,144,180,000,108,180,180,000 ; 31
                    DB      144,072,072,000,144,108,072,000 ; 33
                    DB      144,144,072,000,144,180,072,000 ; 35
                    DB      144,072,108,000,144,108,108,000 ; 37
                    DB      144,144,108,000,144,180,108,000 ; 39
                    DB      144,072,144,000,144,108,144,000 ; 41
                    DB      144,144,144,000,144,180,144,000 ; 43
                    DB      072,072,180,000,144,108,180,000 ; 45
                    DB      144,144,180,000,144,180,180,000 ; 47
                    DB      180,072,072,000,180,108,072,000 ; 49
                    DB      180,144,072,000,180,180,072,000 ; 51
                    DB      180,072,108,000,180,108,108,000 ; 53
                    DB      180,144,108,000,180,180,108,000 ; 55
                    DB      180,072,144,000,180,108,144,000 ; 57
                    DB      180,144,144,000,180,180,144,000 ; 59
                    DB      180,072,180,000,180,108,180,000 ; 61
                    DB      180,144,180,000,180,180,180,000 ; 63
    ; Third register group
                    DB      108,108,108,000,108,144,108,000 ; 1
                    DB      108,180,108,000,108,216,108,000 ; 3
                    DB      108,108,144,000,108,144,144,000 ; 5
                    DB      108,180,144,000,108,216,144,000 ; 7
                    DB      108,108,180,000,108,144,180,000 ; 9
                    DB      108,180,180,000,108,216,180,000 ; 11
                    DB      108,108,216,000,108,144,216,000 ; 13
                    DB      108,180,216,000,108,216,216,000 ; 15
                    DB      144,108,108,000,144,144,108,000 ; 17
                    DB      144,180,108,000,144,216,108,000 ; 19
                    DB      144,108,144,000,144,144,144,000 ; 21
                    DB      144,180,144,000,144,216,144,000 ; 23
                    DB      144,108,180,000,144,144,180,000 ; 25
                    DB      144,180,180,000,144,216,180,000 ; 27
                    DB      144,108,216,000,144,144,216,000 ; 29
                    DB      144,180,216,000,144,216,216,000 ; 31
                    DB      180,108,108,000,180,144,108,000 ; 33
                    DB      180,180,108,000,180,216,108,000 ; 35
                    DB      180,108,144,000,180,144,144,000 ; 37
                    DB      180,180,144,000,180,216,144,000 ; 39
                    DB      180,108,180,000,180,144,180,000 ; 41
                    DB      180,180,180,000,180,216,180,000 ; 43
                    DB      108,108,216,000,180,144,216,000 ; 45
                    DB      180,180,216,000,180,216,216,000 ; 47
                    DB      216,108,108,000,216,144,108,000 ; 49
```

```
                DB      216,180,108,000,216,216,108,000 ; 51
                DB      216,108,144,000,216,144,144,000 ; 53
                DB      216,180,144,000,216,216,144,000 ; 55
                DB      216,108,180,000,216,144,180,000 ; 57
                DB      216,180,180,000,216,216,180,000 ; 59
                DB      216,108,216,000,216,144,216,000 ; 61
                DB      216,180,216,000,216,216,216,000 ; 63
; Fourth register group
                DB      144,144,144,000,144,180,144,000 ; 1
                DB      144,216,144,000,144,252,144,000 ; 3
                DB      144,144,180,000,144,180,180,000 ; 5
                DB      144,216,180,000,144,252,180,000 ; 7
                DB      144,144,216,000,144,180,216,000 ; 9
                DB      144,216,216,000,144,252,216,000 ; 11
                DB      144,144,252,000,144,180,252,000 ; 13
                DB      144,216,252,000,144,252,252,000 ; 15
                DB      180,144,144,000,180,180,144,000 ; 17
                DB      180,216,144,000,180,252,144,000 ; 19
                DB      180,144,180,000,180,180,180,000 ; 21
                DB      180,216,180,000,180,252,180,000 ; 23
                DB      180,144,216,000,180,180,216,000 ; 25
                DB      180,216,216,000,180,252,216,000 ; 27
                DB      180,144,252,000,180,180,252,000 ; 29
                DB      180,216,252,000,180,252,252,000 ; 31
                DB      216,144,144,000,216,180,144,000 ; 33
                DB      216,215,144,000,216,252,144,000 ; 35
                DB      216,144,180,000,216,180,180,000 ; 37
                DB      216,216,180,000,216,252,180,000 ; 39
                DB      216,144,216,000,216,180,216,000 ; 41
                DB      216,216,216,000,216,252,216,000 ; 43
                DB      144,144,252,000,216,180,252,000 ; 45
                DB      216,216,252,000,216,252,252,000 ; 47
                DB      252,144,144,000,252,180,144,000 ; 49
                DB      252,216,144,000,252,252,144,000 ; 51
                DB      252,144,180,000,252,180,180,000 ; 53
                DB      252,216,180,000,252,252,180,000 ; 55
                DB      252,144,216,000,252,180,216,000 ; 57
                DB      252,216,216,000,252,252,216,000 ; 59
                DB      252,144,252,000,252,180,252,000 ; 61
                DB      252,216,252,000,252,252,252,000 ; 63

; Gray shades palette. Notice that the pattern in the first 64
; registers is repeated 3 times
GRAY_SHADES     DB      000,000,000,000,004,004,004,000 ; 1
                DB      008,008,008,000,012,012,012,000 ; 3
                DB      016,016,016,000,020,020,020,000 ; 5
                DB      024,024,024,000,028,028,028,000 ; 7
                DB      032,032,032,000,036,036,036,000 ; 9
                DB      040,040,040,000,044,044,044,000 ; 11
                DB      048,048,048,000,052,052,052,000 ; 13
                DB      056,056,056,000,060,060,060,000 ; 15
                DB      064,064,064,000,068,068,068,000 ; 17
                DB      072,072,072,000,076,076,076,000 ; 19
                DB      080,080,080,000,084,084,084,000 ; 21
                DB      088,088,088,000,092,092,092,000 ; 23
                DB      096,096,096,000,100,100,100,000 ; 25
                DB      104,104,104,000,108,108,108,000 ; 27
                DB      112,112,112,000,116,116,116,000 ; 29
                DB      120,120,120,000,124,124,124,000 ; 31
                DB      128,128,128,000,132,132,132,000 ; 33
                DB      136,136,136,000,140,140,140,000 ; 35
                DB      144,144,144,000,148,148,148,000 ; 37
                DB      152,152,152,000,156,156,156,000 ; 39
                DB      160,160,160,000,164,164,164,000 ; 41
                DB      168,168,168,000,172,172,172,000 ; 43
                DB      176,176,176,000,180,180,180,000 ; 45
                DB      184,184,184,000,188,188,188,000 ; 47
                DB      192,192,192,000,196,196,196,000 ; 49
                DB      200,200,200,000,204,204,204,000 ; 51
                DB      208,208,208,000,212,212,212,000 ; 53
                DB      216,216,216,000,220,220,220,000 ; 55
                DB      224,224,224,000,228,228,228,000 ; 57
                DB      232,232,232,000,236,236,236,000 ; 59
                DB      240,240,240,000,244,244,244,000 ; 61
                DB      248,248,248,000,252,252,252,000 ; 63
;
                DB      000,000,000,000,004,004,004,000 ; 1
                DB      008,008,008,000,012,012,012,000 ; 3
                DB      016,016,016,000,020,020,020,000 ; 5
                DB      024,024,024,000,028,028,028,000 ; 7
                DB      032,032,032,000,036,036,036,000 ; 9
                DB      040,040,040,000,044,044,044,000 ; 11
                DB      048,048,048,000,052,052,052,000 ; 13
                DB      056,056,056,000,060,060,060,000 ; 15
                DB      064,064,064,000,068,068,068,000 ; 17
                DB      072,072,072,000,076,076,076,000 ; 19
                DB      080,080,080,000,084,084,084,000 ; 21
                DB      088,088,088,000,092,092,092,000 ; 23
                DB      096,096,096,000,100,100,100,000 ; 25
                DB      104,104,104,000,108,108,108,000 ; 27
                DB      112,112,112,000,116,116,116,000 ; 29
                DB      120,120,120,000,124,124,124,000 ; 31
                DB      128,128,128,000,132,132,132,000 ; 33
                DB      136,136,136,000,140,140,140,000 ; 35
                DB      144,144,144,000,148,148,148,000 ; 37
                DB      152,152,152,000,156,156,156,000 ; 39
                DB      160,160,160,000,164,164,164,000 ; 41
                DB      168,168,168,000,172,172,172,000 ; 43
                DB      176,176,176,000,180,180,180,000 ; 45
                DB      184,184,184,000,188,188,188,000 ; 47
                DB      192,192,192,000,196,196,196,000 ; 49
                DB      200,200,200,000,204,204,204,000 ; 51
                DB      208,208,208,000,212,212,212,000 ; 53
                DB      216,216,216,000,220,220,220,000 ; 55
                DB      224,224,224,000,228,228,228,000 ; 57
                DB      232,232,232,000,236,236,236,000 ; 59
                DB      240,240,240,000,244,244,244,000 ; 61
                DB      248,248,248,000,252,252,252,000 ; 63
;
                DB      000,000,000,000,004,004,004,000 ; 1
                DB      008,008,008,000,012,012,012,000 ; 3
                DB      016,016,016,000,020,020,020,000 ; 5
                DB      024,024,024,000,028,028,028,000 ; 7
                DB      032,032,032,000,036,036,036,000 ; 9
                DB      040,040,040,000,044,044,044,000 ; 11
                DB      048,048,048,000,052,052,052,000 ; 13
                DB      056,056,056,000,060,060,060,000 ; 15
                DB      064,064,064,000,068,068,068,000 ; 17
                DB      072,072,072,000,076,076,076,000 ; 19
                DB      080,080,080,000,084,084,084,000 ; 21
                DB      088,088,088,000,092,092,092,000 ; 23
                DB      096,096,096,000,100,100,100,000 ; 25
                DB      104,104,104,000,108,108,108,000 ; 27
                DB      112,112,112,000,116,116,116,000 ; 29
                DB      120,120,120,000,124,124,124,000 ; 31
                DB      128,128,128,000,132,132,132,000 ; 33
                DB      136,136,136,000,140,140,140,000 ; 35
                DB      144,144,144,000,148,148,148,000 ; 37
                DB      152,152,152,000,156,156,156,000 ; 39
                DB      160,160,160,000,164,164,164,000 ; 41
                DB      168,168,168,000,172,172,172,000 ; 43
                DB      176,176,176,000,180,180,180,000 ; 45
                DB      184,184,184,000,188,188,188,000 ; 47
                DB      192,192,192,000,196,196,196,000 ; 49
                DB      200,200,200,000,204,204,204,000 ; 51
                DB      208,208,208,000,212,212,212,000 ; 53
                DB      216,216,216,000,220,220,220,000 ; 55
                DB      224,224,224,000,228,228,228,000 ; 57
                DB      232,232,232,000,236,236,236,000 ; 59
                DB      240,240,240,000,244,244,244,000 ; 61
                DB      248,248,248,000,252,252,252,000 ; 63
;
                DB      000,000,000,000,004,004,004,000 ; 1
                DB      008,008,008,000,012,012,012,000 ; 3
                DB      016,016,016,000,020,020,020,000 ; 5
                DB      024,024,024,000,028,028,028,000 ; 7
                DB      032,032,032,000,036,036,036,000 ; 9
                DB      040,040,040,000,044,044,044,000 ; 11
                DB      048,048,048,000,052,052,052,000 ; 13
                DB      056,056,056,000,060,060,060,000 ; 15
                DB      064,064,064,000,068,068,068,000 ; 17
                DB      072,072,072,000,076,076,076,000 ; 19
                DB      080,080,080,000,084,084,084,000 ; 21
                DB      088,088,088,000,092,092,092,000 ; 23
                DB      096,096,096,000,100,100,100,000 ; 25
                DB      104,104,104,000,108,108,108,000 ; 27
                DB      112,112,112,000,116,116,116,000 ; 29
                DB      120,120,120,000,124,124,124,000 ; 31
                DB      128,128,128,000,132,132,132,000 ; 33
                DB      136,136,136,000,140,140,140,000 ; 35
                DB      144,144,144,000,148,148,148,000 ; 37
                DB      152,152,152,000,156,156,156,000 ; 39
                DB      160,160,160,000,164,164,164,000 ; 41
                DB      168,168,168,000,172,172,172,000 ; 43
                DB      176,176,176,000,180,180,180,000 ; 45
                DB      184,184,184,000,188,188,188,000 ; 47
                DB      192,192,192,000,196,196,196,000 ; 49
                DB      200,200,200,000,204,204,204,000 ; 51
                DB      208,208,208,000,212,212,212,000 ; 53
                DB      216,216,216,000,220,220,220,000 ; 55
                DB      224,224,224,000,228,228,228,000 ; 57
                DB      232,232,232,000,236,236,236,000 ; 59
                DB      240,240,240,000,244,244,244,000 ; 61
                DB      248,248,248,000,252,252,252,000 ; 63
;
DATA    ENDS

;***************************************************************
;                          code segment
;***************************************************************
```

```
;
CODE    SEGMENT
        ASSUME  CS:CODE
        .
        .
        .
; Call HSPAL to save current palette
        MOV     AX,57           ; Code number for this service
        LEA     BX,HSPAL_DATA   ; Pointer to data block
        CALL    AI_SERVICE
        .
        .
        .
; Initialize DAC registers for 256-color mode in the following
; format:
;               7 6 5 4 3 2 1 0  <= bits
;               |_| |_| |_| |_|
;                I   R   G   B
;*********************|
;  set LUT registers  |
;*********************|
; Set address of color table in HLDPAL data area
        PUSH    DS              ; DS to stack
        POP     PAL_SEG         ; Store segment in variable
        LEA     SI,IRGB_SHADES  ; Pointer to offset of address
        MOV     PAL_OFF,SI      ; Store offset
; Call HLDPAL to set palette registers
        MOV     AX,19           ; Code number for this service
        LEA     BX,HLDPAL_DATA  ; Pointer to data block
        CALL    AI_SERVICE
        .
        .
        .
; Initialize DAC registers for 64 gray shades, repeated 4 times
;*********************|
;  set LUT registers  |
;*********************|
; Set address of color table in HLDPAL data area
        PUSH    DS              ; DS to stack
        POP     PAL_SEG         ; Store segment in variable
        LEA     SI,GRAY_SHADES  ; Pointer to offset of address
        MOV     PAL_OFF,SI      ; Store offset
; Call HLDPAL to set palette registers
        MOV     AX,19           ; Code number for this service
        LEA     BX,HLDPAL_DATA  ; Pointer to data block
        CALL    AI_SERVICE
        .
        .
        .
; Call HRPAL to restore original palette
        MOV     AX,58           ; Code number for this service
        LEA     BX,HSPAL_DATA   ; Pointer to saved palette data
        CALL    AI_SERVICE
; Notice that the same data area in which the palette was saved
; is used during the restore operation
        .
        .
        .
CODE    ENDS
```

In addition to the three palette commands mentioned above, the AI contains a function named HQDFPAL (query default palette) that reports the default setting of the first 16 palette registers. HQDFPAL appears to be of little practical use, since the setting of all palette registers can be obtained by means of the HSPAL (save palette) function, and the default settings of the first 16 registers is usually known beforehand (see Table 11-3).

11.5.3 Geometrical Functions

Drawing operations on the XGA and 8514/A Adapter Interface are limited to straight line segments. The other geometrical functions are rectangular fill area fill operations.

Drawing Straight Lines

The AI documentation classifies the line drawing commands into three types: vertex, offset, and disjoint lines. All three line types are of the polyline category, since several line segments can be drawn with the same command. In all AI line drawing commands the characteristics of the line depend on the selected line type and width, as well as on the active color mix and comparison. The color of the line and its background is determined by the setting of the foreground and background colors.

HLINE (polyline at given position) and HCLINE (polyline at current position) are vertex-type commands. Both commands require a parameter block that encodes a set of coordinate points. The draw operation connects these coordinate points by means of straight line segments.

HRLINE (relative polyline at given position) and HCRLINE (relative polyline at current position) are offset-type commands. In HRLINE the start point of the polyline is the coordinate of a screen point. In the HCRLINE command the polyline starts at the current point. The remaining points in the polyline are described as offsets from the start point or the from the previous end point. The offsets are encoded as a 1-byte signed integer for the x coordinate and another one for the y coordinate. Since each offset is encoded in 1 byte, its range is limited to -128 to +127 pixels.

The disjoint line command is named HDLINE. This function is part of the XGA extended set, therefore, it is not available in 8514/A systems. In HDLINE the polyline is described by two coordinate points for each line segment; one marks the start of the line and the next one its end point. Since each line is described independently, the line segments that form the polyline can be disconnected from each other.

The following code fragment shows drawing a four-segment polyline using the HLINE command.

```
;****************************************************************
;                          data segment
;****************************************************************
DATA     SEGMENT
         .
         .
         .
; HLINE (polyline at given position)
HLINE_DATA       DW      18      ; Length of data block
                 DW      500     ; x coordinate of first point
```

```
                DW      300     ; y coordinate of first point
                DW      600     ; next x coordinate
                DW      300     ; next y coordinate
                DW      600     ; x
                DW      350     ; y
                DW      700     ; x
                DW      350     ; y
                DW      700     ; x
                DW      200     ; y
        .
        .
        .
DATA    ENDS
;****************************************************************
;                           code segment
;****************************************************************
;
CODE    SEGMENT
        ASSUME  CS:CODE
        .
        .
;*********************|
;     draw polyline   |
;*********************|
POLYGON:
; Call HSCOL (set color)
        MOV     FORE_COL,00001001B      ; Bright blue
        MOV     AX,7            ; Code number for this service
        LEA     BX,HSCOL_DATA   ; Pointer to data block
        CALL    AI_SERVICE
; Use the HLINE (polyline at given position) to draw a polyline
        MOV     AX,0            ; Code number for this service
        LEA     BX,HLINE_DATA   ; Pointer to data block
        CALL    AI_SERVICE
        .
        .
        .
CODE    ENDS
```

Rectangular Fill

The AI provides a service named HRECT (fill rectangle) which can be used to fill a rectangular area using the current foreground color and mix as well as an optional fill pattern defined by the caller. The optional pattern, which can be monochrome or color, is enabled by means of the HSPATT (set pattern shape) command. The rectangular fill operation can be conveniently used to clear a window within the viewport, or even the entire display. Notice that the HEGS (erase graphics screen) command can also be used to clear the entire display area. HEGS is independent of colors and mixes but is limited by the scissors and enabled planes.

The following code fragment shows the use of a rectangular fill operation in an XGA or 8514/A system.

```
;****************************************************************
;                           data segment
;****************************************************************
DATA    SEGMENT
        .
        .
        .
; Data block for rectangle draw
HRECT_DATA      DW      8       ; Length of data block
RECT_X          DW      0       ; x coordinate of top-left corner
RECT_Y          DW      0       ; y coordinate of top-left corner
RECT_WIDTH      DW      0       ; Width (1 to 1024)
RECT_HIGH       DW      0       ; Height (1 to 768)
        .
        .
        .
DATA    ENDS

;****************************************************************
;                           code segment
;****************************************************************
;
CODE    SEGMENT
        ASSUME  CS:CODE
        .
        .
        .
; Fill a rectangular area using HRECT
        MOV     RECT_X,100      ; x origin
        MOV     RECT_Y,50       ; y origin
        MOV     RECT_WIDTH,500  ; Width, in pixels
        MOV     RECT_HIGH,200   ; Height, in pixels
        MOV     AX,32           ; Code number for this service
        LEA     BX,HRECT_DATA   ; Pointer to data block
        CALL    AI_SERVICE
        .
        .
        .
CODE    ENDS
```

Area Fill

An application using the AI services can define a closed area before it is drawn and then fill its enclosed boundary with a solid color or a pattern. The HBAR (begin area definition) command is used to mark the start of the draw or move commands that will delimit the area to be filled. If the figure defined after the HBAR command is not properly closed, that is, if its start and end points do not coincide, it is closed automatically by the AI software. The actual fill operation is performed by means of the HEAR (end area definition) command. A control byte in the HEAR parameter area allows selecting one of three operations modes: fill area, suspend area definition, or abort. The control setting to suspend the area definition has the effect of leaving the presently defined area in an internal AI buffer until another HBAR or HEAR command is executed. Area fill operations take place using the current foreground color, as well as the pattern and mix.

The following code fragment shows the definition, drawing, and filling of a polygon.

```
;****************************************************************
;                           data segment
;****************************************************************
DATA    SEGMENT
```

```
        .
        .
        .
; Data for connected straight line segments to form a 7-segment
; polygon
HCLINE_DATA     DW      26      ; Length of data block
                                ; for 14 coordinate points
X1              DW      562     ; x coordinate of first end point
Y1              DW      384     ; y coordinate of first end point
X2              DW      700     ; Second pair of x,y coordinates
Y2              DW      500
X3              DW      520     ; Third pair of x,y coordinates
Y3              DW      550
X4              DW      400     ; Fourth pair of x,y coordinates
Y4              DW      500
X5              DW      450     ; Fifth pair of x,y coordinates
Y4              DW      384
X6              DW      530     ; Sixth pair of x,y coordinates
Y6              DW      450
X7              DW      512     ; Last pair of x,y coordinates
Y7              DW      384     ; are on screen center
        .
        .
        .
DATA    ENDS

;****************************************************************
;                          code segment
;****************************************************************
;
CODE    SEGMENT
        ASSUME  CS:CODE
        .
        .
        .
; Call HSCP (set current coordinate position)
; Coordinates are set at the center of the screen on 1024 by 768
; pixels modes
        MOV     NEW_X,512       ; Middle of screen column
        MOV     NEW_Y,384       ; Middle of screen row
        MOV     AX,4            ; Code number for this service
        LEA     BX,HSCP_DATA    ; Pointer to data block
        CALL    AI_SERVICE
; Call HBAR to begin fill area
        MOV     AX,5            ; Code number for this service
        LEA     BX,DUMMY        ; Pointer to data block
        CALL    AI_SERVICE
; Call HCLINE (draw line at current coordinate position)
; Coordinates of the line's start point were set by the HSCP
; service. Coordinates of polygon points already in data block
        MOV     AX,1            ; Code number for this service
        LEA     BX,HCLINE_DATA  ; Pointer to data block
        CALL    AI_SERVICE
; Call HEAR to fill area
        MOV     AX,6            ; Code number for this service
        LEA     BX,HEAR_DATA    ; Pointer to data block
        CALL    AI_SERVICE
        .
        .
        .
```

```
CODE    ENDS
```

11.5.4 Raster Operations

The XGA and 8514/A AI supports two types of raster operations: polymarker display and bitBLTs. These functions were described in Sections 11.4.4 and 11.4.6 respectively. In addition, the extended XGA AI services provide a means for manipulating on and off screen bitmaps. The bitmap functions are not available in 8514/A systems.

Polymarkers

Polymarkers are useful in displaying one or more copies of a bitmapped object. A typical use is in the animated display of one or more mouse-controlled screen objects. The following code fragment shows the display of two copies of a marker symbol.

```
;****************************************************************
;                          data segment
;****************************************************************
DATA    SEGMENT
        .
        .
        .
; Data area for HSMARK (define marker symbol)
HSMARK_DATA     DW      14      ; Length of data block
MARK_WIDE       DB      8       ; Pixel width of marker symbol
MARK_HIGH       DB      16      ; Pixel height of marker
MARK_TYPE       DB      0       ; 7 6 5 4 3 2 1 0 <= BITS
                                ; | |_|_|_|_|_|_|_ reserved (0)
                                ; |____________ 0 = monochrome
                                ;               1 = multicolor
                DB      0       ; Reserved
MARK_SIZE       DW      16      ; Number of bytes in marker image
                                ; size = ((width * height)+7)/8
MARK_OFF        DW      0       ; Offset of marker image map
MARK_SEG        DW      0       ; Segment of marker image map
M_COLOR_OFF     DW      0       ; Offset of color image map
M_COLOR_SEG     DW      0       ; Segment of color image map
;
; Bitmap for marker image
; Marker image is a vertical arrow symbol
MARK_MAP        DB      00100100B       ; 1
                DB      00111100B       ; 2
                DB      00111100B       ; 3
                DB      00111100B       ; 4
                DB      00011000B       ; 5
                DB      00011000B       ; 6
                DB      00011000B       ; 7
                DB      00011000B       ; 8
                DB      00011000B       ; 9
                DB      00011000B       ; 10
                DB      00011000B       ; 11
                DB      11111111B       ; 12
                DB      01111110B       ; 13
                DB      00111100B       ; 14
                DB      00011000B       ; 15
                DB      00011000B       ; 16
;
; Marker display command
HMRK_DATA       DW      8       ; Length of data block
```

```
MARKER_X0       DW      40      ; x coordinate of first marker
MARKER_Y0       DW      500     ; y coordinate of first marker
MARKER_X1       DW      55      ; x coordinate of second marker
MARKER_Y1       DW      500     ; y coordinate of second marker
        .
        .
        .
DATA    ENDS

;****************************************************************
;                          code segment
;****************************************************************
;
CODE    SEGMENT
        ASSUME  CS:CODE
        .
        .
        .
;**********************|
;   marker display     |
;**********************|
; Display monochrome marker (down arrow) stored at MARK_MAP
; First use HSMARK to define the marker bitmap
; Set address marker bitmap in control block variables
        PUSH    DS              ; Data segment
        POP     MARK_SEG        ; Store in variable
        LEA     SI,MARK_MAP     ; Offset of marker bitmap
        MOV     MARK_OFF,SI     ; Store offset of bitmap
; Call HSMARK
        MOV     AX,52           ; Code number for this service
        LEA     BX,HSMARK_DATA  ; Pointer to data block
        CALL    AI_SERVICE
; Call HMRK (display markers)
        MOV     AX,50           ; Code number for this service
        LEA     BX,HMRK_DATA    ; Pointer to data block
        CALL    AI_SERVICE
        .
        .
        .
CODE    ENDS
```

BitBLT

BitBLT operations in the AI allow read, write, and copy functions, as described in Section 11.4.6. Except for the polymarker function, bitBLT provides the only way in which an 8514/A application can read, write, or copy a bi map. The following code fragment shows two bitBLT operations, first, a bitmapped image of a running boar target, resident in RAM, is displayed using a bitBLT write operation. Second, the displayed image is copied to another screen position.

```
;*****************************************************************
;                           data segment
;*****************************************************************
DATA    SEGMENT
        .
        .
        .
; Data for bitBLT write operation
HBBW_DATA       DW      10      ; Length of data block
WR_FORMAT       DW      0       ; Format
                                ; 0000H = across the planes
                                ; 0008H = through the planes
WR_WIDTH        DW      48      ; Block's pixel width
WR_HEIGHT       DW      19      ; Pixel rows in block
DEST_X          DW      100     ; x coordinate for display
DEST_Y          DW      500     ; y coordinate for display
;
; Data for bitBLT chain image operation
HBBCHN_DATA     DW      6       ; Length of data block
BBLOK_OFF       DW      0       ; Offset of image map
BBLOK_SEG       DW      0       ; Segment of image map
BBLOK_SIZE      DW      114     ; Byte size of image buffer
;
; Data block for bit block copy
HBBC_DATA       DW      16      ; Length of data block
BLT_FORMAT      DW      8       ; Format
                                ; 0000H = across the planes
                                ; 0008H = through the planes
BLT_WIDTH       DW      60      ; Block's pixel width
BLT_HEGHT       DW      20      ; Pixel rows in block
PLANE_NUM       DB      0       ; Bit plane for across plane
                                ; mode
                DB      0       ; Reserved value
SOURCE_X        DW      20      ; x coordinate of source image
SOURCE_Y        DW      490     ; y coordinate of source
DESTIN_X        DW      200     ; x coordinate of destination
DESTIN_Y        DW      500     ; y coordinate of destination
;
;****************************|
;   bitmapped image in RAM   |
;****************************|
; Bitmap for a running boar target
; Bitmap dimensions are 6 bytes (48 pixels) by 19 rows
BOAR_MAP        DB      01FH,080H,00FH,0FFH,0F0H,000H ; 1
                DB      000H,043H,0F0H,081H,00EH,000H ; 2
                DB      000H,03CH,001H,03CH,081H,000H ; 3
                DB      000H,040H,002H,042H,040H,0C0H ; 4
                DB      000H,040H,004H,099H,020H,030H ; 5
                DB      000H,080H,005H,024H,0A0H,00CH ; 6
                DB      000H,080H,005H,05AH,0A0H,003H ; 7
                DB      000H,080H,005H,05AH,0A0H,001H ; 8
                DB      007H,000H,005H,024H,0A0H,01EH ; 9
                DB      008H,000H,004H,099H,020H,060H ; 10
                DB      008H,000H,002H,042H,047H,080H ; 11
                DB      010H,000H,001H,03CH,088H,000H ; 12
                DB      028H,000H,000H,081H,007H,080H ; 13
                DB      05FH,0C1H,0F0H,03FH,000H,040H ; 14
                DB      0FCH,03EH,00FH,0FCH,000H,0B0H ; 15
                DB      014H,000H,000H,002H,061H,060H ; 16
                DB      024H,000H,000H,001H,099H,000H ; 17
                DB      078H,000H,000H,000H,006H,080H ; 18
                DB      000H,000H,000H,000H,001H,0C0H ; 19
        .
        .
        .
DATA    ENDS

;*****************************************************************
;                          code segment
;*****************************************************************
;
CODE    SEGMENT
        ASSUME  CS:CODE
        .
        .
        .
;*********************|
;   bitBLT operations  |
;*********************|
; BitBLT bitmap of boar from memory to video
; Call HBBW (bit block write)
        MOV     AX,21           ; Code number for this service
        LEA     BX,HBBW_DATA    ; Pointer to data block
        CALL    AI_SERVICE
; Call HBBCHN to chain bit block
; Set address marker bitmap in control block variables
        PUSH    DS              ; Data segment
        POP     BBLOK_SEG       ; Store in variable
        LEA     SI,BOAR_MAP     ; Offset of marker bitmap
        MOV     BBLOK_OFF,SI    ; Store offset of bitmap
; Call HBBCHN service
        MOV     AX,24           ; Code number for this service
        LEA     BX,HBBCHN_DATA  ; Pointer to data block
        CALL    AI_SERVICE
; Re-display boar image using a bit block copy
; Call HBBC (bit block copy)
        MOV     AX,25           ; Code number for this service
        LEA     BX,HBBC_DATA    ; Pointer to data block
        CALL    AI_SERVICE
        .
        .
        .
CODE    ENDS
```

11.5.5 Character Fonts

XGA and 8514/A systems are furnished with disk-based character fonts that can be used in text display operations. Since the BIOS text functions do not operate on the XGA and 8514/A, the use of disk-based fonts is the simplest option for text display in the advanced function modes. In the loading of a disk-based font file the application is left to its own resources, since the AI provides no command to perform this operation. In addition to loading the font file into RAM, the application must also inform the AI of the font's address and select the desired character set. The following code fragment shows the necessary operations for loading a disk-resident font file into RAM, for initializing the necessary AI parameter blocks, and for selecting a character set for text and alphanumeric operations.

```
;****************************************************************
;                          data segment
;****************************************************************
DATA    SEGMENT
        .
        .
        .
;**********************|
; text operations data |
;**********************|
; Parameter block for HSCS (text select character set)
HSCS_DATA       DW      4       ; Length of data block
FONT_OFF        DW      0       ; Offset of loaded font
FONT_SEG        DW      0       ; Segment of loaded font
;
; Parameter block for ASFONT (alpha select character set)
ASFONT_DATA     DW      6       ; Length of data block
                DB      0       ; Font number
                DB      0       ; Reserved
AFONT_OFF       DW      0       ; Offset of loaded font
AFONT_SEG       DW      0       ; Segment of loaded font
;
;**********************|
;      fonts           |
;**********************|
; ASCIIZ filename for XGA 85-by-38 font
F1220_NAME      DB      'STAN1220.FNT',00H
FONT_HANDLE     DW      0       ; Handle for font file
;
;**********************|
;   storage for font   |
;**********************|
; Font header area
FONT_BUF        DW      0       ; Number of code pages
                DW      0       ; Default code page (0 to 4)
                DW      0       ; Alternate default (0 to 4)
                DD      0       ; 4-byte ID string ('437',0)
PAGE_1_OFFSET   DW      0       ; Offset of CSD within file
                DD      0       ; 4-byte ID string ('850',0)
                DW      0       ; Offset of CSD within file
                DD      0       ; 4-byte ID string ('860',0)
                DW      0       ; Offset of CSD within file
                DD      0       ; 4-byte ID string ('863',0)
                DW      0       ; Offset of CSD within file
                DD      0       ; 4-byte ID string ('865',0)
                DW      0       ; Offset of CSD within file
;
; Character set definition block for first code page
                DB      0       ; Reserved
                DB      0       ; Font type:
                                ; 0 = multiplane image
                                ; 3 = short vector font
                DB      0       ; Reserved
                DD      0       ; Reserved
CELL_WIDTH      DB      0       ; Pixel width of character cell
CELL_HEIGHT     DB      0       ; Pixel height of cell
                DB      0       ; Reserved
                DW      0       ; Cell size
CSD_FLAGS       DW      0       ; Flag bits:
                                ; Bit 14 ... 0 = single-plane
                                ;            1 = multiplane
                                ;      13 ... 0 = not prop. space
                                ;            1 = prop. space
IDX_TABLE_O     DW      0       ; Offset of index table
IDX_TABLE_S     DW      0       ; Segment of index table
                DW      0       ; Offset of envelope table
                DW      0       ; Segment of envelope table
                DB      0       ; Initial code point
                DB      0       ; Final code point
CSD_TABLE_O     DW      0       ; Offset of character definition
CSD_TABLE_S     DW      0       ; Segment of character definition
                DB      14250 DUP (00H)
        .
        .
        .
DATA    ENDS
;
;****************************************************************
;                          code segment
;****************************************************************
;
CODE    SEGMENT
        ASSUME  CS:CODE
        .
        .
        .
;**********************|
;    load font file    |
;**********************|
; Before using text display operations one of the four font files
; provided with the adapter must be loaded into RAM
        LEA     DX,F1220_NAME   ; Filename for XGA 12x20 font
        LEA     DI,FONT_BUF     ; Buffer for storing font
        CALL    XGA_FONT        ; Local procedure to load font
; Carry set if error during font load
        JNC     OK_XGA_FONT     ; Go if no error
;**********************|
;   font load error    |
;**********************|
; At this point the application must provide a handler to take
; care of the error that occurred during the font load operation
        .
        .
        .
;**********************|
; init parameter block |
;**********************|
; The AI is informed of the address of the loaded font by means
; of the HSCS (set character set) function
OK_XGA_FONT:
        PUSH    DS              ; DS to stack
        PUSH    DS              ; twice
        POP     FONT_SEG        ; Store in parameter block
; Alphanumeric display operations require a separate parameter
; block initialization
        POP     AFONT_SEG       ; For alphanumeric operations
; The offset of the font's character set definition block is
; located at byte 10 of the font header
        LEA     SI,FONT_BUF     ; Offset of font buffer
        MOV     BX,[SI+10]      ; Get offset of first code page
        ADD     BX,SI           ; Add offset to pointer
        MOV     FONT_OFF,BX     ; Store pointer in block
        MOV     AFONT_OFF,BX    ; For alphanumeric operations
;**********************|
; update font pointers |
;**********************|
; Update pointers in character set definition area by adding
; the load address of the font (in SI)
        ADD     IDX_TABLE_O,SI  ; Add to index table offset
        ADD     CSD_TABLE_O,SI  ; and to CSD table offset
; AX still holds the segment address. Store segment portion of
; address
        MOV     IDX_TABLE_S,AX  ; In index table
        MOV     CSD_TABLE_S,AX  ; In character set table
;**********************|
; select character set |
;**********************|
; Call HSCS (set character set) function
        MOV     AX,37           ; Code number for this service
        LEA     BX,HSCS_DATA    ; Pointer to data block
        CALL    AI_SERVICE
        .
        .
        .
;****************************************************************
;                           procedures
;****************************************************************
XGA_FONT        PROC    NEAR
; Read an XGA or 8514-a font file into RAM
; On entry:
;         DS:DX --> ASCIIZ filename for font file
;                   (must be in the current path)
;         DS:DI --> RAM buffer for font storage
;
; On exit:
;       Carry clear if font read and stored in buffer
;       Carry set if file not found or disk error
;
; Open font file using MS-DOS service
        PUSH    DI              ; Save entry pointer
        MOV     AH,61           ; DOS service request number
                                ; to open file (handle mode)
        MOV     AL,2            ; Read/write access
        INT     21H
        POP     DI              ; Restore pointer
; File opened?
        JNC     OK_XOPEN        ; Go if no error code
;**********************|
;    disk open error   |
;**********************|
; Open operation failed. Set carry flag and return to caller
        STC                     ; Signal error
        RET
;**********************|
;   read font into RAM |
;**********************|
OK_XOPEN:
        MOV     FONT_HANDLE,AX          ; Store file handle
NEW_128:
        MOV     BX,FONT_HANDLE
        LEA     DX,DATA_BUF             ; Buffer for data storage
        PUSH    DI              ; Save buffer pointer
; Use MS-DOS service to read 128 bytes
        PUSH    CX              ; Save entry CX
        MOV     AH,63           ; MS-DOS service request
        MOV     CX,128          ; Bytes to read
        INT     21H
        POP     CX              ; Restore
; 128 bytes read into buffer
        POP     DI              ; Restore buffer pointer
        CMP     AX,0            ; Test for end of file
        JNE     MOVE_128        ; Go if not at end of file
;**********************|
;    end of file       |
;**********************|
        MOV     BX,FONT_HANDLE  ; Handle for font file
; Close file using MS-DOS service
        MOV     AH,62           ; DOS service request
        INT     21H
        JMP     END_OF_READ
;**********************|
;   move sector to     |
;    font buffer       |
;**********************|
; At this point DATA_BUF holds 128 bytes from disk file
; DI --> storage position in the font's buffer
MOVE_128:
        MOV     CX,128          ; Byte counter
        LEA     SI,DATA_BUF     ; Pointer to data just read
PLACE_128:
        MOV     AL,[SI]         ; Byte from DATA_BUF
        MOV     [DI],AL         ; Into font's buffer
        INC     SI              ; Bump pointers
        INC     DI
        LOOP    PLACE_128       ; Continue until all sector read
; At this point the 128 bytes read from the disk file are stored
; in the font's buffer
        JMP     NEW_128
END_OF_READ:
        CLC
        RET
XGA_FONT        ENDP
        .
        .
        .
CODE    ENDS
```

11.5.6 Displaying Text

Once the preparatory operations described in Section 11.5.5 have been successfully executed, the application is able to use AI commands to display text characters and strings. Two types of text display services are available in the AI: string and alphanumeric commands.

Character String Operations

The character string commands are HCHST (character string at given position) and HCCHST (character string at current position). AI string display operations allow positioning the text characters at a screen pixel boundary. This offers a level of control that exceeds the one in BIOS text display services. The following code fragment shows the display of a character string using HCHST.

```
;****************************************************************
;                          data segment
;****************************************************************
DATA    SEGMENT
        .
        .
        .
HCHST_DATA_1    DW      59      ; Length of data block
PIXEL_COL       DW      150     ; Column address for start
PIXEL_ROW       DW      20      ; Row address for start
                DB      'XGA and 8514/A Adapter Interface'
                DB      ' bitBLT Operations Demo'
;
;**********************|
;     color data       |
;**********************|
; Parameter blocks for foreground and background colors
; Foreground color
HSCOL_DATA      DW      4       ; Length of data block
FORE_COL        DB      0F0H    ; 8-bit color code
                DB      0       ; Padding for double word
                DW      0
; Background color
HSBCOL_DATA     DW      4       ; Length of data block
BACK_COL        DB      11110000B       ; Bright red in 2-bit
                                        ; IRGB format
                DB      0       ; Padding for double word
                DW      0
        .
        .
        .
DATA    ENDS
;
;****************************************************************
;                          code segment
;****************************************************************
;
CODE    SEGMENT
        ASSUME  CS:CODE
        .
        .
        .
;**********************|
;   select colors      |
;**********************|
;
; AI string commands perform text display operations at the pixel
; level. First set foreground color to bright red
        MOV     FORE_COL,00001100B      ; Bright red
        MOV     AX,7            ; Code number for this service
        LEA     BX,HSCOL_DATA   ; Pointer to data block
        CALL    AI_SERVICE
; Now set the background color to dark blue
        MOV     BACK_COL,00000001B      ; Dark blue
        MOV     AX,10           ; Code number for this service
        LEA     BX,HSBCOL_DATA  ; Pointer to data block
        CALL    AI_SERVICE
```

```
;**********************|
;  display text string |
;**********************|
; Call HCHST (display character string at given position)
        MOV     AX,38           ; Code number for this service
        LEA     BX,HCHST_DATA_1         ; Pointer to data block
        CALL    AI_SERVICE
        .
        .
        .
CODE    ENDS
```

Alphanumeric Operations

Alphanumeric commands in the AI can be easily identified since their names start with the letter "A". In Section 11.5.5 we saw the use of the ASFONT (alpha select character set) to inform the adapter of the address of the character map resident in RAM and to select a character set. The other preparatory operations described in Section 11.5.5 must also be performed in order for an application to use the alphanumeric commands.

One difference between the string display commands and the alphanumeric commands is that the string commands allow positioning of the text characters at the screen pixel level while the alphanumeric commands use a screen grid of the size of the character cells. Table 11-11 shows the cell size of the different font files furnished with XGA and 8514/A systems.

Table 11-11

XGA and 8514/A Font Files and Text Resolution

FILE NAME	SCREEN SIZE	CHARACTER SIZE		ALPHA MODE GRID	
		WIDTH	HEIGHT	COLUMNS	ROWS
STAN1220.FNT	1024 by 768	12	20	85	38
STAN1223.FNT	1024 by 768	12	23	85	33
STAN0814.FNT	640 by 480	8	14	80	34
	1024 by 768	8	14	128	54
STAN0715.FNT	1024 by 768	7	15	146	52

On the other hand, the AI alphanumeric commands allow the attributes of each character to be individually controlled. In addition, alphanumeric commands provide the control and display of a cursor character. Since the blinking attribute is not available in XGA and 8514/A systems, this alphanumeric cursor is nothing more than a static graphics symbol, which must be handled by the application. The grid for cursor operations is also determined by the character size.

There are two alphanumeric display commands in the AI. The command named ABLOCKMFI (write character block in mainframe interactive mode) is designed to simulate character display in a mainframe environment. The command ABLOCKCGF (write character block in CGA mode) is designed to simulate the display controls in the IBM Color Graphics Adapter. The following code fragment shows the use of alphanumeric commands in cursor and text display operations.

```
;****************************************************************
;                          data segment
;****************************************************************
DATA    SEGMENT
        .
        .
        .
; ASCUR (set cursor shape)
ASCUR_DATA      DW      3       ; Length of data block
                DB      16      ; Cursor start line
                DB      19      ; Cursor stop line
CUR_SHAPE       DB      00      ; Cursor attribute:
                                ; 00 = normal
                                ; 01 = hidden
                                ; 02 = left arrow
                                ; 03 = right arrow
; ACURSOR (set cursor position)
ACURSOR_DATA    DW      2       ; Length of data block
CUR_COLUMN      DB      0       ; Cursor column
CUR_ROW         DB      0       ; Cursor row
;
; ASFONT (select character set)
ASFONT_DATA     DW      6       ; Length of data block
                DB      0       ; Font number
                DB      0       ; Reserved
AFONT_OFF       DW      0       ; Offset of loaded font
AFONT_SEG       DW      0       ; Segment of loaded font
;
; ABLOCKCGA (writes a block of characters in CGA emulation mode)
ABLOCKCGA_DATA  DW      10      ; Length of data block
COL_START       DB      0       ; Start column for display
ROW_START       DB      0       ; Start row for display
CHAR_WIDE       DB      0       ; Width of block (characters)
CHAR_HIGH       DB      0       ; Height of block (characters)
STRING_OFF      DW      0       ; Offset of string address
STRING_SEG      DW      0       ; Segment of string address
BUF_WIDE        DB      85      ; Characters per row displayed
ATTRIBUTE       DB      0       ; 7 6 5 4 3 2 1 0 <= BITS
                                ; | | | | | | | |
                                ; | | | | | | |_|__ font (0 to 3)
                                ; | | | | |_|______ reserved
                                ; | | | |________ 1 = transparent
                                ; | | |           0 = opaque
                                ; | | |__________ overstrike
                                ; | |____________ reverse video
                                ; |______________ underscore
;
;                            __________ background color
;                            |    _____ foreground color
;                            |   |
; String for ABLOCKCGA  C  |--||--|
STRING_1        DB      'T',00001001B
                DB      'h',00001001B
                DB      'i',00001001B
                DB      's',00001001B
                DB      ' ',00001001B
                DB      'i',00001100B
                DB      's',00001100B
                DB      ' ',00001100B
                DB      'a',00001010B
                DB      ' ',00001010B
                DB      't',00011100B
                DB      'e',00011100B
                DB      's',00011100B
                DB      't',00011100B
        .
        .
        .
DATA    ENDS
;
;****************************************************************
;                          code segment
;****************************************************************
;
CODE    SEGMENT
        ASSUME  CS:CODE
        .
        .
        .
;**********************|
;  alphanumeric text   |
;**********************|
; AI commands that start with the prefix letter A are used to
; perform alphanumeric operations at the character cell level.
; The alphanumeric commands allow controlling the attribute of
; each individual character displayed.
;**********************|
;   cursor operations  |
;**********************|
; Display cursor
        MOV     CUR_COLUMN,30   ; Column number
        MOV     CUR_ROW,4       ; Row number
;
; Call ASCUR (set cursor shape)
        MOV     AX,45           ; Code number for this service
        LEA     BX,ASCUR_DATA   ; Pointer to data block
        CALL    AI_SERVICE
; CAll ACURSOR (set cursor position)
        MOV     AX,44           ; Code number for this service
        LEA     BX,ACURSOR_DATA ; Pointer to data block
        CALL    AI_SERVICE
; Call ASFONT (select font)
; Code assumes that the address of the RAM-resident font has
; been previously set in the parameter block
        MOV     AX,46           ; Code number for this service
        LEA     BX,ASFONT_DATA  ; Pointer to data block
        CALL    AI_SERVICE
; Display text message using ABLOCKCGA function
; Set display parameters in control block variables
        MOV     COL_START,20    ; Start at column 20
        MOV     ROW_START,30    ; and at row number 30
        MOV     CHAR_WIDE,14    ; Characters wide
        MOV     CHAR_HIGH,1     ; Characters high
        PUSH    DS              ; Data segment
        POP     STRING_SEG      ; Store in variable
        LEA     SI,STRING_1     ; Offset of text string
        MOV     STRING_OFF,SI   ; Store offset of string
; Call ABLOCKCGA
        MOV     AX,41           ; Code number for this service
        LEA     BX,ABLOCKCGA_DATA       ; Pointer to data block
```

```
          CALL    AI_SERVICE
          .
          .
          .
CODE      ENDS
```

Chapter 12

XGA Hardware Programming

Topics:

- The XGA hardware
- XGA features and architecture
- Initializing the XGA system
- Processor access to XGA video memory
- Programming the XGA graphics coprocessor
- The XGA sprite
- Using the book's XGA library

This chapter describes the XGA architecture and its programmable hardware components.and iIllustrates XGA programming by manipulating the video hardware directly and by accessing video memory. It describes the XGA graphics coprocessor, its capabilities, initialization, and programming, and also the XGA sprite, its hardware elements, and the programming of sprite operations. The chapter concludes with a listing of the procedures in the GRAPHSOL library furnished with the book.

12.1 XGA Hardware Programming

Chapter 11 discusses the XGA Adapter Interface software and how it can be used in programming 8514/A and XGA systems. However, the AI has some limitations. At the system level the use of AI services would almost certainly be discarded for reasons of code autonomy. The applications programmer can also find objections to using the AI, particularly its limited services and its performance penalty. In summary, one or more of the following reasons will often determine that the programmer uses direct access to the XGA hardware:

1. The process of loading and initializing the Adapter Interface cannot be conveniently performed at the program's level.
2. The services provided by the Adapter Interface are insufficient for the program's purpose.

3. The performance of the adapter interface services do not meet the requirements of the code.

In the case of system programs, device drivers, and other low-level graphics software, the decision will often be to not use the AI at all, especially if objection number one, listed above, is applicable. Then the programmer would take control of the XGA hardware and proceed with the XGA device as described in Chapters 2 to 5 regarding the VGA system. Although, even when assuming control over the hardware, it is possible that the software developers could benefit from using the character fonts furnished with the AI.

On the other hand, most graphics applications could be developed either by using AI services exclusively or in a mixed environment in which the code complements the AI services with direct hardware programming. For example, an application could be designed to use the AI services when their control and performance is at an acceptable level. In this manner, the AI commands described in Chapter 6 can be useful and convenient in initializing the XGA, setting the color palette, loading font files into RAM, display-ing text messages, clearing the screen, and closing the adapter. All of the above are functions in which performance is often not an important issue. At the same time, the application may assume direct control of the XGA hardware in setting individual pixels, drawing lines and geometrical figures, performing bitBlt operations to and from video memory, manipulating graphics markers, and other functions in which control or performance factors are important.

12.1.1 XGA Programming Levels

Regarding the XGA and system hardware the graphics programmer can operate at four different levels. The first and highest level is the graphics functions offered by operating systems and graphics environments. Such is the case in applications that execute under the Windows and OS/2 operating systems and use the graphics services provided by the system software. The second level of XGA programming is by means of the AI services discussed in Chapter 6. The third level is by programming the XGA registers and the graphics coprocessor. The fourth and lowest level of XGA graphics programming is by accessing video memory directly. Graphics programming in high-level environments such as the Windows and OS/2 operating systems are outside of the subject matter of this book. XGA programming by means of AI services was discussed in Chapter 6. The present chapter is devoted to programming the XGA graphics coprocessor and accessing XGA video memory directly.

These same four levels of programming are possible in 8514/A systems. Since the 8514/A is no longer state-of-the-art we have not included its low-level programming. Readers interested in programming the 8514/A at the register level should consult *Graphics Programming for the 8514/A* by Jake Richter and Bud Smith (see Bibliography), as well as the 8514/A documentation available from IBM.

12.2 XGA Features and Architecture

Figure 11.2 shows the elements of the XGA system. The XGA is furnished as an optional adapter card for microchannel computers equipped with the 80386, 80386SX, or 486 processor. The XGA system is integrated in the motherboard of the Model 90 XP

486. Sections 6.0 and 6.1 (Chapter 6) describe the evolution of the XGA from the 8514/A adapter, its comparative features as well as its presentation. To the programmer the XGA system presents the following interesting features:

1. It includes all VGA modes and is compatible with VGA at the register level. That is, software developed for VGA can be expected to run satisfactorily in XGA. One exception is programs that make use of the VGA video space for other purposes. For example, a popular VGA enhancement for the Ventura Publisher typesetting program, called Soft Kicker, will not operate in the VGA modes of an XGA system.

2. XGA includes a 132-column text mode that represents a substantial enhancement to the 80-column text modes of the VGA. This mode requires an XGA system equipped with the appropriate video display. At this time no BIOS support is provided for the 132-column mode or for XGA graphics operations.

3. The XGA Extended Graphics modes, or enhanced modes, provide a maximum resolution of 1,024 by 768 pixels in 256 colors, which can be selected from a palette of 256K colors. The enhanced modes also provide a 64-by-64 pixels hardware-controlled graphics object, whose shape is defined by the application. This graphics object, called the sprite, is usually animated by mouse movements and used to non-destructively overlay a displayed image. The XGA graphics modes support systems with multiple video displays.

4. The XGA direct color mode, also called the palette bypass mode, is capable of displaying 65,536 colors on a 640-by-480 pixel grid. In this mode the pixel color is encoded in a 16-bit value that is used to set the red, blue, and green electron guns without intervention of the LUT registers.

12.2.1 The XGA Graphics Coprocessor

One characteristic of XGA hardware that differentiates it from VGA and SuperVGA systems is the presence of a graphics coprocessor chip. Much of the enhanced performance of the XGA system is due to this device. The following are the most important features of the graphics coprocessor:

1. The coprocessor can obtain control of the system bus in order to access video and system memory independently of the central processor. This bus-mastering feature allows the coprocessor to perform graphics operations while the main processor is executing other functions.

2. The graphics coprocessor can directly perform drawing operations. These include straight lines, filled rectangles, and bit block transfers.

3. The coprocessor provides support for saving its own register contents. This feature is useful in a multitasking environment.

4. The coprocessor supports several logical and arithmetic mixes including OR, AND, XOR, NOT, source, destination, add, subtract, average, maximum, and minimum operands.

5. The coprocessor can manipulate images encoded in 1, 2, 4, or 8 bits per pixel formats. Pixel maps can be defined as coded in Intel or Motorola data storage formats.

6. The coprocessor can be programmed to generate system inter-rupts. These interrupts can occur when the coprocessor operation has completed, an access to the

coprocessor was rejected, a sprite operation completed, or at the end or start of the screen blanking cycle.

The coprocessor registers are memory-mapped. To an application, programming the coprocessor consists of reading and storing data into these reserved memory addresses. In contrast, the XGA main registers are port-mapped and programming consists of reading and writing to these dedicated ports.

The execution of a coprocessor operation consists of the following steps:

1. The system microprocessor reads and writes data to coprocessor registers that must be initialized for the operations.
2. The coprocessor operation starts when a command is written to its Pixel Operations register.
3. The coprocessor executes the programmed operation. During this time the system microprocessor can be performing other tasks. The only possible interference between processor and coprocessor is when both are accessing the bus simultaneously. In this case the access takes place according to the established priorities.
4. At the conclusion of the programmed operation the graphics coprocessor informs the system and becomes idle.

12.2.2 VRAM Memory

Since the XGA is a memory-mapped system the color code for each screen pixel is encoded in video RAM. How many units of memory are used to encoded the pixel's color depends on the adopted format. Possible values are of 1, 2, 4, 8, and 16 bits per pixel. The number of colors are respective powers of 2, as shown in Table 12-1.

Table 12-1

Pixel to Memory Mapping in XGA Systems

BITS-PER-PIXEL	POWER OF 2	NUMBER OF COLORS
1	2^1	2
2	2^2	4
4	2^4	16
8	2^8	256
16	2^{16}	65536

Notice that the 256 and 65,536 color modes are available only in XGA systems with maximum on-board RAM (1Mb). The total amount of VRAM required depends on the number of screen pixels and the number of encoded colors. For example, to store the contents of the entire XGA screen at 1,024-by-768 pixels resolution requires a total of 786,432 memory units. In the 8-bits per pixel format the number of memory units is of 786,432 bytes (8 bits per byte). However, this same screen can be stored in 98,304 bytes if each screen pixel is represented in a single memory bit (786,432 / 8 = 98,304).

Therefore the video memory space of an XGA system in 1,024-by-768 pixel mode, with each pixel encoded in 256 colors, exceeds by far the limit of an 80x86 segment register (65,536). Therefore an application accessing video memory directly while executing in 80x86 real mode requires some sort of memory banking mechanism by which to access a total of 768,432 bytes of VRAM memory. In fact, a minimum of 12 memory banks of 65,536 bytes are required to encode the 768,432 XGA pixels in 1,024-by-768 pixel mode in 256 colors. This banking mechanism is discussed in detail later in Section 12.3

Video Memory Apertures

In general, an XGA system can access video memory by means of three different apertures, described as follows:

1. The largest memory aperture is of a 22-bit space. This range of 4Mb allows addressing four times the maximum VRAM that can be present in an XGA system. The 4Mb address space must be represented in an 80386 or 486 extended register. This is the aperture used by the XGA graphics coprocessor.
2. The second possible aperture into video memory is of 1Mb. Since this is also the maximum VRAM that can be present in an XGA system, the 1Mb aperture allows addressing all video memory consecutively by means of an 80386 or 486 extended register.
3. The third possible aperture is of 16 banks of 64K each. This aperture, which is the only one possible in the MS-DOS environment, requires bank switching to access the maximum VRAM.

Notice that in a particular display mode not all 16 banks are required to access the mapped video memory space.

Data Ordering Schemes

XGA memory mapping can be according to the Intel or the Motorola storage conventions. The XGA hardware allows selecting the Intel or Motorola formats for every operation that accesses a pixel map or image stored in system or video memory. In the Intel conven-tion, also known as the little-endian addressing scheme, the smallest element (little end) of a number is stored at the lowest numbered memory location. In the Motorola convention, known as big-endian addressing, the largest element (big end) is stored at the lowest numbered memory location. Table 12-2 shows the results of storing bytes, words, and doublewords according to the Intel and the Motorola conventions.

Table 12-2

Data Storage According to the Intel and Motorola Conventions

DATA	STORAGE UNIT	INTEL	MOTOROLA
00 11 AA FF	byte	00|11|AA|FF	00|11|AA|FF
00 11 AA FF	word	11 00|FF AA	00 11|AA FF
00 11 AA FF	doubleword	FF AA 11 00	00 11 AA FF
		low => high	low => high

Notice that since the unit of memory storage in IBM microcomputers is 1 byte, the Intel and Motorola storage schemes are identical in byte-ordered data. Also that the value of bits within the stored byte is in the conventional format, that is, the low order bit (bit number 0) is located at the rightmost position.

12.2.3 The XGA Display Controller

Another programmable device of the XGA system is the Display Controller chip. This IC contains the color look-up table, the CRT Controller, and the hardware registers for the operation of a special cursor, called the sprite (see Section 12.5). The XGA display controller registers are a superset of the VGA registers. As in the VGA, these registers are mapped into the systems I/O space. Therefore they appear to the programmer as input and output ports.

The base address of the XGA display controller is port 21x0H. The variable x in the port number depends on the instance of the XGA adapter. Recall that more than one XGA system can co-exist in a microcomputer. The instance is the number that corresponds to a particular XGA adapter or motherboard implementation. The user can change the instance number of an installed XGA adapter by means of the setup procedures provided by the reference diskette. The default instance value for a single XGA adapter card is 6, which determines a base address for the Display Controller of 2160H. Notice that the instance number replaces the variable x in the general formula.

The programmable registers in the XGA Display Controller are in the range 21x0H to 21xFH. Here again the variable x represents the instance number. Table 12-3 shows some of the Display Controller registers and the values to which they must be initialized during mode setting.

The Display Controller registers are divided into two groups: direct access and indexed access registers. The direct access registers are the ten registers in the range 21x0H to 21x9H. The indexed access registers are related to the Index register (port 21xAH) and the data registers (ports 21xBH to 21xFH). The index values are in the range 04H to 70H but not all values in this range are actually used in XGA. The direct access registers in the Display Controller are programmed by means of IN or OUT instructions to the corresponding port; for example, the Memory Access Mode register, at 21x9H, can be programmed for 8 bits per pixel and Intel data format as follows:

```
; Programming a direct access register of the XGA Display
; Controller group
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,9            ; Add offset of Memory Access
                                ; Mode register
        MOV     AL,00000011B    ; Bitmap for Intel format
                                ; and 8 bits per pixel
        OUT     DX,AL
```

The above code fragment assumes that the base address of the Display Controller register groups has been previously determined and is stored in the variable XGA_REG_BASE. The operations necessary for determining this base address are shown in Section 12.2.

Programming the indexed access registers takes place in two steps: first, the desired register is selected by writing a value to the Index register at port 21xAH; second, data is read or written to the register by means of the data registers in the range 21xBH to 21xFH. The following fragment shows writing all one bits (FFH) to the Palette Mask register at offset 64H of the Index register.

```
; Programming an indexed access register of the XGA Display
; Controller group
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; Add offset of Index register
        MOV     AL,64H          ; Select Palette Mask register
                                ; at offset 64H
        MOV     AH,0FFH         ; Data byte to write
        OUT     DX,AX           ; Select and write data
```

Notice that the 80x86 instruction OUT DX,AX writes the value in AL to the port number in DX and the value in AH to the port number in DX + 1. The result is that by using this form of the OUT instructions we can select and access the register with a single operation.

The following Display Controller registers are particularly interesting to the programmer:

1. The Interrupt Enable register (located at base address plus 4) is used to unmask the interrupt or interrupt sources that will be used by the software.
2. The Operating Mode register (located at the base address) is usually set to extended graphics mode.
3. The Aperture Control register (located at base address plus 1) allows enabling the 64K memory aperture mentioned in Section 12.1.2. as well as selecting the start address of video memory either at A0000H or at B0000H. Most applications executing under MS-DOS use A0000H, the VGA start address for dot addressable graphics.
4. The Memory Access Mode register (located at base address plus 9) allows selecting the number of bits per pixel and the Intel or Motorola data format.

12.3 Initializing the XGA System

The first XGA programming operation usually consists of initializing and enabling the video system. The simplest initialization method is by means of the AI services described in Chapter 6. An application that is to access the XGA exclusively by means of AI services need do nothing more than use the HOPEN and HINIT functions to initialize the system. However, programs that access the XGA directly must often perform additional initialization operations. Two possibilities can be considered:

1. Programs can use the AI HINIT and HOPEN services and, in addition, perform other initialization operations so as to enable the use of AI services and direct access to XGA hardware simultaneously.
2. A program can rely entirely on its own hardware initialization routines, and not use the AI HINIT and HOPEN functions.

Which method is adopted depends on the program's characteristics. If the software is to use both, AI services and direct access methods, then the HINIT and

HOPEN functions are necessary. On the other hand, programs that do not use AI services can perform the necessary hardware initialization operations. Notice that the AI is a software black box which manipulates registers and video memory in ways that are not visible to the application. This creates additional problems for programs that mix AI services and direct access methods.

The following discussion relates to direct initialization of the XGA system. The use of the AI HINIT and HOPEN was explained in Chapter 6.

12.3.1 Locating the XGA Hardware

The first initialization task consists of locating the XGA components in the system's space. The necessary information is found in the PS/2 Programmable Option Select (POS) registers. Figure 12-1 shows important POS data related to the XGA hardware.

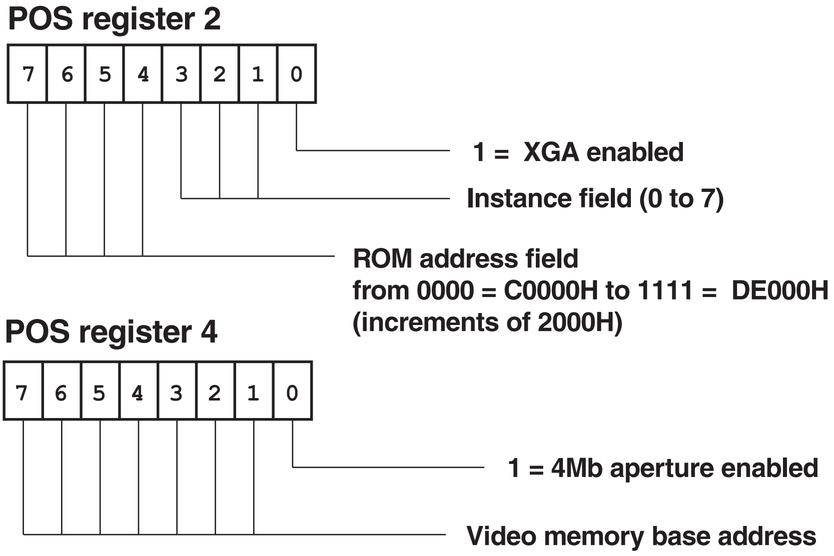


Figure 12-1 *XGA Data in POS Registers*

The first step in reading the POS registers is determining where these registers are located. BIOS service number 196, sub-service number 0, of INT 15H, returns the POS registers base address in the DX register. The following code fragment shows the required processing.

```
;**********************|
;   get POS address    |
;**********************|
; Use service number 196, INT 15H, with AL = 0 to determine base
; address of Programmable Option Select (POS) registers
        MOV     AX,0C400H       ; AH = C4H (service request)
                                ; AL = 0 (sub-service)
        INT     15H             ; BIOS interrupt
                                ; for microchannel machines only
        JNC     VALID_POS       ; Go if POS address returned
        JMP     NO_XGA          ; Error - not microchannel
```

```
VALID_POS:
        MOV     XGA_POS,DX   ; Save base address of POS
; An XGA system can be located on the motherboard or in one
; of 9 possible slots. Initialize CX = 0 for motherboard XGA
; CX = 1 to 9 for XGA in adapter card
        XOR     CX,CX           ; Start with motherboard
        CLI                     ; Interrupts off
        .
        .
        .
```

Not all POS values encode XGA data. The valid range for XGA systems is 8FD8H to 8FDBH. Service number 196, sub-service number 1, of INT 15H can be used to enable each one of 9 possible slots for setup. Then the value stored at the POS register base is read and compared to the valid range. If the value is within the range an XGA adapter or motherboard implementation has been detected. In this case the POS registers contain data required for the initialization of the XGA system. The following code fragment illustrates the required processing.

```
 ; Use BIOS service 196, sub-service number 1, to enable slot
; for setup
GET_POS_0:
        MOV     AH,0C4H         ; BIOS service
        MOV     AL,01H          ; Sub-service number
        MOV     BX,CX           ; Slot number to BX
        INT     15H
; Slot enabled for setup
        MOV     DX,XGA_POS      ; POS register 0 and 1
        IN      AX,DX           ; Read ID low and high bytes
; Valid range for XGA systems is 8FD8H to 8FDBH
        CMP    AX,08FD8H        ; Test low limit
        JAE    TEST_HIGH_LIM    ; Go if equal or greater
; At this point the POS reports that system is not an XGA
; adapter
NOT_XGA_POS:
        INC     CX              ; CX is options counter
        CMP     CX,9            ; Done all slots?
        JB      GET_POS_0       ; Go if not at last slot
        JMP     NO_XGA          ; No XGA exit
TEST_HIGH_LIM:
        CMP     AX,08FDBH       ; Test high limit of range
        JA      NOT_XGA_POS     ; Go if out of range
;**********************|
;     XGA found        |
;**********************|
        CLI                      ; Disable interrupts
; Test if XGA is in motherboard
        CMP    CX,0              ; 0 is motherboard value
        JNE    XGA_CARD          ; Go if not on the motherboard
;**********************|
;  motherboard XGA     |
;**********************|
; Port 94H is used to enable and disable motherboard video
        MOV    AL,0DFH          ; Bit 5 = 0 for video setup
        MOV    DX,94H           ; 94H is system board enable
        OUT    DX,AL
        JMP    SHORT GET_POS    ; Skip slot setup
;**********************|
;       XGA card       |
;*********************|
XGA_CARD:
        MOV    AX,0C401H        ; Place adapter in setup mode
        MOV    BX,CX            ; Slot number to BL
        INT    15H
;*********************|
;  save POS registers  |
;*********************|
GET_POS:
        MOV     DX,XGA_POS      ; Get POS record for the slot id
        ADD     DX,2            ; POS register at offset 2
        IN      AL,DX           ; Read data byte
        MOV     POS_2,AL        ; and store it
        INC     DX              ; Next POS register
        INC     DX              ; is number 4
        IN      AL,DX           ; Get contents
        MOV     POS_4,AL        ; Store it
; At this point POS registers 2 and 4 have been saved in
; variables
;*********************|
;   re-enable video    |
;*********************|
; Test for XGA in motherboard
        CMP    CX,0             ; Treat the motherboard
                                ; differently
        JNE    XGA_ADAPTER      ; Go if not in motherboard
; XGA in motherboard. Set bit 5 in port 94H to re-enable video
        MOV    AL,0FFH           ; All bits set
        OUT    094H,AL
        JMP    SHORT REG_BASE
XGA_ADAPTER:
        MOV    AX,0C402H         ; Enable the slot for normal
        MOV    BX,CX             ; operation
        INT    15H
        .
        .
        .
```

The next step in the XGA initialization is calculating the XGA Display Controller register base by adding the instance value to the template 21x0H mentioned in Section 12.1.3. The following code fragment shows the necessary manipulation of the instance bits.

```
;*********************|
; calculate and store  |
; XGA register base    |
;*********************|
REG_BASE:
        STI                     ; Interrupts on again
        MOV    AL,POS_2         ; Get value at POS register 2
        AND    AX,0EH           ; Mask out all bits except
                                ; instance
        SHL    AX,1             ; Multiply instance by 8
        SHL    AX,1             ; to move to second digit
        SHL    AX,1             ; position
        ADD    AX,2100H         ; Add instance to base address
        MOV    XGA_REG_BASE,AX  ; Store result in variable
```

12.3.2 Setting the XGA Mode

Once the XGA Display Controller register base has been established the initialization usually proceeds to set the XGA hard-ware in a pre-established display mode. Although the XGA display modes are unofficial, Table 12-3 shows the ones mentioned in IBM's documentation.

Table 12-3

XGA Modes

MODE NUMBER	TYPE	HORIZONTAL PIXELS	VERTICAL PIXELS	COLORS
1	132-column text			
2	graphics	1024	768	256
3	graphics	1024	768	16
4	graphics	640	480	256
5	direct color	640	480	65536

The fundamental mode setting operation consists of loading most of the Display Controller registers with pre-established values. These values are listed in the XGA Video Subsystem section of the IBM Technical Reference Manual for Options and Adapters, document number 504G-3287-000. This document can be obtained from IBM Literature Department. Table 12-4 lists the Display Controller registers that must be initialized during mode setting.

Table 12-4

XGA Display Controller Register Initialization Settings

		2	3	4	5	<= MODE
		1024	1024	640	640	<= rows
ADDRESS/		768	768	480	480	<= columns
INDEX	REGISTER NAME	256	16	256	65536	<= colors
21x4	Interrupt Enable	00H	00H	00H	00H	All interrupts OFF
21x5	Interrupt Status	8FH	8FH	8FH	8FH	Reset interrupts
21x0	Operating Mode	04H	04H	04H	04H	Graphics modes
21xA	Index Register					
64	Palette mask	00H	00H	00H	00H	Blank display
21x1	Aperture Control	01H	01H	01H	01H	64K at A0000H
21x8	Aperture Index	00H	00H	00H	00H	|
21x6	Video Mem. Ctrl.	00H	00H	00H	00H	|----- Initial values
21x9	Memory Access Mode	03H	02H	03H	04H	|
21xA	Index Register					
50	Display mode 1	01H	01H	01H	01H	Prepare for reset
50	Display mode 1	00H	00H	00H	00H	Reset CRT
10	x total low	9DH	9DH	63H	63H	|------ initial values
11	x total high	00H	00H	00H	00H	|
12	x display end low	7FH	7FH	4FH	4FH	|
13	x display end high	00H	00H	00H	00H	|
14	x blank start low	7FH	7FH	4FH	4FH	|
15	x blank start high	00H	00H	00H	00H	|
16	x blank start low	9DH	9DH	63H	63H	|
17	x blank end high	00H	00H	00H	00H	|

(continued)

Table 12-4

XGA Display Controller Register Initialization Settings **(continued)**

		2	3	4	5	<= MODE
		1024	1024	640	640	<= rows
ADDRESS/		768	768	480	480	<= columns
INDEX	REGISTER NAME	256	16	256	65536	<= colors
21xA	Index Register					| ------ initial values
18	x sync start low	87H	87H	55H	55H	|
19	x sync start high	00H	00H	00H	00H	|
1A	x sync end low	9CH	9CH	61H	61H	|
1B	x sync end high	00H	00H	00H	00H	|
1C	x sync position	40H	40H	00H	00H	|
1E	x sync position	04H	04H	00H	00H	|
20	y total low	30H	30H	0CH	0CH	|
21	y total high	03H	03H	02H	02H	|
22	y display end low	FFH	FFH	DFH	DFH	|
23	y display end high	02H	02H	01H	1H	|
24	y blank start low	FFH	FFH	DFH	DFH	|
25	y blank start high	02H	02H	01H	01H	|
26	y blank start low	30H	30H	0CH	0CH	|
27	y blank end high	03H	03H	02H	02H	|
28	y sync start low	00H	00H	EAH	EAH	|
29	y sync start high	03H	03H	01H	01H	|
2A	y sync end	08H	08H	ECH	ECH	|
2C	y line comp low	FFH	FFH	FFH	FFH	|
2D	y line comp high	FFH	FFH	FFH	FFH	|
36	Sprite control	00H	00H	00H	00H	|
40	Start address low	00H	00H	00H	00H	|
41	Start address med	00H	00H	00H	00H	|
42	Start address high	00H	00H	00H	00H	|
43	Buffer pitch low	80H	40H	50H	A0H	|
44	Buffer pitch high	00H	00H	00H	00H	|
54	Clock select 1	0DH	0DH	00H	00H	|
51	Display mode 2	03H	02H	03H	04H	|
70	Clock select 2	00H	00H	00H	00H	|
50	Display mode 1	0FH	0FH	C7H	C7H	|
At this point XGA palette registers must be loaded and memory must be cleared						
55	Border color	00H	00H	00H	00H	|
60	Sprite/Pal low	00H	00H	00H	00H	|
61	Sprite/Pal high	00H	00H	00H	00H	|
62	Sprite pre low	00H	00H	00H	00H	|
63	Sprite pre high	00H	00H	00H	00H	|
64	Palette mask	FFH	FFH	FFH	FFH	Make visible

The registers in Table 12-4 are listed in the order in which they must be set. Notice that before the last group of registers are set, the initialization routine must load the XGA palette and clear all video memory. Failure to do this last operation could result in the display of random data at the conclusion of the mode setting operation. The actual coding can be based on data stored in two arrays: one holds the values for the first group of Display Controller registers and the second one for the group of registers to be initialized after the palette is loaded and the screen cleared. The following fragment demonstrates the necessary manipulations.

```
DATA      SEGMENT

; Mode number ----|
; 640x480x65536   5 --------------------|
; 640x480x256     4 ---------------|    |
; 1024x768x16     3 ----------|    |    |
; 1024x768x256    2 -----|    |    |    |
; Index ------------|    |    |    |    |
; Register ----|    |    |    |    |    |
;             _|__ _|__ _|__ _|__ _|__ _|__
XGA_V1  DB    004H,000H,000H,000H,000H,000H ; Interrupt enable
        DB    005H,000H,08FH,08FH,08FH,08FH ; Interrupt status
        DB    000H,000H,004H,004H,004H,004H ; Operating mode
        .
        .     (missing values as in Table 12-4)
        .
        DB    00AH,050H,00FH,00FH,0C7H,0C7H ; Display mode 1
        DB    0FFH,0FFH,0FFH,0FFH,0FFH,0FFH ; End of the list
;
XGA_V2  DB    00AH,055H,000H,000H,000H,000H ; Border color
        .
        .     (missing values as in Table 12-4)
        .
        DB    00AH,064H,0FFH,0FFH,0FFH,0FFH ; Palette mask
        DB    0FFH,0FFH,0FFH,0FFH,0FFH,0FFH ; End of the list
;
 Variables used by the XGA_MODE procedure
MODE          DW    0           ; Mode number
; Previously initialized base address of the XGA Display
; Controller (see Section 12.2.1)
XGA_REG_BASE  DW    0           ; Address variable
;
DATA    ENDS

CODE    SEGMENT
        .
        .
        .
XGA__MODE    PROC     NEAR
; Procedure to initialize an XGA graphics mode by setting the
; video system registers directly
; On entry:
;         AL = mode number (valid range is 2 to 5)
; On exit:
;        carry clear if no error
;
        MOV     AH,0            ; Clear high byte
        MOV     MODE,AX         ; Mode to variable
        CMP     MODE,6          ; Mode number out of range?
        JB      TEST_MODE1      ; Go if less than 9
        JMP     BAD_MODE        ; illegal entry value for mode
; Mode 0 = VGA BIOS mode number 3
; Mode 1 = 132 column VGA text mode
; These modes are not valid
TEST_MODE1:
        CMP     MODE,1          ; 80-column VGA text mode?
        JA      VALID_MODE      ; Go if range is > 1
        JMP     BAD_MODE        ; Error exit for invalid mode
;**********************|
;  initialize first    |
;   register group     |
;*********************|
VALID_MODE:
; The table at XGA_V1 contains the values to be sent to the
; XGA register in order to initialize the corresponding mode
        LEA     SI,XGA_V1       ; Point to start of values table
        MOV     BX,MODE         ; Use mode as an offset
        CALL    INIT_REG_BLK    ; Local init procedure
;*********************|
;    init palette      |
;*********************|
; Palette initialization at this point
; Notice that this routine must be mode-specific
        .
        .
        .
;*********************|
; clear video memory   |
;*********************|
; Video memory cleared at this point
; Notice that this routine must be mode-specific
        .
        .
        .
;*********************|
;  initialize second   |
;   register group     |
;*********************|
; The table at XGA_V2 contains the values to be sent to the
; XGA register in order to initialize the second group of XGA
; registers
        LEA     SI,XGA_V2       ; Point to start of values table
        MOV     BX,MODE         ; Use mode as an offset
        CALL    INIT_REG_BLK    ; Local init procedure
;
        MOV    XGA_CURBK,-1      ; Reset the bank counter
        MOV    AX,MODE           ; Remember the mode we're in
        MOV    XGA_CUR_MODE,AX
        MOV    AX,1              ; Return ok
        RET
BAD_MODE:
        MOV    AX,0              ; Return failure
        RET
XGA_MODE    ENDP
;
INIT_REG_BLK    PROC    NEAR
; Auxiliary procedure for XGA_SET_MODE
; Initialize block of XGA register until FFH is found
; On entry:
;         SI --> formatted register data
;         BX = display mode
; The value at offset 0 of XGA_V1 is the register number
; The value at offset 1 is the index register number if the
; register is 0AH. The remaining entries is register data for
; each mode
REG_DATA:
        MOV     DX,XGA_REG_BASE ; XGA register base
        MOV     AH,0            ; High byte of offset is 0
        MOV     AL,[SI]         ; Low byte of offset
; Register value 0FFH marks the end of the table
```

```
        CMP     AL,0FFH          ; End of the table?
        JE      END_OF_BLOCK     ; End of register setup
        ADD     DX,AX            ; Add register offset to base
        CMP     AL,0AH           ; Test for an index register
        JE      INDEXED          ; Go if index register
; At this point register is not at offset 0AH, therefore data
; is output directly
        MOV     AL,[SI+BX]       ; Get data value from table
        OUT     DX,AL            ; and send to port
        JMP     SHORT NEXT_REG   ; Continue
INDEXED:
        MOV     AL,[SI+1]        ; Get index register number
        MOV     AH,[SI+BX]       ; Get data byte from table
        OUT     DX,AX            ; Output data to index register
NEXT_REG:
        ADD    SI,6              ; Index to next register in table
        JMP    REG_DATA
END_OF_BLOCK:
        RET
INIT_REGT_BLK    ENDP
```

An XGA initialization routine can be found in the procedure named INIT_XGA contained in the XGA2 module of the GRAPHSOL library included in the book's software. Because of the complexities in the design of mode-specific palette initialization and screen clearing routines for all XGA modes, the INIT_XGA procedure does not perform these operations.

12.3.3 Loading the XGA Palette

Color display in XGA systems is by means of a Color Look-up Table (LUT), a Digital-to-Analog converter (DAC) and associated hardware. The actual structure is reminiscent, although not identical, of the one used in VGA systems. The XGA palette was described in Section 6.1.4. Bit plane mapping for a 256-color mode can be seen in Figure 6.3. The XGA color palette registers can be set by means of the HLDPAL AI service described in Section 6.4.2. In addition, a program can assume control of the XGA palette hardware and set its values directly.

We saw that XGA palette data consists of red, blue, and green values that are stored in corresponding registers. The mechanism resembles the one used by the VGA palette in the 256 color modes. However, the XGA palette is a simpler device than the one in VGA since no Palette or Color Select registers are used (see Figure 3.8). In other words, the XGA palette consists of 256 registers in which the red, blue, and green DAC values are stored. A pixel color is nothing more than a palette register number; the actual color in which the pixel is displayed depends on the value stored in the corresponding Palette register.

The XGA palette consists of 256 locations, each location divided into three fields. The first field corresponds to the red DAC value, the second one to the blue, and the third field to the green. The XGA allows two update modes: in the 3-value update mode data is written to the palette registers in groups of three items representing the red, blue, and green colors. In the 4-value update mode data is written in groups of four items, the first three represent the red, blue, and green values, and the fourth item is a padding byte which is ignored by the hardware. The 3-value sequence is similar to the one used in VGA systems. The 4-value sequence is the one used by the

AI HLDPAL function. The update mode is selected by means of bit 2 of the Palette Sequence register. Notice that in the XGA palette the 6 high-order bits are significant while in VGA the significant bits are the 6 low ones (see Figure 6.5).

The following code fragment shows the necessary processing for setting the 256 XGA palette registers from an array in RAM.

```
DATA     SEGMENT
;
; Double-bit IRGB palette in the following format
;            7 6 5 4 3 2 1 0  <= Bits
;            I I R R G G B B  <= Color codes
;
;                      | R   B   G       R   B   G      | REG
IRGB_SHADES     DB      000,000,000,000,036,072,036,000 ; 1
                DB      036,108,036,000,036,144,036,000 ; 3
                .       (missing data as in the code fragment
                .        in Section 6.4.2)
                .
                DB      252,144,252,000,252,180,252,000 ; 254
                DB      252,216,252,000,252,252,252,000 ; 255
;
; Previously initialized base address of the XGA Display
; Controller (see Section 12.2.1)
XGA_REG_BASE  DW    0           ; Address variable

DATA    ENDS
;
;
CODE    SEGMENT
        .
        .
        .
; Code to set 256 XGA Palette registers
; On entry:
;         SI --> 1024-byte color table in RGBx format
; Assumes that XGA system is set in a graphics mode
;
        LEA    SI,IRGB_SHADES   ; Pointer to data array
        MOV    DX,XGA_REG_BASE  ; Base address of XGA Display
                                ; Controller register
; Select Index register at offset 0AH
        ADD    DX,0AH           ; To Index register
; Write 00H (in AH) to Palette Mask register (64H)
; This value is ANDed with display memory. Clearing all bits
; makes the palette invisible during setup
        MOV    AX,0064H          ; make invisible
        OUT    DX,AX
; Write 00H (in AH) to Border Color register (55H)
        MOV    AX,0055H          ; border color
        OUT    DX,AX
; Write 00000100B (in AH) to Palette Sequence register (66H) to
; select four-color write mode (RGBx) and to start with the
; Red color code
        MOV     AX,0466H          ; Palette Sequence register
        OUT     DX,AX
; Write 00H (in AH) to Palette Index register low (60H)
; and high (61H) to select first DAC register
        MOV     AX,0060H          ; Start at palette 0
        OUT     DX,AX
        MOV     AX,0061H          ; Sprite index high
        OUT     DX,AX
; SI --> table of palette colors
        MOV     CX,1024         ; Counter for 256 * 4
        MOV     AX,065H         ; Select Data register
        OUT     DX,AL
        INC     DX              ; Point to first register
; Loop to send 4 blocks of 256 bytes each to port 065H
NEW_PALETTE:
        MOV     AL,[SI]         ; Get byte from table
        OUT     DX,AL           ; Send to port
        INC     SI              ; Bump table pointer
        LOOP    NEW_PALETTE
;
        DEC     DX              ; Back to Select register
; Write FFH (in AH) to Palette Mask register (64H)
; This value is ANDed with display memory. Setting all bits
; makes the palette visible again
        MOV    AX,0FF64H        ; All bits set
        OUT    DX,AX            ; To make visible
; At this point all Palette registers have been loaded from
; the data array supplied on entry
        .
        .
        .
```

The procedure named XGA_PALETTE in the XGA2 module of the GRAPHS-OL library, furnished with the book, can be used to perform palette loading. The code in this procedure is similar to the one listed above.

12.4 Processor Access to XGA Video Memory

An application can access XGA video memory through the CPU or by means of the XGA graphics coprocessor. Coprocessor programming is discussed in Section 12.5. The present discussion relates to accessing the XGA video memory space by means of the 80386 or 486 Central Processing Unit.

The system processor can access XGA memory to perform write and read operations. The write operation sets one or more screen pixels to the value stored in a processor register. The read operation transfers a pixel's value into a processor register. In Section 12.1.2 we saw that the XGA system can configure video memory by means of three possible apertures. The 4Mb aperture is the one used by the graphics coprocessor. Using this memory aperture will be discussed later in this chapter. The 1Mb memory aperture is typically used in multitasking environments.

MS-DOS applications usually access XGA video memory by means of multiple memory banks of 64K each. This is called the 64K aperture. Before this aperture is used the program must make sure that the Aperture Control register (at base address plus 1) has been initialized to the value 01H (see Table 12-4). The banks' structure at this aperture depends on the display mode. At the 1,024 by 768 modes the 64K aperture can be visualized as 12 memory blocks of 64K each. This visualization is shown in Figure 12-2, on the following page

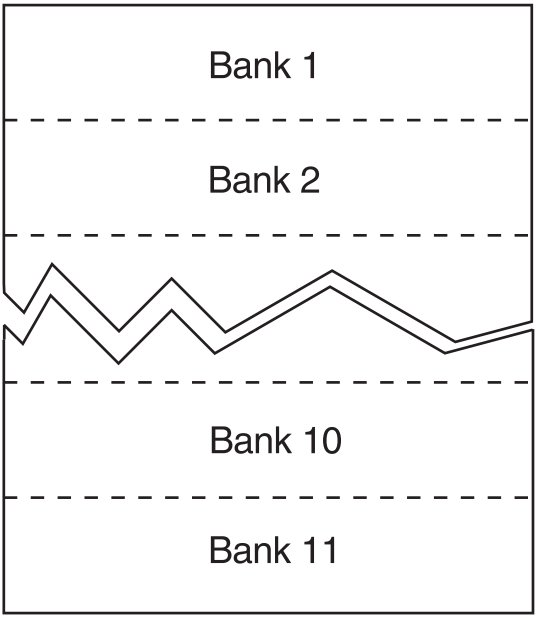


Figure 12-2 *Block Structure in XGA 64K Aperture*

Notice that, when using the 64K aperture, the start address for the video memory in each bank is selected by means of the Aperture Control register. The valid values are A0000H and B0000H. The first one coincides with the base address used in VGA graphics modes. If the start address of A0000H is selected, then each bank extends from A0000H to B0000H. Which bank is currently selected depends on the setting of the Aperture Index register, located at base address plus 8 of the XGA Display Controller group. If the base address of the Display Controller group is stored in the variable XGA_REG_BASE and the bank number in the AL register, then enabling the bank can be coded as follows:

```
MOV     DX,XGA_REG_BASE    ; XGA base register address
ADD     DX,08H             ; Aperture Index register
OUT     DX,AL              ; Bank number is in AL
```

The total number of banks available depends on the display mode selected. We saw that 12 banks of 65,536 memory units are needed to encode all the pixels in the 1,024 by 768 modes. However, in the 640 by 480 pixel mode each full screen consists of 307,200 pixels, which require only 5 memory banks of 65,536 units each.

12.4.1 Setting Screen Pixels

In order to set a screen pixel the display logic must take into account whether the base address of the video buffer for the 64K aperture is located at A000H or at B000H. In addition, the code must perform the necessary bank switching operation. Processing performance in this case can be improved by storing the value of the currently selected bank in a memory variable so that bank switching can be bypassed if the pixel is located in the currently selected bank. The following code fragment writes a data byte to a video memory address. This fragment does not take into account the currently selected bank.

```
; Write a screen pixel accessing XGA memory directly
; On entry:
;       CX = x coordinate of pixel
;       DX = y coordinate of pixel
;       BL = pixel color in 8-bit format
; Note: code assumes that XGA is in a 1024 by 768 pixel mode
;       in 256 colors and that A0000H is the start address for
;       the video buffer using the 64K aperture
;
; Set ES to video buffer base address
        MOV     AX,0A000H       ; Base for all graphics modes
        MOV     ES,AX           ; To ES segment
        MOV     AL,BL           ; Color to AL
; Get address in XGA system
        CLC                     ; Clear carry flag
        PUSH    AX              ; Save color value
        MOV     AX,1024         ; 1024 dots per line
        MUL     DX              ; DX holds line count of address
        ADD     AX,CX           ; Plus this many dots on the line
        ADC     DX,0            ; Answer in DX:AX
                                ; DL = bank, AX = offset
        MOV     BX,AX           ; Save offset in BX
        MOV     AX,DX           ; Move bank number to AL
;*********************|
;    change banks     |
;*********************|
        MOV     DX,XGA_REG_BASE ; XGA base register address
        ADD     DX,08H          ; Aperture index register
        OUT     DX,AL           ; Bank number is in AL
        POP     AX              ; Restore color value
;*********************|
;   set the pixel     |
;*********************|
        MOV     ES:[BX],AL      ; Write the dot
        .
        .
        .
```

The procedure named XGA_PIXEL in the XGA2 module of the GRAPHSOL library sets a screen pixel using processing similar to that shown in the above code sample. A routine to set the entire screen to a specific color value can be simplified by using 80x86 string move instructions. The following code fragment shows the processing necessary to clear the entire vide display in an XGA 1,024-by-768 pixel mode.

```
; Clear video memory using block move
        MOV     AX,0A000H       ; Video memory base address
        MOV     ES,AX           ; To the ES register
        MOV     BL,0            ; BL is bank counter
; Select bank
NEXT_BANK:
        MOV     DX,XGA_REG_BASE ; Select Page
        ADD     DX,08H          ; To Aperture Index register
        MOV     AL,BL           ; Bank number
        OUT     DX,AL           ; Select bank in AL
; Write 65536 bytes of 00H in current bank
        MOV     CX,0FFFFH       ; CX is byte counter
        MOV     AX,0            ; Attribute to place in VRAM
        CLD                     ; Forward direction
        MOV     DI,0            ; Start of block
        REP     STOSB           ; Store 65536 bytes
; Bump bank
        INC     BL
        CMP     BL,12           ; 12 is past last bank
        JNE     NEXT_BANK
        .
        .
        .
```

The procedure named XGA_CLS in the XGA2 module of the GRAPHSOL library clears the screen using processing similar to the one listed above.

12.4.2 Reading Screen Pixels

A write routine that accesses the video memory space through the Central Processing Unit can be easily converted to read the value of screen pixels. The conversion consists mainly of changing the write operation for a read operation and in making other minor register adjustments. The following code fragment can be used to read the value of a screen pixel into a CPU register.

```
; Read a screen pixel accessing XGA memory directly
; On entry:
;       CX = x coordinate of pixel
;       DX = y coordinate of pixel
; On exit:
;       BL = pixel color
; Note: code assumes that XGA is in a 1024 by 768 pixel mode
;       in 256 colors and that A0000H is the start address for
;       the video buffer using the 64K aperture
; Set ES to video buffer base address
        MOV     AX,0A000H       ; Base for all graphics modes
        MOV     ES,AX           ; To ES segment
; Get address in XGA system
        CLC                     ; Clear carry flag
        PUSH    AX              ; Save color value
        MOV     AX,1024         ; 1024 dots per line
        MUL     DX              ; DX holds line count of address
        ADD     AX,CX           ; Plus this many dots on the line
        ADC     DX,0            ; Answer in DX:AX
                                ; DL = bank, AX = offset
        MOV     BX,AX           ; Save offset in BX
        MOV     AX,DX           ; Move bank number to AL
;**********************|
;    change banks      |
;**********************|
        MOV     DX,XGA_REG_BASE ; XGA base register address
        ADD     DX,08H          ; Aperture Index register
        OUT     DX,AL           ; Bank number is in AL
        POP     AX              ; Restore color value
;**********************|
;   read the pixel     |
;**********************|
        MOV     BL,ES:[BX]      ; Read pixel in BL
        .
        .
        .
```

12.4.3 Programming the XGA Direct Color Mode

Mode number 4 in Table 12-3 is called the direct color mode. It consists of 640 by 480 pixels in 65,536 colors. Notice that this mode is available in XGA systems equipped with the full maximum VRAM of 1,024K. The XGA direct color mode presents some unique characteristics, among them the most extensive color range. In this mode the pixel color is determined by a 16-bit value, which encodes 65,536 colors that can be represented. The actual pixel color is generated independently of the setting of the DAC registers, for which reason the direct color mode has also been referred to as the palette bypass mode. The color encoding of the 16-bit value for the direct color mode is shown in Figure 12-3.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RED (5 bits)					GREEN (6 bits)						BLUE (5 bits)				

Figure 12-3 *Bitmapping in XGA Direct Color Mode*

Notice that the color bitmap in Figure 12-3 contains 5 bits for the blue and red elements and 6 bits for the green element. This 5-6-5 configuration allows 64 shades of green and 32 shades of both blue and red colors. The argument in favor of having more shades of green than of red and blue is that the human eye is more sensitive to the green portion of the spectrum.

The Direct Color Palette

Although the DAC registers are bypassed during direct color mode operation, the IBM documentation states that the DAC registers must be loaded with specific data for operating in the Direct Color mode. Table 12-5 shows the values recommended by IBM.

Notice that bit 7 of the Border Color register (at offset 55H) is used to select between the first and second group of values to be entered in the direct color palette. Also that the red and blue components are always zero, while the green component is incremented by 8 for each successive register. The following code fragment allows setting the Palette registers for the direct color mode.

```
; Code to set 256 XGA Palette registers for the 65535 color mode
; Note: the values are those recommended by IBM
; Code assumes that XGA system is set in a graphics mode
;
        MOV     DX,XGA_REG_BASE   ; Wait for a retrace
        ADD     DX,0AH            ; To index register
; Write 00H (in AH) to Palette Mask register (64H)
; This value is ANDed with display memory. Clearing all bits
; makes the palette invisible during setup
        MOV     AX,0064H          ; Make invisible
        OUT     DX,AX
; Write 00H (in AH) to Palette Sequence register (66H) to enable
; three-color write mode (RGB) and to start with the
; R color code
        MOV      AX,0066H         ; Palette sequence register
        OUT      DX,AX
```

Table 12-5

Palette Values for XGA Direct Color Mode

LOCATION	BORDER COLOR BIT 7	RED	BLUE	GREEN
0	1	0	0	0
1	1	0	0	8
2	1	0	0	16
3	1	0	0	24
.	.	.	.	.
.	.	.	.	.
31	1	0	0	256
32	1	0	0	0
33	1	0	0	8
.	.	.	.	.
.	.	.	.	.
126	1	0	0	240
127	1	0	0	248
128	0	0	0	0
129	0	0	0	8
130	0	0	0	16
131	0	0	0	24
.	.	.	.	.
.	.	.	.	.
159	0	0	0	256
160	0	0	0	0
161	0	0	0	8
.	.	.	.	.
.	.	.	.	.
254	0	0	0	240
255	0	0	0	248

```
; Write 00H (in AH) to Palette Index register low (60H)
; and high (61H) to select first DAC register
        MOV     AX,0060H        ; Start at palette 0
        OUT     DX,AX
        MOV     AX,0061H        ; Also set the Sprite Index
        OUT     DX,AX           ; High register
;**********************|
; first 128 registers  |
;**********************|
; Write 80H (in AH) to Border Color register (55H) to select
; first group of 128 registers
        MOV     AX,8055H        ; Border Color bit 7 set
        OUT     DX,AX
        CALL    LOAD_128        ; Local procedure
;**********************|
; second 128 registers |
;**********************|
; Write 00H (in AH) to Border Color register (55H) to select the
; second group of 128 registers
        MOV     AX,0055H        ; Border Color bit 7 clear
        OUT     DX,AX
        CALL    LOAD_128        ; Local procedure
; Write FFH (in AH) to Palette Mask register (64H)
; This value is ANDed with display memory. Setting all bits
; makes the palette visible again
        MOV    AX,0FF64H        ; All bits set
        OUT    DX,AX            ; To make visible
        .
        .
;****************************************************************
LOAD_128        PROC    NEAR
; Auxiliary procedure for XGA_DC_PALETTE to load a group of 128
; DAC registers with the recommended values
        MOV     DX,XGA_REG_BASE ; Base address
        ADD     DX,0AH          ; Index register
        MOV     AX,0065H        ; Select data register
        OUT     DX,AL
        INC     DX              ; To data register
        MOV     BX,0            ; BX is value for blue register
        MOV     CX,128          ; Counter for 128 registers
; Loop to send 3 bytes to 128 registers
DC_128:
        MOV     AL,0            ; Send red
        OUT     DX,AL           ; Send to port
        JMP     SHORT $ + 2     ; I/O delay
        OUT     DX,AL           ; Send blue
        MOV     AL,BL           ; Load green value
        OUT     DX,AL           ; Send green
        ADD     BL,8            ; Bump green value in BL
                                ; Wraps around automatically
        LOOP    DC_128
        DEC     DX              ; Back to Index register
        RET
LOAD_128        ENDP
```

The procedure named DC_PALETTE in the XGA2 module of the GRAPHSOL library can be used to set the XGA Palette registers to the direct color mode.

Pixel Operations in Direct Color Mode

The programmer working in the direct color mode has fewer options than in other XGA modes. In the first place there is no AI support for direct color mode operations. Another limitation is that the XGA graphics coprocessor (discussed in Section 12.4) is not operational in the direct color mode. In the direct color mode the actual setting of screen pixels is performed with a word write operation, as shown in the following code fragment.

```
; Word write operation for 16-bit per pixel mode
; AX = 16-bit color code in 5-6-5 format
; BX = offset into video buffer
; ES = video memory segment (A000H or B000H)
;
        MOV        ES:[BX],AX         ; Writes the pixel
```

In the direct color mode the programmer must take into account that each screen pixel is mapped to two video buffer bytes. For example, the tenth pixel from the start of the first screen row is located 20 bytes from the start of the buffer. By the same token, each pixel is at a word boundary in the video buffer. The display routine must make the necessary adjustment, as in the following code fragment.

```
; Display 10 pixels in the brightest red color at the center
; of the first screen row while in XGA direct color mode
; Assumes:
;     1. ES = video buffer base address (A000H or B000H)
;     2. Direct color palette has been loaded
;     3. Mode number 6 (640 by 480 in 65,536 colors) has been set
;     4. XGA_REG_BASE variable holds base address of XGA Display
;        Controller
; First select video bank number 0
        MOV     DX,XGA_REG_BASE ; Select Page
        ADD     DX,08H          ; To Aperture Index register
        MOV     AL,0            ; Bank number
        OUT     DX,AL           ; Select bank in AL
; Setup operational variables
        MOV     CX,10           ; Counter for 10 pixels
        MOV     AX,0F800H       ; All red bits set
        MOV     DI,640          ; Offset pointer to word number
                                ; 320 on first screen row
; Write 10 bytes of AX into video memory
SET_10_PIXS:
        MOV     ES:[DI],AX      ; Write to memory
        ADD     DI,2            ; Bump pointer to next word
        LOOP    SET_10_PIXS
        .
        .
```

Notice in the above code fragment that the value initially loaded into the buffer pointer register (DI) is the word offset of the first pixel to be set. Also that the pointer is bumped to the next word (ADD DI,2) in each iteration of the loop.

12.5 Programming the XGA Graphics Coprocessor

To a programmer the most important XGA hardware component is the graphics coprocessor chip. The general features of the XGA graphics coprocessor were discussed in Section 12.1.1. The present discussion relates to performing graphics operations by programming the XGA coprocessor. The reader should notice that the XGA Graphics Coprocessor is a complex and sophisticated IC. In the following sections we will cover only its programming at the elementary level. A detailed technical description of this device, as well as of the XGA system in general, can be found in the XGA Video Subsystem section of the IBM Technical Reference Manual for Options and Adapters, document number 504G-3287-000. This document can be obtained from the IBM Literature Department.

To the programmer the XGA graphics coprocessor appears as a set of memory-mapped registers. The area of memory devoted to these registers is called the coprocessor's address space. Table 12-6 is a map of the coprocessor registers.

The coprocessor registers can be accessed using either the Intel or the Motorola data formats. Table 12-6 represents the register structure in the Intel format. Most coprocessor registers are write only. The second column in Table 12-6 shows which registers can be read by the CPU. Notice that the Current Virtual Address, State A Length, and State B Length registers are read-only. Software should not write to these registers. The Page Directory Base Address and the Current Virtual Address registers (offset plus 0 and plus 4 respectively) are used only in a virtual memory environment. Real mode programs, such as those executing in MS-DOS, need not access these registers.

Table 12-6

XGA Graphic Coprocessor Register Map

HEX OFFSET	READ /WRITE	+0	+1	+2
0	W	Page Directory Base Address		
4	R	Current Virtual Address		
8				
C	R	State A Length	State B Length	
10	R/W W		Coprocessor Control	Pixel Map Index
14	W	Pixel Map n Base Pointer		
18	W	Pixel Map n Width		Pixel Map n Height
1C	W	Pixel Map format		
20	R/W	Bresenham Error Term		
24	W	Bresenham K1 Term		
28	W	Bresenham K2 Term		
2C	W	Direction Steps		
.				
.				
44				
48	W	Foreground Mix	Background Mix	Destination Color Compare Condition
4C	W	Destination Color Compare Value		
50	W	Pixel Bit Mask		
54	W	Carry Chain Mask		
58	W	Foreground Color		
5C	W	Background Color		
60	W	Operations Dimemsion 1		Operations Dimension 2
64				
68				
6C	W	Map Mask Origin x Offset		Map Mask Origin y Offset
70	R/W	Source Map x Coordinate		Source Map y Coordinate
74	R/W	Pattern Map x Coordinate		Pattern Map y Coordinate
78	R/W	Destination Map x Coordinate		Destination Map y Coordinate
7C	W	Pixel Operations		

The XGA coprocessor can access all memory in the system and treats video memory and system memory in the same fashion. Once the coprocessor is informed of the VRAM address it uses it to determine if the memory access is local or remote. In remote accesses the coprocessor obtains direct control of the bus. This capability of the coprocessor improves XGA performance by allowing the CPU to continue executing code while the coprocessor manipulates memory data.

The XGA Graphics Coprocessor is designed to take advantage of the 80386 instruction set. Since XGA requires an 80386 CPU, XGA programs can safely use 80386 instructions without fear of hardware incompatibility. Therefore, in the code samples that follow we have used 80386/486 instructions when programming coprocessor operations.

12.5.1 Initializing the Coprocessor

The initial action taken by a program that accesses the XGA coprocessor is its initialization. The first two steps in coprocessor initialization consist of calculating and storing two data items required in programming this device: the base address of the coprocessor register space and the physical address of the start of video memory. Notice that the video memory address used by the coprocessor corresponds with the 4Mb aperture mentioned in Section 12.1.2. The data for calculating these addresses is found in the XGA POS registers (Section 12.2.1 and Figure 12-1). In addition, the initialization routine should make certain that the appropriate value is stored at the Memory Access Mode Register of the XGA Display Controller group.

Obtain the Coprocessor Base Address

The coprocessor base address is calculated from the ROM address field in POS register 2 (see Figure 12-1) and from the instance field in this same POS register. The coprocessor address formula is

```
coprocessor address = (((i * 128) +1C00H) + (R + 2000H) + C000H)
```

where i is the instance and R is the value in the ROM field of POS register 2. The code for calculating the coprocessor address is as follows:

```
DATA      SEGMENT

; The following variables are loaded from the XGA POS registers
; as shown in the code sample in Section 12.2.1
POS_2     DW         ????       ; POS register 2
POS_2     DW         ????       ; POS register 4

DATA      ENDS

CODE      SEGMENT
          .
          .
          .
; Calculate coprocessor base address
; Code assumes that the POS_2 and POS_4 variables have been
; initialized to the contents of the corresponding POS registers
; Coprocessor base address is calculated as follows:
; ROM address = (ROM field + 2000H) + C0000H
; COP address = (((Instance * 128) + 1C00H) + ROM address)
;
; First calculate ROM address from data in POS register 2
        MOV     EAX,0           ; Clear EAX
        MOV     AL,POS_2        ; Get POS register 2
        AND     EAX,0F0H        ; Preserve ROM bits
        SHR     EAX,4           ; Shift ROM to low nibble
        MOV     ECX,2000H       ; Multiplier
        MUL     ECX             ; EAX * ECX in EAX
        ADD     EAX,0C0000H     ; Add constant
        MOV     EBX,EAX         ; Store ROM address in EBX
; EBX now holds ROM address
; Instance is stored in bits 1-3 of POS register 2
        MOV     EAX,0           ; Clear EAX
        MOV     AL,POS_2        ; Get POS register 2
        AND     EAX,0EH         ; Preserve Instance bits
        SHR     EAX,1           ; Shift right Instance bits
        MOV     ECX,128         ; Multiplier to ECX
        MUL     ECX
        ADD     EAX,1C00H       ; Add constant from formula
; Add ROM address
```

```
        ADD     EAX,EBX
        SHR     EAX,4           ; Shift right one nibble to
                                ; to obtain segment value
; Store segment value in GS
        MOV     GS,AX           ; Move segment into GS
;
```

Notice that the segment value of the coprocessor base address is stored in segment register GS. This is consistent with the notion of making full use of the 80386 architecture and instruction set.

Obtain the Video Memory Address

The physical address of video memory is a 32-bit value determined from the video memory base address field in POS register 4 and from the instance field in POS register 2 (see Figure 12-1). The address is formed by re-locating the POS data items as shown in Figure 12-4.

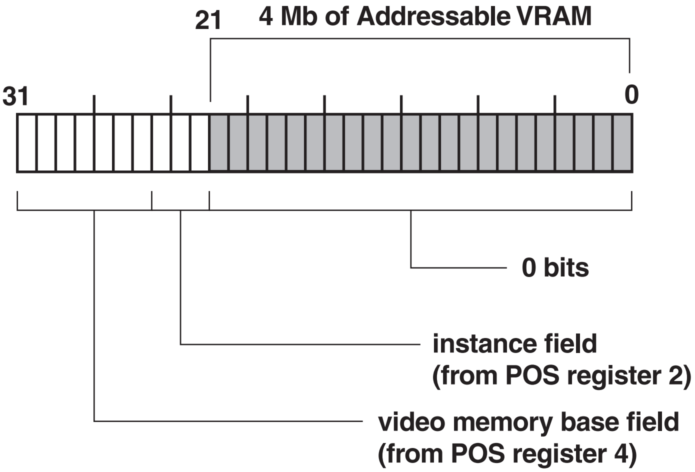


Figure 12-4 *Physical Address of Video Memory Bitmap*

The required processing for calculating the VRAM physical address is shown in the following code fragment.

```
;**********************|
;    get VRAM base     |
;**********************|
; First get the video memory field in POS register 4
        MOV     AL,POS_4        ; VRAM field
        AND     AL,11111110B    ; Clear low bit
        SHL     AX,8            ; Shift to high position
; Now get instance bits in POS register 2
        MOV     BL,POS_2        ; Instance field
        AND     BL,00001110B    ; Mask out other bits
        MOV     BH,0            ; Clear high part of BX
        SHL     BX,5            ; Move instance bits to position
        OR      AX,BX           ; OR with B bits (in AX)
        MOV     FS,AX           ; Store in FS segment
```

Notice that the high-order part (16 bits) of the VRAM physical address is now stored in the FS segment register. The 80386 FS segment is a convenient storage for this value, which must later be used in coprocessor programming.

Select Access Mode

Coprocessor operation requires that the Memory Access Mode register of the Display Controller be set to 1, 2, 4, or 8 bits per pixel and to the Intel or Motorola data storage format. In the PC environment with a fully equipped XGA (1Mb of VRAM) the coprocessor is typically set to 8 bits per pixel and to match the Intel format of the CPU. The following code fragment shows selecting the access mode for coprocessor operation.

```
; Select Intel order and 8 bits per pixel in the Memory Access
; Mode register (offset + 9)
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,9            ; To Mode register
        MOV     AL,03H          ; 7 6 5 4 3 2 1 0   <= bitmap
                                ; | | | | | | | |   Bits/pixel
                                ; | | | | | |_|_|__ 000 = 1 bit
                                ; | | | | |         001 = 2 bits
                                ; | | | | |         010 = 4 bits
                                ; | | | | |        *011 = 8 bits
                                ; | | | | |         100 = 16 bits
                                ; | | | | |         FORMAT:
                                ; | | | | |___ *0 = Intel
                                ; | | | |        1 = Motorola
                                ; |_|_|_|____ RESERVED
                                ; 03H = 00000011B
        OUT     DX,AL
```

At this point the coprocessor is ready for use. The procedure INIT_COP in the XGA2 module of the GRAPHSOL library uses similar processing to initialize the coprocessor. The programmer must consider that if this initialization code is used, the software must make sure that the 80386 segment registers FS and GS are preserved, since their contents are repeatedly required in setting up the coprocessor operations.

12.5.2 Coprocessor Operations

The XGA graphics coprocessor can autonomously perform drawing operations in parallel with the CPU. The coprocessor can execute in 1, 2, 4, and 8 bits per pixel formats, but not in the direct color mode described in Section 12.3.3. The execution of a coprocessor operation requires the following steps:

1. The CPU initializes the coprocessor registers to be used in the operation.
2. Coprocessor operation starts when the CPU writes a command to the Pixel Operations register.
3. The coprocessor executes the programmed operation. During this time the system microprocessor can be performing other tasks.

The graphics functions that can be performed by the coprocessor are pixel block transfer (abbreviated pixBlt), line draw, and draw and step.

The programmer can set up the coprocessor so that it generates an interrupt at the conclusion of its operations. This mechanism can be used in optimizing parallel processing, in task switching in a multitasking environment, in error recovery, and in synchronizing coprocessor access. The coprocessor Operation Complete interrupt is enabled by setting bit 7 of the Interrupt Enable register of the Display Controller group. The interrupt source is identified by testing the corresponding bit in the Interrupt Status register of the Display Controller group (see Figure 9.4 and Figure 9.5 in Chapter 9). Notice that this is set if an interrupt occurred, regardless of the setting of the Interrupt Enable register.

Synchronizing Coprocessor Access

Since the coprocessor operates asynchronously regarding the CPU, the central processor must wait until the coprocessor has con-cluded its previous operation before issuing a new command. This can be performed in two ways: by enabling the Coprocessor Operation Complete interrupt described in the previous paragraph or by polling the busy bit in the coprocessor Control register. Both methods are quite feasible, each having its advantages and disadvantages.

An XGA interrupt handler for testing the conclusion of coprocessor operation (or any other XGA interrupt for that matter) is designed to intercept vector 0AH, which corresponds with the IRQ2 line of the system's Interrupt Controller. Since this interrupt can be shared, the handler must first make sure that the interrupt was caused by the coprocessor. This requires testing bit 7 of the Interrupt Status register (at offset 05H). If the Coprocessor Operation Complete bit is set, then the code can proceed with the next coprocessor operation. At this time the code must write 1 to bit number 7 in order to clear the interrupt condition so that the next interrupt can take place.

Since polling the busy bit is easier to implement in software this is the method illustrated in the present section. The main objection to polling for hardware not busy is that it slows down operations since the coprocessor must pause execution to read its own Control register. This can be partially overcome by designing routines that includes a delay loop so that so that the coprocessor is not polled constantly. The following procedure from the XGA2 module of the GRAPHSOL library polls bit 7 of the coprocessor Control register to test for a not-busy condition. The COP_RDY procedure is called by the drawing routines in the XGA2 module before emitting a new coprocessor command. The delay period in the wait loop is an arbitrary value.

```
COP_RDY          PROC    NEAR
; Poll bit 7 of coprocessor Control register (offset 11H) to
; determine if coprocessor is busy, if so, wait until ready
; Code assumes that GS segment holds coprocessor base address
        PUSH    AX               ; Save context
        PUSH    CX
TEST_COP:
        MOV     AL,GS:[+11H]     ; Read control register
        TEST    AL,10000000B     ; Test bit 7
        JZ      COP_READY        ; Go if bit is clear
        MOV     CX,100           ; Counter for wait loop
; A 100 iteration wait loop is introduced so that the coprocessor
; is not polled constantly, since constant polling would slow
; down execution
```

```
WAIT_100:
        NOP                     ; Delay
        NOP
        LOOP    WAIT_100        ; Wait
        JMP     TEST_COP        ; Test again after wait
COP_READY:
        POP     CX              ; Restore context
        POP     AX
        RET
COP_RDY         ENDP
```

General Purpose Maps

The XGA graphics coprocessor can operate on three general purpose pixel maps, designated as Map A, Map B, and Map C in the IBM literature. The identification letters A, B, and C are sometimes generically represented by the variable n, as is the case in the Pixel Map n Base Pointer designation used in Table 12-6. Notice that, in actual coding, Map n is either Map A, Map B, or Map C. Pixel maps can be located in system or in video memory. The maximum size of a map is of 4,096 by 4,096 pixels.

The following coprocessor registers are related to pixel maps:

1. The Pixel Map n Base Pointer register (at offset 14H) contains the map's start address.
2. The Pixel Map n Width register (at offset 18H) determines the horizontal dimension of the pixel map and the Pixel Map n Height register (at offset 1AH) determines its vertical dimension. The values loaded into these registers must be one less than the required size.
3. The Pixel Map Format register (at offset 1CH) determines if the map is in 1, 2, 4, or 8 bits per pixel and whether it is encoded in Intel or Motorola data format.
4. The Pixel Map Index register (at offset 12H) is used to determine if the mask map is of type A, B, C, or M. The different mask map types are explained in the following paragraphs.

The x and y coordinates of a pixel map are based on the same convention used for the video display, that is, the top-left corner of the pixel map has coordinates x = 0, y = 0. The value of x increases to the right and the value of y increases downward. The pixel map coordinate system conventions and dimensions are shown in Figure 12-5.

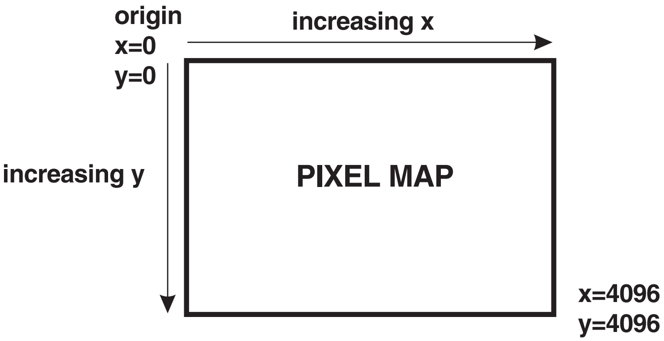


Figure 12-5 *Pixel Map Origin and Dimensions*

In relation to the coprocessor operation a pixel map can represent a source, a destination, or a pattern. The following cases represent common bitBlt operations:

1. In displaying a bitmap stored in the applications address space the source map is the application's data, and the destina-tion map is a location in video memory.
2. In an operation that consists of reading video data into system memory the source is a VRAM map and the destination a location in the application's memory space.
3. An operation that copies a video image into another screen area has both source and destination in video memory.
4. The coprocessor can also copy an area of user memory into another one. In this case both source and destination maps are located in the application's memory space.

The pattern map is used in determining if a pixel is considered a foreground or a background. A value of 1 indicates a foreground and a value of 0 a background. This action is shown later in this section.

The Mask Map

The mask map is an additional type of pixel map closely related to the destination map. The mask map, also called Map M, is used to protect the destination map on a pixel-by-pixel basis. In contrast with the other general purpose maps, the mask map is always fixed to a 1 bit per pixel ratio. A 0 bit in the Mask Map (inactive mask) protects the corresponding destination pixel from update, while a 1 bit allows the pixel's normal update.

The x and y dimensions of the mask map can be equal or less than the corresponding coordinates in the destination map. If the mask map and destination map have the same dimensions, then masking is a simple bit to pixel relation. If the mask map is smaller than the destination map then a scissoring operation is performed. In this respect the mask map action can be in one of three modes, as follows:

1. Mask Map Disabled. In this mode the Mask Map is ignored.
2. Mask Map Boundary Enabled. In this mode the Mask Map performs an outline scissoring action similar to a rectangular window. The contents of the Mask Map are ignored.
3. Mask Map Enabled. In this mode the mask map's border acts as a scissoring rectangle, at the same time its contents provide a pixel by pixel masking operation.

The making mode is selected by a 2-bit field in the Pixel Operations register. The difference between the Mask Map Enabled and the Boundary Enabled modes can be seen in Figure 12-6, on the following page.

Notice that the action of a mask map in the Boundary Enabled mode is identical to that of a mask map of all one bits. The difference is that the Boundary Enabled mask map consumes no memory while a normal mask map can take up as much as 94Kb in 1,024-by-768 pixels resolution.

In addition to the map address the program can define the pixel map's x and y coordinates. These value can be interpreted as offsets within the map. For example, if the destination pixel map is the video screen, the physical address of VRAM is entered in the Pixel Map n Base Pointer register and the actual position within the video display is determined by the x and y coordinates entered in the Destination Map x Coordinate and Destination Map y Coordinate registers. On the other hand, if the pixel map is within the application's address space, the offset is usually zero. This value signals the start of the pixel map as the reference position, however; the coordinates can be changed to indicate another position within the defined rectangle.

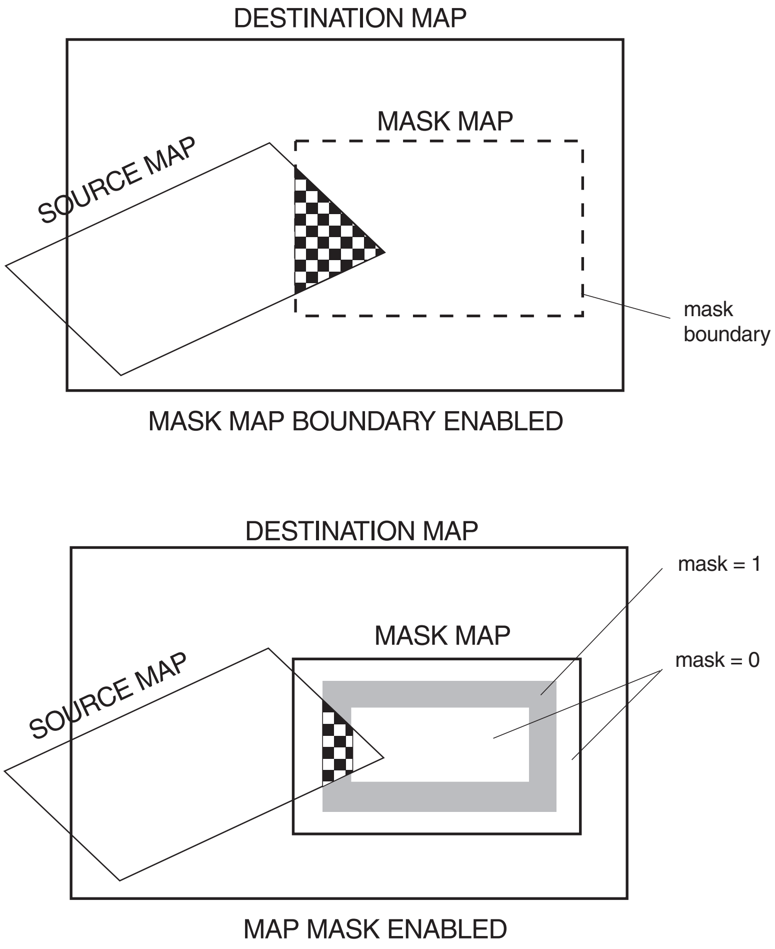


Figure 12-6 *Mask Map Scissoring Operations*

Coordinate registers for source and pattern pixel maps are available at offset 70H and 74H (see Table 12-6). However, there are no x and y coordinate registers for the mask map, because its origin is assumed to coincide with that of the destination map. Nevertheless, if the mask map is smaller than the destination map it becomes necessary to locate the mask map within the destination map. This is done by means of the Mask Map Origin x Offset and the Mask Map Origin y Offset registers at offset 6CH and 6EH respectively. The use of these mask map offset values is shown in Figure 12-12.

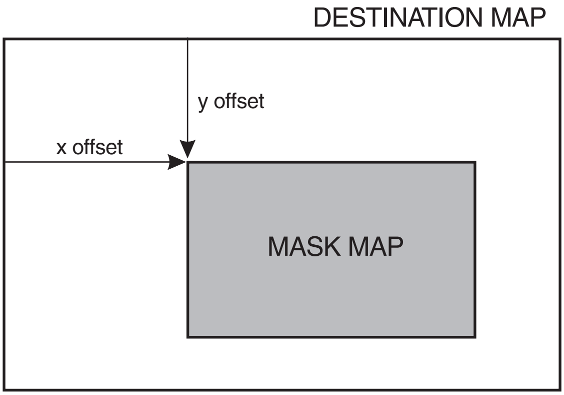


Figure 12-7 *Mask Map x and y Offset*

Pixel Attributes

The coprocessor generates a pixel with specific attributes by combining the source, pattern, and destination, according to a certain mix mode. The pattern pixel map, if used, serves as a filter to determine if a bit corresponds to a foreground or a background pixel. A value of 1 in the pattern pixel map determines that the bit is mapped to a foreground pixel, a value of 0 determines that the bit is mapped to a background pixel. If no pattern map is used then the foreground and background sources can be a specific color or determined by the color encoding stored in a source map. If the foreground source is a specific color, it is stored at the Foreground Color register at offset 58H. The background color is stored at the register at offset 5CH. The elements that take part in determining a pixel's attributes are shown in Figure 12-8.

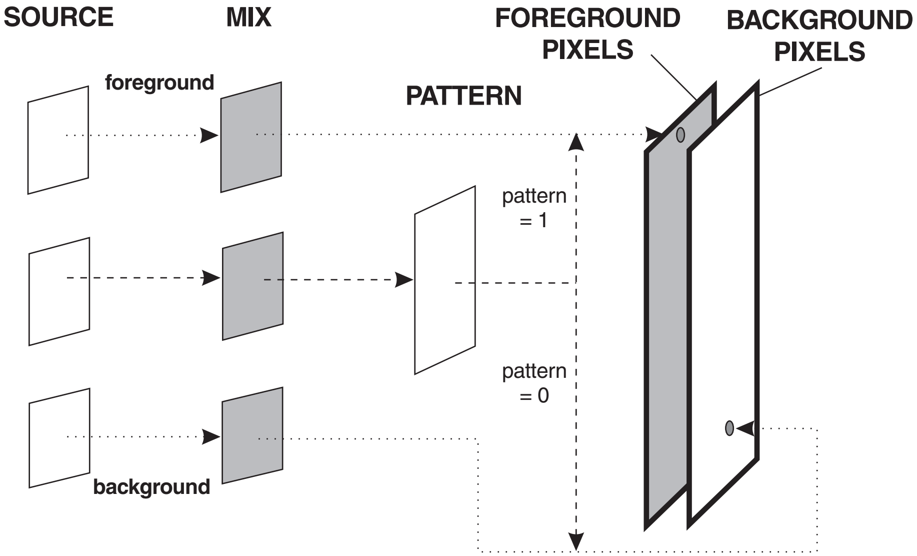


Figure 12-8 *Determining the Pixel Attribute*

Pixel Masking and Color Compare Operations

In addition, it is possible to protect individual pixels by masking. The Pixel Bit Mask register (offset 50H) is used for this purpose. A value of 1 in the Pixel Bit Mask register enables the corresponding pixel for update, while a value of 0 determines that the pixel is excluded from the update operation. Notice that the Pixel Bit Mask is related to the adopted format. In an 8-bits per pixel mode, the Pixel Bit Mask has active the 8 low order bits of the register, while in a 2 bits per pixel mode only the lowest 2 bits are used.

The coprocessor also allows a color compare operation that further inhibits certain pixel patterns from upgrade. The Destination Color Compare Value register (offset 4CH) is used for storing the bitmap to be used in the comparison. As with the Pixel bitmap register, the number of bits effectively used in the color compare operation depends on the number of bits per pixel in the adopted format. Several color compare conditions are allowed. The code for the selected condition is stored in the Destination Color Compare Condition register (offset 4AH). Table 12-7 lists the condition codes and the respective action.

Table 12-7

Destination Color Compare Conditions

CODE	BINARY	CONDITION
0	000	Always true (disable update)
1	001	Destination > color compare value
2	010	Destination = color compare value
3	011	Destination < color compare value
4	100	Always false (enable update)
5	101	Destination > = color compare value
6	110	Destination < > color compare value
7	111	Destination < = color compare value

Mixes

In Figure 12-8 we see that the attribute of the destination pixels depends upon a mix. The mix is a logical or arithmetic operation used in combining the source and the destination bitmaps. The mix is selected independently for the foreground and the background pixels (see Figure 12-8). The foreground mix is entered into the Foreground Mix register (offset 48H) and the background mix into the Background Mix register (offset 49H). The actual mix operation is determined by a mix code. The mix codes and action are shown in Table 12-8.

The word saturate in Table 12-8 means that if the result of an addition or subtraction operation is greater than 1, the final result is left at 1, while if it is smaller than 0 it is left at 0.

Table 12-8

Logical and Arithmetic Mixes

CODE	HEX	ACTION
0	00H	Zeros
1	01H	Source AND destination
2	02H	Source AND NOT destination
3	03H	Source
4	04H	NOT source AND destination
5	05H	Destination
6	06H	Source XOR destination
7	07H	Source OR destination
8	08H	NOT source AND NOT destination
9	09H	Source XOR NOT destination
10	0AH	NOT destination
11	0BH	Source OR NOT destination
12	0CH	Source NOT destination
13	0DH	NOT source OR destination
14	0EH	NOT source OR NOT destination
15	0FH	Ones
16	10H	Maximum
17	11H	Minimum
18	12H	Add with saturate
19	13H	Destination minus source (with saturate)
20	14H	Source minus destination (with saturate)
21	15H	Average
22	16H	|
.	.	> Reserved
255	FFH	|

Pixel Operations

The coprocessor starts executing the programmed operation when data is written to the Pixel Operations register (offset 7CH). The one exception to this statement is the draw and step command which is initiated by writing to the Direction Steps register (at offset 2CH). The Pixel Operations register also defines the flow of data during coprocessor operations. Figure 12-9, on the following page, is a bitmap of the Pixel Operations register.

The action performed by each field of the Pixel Operations register is explained in the discussion of the various coprocessor commands contained in the sections that follow.

12.5.3 PixBlt Operations

A pixel block transfer operation (pixBlt) consists of moving rectangular memory block from a source area to a destination area. Both the source and the destination can be system or video memory. The dimensions of the pixel rectangles are entered into the Operations Dimension registers; the width into Operations Dimension 1 and the height into Operation Dimension 2. The pixBlt can be programmed to start at any one of the four corners of the rectangle. The operation always proceeds in the direction of the diagonally opposite corner. The direction is entered into the Pixel Operations register (at offset 7CH).

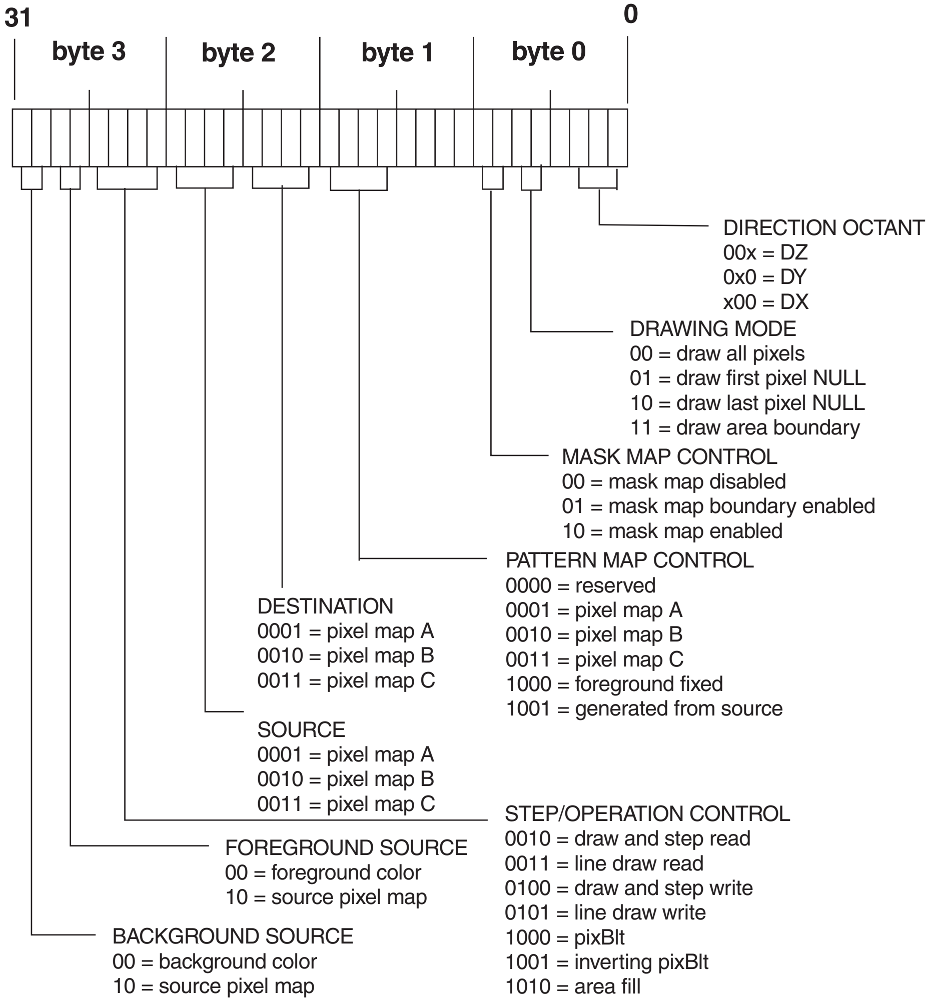


Figure 12-9 *Pixel Operations Register Bitmap*

Rectangular Fill PixBlt

Perhaps the simplest pixBlt operations is filling a rectangular screen area using the Foreground Color register as source data. The following code fragment shows the coprocessor commands necessary to perform this form of pixBlt.

```
; Use graphics coprocessor to perform a pixBlt on a rectangular
; screen area
; Code assumes XGA 1024 by 768 mode in 256 colors (8 bits per
; pixel)
; At this point:
;         CX = x coordinate of top-left corner
;         DX = y coordinate of top-left corner
;         SI = width of rectangle, in pixels
;         DI = height of rectangle, in pixels
;         BL = 8-bit color code
;         segment register setting:
;         GS = Coprocessor base address (Section 12.4.1)
;         FS = VRAM base address (Section 12.4.2)
;
;**********************|
;   test for not busy  |
;**********************|
        CALL    COP_RDY         ; Routine developed in
                                ; Section 12.4.2
; At this point the coprocessor is not busy
;**********************|
;   prepare to pixBlt  |
;**********************|
; Memo: GS holds the coprocessor base address (see Section 12.4.1)
        MOV     AL,01H          ; Data value for Map A
        MOV     GS:[+12H],AL    ; Write to Pixel Map Index
        MOV     AX,0H           ; Data value for VRAM low
        MOV     GS:[+14H],AX    ; Write to pix map base address
; Memo: FS register holds the high order word of VRAM address.
; ( see Section 12.4.2)
        MOV     AX,FS           ; Data for VRAM high
        MOV     GS:[+16H],AX    ; Write to pix map segment
                                ; address
; Code assumes 1024 by 768 pixel mode and Intel format
        MOV     AX,1023         ; Value for pix map width
        MOV     GS:[+18H],AX    ; Write to Width register
        MOV     AX,767          ; Value for pix map height
        MOV     GS:[+20H],AX    ; Write to Height register
        MOV     AL,3            ; Select Intel order and 8 bits
                                ; per pixel
        MOV     GS:[+1CH],AL    ; Write to Format register
;**********************|
;  enter pixBlt data   |
;**********************|
        MOV     AL,03H          ; Select source mix mode
        MOV     GS:[+48H],AL    ; Write to Mix register
; Write color (in BL) to foreground register
        MOV     GS:[+58H],BL    ; Write to Foreground Color
                                ; register
; Write coordinates of rectangle's start point to coprocessor
; registers
        MOV     GS:[+78H],CX    ; Write to Destination x Address
                                ; register
        MOV     GS:[+7AH],DX    ; Write to Destination y Address
                                ; register
; Store width in Operations Dimension 1 register
        MOV     GS:[+60H],SI    ; Write to Operation Dimension 1
; Store height in Operations Dimension 2 register
        MOV     GS:[+62H],DI    ; Write to Operation Dimension 2
;**********************|
;  setup pix operation |
;      registers       |
;**********************|
; Bitmap of Pixel Operations register for pixBlt operation:
; byte 3 = bbss|pppp
;                     bb = background source
;                          00 = fixed register pixBlt
;                     ss = foreground source
;                          00 = fixed register pixBlt
;                     pppp = step/operation control
;                          1000 = pixBlt
;                     BYTE 3 = 00001000B = 08H
; byte 2 = SSSS|DDDD
;                     SSSS = source
;                          0001 = pixel map A
;                     DDDD = destination
;                          0001 = pixel map A
;                     BYTE 2 = 00010001B = 11H
; byte 1 = PPPP|0000
;                     PPPP = pattern map control
;                          1000 = foreground fixed
;                     BYTE 1 = 10000000B = 80H
; byte 0 = mm00|0oox
;                     mm = mask pixel map
;                          00 = mask map disabled
;                     oox = octant bits (x = don't care)
;                          00 = start at top left and move
;                               right and down
;                     BYTE 0 = 00000000B = 00H
;**********************|
;    execute pixBlt    |
;**********************|
; Coprocessor operation commences when data is written to the
; Pixel Operations register
        MOV     EAX,08118000H   ; Value from bitmap
        MOV     GS:[+7CH],EAX   ; Write to Pixel Operations
                                ; register
```

If XGA is initialized to 1,024 by 768 pixels in 256 colors, and if on entry to the above code fragment the CX register holds 512, the DX register holds 384, the SI register holds 100, the DI register holds 80, and BL = 00001100B, then an 100-by-80 pixel rectangle is drawn with its left-top corner at the center of the screen. If the default palette is active, the color of the rectangle is bright red.

Notice, in the above example, that the direction octant bits in byte 0 of the Pixel Operations register (see Figure 12-9) determine the direction in which the pixBlt takes place. For performing a non-overlapping pixBlt the direction octant bits are normally set to zero. However, if the source and destination rectangles overlap, the direction octant bits must be used in order to avoid pixel corruption. Table 12-9 shows the action of the direction octant bits in pixBlt operations. Notice that these bits are interpreted differently during the coprocessor line draw functions.

Table 12-9

Action of the Direction Octant Bits During PixBlt

VALUE	ACTION
00x	From top-left to bottom-right
10x	From top-right to bottom-left
01x	From bottom-left to top-right
11x	From bottom-right to top- left
Legend: x = don't care	

The procedure named COP_RECT in the XGA2 module of the GRAPHSOL library can be used to perform a rectangular fill pixBlt operation. Processing and entry parameters are the same as in the above code fragment.

System Memory to VRAM PixBlt

Another frequent use of the pixBlt operation is to display an image stored in the application's memory space. The processing of a system-to-video-memory pixBlt is similar to the one used in the rectangular fill pixBlt discussed in the preceding paragraphs. The following code fragment is a memory-to-video pixBlt of an image encoded in 1-bit per pixel format.

```
; Use graphics coprocessor to perform a pixBlt operation
; from a source in system memory to a destination in video memory
; Image map is encoded in 1 bit per pixel format
; Code assumes XGA 1024 by 768 mode in 256 colors (8 bits per
; pixel)
; At this point:
;         DS:SI = offset of source bitmap in RAM
;         CX = source map pixel width
;         DX = source map pixel height
;         SI = x coordinate of video image
;         DI = y coordinate of video image
;         BL = 8-bit color code to use in displaying image
;
; Segment register setting:
;         GS = Coprocessor base address (Section 12.4.1)
;         FS = VRAM base address (Section 12.4.2)
;
;**********************|
;   test for not busy  |
;**********************|
        CALL    COP_RDY          ; Routine developed in
                                 ; Section 12.4.2
; At this point the coprocessor is not busy
;**********************|
; map A is destination |
;   (video memory)     |
;**********************|
        PUSH    AX               ; Bitmap offset to stack
        MOV     AL,01H           ; Data value for Map A
        MOV     GS:[+12H],AL     ; Write to Pixel Map index
        MOV     AX,0H            ; Data value for VRAM low
        MOV     GS:[+14H],AX     ; Write to pix map base address
; FS register holds the high order word of VRAM address
        MOV     AX,FS            ; Data for VRAM high
        MOV     GS:[+16H],AX     ; Write to pix map segment
                                 ; address
; Destination map is 1024 by 768 pixel mode and Intel format
        MOV     AX,1023          ; Value for pix map width
        MOV     GS:[+18H],AX     ; Write to Width register
        MOV     AX,767           ; Value for pix map height
        MOV     GS:[+20H],AX     ; Write to Height register
; Bitmap of Pixel Format register:
; 7 6 5 4 3 2 1 0 <= bits
; | | | | | |_|_|______ pixel image size (* = selected value)
; | | | | |             000 = 1 bit per pixel
; | | | | |             001 = 2 bits per pixel
; | | | | |             010 = 4 bits per pixel
; | | | | |             *011 = 8 bits per pixel
; | | | | |_______ format control
; | | | |           1 = Motorola order
; | | | |          *0 = Intel order
; |_|_|_|________________________________ RESERVED
;
        MOV     AL,3            ; Select Intel order and 8 bit
                                ; per pixel
        MOV     GS:[+1CH],AL    ; Write to Format register
;**********************|
;    map B is source   |
;    (system memory)   |
;**********************|
        MOV     AL,02           ; Data value for Map B
        MOV     GS:[+12H],AL    ; Write to Pixel Map index
; AX = offset of source bitmap (in stack)
; DS = segment of source bitmap
; To convert logical address to physical address the segment
; value is shifted left 4 bits and the offset added
        MOV     EAX,0           ; Clear 32 bits
        MOV     AX,DS           ; Segment to AX
        SHL     EAX,4           ; Shift segment 4 bits
        POP     BP              ; Offset to BP
        ADD     AX,BP           ; Add offset to segment
        MOV     GS:[+14H],EAX   ; Write to pix map base address
; Dimensions of source map are in CX and DX registers
        DEC     CX
        DEC     DX
        MOV     GS:[+18H],CX    ; Write to Width register
        MOV     GS:[+20H],DX    ; Write to Height register
; Bitmap of pixel format register:
; 7 6 5 4 3 2 1 0 <= bits
; | | | | | |_|_|______ pixel image size (* = selected value)
; | | | | |             *000 = 1 bit per pixel
; | | | | |              001 = 2 bits per pixel
; | | | | |              010 = 4 bits per pixel
; | | | | |              011 = 8 bits per pixel
; | | | | |_______ format control
; | | | |          *1 = Motorola order
; | | | |           0 = Intel order
; |_|_|_|________________________________ RESERVED
        MOV     AL,08H          ; Select Motorola order and 1
                                ; bit per pixel
        MOV     GS:[+1CH],AL    ; Write to Format register
;**********************|
;    select mix mode   |
;**********************|
        MOV     AL,03H          ; Select source mix mode
        MOV     GS:[+48H],AL    ; Write to Mix register
; Write color (in BL) to foreground register
        MOV     GS:[+58H],BL    ; Write to Foreground Color
                                ; register
; Write coordinates of source and destination
; Source coordinates are 0,0, destination coordinates are in SI
; and DI
        MOV     AX,0            ; Source coordinates
        MOV     GS:[+70H],AX    ; Write to Source x Address
        MOV     GS:[+72H],AX    ; Write to Source y Address
        MOV     GS:[+78H],SI    ; Write to Destination x Address
        MOV     GS:[+7AH],DI    ; Write to Destination y Address
```

```
; Store width in Operations Dimension 1 register
        MOV     GS:[+60H],CX    ; Write to Operation Dimension 1
; Store height in Operations Dimension 2 register
        MOV     GS:[+62H],DX    ; Write to Operation Dimension 2
;**********************|
; set up Pix Operation |
;      registers       |
;**********************|
; Bitmap of Pixel Operations register for pixBlt operation:
; byte 3 = bbss|pppp
;                     bb = background source
;                          00 = background color
;                     ss = foreground source
;                          00 = foreground color
;                     pppp = function
;                          1000 = pixBlt
;                     BYTE 3 = 00001000B = 08H
; byte 2 = SSSS|DDDD
;                     SSSS = source pixel map
;                          0010 = pixel map B
;                     DDDD = destination pixel map
;                          0001 = pixel map A
;                     BYTE 2 = 00100001B = 21H
; byte 1 = PPPP|0000
;                     PPPP = pattern pixel map
;                          0010 = pixel map B
;                     BYTE 1 = 00100000B = 20H
; byte 0 = mm00|0oox (* = values for this operation)
;                     mm = mask pixel map
;                          00 = mask map disabled
;                     oox = octant bits (x = don't care)
;                          00 = start at top left and move
;                               right and down
;                     BYTE 0 = 00000000B = 00H
;**********************|
;    execute pixBlt    |
;**********************|
; Coprocessor operation commences when data is written to the
; Pixel Operations register
        MOV     EAX,008212000H  ; Value from bitmap
        MOV     GS:[+7CH],EAX   ; Write to Pixel Operations
                                ; register
```

The procedure named COP_SYSVID_1 in the XGA2 module of the GRAPHSOL library can be used to perform a system memory to VRAM pixBlt operation. Processing and entry parameters are the same as in the above code fragment. The procedure named COP_SYSVID_8, also in the GRAPHSOL library, assumes an 8-bit per pixel encoding in the source bitmap. This last procedure can be used to display a memory stored image in 1,024 by 768 pixels in 256 colors.

12.5.4 Line Drawing Operations

The XGA draws a straight line following a method originally described by J.E. Bresenham (*IBM Systems Journal*, 1965) and since known as Bresenahm's algorithm. Bresenham's method is based on the differential equation for the slope of a straight line, which states that the difference between the y coordinates divided by the difference between the x coordinates is a constant. This constant, usually called the slope, is designated by the letter m. The formula is:

$$m = \frac{Dy}{Dx}$$

where Dy is the difference between the y values and Dx the difference between the x values. Therefore y can be expressed as a function of x, as follows:

$$y = mx$$

Bresenham's algorithm, as implemented on XGA, requires that all parameters be normalized to the first octant (octant number 0). Figure 12-10 shows the octant numbering in the Cartesian plane.

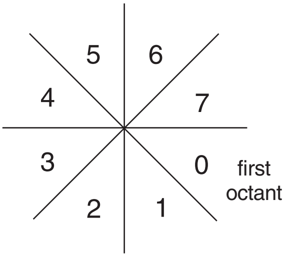


Figure 12-10 *Octant Numbering in the Cartesian Plane*

Reduction to the First Octant

The octant is selected by the octant field bits in the Pixel Operations register (see Figure 12-9). The1-bit values designated DX, DY, and DZ have the following meaning:

1. DX encodes the direction of the x values in reference to the line's start point. DX = 0 if x is in the positive direction and DX = 1 if it is in the negative direction.

2. DY encodes the direction of the y values in reference to the line's start point. DY = 0 if y is in the positive direction and DY = 1 if it is in the negative direction.

3. DZ encodes the relation between the absolute value of the x and y coordinates. DZ = 0 if $|x| > |y|$, and DZ = 1 otherwise.

The following rules allows normalizing any line defined by its start and end points to the first octant:

1. If the end x coordinate is smaller than the start x coordinate set the DX bit in the Pixel Operations register.

2. If the end y coordinate is smaller than the start y coordinate set the DY bit in the Pixel Operations register.

3. If the difference between the y coordinates is greater than or equal to the difference between the x coordinates set the DZ bit in the Pixel Operations register.

4. After the octant bits DX, DY, and DZ are set according to the above rules, the code can use the unsigned difference between y coordinates (delta y or Dy) and the unsigned difference between x coordinates (delta x or Dx) in the remaining calculations.

Calculating the Bresenham Terms

Three coprocessor registers are use to encode values that result from applying Bresenham's algorithm, these are the Bresenham Error Term register (offset 20H), the Bresenham K1 Term register (offset 24H), and the Bresenham K2 Term register (offset 28H).

The Bresenham K1 constant is calculated by the formula:

$$Term\ K1 = 2 \times Dy$$

Recall that Dy is the absolute difference between y coordinates, and Dx the absolute difference between x coordinates. The Bresenham K2 constant is calculated by the formula:

$$Term\ K2 = 2 \times (Dy - Dx)$$

Finally, the Bresenham error term is calculated by the formula:

$$Term\ E = (2 \times Dy) - Dx$$

The Bresenham terms are entered into the corresponding coprocessor registers (see Table 12-6). The Operation Dimension 1 register (at offset 60H) is loaded with the value of Dx. The following code fragment shows the necessary processing for drawing a straight line using the XGA coprocessor.

```
; Use graphics coprocessor to draw a straight line
; Code assumes XGA 1024 by 768 mode in 256 colors (8 bits per
; pixel)
; At this point:
;         CX = x pixel coordinate of line start
;         DX = y pixel coordinate of line start
;         SI = x pixel coordinate of line end
;         DI = y pixel coordinate of line end
;         BL = 8-bit color code
;         segment register setting:
;         GS = Coprocessor base address (Section 12.4.1)
;         FS = VRAM base address (Section 12.4.2)
;
;**********************|
;   test for not busy  |
;**********************|
        CALL    COP_RDY         ; Routine developed in
                                ; Section 12.4.2
; At this point the coprocessor is not busy
;**********************|
;   prepare to draw    |
;**********************|
; Prime coprocessor registers
        MOV     AL,01H          ; Data value for Map A
        MOV     GS:[+12H],AL    ; Write to pixel map index
        MOV     AX,0H           ; Data value for VRAM low
        MOV     GS:[+14H],AX    ; Write to pix map base address
; FS register holds the high order word of VRAM address. This
; value is calculated by the INIT_COP routine in this module
        MOV     AX,FS           ; Data for VRAM high
        MOV     GS:[+16H],AX    ; Write to pix map segment
                                ; address
; Code assumes 1024 by 768 pixel mode and Intel format
        MOV     AX,1023         ; Value for pix map width
        MOV     GS:[+18H],AX    ; Write to Width register
        MOV     AX,767          ; Value for pix map height
        MOV     GS:[+20H],AX    ; Write to Height register
        MOV     AL,3            ; Select Intel order and 8 bits
                                ; per pixel
        MOV     GS:[+1CH],AL    ; Write to Format register
;*********************|
;     mix, color and  |
;      coordinates    |
;*********************|
        MOV     AL,03H          ; Select source mix mode
        MOV     GS:[+48H],AL    ; Write to Mix register
; Write color (in BL) to Foreground register
        MOV     GS:[+58H],BL    ; Write to Foreground Color
                                ; register
; Write coordinates of line start point to coprocessor registers
        MOV     GS:[+78H],CX    ; Write to Destination x Address
                                ; register
        MOV     GS:[+7AH],DX    ; Write to Destination y Address
                                ; register
;*********************|
;  reduce to octant 0 |
;*********************|
;        CX = x pixel coordinate of line start
;        DX = y pixel coordinate of line start
;        SI = x pixel coordinate of line end
;        DI = y pixel coordinate of line end
; Octant bits in Pixel Operations register as follows:
; xxxx x210
;       |||_______ DZ bit = 0 if |x| > |y|
;       ||________ DY bit = 0 if y is positive (DI >= DX)
;       |_________ DX bit = 0 if x is positive (SI >= CX)
; BL will hold octant bits
        MOV     BL,0            ; Clear Octant register
        CMP     SI,CX           ; Test for DX bit
        JGE     DX_ISOK         ; Go if horizontal line
; At this point SI < CX, therefore DX bit must be set
        OR      BL,00000100B    ; DX bit is now set in BL
        XCHG    SI,CX           ; Exchange so that CX > SI
DX_ISOK:
; Now test DX bit condition
        CMP     DI,DX           ; Test for DY bit
        JGE     DY_ISOK         ; Go if horizontal line
; At this point DI < DX, therefore DY bit must be set
        OR      BL,00000010B    ; DY bit is now set in BL
        XCHG    DI,DX           ; Exchange so that DX > DI
; Now test DX bit condition
DY_ISOK:
        SUB     DI,DX           ; Find |y|
        XCHG    DX,DI           ; |y| to DX
        SUB     SI,CX           ; and |x|
        XCHG    CX,SI           ; |x| to CX
        CMP     CX,DX           ; Is |x| > |y|
        JG      BRZ_TERMS       ; Go to leave DZ = 0
; At this point |x| <= |y|, therefore DZ bit must be set
; and |y| must be exchanged with |x|
        OR      BL,00000001B    ; Set DZ bit
        XCHG    CX,DX           ; Exchange
;**********************|
;   Bresenham terms    |
;     calculations     |
;**********************|
BRZ_TERMS:
; Bresenham terms:
;     Term E (error) = (2 * |y|) - |x|
;     Term K1        = 2 * |y|
;     Term K2        = 2 * (|y| - |x|)
; AT this point CX = |x| and DX = |y|
; First store |x| in Operations Dimensions register
        MOV     GS:[+60H],CX    ; Write to Operation Dimension 1
                                ; register
; Then calculate Term E
        PUSH    DX              ; Save |y|
        ADD     DX,DX           ; 2 * |y|
        SUB     DX,CX           ; - |x|
        MOV     SI,DX           ; Store Term E in SI
        POP     DX              ; Restore |y|
        PUSH    CX              ; and save |x|
        MOV     CX,DX           ; |y| to CX
        ADD     CX,CX           ; Calculate 2 * |y|
        MOV     DI,CX           ; Store Term K1 in DI
        POP     CX              ; Restore |x| from stack
        SUB     DX,CX           ; |y| - |x|
        ADD     DX,DX           ; times 2
; DX = Term K2
        MOV     GS:[+20H],SI    ; Write to Error Term register
        MOV     GS:[+24H],DI    ; Write to K1 register
        MOV     GS:[+28H],DX    ; Write to K2 register
; Bitmap of Pixel Operations register:
; byte 3 = 0000|0101 = line draw write operation
; byte 2 = 0001      = source pixel map is map A
;               0001 = destination pixel map is map A
; byte 1 = 1000|rrrr = special code for foreground and all 1s
; byte 0 = 00   0    = Mask map disabled
;            00      = Drawing mode for all pixels drawn
;                      OCTANT DATA:
;                0   = DX = 0 for x in positive direction
;                 0  = DY = 0 for y in positive direction
;                  0 = DZ = 0 for |x| > |y|
;**********************|
;   execute operation  |
;**********************|
        MOV     EAX,05118000H   ; All bits except octant
; BL holds octant bits
        OR      AL,BL           ; OR-in octant bits
        MOV     GS:[+7CH],EAX   ; Write to Pixel Operations
                                ; register
```

12.6 The XGA Sprite

Many graphics programs, at both the system and the application level, must manipulate some sort of animated screen marker image. A typical example of screen marker is a mouse-controlled pointer or icon often used to facilitate selecting from option boxes or menus. Since the marker image overlays the screen, the software has to find some way of saving and restoring the screen contents as this image is translated over the pixel grid. In our discussion of animation techniques (see Chapter 31) we describe how the XOR operation is used in VGA graphics to display and erase an icon without affecting the screen contents. In XGA, the operation of a small screen pointer icon is considerably simplified thanks to a device called the sprite.

The XGA sprite mechanism consists of hardware elements designed to store and display a small graphics object. The sprite operation is independent of the video display function. The maximum size of the sprite image is 64 by 64 pixels. This image is stored in a 32K static RAM chip (which is not part of video memory) called the sprite buffer. This buffer is used for storing alphanumeric characters when XGA is in a VGA mode or in its proprietary 132-column text mode. The main advantage of the XGA sprite is that it does not affect the image currently displayed, therefore the XGA programmer need not worry about preserving the video image as the sprite is moved on the screen. This action can be best visualized as a transparent overlay that is moved over the picture without changing it. Figure 12-11 shows the structure of the sprite buffer.

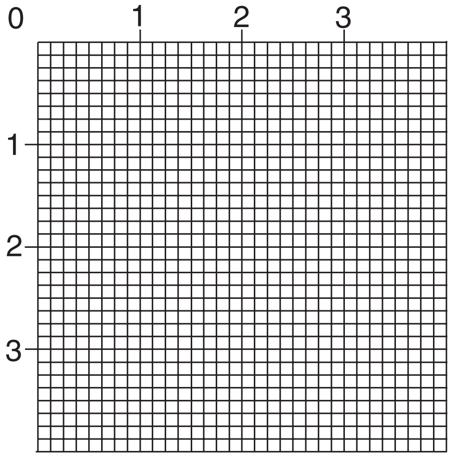


Figure 12-11 *XGA Sprite Buffer*

The XGA registers related to sprite image display and control are located in the indexed access registers of the Display Controller group (see Section 12.1.3). Table 12-10 lists the location and purpose of the sprite-related registers.

The displayed sprite can be smaller than 64 by 64 pixels. In this case the software controls which part of the sprite image is displayed by means of the Sprite Horizontal Preset (offset 32H) and Sprite Vertical Preset registers (offset 35H) in the Display Controller (see Table 12-10). However, the sprite image always extends to the full 64-bit length and width of the sprite buffer. Nevertheless, transparent sprite codes can be used to locate the sprite image within the pixel rectangle defined by the 64-byte sprite buffer. The elements used in controlling the size of the sprite image are shown in Figure 12-12.

Table 12-10

Sprite-Related Registers in the Display Controller

INDEX REGISTER OFFSET	REGISTER NAME
30H	Sprite horizontal start, low part
31H	Sprite horizontal start, high part
32H	Sprite horizontal preset
33H	Sprite vertical start, low part
34H	Sprite vertical start, high part
35H	Sprite vertical preset
36H	Sprite control register
38H	Sprite color 0, red component
39H	Sprite color 0, green component
3AH	Sprite color 0, blue component
3BH	Sprite color 1, red component
3CH	Sprite color 1, green component
3DH	Sprite color 1, blue component
60H	Sprite/palette index, low part
61H	Sprite/palette index, high part
62H	Sprite/palette prefetch, low part
63H	Sprite/palette prefetch, high part
6AH	Sprite data
6BH	Sprite prefetch save (RESERVED)

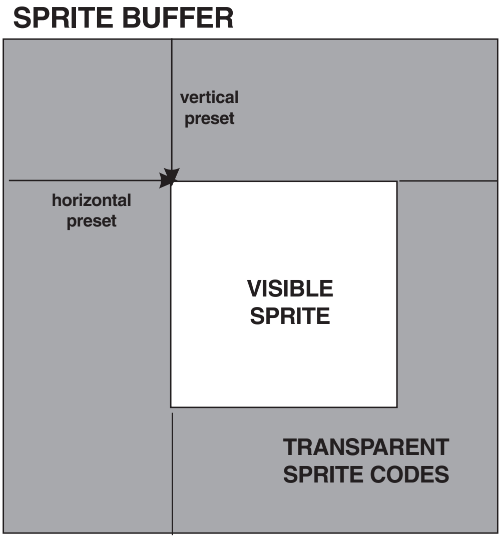


Figure 12-12 *Visible Sprite Image Control*

12.6.1 The Sprite Image

The sprite image consists of 64 by 64 pixels. Each sprite image pixel can have one of four attributes. The storage structure is in Intel data format and encoded in 2 bits per pixel. The bit codes for the sprite image is shown in Table 12-11.

Table 12-11

Sprite Image Bit Codes

BIT CODE	ACTION
00	Pixel displayed in sprite color 0
01	Pixel displayed in sprite color 1
10	Transparent (image pixel is visible)
11	Complement (one's complement of image pixel is visible)

The location of the sprite image within the viewport is determined by the Sprite Horizontal Start and Sprite Vertical Start registers (see Table 12-11). Both of these registers are word-size; however, the valid range of values is limited to 0 to 20,412. The low-order bit in the Sprite Control register (offset 36H) determines the sprite's visibility. The sprite is displayed when this bit is set and is invisible if the bit is cleared.

Encoding of Sprite Colors and Attributes

In Section 12.1 5 we mentioned that the sprite's attributes are coded into a 2-bit field. The first two codes refer to sprite color attributes, the third code defines a transparent attribute, and the last one a one's complement operation (see Table 12-10). The sprite colors 0 and 1 are determined by the setting in two sets of registers in the Display Controller group; registers 38H to 3AH select the red, green, and blue values of sprite color 0, while registers 3BH to 3DH select the same values in sprite color 1. In this manner, if the first byte in the sprite buffer is encoded with the value 01010101B, then the first 4 bits in the sprite are displayed using the color value for sprite color 1. Figure 12-13 shows how the sprite pixels are mapped to the binary values stored in the sprite buffer.

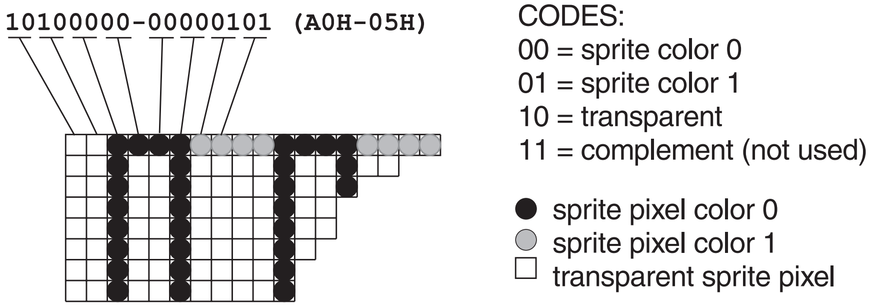


Figure 12-13 *Bit-to-Pixel Mapping of Sprite Image*

In summary, the attribute of each sprite pixel corresponds to the 2-bit code stored in the sprite buffer. Therefore, designing a sprite image is a matter of installing the red, green, and blue values for each sprite color and then composing a pixel map using the 2-bit values in Table 12-10. The Sprite Horizontal and Vertical Preset registers can be used to adjust a sprite image that does not coincide with the top-left corner of the map stored in the sprite buffer.

Loading the Sprite Image

Once the sprite map has been composed and stored in an application's memory variable, the software must proceed to set the sprite color registers and load the image into the sprite buffer. The following code fragment assumes that the sprite colors and bitmap have been placed in a formatted parameter block. From this data the sprite color values and image are loaded into the corresponding Display Controller registers.

```
DATA     SEGMENT
         .
         .
         .
;**********************|
;    sprite data       |
;**********************|
; The 64 by 64 pixel sprite is defined at 64 lines of 4
; doublewords per line
;
; First 6 bits of the sprite color are significant
; In this example color number 0 is bright red and color number 1
; is bright white
SPRITE_MAP_0    DB      11111100B       ; Red for color 0
                DB      0               ; Green for color 0
                DB      0               ; Blue for color 0
                DB      11111100B       ; Red for color 1
                DB      11111100B       ; Green for color 1
                DB      11111100B       ; Blue for color 1
; The 64 by 64 pixel sprite is defined as 64 lines of 4
; doublewords per line, encoded as follows:
; 00H = 00 00 00 00 B = 4 pixels in sprite color 0
; 55H = 01 01 01 01 B = 4 pixels in sprite color 1
; AAH = 10 10 10 10 B = 4 transparent pixels
; FFH = 11 11 11 11 B = 4 pixels in one's complement of image
;
                DD      256 DUP (0055AAFFH)
DATA      ENDS
;
CODE      SEGMENT
          .
          .
          .

; Load sprite image and select color registers
; On entry:
;         DS:SI --> caller's sprite image formatted as follows:
;
;   OFFSET    UNIT      CONTENTS
;     0       byte      6 low bits are RED for sprite color 0
;     1       byte      6 low bits are GREEN for sprite color 0
;     2       byte      6 low bits are BLUE for sprite color 0
;     3       byte      6 low bits are RED for sprite color 1
;     4       byte      6 low bits are GREEN for sprite color 1
;     5       byte      6 low bits are BLUE for sprite color 1
;     6       16 bytes per 64 rows (1024 bytes) encoding the
;             sprite image at 2 bits per pixel
;   1030      end of sprite image
;
; Code assumes that the variable XGA_REG_BASE holds the XGA
; register base address (see Section 12.2.1)
```

```
;**********************|
; set sprite color 0   |
;**********************|
; Load sprite color 0 registers using values in parameter block
; supplied by caller (DS:SI)
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 38H is Sprite Color 0, red value
        MOV     AL,38H          ; Sprite register
        MOV     AH,[SI]         ; Data from caller's buffer
        INC     SI              ; Bump pointer to next byte
        OUT     DX,AX           ; Write data
;
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 39H is Sprite Color 0, green value
        MOV     AL,39H          ; Sprite register
        MOV     AH,[SI]         ; Data from caller's buffer
        INC     SI              ; Bump pointer to next byte
        OUT     DX,AX           ; Write data
;
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 3AH is Sprite Color 0, blue value
        MOV     AL,3AH          ; Sprite register
        MOV     AH,[SI]         ; Data from caller's buffer
        INC     SI              ; Bump pointer to next byte
        OUT     DX,AX           ; Write data
;**********************|
; set sprite color 1   |
;**********************|
; Load sprite color 1 registers to GREEN
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 3BH is Sprite Color 1, red value
        MOV     AL,3BH          ; Sprite register
        MOV     AH,[SI]         ; Data from caller's buffer
        INC     SI              ; Bump pointer to next byte
        OUT     DX,AX           ; Write data
;
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To index register
; Index register 3CH is Sprite Color 1, green value
        MOV     AL,3CH          ; Sprite register
        MOV     AH,[SI]         ; Data from caller's buffer
        INC     SI              ; Bump pointer to next byte
        OUT     DX,AX           ; Write data
;
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 3DH is Sprite Color 1, blue value
        MOV     AL,3DH          ; Sprite register
        MOV     AH,[SI]         ; Data from caller's buffer
        INC     SI              ; Bump pointer to next byte
        OUT     DX,AX           ; Write data
;**********************|
;   prepare to load    |
;     sprite image     |
;**********************|
; First set the Sprite Index registers to zero
```

```
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 60H is Sprite/Palette index Low
        MOV     AX,0060H        ; 00 to register at offset 60H
        OUT     DX,AX           ; Write data
; Reset to base
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To index register
; Index register 61H is Sprite/Palette index High
        MOV     AX,0061H        ; 00 to register at offset 60H
        OUT     DX,AX           ; Write data
; Select the Sprite Data register at offset 6AH
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
        MOV     AL,06AH         ; Offset of Data register
        OUT     DX,AL           ; Select the Sprite Data register
;*********************|
;  load sprite image   |
;*********************|
; DS:SI --> buffer area containing the sprite bitmapped image in
;           2 bits per pixel format, as follows:
;           00 = sprite color 0
;           01 = sprite color 1
;           10 = transparent pixel
;           11 = complement pixel
        MOV     CX,512          ; Word item counter
SPRITE_DATA:
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0CH          ; To second Data register
        MOV     AX,[SI]         ; Get data from buffer
        OUT     DX,AX           ; Send to data port
        INC     SI              ; Bump data pointer
        INC     SI              ; to next word
        LOOP    SPRITE_DATA     ; Repeat 512 times
; At this point sprite color and image have been loaded
```

The procedure named SPRITE_IMAGE in the XGA2 module of the GRAPHSOL library loads the sprite image and colors using the same processing as in the above code fragment.

12.6.2 Displaying the Sprite

As mentioned in Section 12.5.1, if the low-order bit of the Sprite Control register is set, the sprite image is displayed on the video screen. The position at which it is displayed is determined by the setting of the Sprite Horizontal Start and Vertical Start registers (see Table 12-11). The following code fragment displays the sprite image at the screen coordinates supplied by the caller.

```
; Display sprite image at coordinates furnished by the caller
; as follows:
;         BX = x coordinate of sprite location (0 to 1023)
;         CX = y coordinate of sprite location (0 to 767)
; Code assumes that the variable XGA_REG_BASE holds the XGA
; register base address (see Section 12.2.1)
;
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 30H is Sprite x Start LOW register
        MOV     AH,BL           ; Value to Start register
        MOV     AL,30H          ; Address of Start x Low
        OUT     DX,AX           ; Write data
;
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 31H is Sprite x Start HIGH register
        MOV     AH,BH           ; Value to start register
        MOV     AL,31H          ; Address of Start register
        OUT     DX,AX           ; Write data
; Set Sprite x Preset register to 0
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 32H is Sprite x Preset register
        MOV     AH,00           ; Value to preset register
        MOV     AL,32H          ; Address of Start register
        OUT     DX,AX           ; Write data
; Select y coordinate registers
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 33H is Sprite y Start LOW register
        MOV     AH,CL           ; Value to start register
        MOV     AL,33H          ; Address of Start x Low
        OUT     DX,AX           ; Write data
;
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 34H is Sprite y Start HIGH register
        MOV     AH,CH           ; Value to Start register
        MOV     AL,34H          ; Address of Start register
        OUT     DX,AX           ; Write data
; Set Sprite y Preset register to 0
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 35H is Sprite x Preset register
        MOV     AH,00           ; Value to preset register
        MOV     AL,35H          ; Address of Start register
        OUT     DX,AX           ; Write data
;*********************|
;    display sprite   |
;*********************|
; Sprite is displayed by setting bit 0 of the Sprite Control
; register at offset 36H
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Sprite control register offset is 36H
        MOV     AH,01           ; Value to start register
        MOV     AL,36H          ; Address of Start register
        OUT     DX,AX           ; Write data
; At this point the sprite has been displayed
```

The procedure named SPRITE_AT in the XGA2 module of the GRAPHSOL library displays the sprite using the same processing as in the above code fragment. To turn off the sprite the program need only clear the low-order bit in the Sprite Control Register. This operation is performed by the SPRITE_OFF procedure in the XGA2 module of the GRAPHSOL library.

12.7 Using the XGA Library

The GRAPHSOL library furnished in the book's software package includes two modules that contain XGA specific routines: XGA1.ASM and XGA2.ASM. XGA1.ASM contains procedures that use the AI services described in Chapter 6. The purpose of this module is to simplify initializing the XGA system and the AI software as well as to facilitate the use of AI text services. The module XGA2.ASM of the XGA library contains routines that access the XGA registers directly. These procedures serve to initialize the XGA system, to select the display mode, to set an individual screen pixel using the 1,024 by 768 pixel definition in 256 colors,to perform the display of geometrical figures, and to load and manipulate the sprite.

In addition to the routines in the GRAPHSOL library, XGA programs can also use several procedures in the VGA modules of GRAPHSOL.L-IB. The use of the VGA procedures by an XGA system requires a previous call to the SET_DEVICE routine in the VGA3 module. For XGA systems this call is made with the AL register holding the character "X." The call sets a device-specific display switch to the XGA display routine. This enables the use of several geometrical display routines in the VGA3 module, including the named BRESENHAM, LINE_BY_SLOPE, DISPLAY_LINE, CIRCLE, ELLIPSE, PARABO-LA, and HYPERBOLA. Also the following procedures in the VGA2 module: FINE_TEXT, FINE_TEXTHP, and MULTITEXT, as well as the corresponding text display support routines, such as FONT_TO_RAM and READ_HPFONT. Information regarding the VGA text display and geometrical routines can be found in Chapters 4 and 5 as well as in the source files VGA2.ASM and VGA3.ASM contained in the book's software.

12.7.1 Procedures in the XGA1.ASM Module

OPEN_AI

Initialize Adapter Interface software.

```
Receives:

          Nothing

Returns:

         Carry clear if AI initialized

         Carry set if error
```

CLOSE_AI

Erase video and close Adapter Interface.

```
Receives:

          Nothing

Returns:

         Nothing
```

AI_FONT

Read an XGA or 8514/A font file into RAM to enable text display using AI functions.

```
Receives:
         Far pointer to ASCIIZ filename for font file
         (must be in the current path)
Returns:
        Carry clear if font read and stored in buffer
        Carry set if file not found or disk error
```

AI_COLOR

Set foreground and background colors for AI services.

```
Receives:
         1. Byte integer of foreground color
         2. Byte integer of background color
Returns:
         Nothing
Action:
         Foreground and background colors selected
```

AI_CLS

Clear screen using AI service.

```
Receives:
         Nothing
Returns:
        Nothing
Action:
        Video display is cleared
```

AI_TEXT

Display a text message on XGA screen using an AI service.

```
Receives:
         1. word integer of x pixel coordinate for message
         2. word integer of y pixel coordinate for message
         3. byte integer of foreground color
         4. byte integer of background color
         5. far pointer to text message
Returns:
         Nothing
Action:
         Text message is displayed
```

AI_PALETTE

Initialize 256 DAC color registers from a 4-byte per color table using an AI service.

Receives:

1. far pointer to 1024 byte table of palette colors
   (4 bytes per color encoding)

Returns:

Nothing

Action:

LUT registers are set according to value table furnished by the caller.

AI_COMMAND

Access the services in the XGA/8514-A Adapter Interface.

Receives:

1. word integer of AI service number
2. far pointer to parameter block

Returns:

Nothing

Action:

AI command is executed.

12.7.2 Procedures in the XGA2.ASM Module

XGA_MODE

Initialize an XGA graphics mode by setting the video system registers directly.

Receives:

1. byte integer of XGA mode number

Assumes:

INIT_XGA has been previously called

Returns:

1. carry clear if no error
2. carry set if invalid mode

Action:

XGA system is set to mode requested by the caller. Valid range is 2 to 5, as follows:

Mode number:	Resolution:	Colors:
2	1024 by 756	256
3	1024 by 768	16
4	640 by 480	256

```
        5              640 by 480       65,536
```

INIT_XGA

Initialize XGA registers and report machine setup.

```
Receives:

         Nothing

Returns:

         1. byte integer of machine setup, as follows:

            7 6 5 4 3 2 1 0

            | | | | | | | |___ 1 = XGA in system
            | | | | | | |      0 = no XGA found
            | | | | | | |_____ 1 = XGA color monitor
            | | | | | |        0 = XGA monochrome monitor
            | | | | | |_______ 1 = high resolution (1024 x 768)
            | | | | |          0 = no high resolution
            | | | | |_________ 1 = RAM = 1Mb
            | | | |            0 = RAM = 512Kb
            | | | |___________ 1 = dual monitor system
            | | |              0 = single monitor system
            |_|_|_____________ UNUSED


Action:

        XGA system is initialized and setup is tested. This initialization
is required by many other procedures in this module.
```

XGA_PIXEL_2

Write a screen pixel accessing XGA memory directly while in XGA mode number 2.

```
Receives:

         1. word integer of x coordinate of pixel

         2. word integer of y coordinate of pixel

         3. byte integer of pixel color in 8-bit format

Assumes:

        INIT_XGA has been previously called

Returns:

         Nothing

Action:

         Pixel is set
```

XGA_CLS_2

Clear video memory while in XGA mode number 2 using block move.

```
Receives:

         Nothing

Assumes:

        INIT_XGA has been previously called

Returns:

        Nothing

Action:

        Direct access version of the AI_CLS procedure in the XGA1.ASM
module.
```

XGA_OFF

Turn off XGA video by clearing the Palette Mask register.

```
Receives:

         Nothing

Assumes:

        INIT_XGA has been previously called

Returns:

        Nothing

Action:

        XGA display is disabled
```

XGA_ON

Turn on XGA video by setting the Palette Mask register.

```
Receives:

         Nothing

Assumes:

        INIT_XGA has been previously called

Returns:

        Nothing

Action:

        XGA display is enabled
```

XGA_PALETTE

Load 256 XGA LUT color registers with data supplied by the caller.

```
Receives:

         1. Far pointer of 1024-byte color table in RGBx format

Assumes:
        INIT_XGA has been previously called
Returns:
         Nothing
Action:
         LUT registers are initialed to supplied values. Caller's data to be
formatted in red, blue, green, ignored, pattern.
```

DC_PALETTE

Set 256 XGA palette registers for the direct color mode using values recommended by IBM.

```
Receives:
         Nothing
Assumes:
        INIT_XGA has been previously called
Returns:
        Nothing
Action:
        XGA palette registers are initialized for mode number 5, in 65,536
colors.
```

INIT_COP

Initialize XGA coprocessor. This procedure assumes that the procedure INIT_XGA (in this module) has been previously called and that the POS_x variables have been loaded.

```
Receives:
         Nothing
Returns:
        1. GS = coprocessor base address
        2. FS = base address of video memory (VRAM)
Action:
        Coprocessor is initialized. GS and FS segment registers are set for
calling the coprocessor commands in this module.
```

COP_RECT_2

Graphics coprocessor pixBlt operation on a rectangular screen area.

```
Receives:
         1. word integer of x coordinate of top-left corner
         2. word integer of y coordinate of top-left corner
         3. word integer of rectangle's pixel width
         4. word integer of rectangle's pixel height
         5. byte integer of 8-bit color code
```

```
Assumes:
         1. Mode number 2 (1024 by 768 pixels in 256 colors)
         2. GS and FS segment set by INIT_COP procedure
Returns:
         Nothing
Action:
         Rectangular pixBlt is performed
```

COP_SYSVID_1

Graphics coprocessor pixBlt operation from a source in system memory to a destination in video memory, using an image map encoded in 1-bit per pixel format.

```
Receives:
         1. far pointer to source bitmap in RAM
         2. word integer of pixel map width
         3. word integer of pixel map height
         4. word integer of x coordinate for display
         5. word integer of y coordinate for display
         6. byte integer of 8-bit color value
Assumes:
         1. Mode number 2 (1024 by 768 pixels in 256 colors)
         2. GS and FS segment set by INIT_COP procedure
Returns:
         Nothing
Action:
         PixBlt is performed.
```

COP_SYSVID_8

Graphics coprocessor pixBlt operation from a source in system memory to a destination in video memory, using an image map encoded in 8-bit per pixel format.

```
Receives:
         1. far pointer to source bitmap in RAM
         2. word integer of pixel map width
         3. word integer of pixel map height
         4. word integer of x coordinate for display
         5. word integer of y coordinate for display
Assumes:
         1. Mode number 2 (1024 by 768 pixels in 256 colors)
         2. GS and FS segment set by INIT_COP procedure
```

```
Returns:
         Nothing
Action:
         PixBlt is performed.
```

COP_LINE_2

Draw a line using XGA graphics coprocessor, while in mode number 2 (1,024 by 768 pixels in 256 colors).

```
Receives:
         1. word integer of x coordinate of line start
         2. word integer of y coordinate of line start
         3. word integer of x coordinate of line end
         4. word integer of y coordinate of line end
         5. byte integer of 8-bit color value
Assumes:
         1. Mode number 2 (1024 by 768 pixels in 256 colors)
         2. GS and FS segment set by INIT_COP procedure
Returns:
         Nothing
Action:
         Line is drawn.
```

SPRITE_IMAGE

Load sprite image and install values in Sprite Color registers.

```
Receives:
         1. far pointer to color code and image buffer, formatted
            as follows:
     OFFSET    UNIT      CONTENTS
       0       byte      6 low bits are RED for sprite color 0
       1       byte      6 low bits are GREEN for sprite color 0
       2       byte      6 low bits are BLUE for sprite color 0
       3       byte      6 low bits are RED for sprite color 1
       4       byte      6 low bits are GREEN for sprite color 1
       5       byte      6 low bits are BLUE for sprite color 1
       6       16 bytes per 64 rows (1024 bytes) encoding the
               sprite image at 2 bits per pixel
     1030      end of sprite image
Assumes:
        INIT_XGA has been previously called.
Returns:
        Nothing
Action:
        Sprite image and colors codes are stored in Display Controller
registers
```

SPRITE_AT

Display sprite image at coordinates furnished by the caller.

```
Receives:
        1. word integer of x coordinate of sprite location
           (range is 0 to 1023)
        2. word integer of y coordinate of sprite location
           (range is 0 to 768)
Assumes:
        INIT_XGA has been previously called.
Returns:
        Nothing
Action:
        Sprite is displayed at entry coordinates.
```

SPRITE_OFF

Sprite is turned off by clearing bit 0 of the Sprite Control register.

```
Receives:
        Nothing
Assumes:
        INIT_XGA has been previously called.
Returns:
        Nothing
Action:
        Sprite image is no longer displayed
```

Chapter 13

SuperVGA Programming

Topics:

- SuperVGA programming
- Introducing SuperVGA
- The VESA SuperVGA standard
- The VESA BIOS services
- Programming the SuperVGA system
- The book's SuperVGA Library

This chapter describes the SuperVGA video hardware and its architecture, the VESA SuperVGA standards and the use of the various VESA BIOS Services, also programming the SuperVGA system by accessing the video hardware directly and by the use of the VESA BIOS services. The chapter concludes with a listing of the procedures in the SVGA library furnished with the book.

13.1 Introducing the SuperVGA Chipsets

The term SuperVGA refers to enhancements to the standard VGA modes as furnished in some non-IBM adapters developed for PC compatible computers. The common characteristic of all SuperVGA boards is the presence of graphics features that exceed the VGA standard in definition or color range. In other words, a SuperVGA graphics board is capable of executing, not only the standard VGA modes, but also other modes that provide higher definition or greater color range than VGA. These are usually called the SuperVGA Enhanced Modes.

In the late seventies the proliferation of SuperVGA hardware gave rise to many compatibility problems, due to the fact that the enhanced features of the SuperVGA cards were not standardized; therefore the SuperVGA enhancements in the card produced by one manufacturer were incompatible with the enhancements in a card made by another company. This situation often presented unsurmountable problems to the graphcs application programmer, who would find that an application designed to take advantage of the graphics enhancements in a SuperVGA card would not execute correctly in another one.

At the operating system level these incompatibility problems are easier to correct than at the application level. For example, the manufacturers of SuperVGA boards often furnish software drivers for Windows and Operating System/2. Once the driver is installed, the graphics environment in the operating system will be able to use the enhancements provided by a particular SuperVGA board. By the same token, applications that perform graphics functions by means of operating system services will also take advantage of the SuperVGA enhancements.

On the other hand, graphics applications that control the hardware directly would not be able to take advantage of a system-level driver. Fortunately, some graphics programs are designed with a flexible video interface. In this case, the application software can be more easily adapted to the features of a particular SuperVGA. This is the case with AutoCad, Ventura Publisher, Wordperfect, Lotus 1-2-3, and other high-end graphics applications for the PC. But, for those applications in which the video functions are embedded in the code, the adaptation to a non-standard video mode often implies a major program redesign.

In 1989, in an attempt to solve this lack of standardization, several manufacturers of SuperVGA boards formed the Video Electronics Standards Association (VESA). Most of the companies listed at the beginning of this section are now members of VESA. In October of 1989 VESA released its first SuperVGA standard. The VESA standard defined several enhanced video modes and implemented a BIOS extension designed to provide a few fundamental video services in a compatible fashion. Because of this advantage in compatibility and portability, our treatment of SuperVGA programming focuses on the use of the VESA BIOS functions.

13.1.1 SuperVGA Memory Architecture

In previous chapters we saw that the IBM microcomputer video systems are memory mapped. In VGA the video memory space extends from A0000H to BFFFFH. The 64K space starting at segment base A000H is devoted to graphics and the 64K space starting at segment base B000H is for alphanumeric modes. This means that the total space reserved for video operations is of 128K. But since some systems are set up with two monitors, one operating in alphanumeric modes (base address B000H for monochrome systems and B800H for color systems), the actual video space for graphics operations is practically limited to 64K.

Not much video data can be stored in a 64K memory space. For example, if each screen pixel is encoded in 1 memory byte, then the maximum screen data that can be stored in 65,536 bytes is 256 square pixels. In Chapter 2 we saw that the VGA screen in 640 by 480 pixels resolution requires 307,200 bytes, we also show how the VGA designers were able to compress video data by implementing a latching scheme and a planar architecture. Consequently, in VGA mode number 18 a pixel is encoded into a single memory bit, although it can be displayed in 16 different colors. The latching mechanism (see Figure 2.4) is based on four memory maps of 38,400 bytes each. All four color maps (red, green, blue, and intensity) start at segment base

A000H. The pixel displayed is determined by the value stored in the Bit Mask register of the VGA Graphics Controller group (see Section 2.2.4).

16 Color Extensions

Simple arithmetic shows a memory surplus in many VGA modes. For example, if the resolution is of 640-by-480 pixels, the video data stored in each map takes up 38,400 bytes of the available 65,536. Therefore there are 27,136 unused bytes in each map. The original idea of enhancing the VGA system was based on using this surplus memory to store video data. It is clearly possible to have an 800-by-600 pixel display divided into four maps of 60,000 bytes each, and yet not exceed the 64K space allowed for each color map nor the total 265K furnished with the VGA system.

The 800-by-600 pixel resolution in 16 colors appears as a natural extension to VGA mode number 113. This mode, which was later designated as mode 6AH by the VESA standards, could be programmed in a similar manner as mode number 113. This extension, which could be achieved with minor changes in the VGA hardware, provided a 36 percent increase in the display area.

Another extension to the VGA system is a wider pixel mask register to make possible more than the 16 colors that can be encoded in a 4-bit field. However, this has never been implemented in a SuperVGA system due to performance factors and other hardware considerations.

Memory Banks

In Chapter 7 we saw that the memory structure for VGA 256-color mode number 19 is based not on a the multiplane scheme, but in a much simpler format that maps a memory byte to each screen pixel (see Figure 2.5). In this manner, 256 color combinations can be directly encoded into a data byte, which correspond to the 256 DAC registers of the VGA hardware. The method is straightforward and uncomplicated; however, if the entire video space is to be contained in 64K of memory the maximum resolution would be limited to the 256 square pixels previously mentioned. In other words, a rectangular screen of 320 by 200 pixels nearly fills the allotted 64K.

Therefore, if the resolution for a 256-color mode were to exceed 256 square pixels it would be necessary to find other ways of mapping video memory into 64K of system RAM. The mechanism adopted by the SuperVGA designers was based on the well-known technique known as bank switching. In a bank switching scheme the video display hardware maps several 64K blocks of RAM to different locations in video memory. Addressing of the multi-segment space is by means of a hardware mechanism that selects which video memory area is currently located at the system's aperture. In the SuperVGA implementation the system aperture is usually located at segment base A000H. The entire process is reminiscent of memory page switching in the LIM (Lotus/Intel/Microsoft) Extended Memory environment. Figure 13-1, on the following page, schematically shows mapping of several memory banks to the video space and the map selection mechanism for CPU addressing.

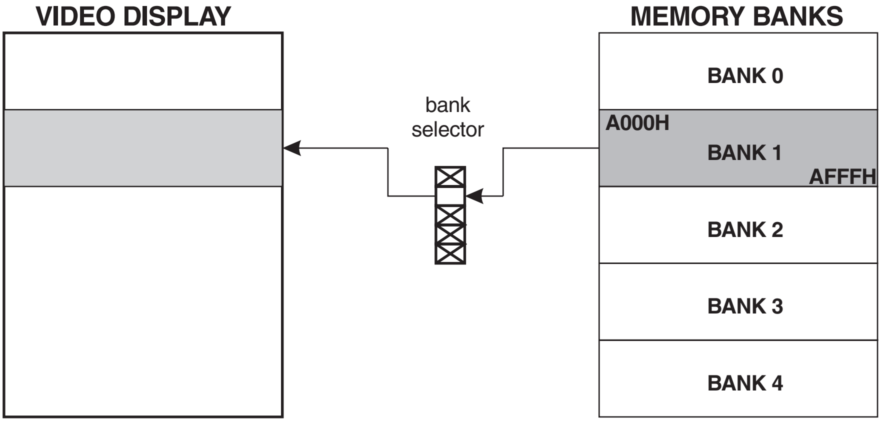


Figure 13-1 *Memory Banks to Video Mapping*

In Chapter 12 we adopted the term aperture from the XGA terminology, which is used to denote the processor's window into video memory. For example, if the addressable area of video memory starts at physical address A0000H and extends to B0000H, we say that the CPU has a 64K aperture into video memory (10000H = 64K). In SuperVGA documentation the word "granularity" is often used in this context. In Figure 13-1 we can see that the bank selector determines which area of video memory is mapped to the processor's aperture. Therefore, the area of the video display can be updated by the processor. In other words, in the memory banking scheme the processor cannot access the entire video memory at once. In Figure 13-1 we can see that we would have to perform 5 bank switches in order to update the entire screen.

256 Color Extensions

The SuperVGA alternative for increasing definition beyond the VGA limit is a banking mechanism similar to the one shown in Figure 13-1. This scheme, in which a memory byte encodes the 256 color combinations for each screen pixel, does away with the pixel masking complications of VGA mode number 113. On the other hand, it introduces the complications of a bank selection device which we already encountered in XGA programming (see Section 7.1.2). The SuperVGA method has no precedent in CGA, EGA, or VGA systems since it is not interleaved nor does it require memory planes or pixel masking. Although it is similar to VGA mode number 19 regarding color encoding, mode number 19 does not require bank switching.

It should be noted that the neat, rectangular window design shown in Figure 13-1 does not always conform with reality. Several implementations of SuperVGA multi-color modes use non-rectangular windows that start and end inside a scan line. This complicates the use of optimizing routines since the software cannot restrict its checking for a window boundary to the start and end of scan lines.

Pixel Addressing

The calculations required for setting an individual pixel in the 256 color modes depend upon the size of the memory banks, the number of pixels per row and of screen rows, and the start address of video memory. Although it is quite feasible to design a routine that performs in different SuperVGA chipsets, the efficiency of such coding would be necessarily low. The VESA standardization offers a solution to the programming complications brought on by different architectures of the various SuperVGA chipsets. In reality, since most SuperVGA systems use a 64K bank size and a processor's window into video memory located at segment base A000H, the variations are reduced to the bank switching operations.

13.2 The VESA SuperVGA Standard

The Video Electronics Standards Association was founded in 1989 with the intention of providing a common programming interface for SuperVGA extended modes. In order to achieve this, each manufacturer furnishes a VESA SuperVGA BIOS extension. The BIOS can be in the adapter ROM or in a TSR routine. Today, most SuperVGA manufacturers are members of VESA and provide a VESA BIOS with their products.

The first release of the VESA SuperVGA standard was published on October 1, 1989 (version 1.0). A second release was published on June 2, 1990 (version 1.1). The present release is dated Sepetember 16, 1998 (version 3.0). The latest version of the standard supports non-VGA systems, flat memory models, and 32-bit operating systems and applications.

13.2.1 VESA SuperVGA Modes

The first element of VESA standardization is the definition of standard modes for the SuperVGA extensions. The VESA mode numbering scheme takes into account that the VGA modes are in the range 0 to 7FH. This range limitation is due to the fact that the VGA BIOS mode setting function (service number 0) uses the high-order bit to determine if video memory is to be cleared. To get around this restriction, the VESA mode number is a word-size value, which is passed to the VESA BIOS in the BX register. Figure 13-2 shows the bitmap of the VESA MODE numbers.

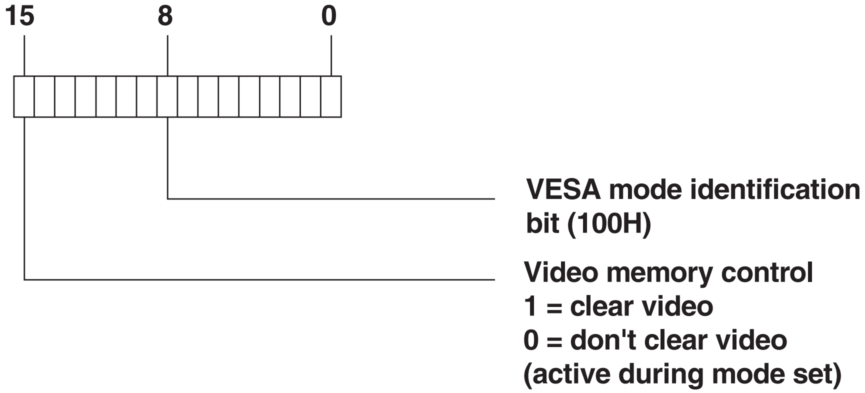


Figure 13-2 *VESA Mode Bitmap*

Notice that bit number 8 identifies a VESA mode. That is, all VESA modes start at number 100H. Notice also that bit number 15 is used during mode set operations to indicate if video memory is to be cleared. Table 13-1 lists the VESA extended modes.

Table 13-1

VESA BIOS Modes

MODE NUMBER		TEXT/	RESOLUTION		
15 BITS	7 BITS	GRAPHICS	PIXELS	COLUMNS/ROWS	COLORS
100H		GRAPHICS	640 by 400		256
101H		GRAPHICS	640 by 480		256
102H	6AH	GRAPHICS	800 by 600		16
103H		GRAPHICS	800 by 600		256
104H		GRAPHICS	1024 by 768		16
105H		GRAPHICS	1024 by 768		256
106H		GRAPHICS	1280 by 1024		16
107H		GRAPHICS	1280 by 1024		256
108H		TEXT		80 by 60	
109H		TEXT		132 by 25	
10AH		TEXT		132 by 43	
10BH		TEXT		132 by 50	
10CH		TEXT		132 by 60	
10DH*		GRAPHICS	300 by 200		32K
10EH		GRAPHICS	320 by 200		64K
10FH		GRAPHICS	320 by 200		16.8Mb
110H		GRAPHICS	640 by 480		32K
111H		GRAPHICS	640 by 480		64K
112H		GRAPHICS	640 by 480		16.8Mb
113H		GRAPHICS	800 by 600		32K
114H		GRAPHICS	800 by 600		64K
115H		GRAPHICS	800 by 600		16.8Mb
116H		GRAPHICS	1024 by 768		32K
117H		GRAPHICS	1024 by 768		64K
118H		GRAPHICS	1024 by 768		16.8Mb
119H		GRAPHICS	1280 by 1024		32K
11AH		GRAPHICS	1280 by 1024		64K
11BH		GRAPHICS	1280 by 1024		16.8Mb

Legend:
* modes after 10DH were introduced in VESA BIOS version 1.2

13.2.2 Memory Windows

The VESA standard accommodates variations in the SuperVGA implementations by recognizing two different types of hardware windows into video memory. The first and simpler type consists of a single window which can be read and written by the CPU. The disadvantage of a read-write window becomes evident when a pixBlt operation crosses the limit of this window, because, in this case, the software is forced to switch banks and the CPU to reset the segment register base during the transfer. This double burden can considerably degrade performance.

A partial solution is to provide separate windows for read and write operations. One possible option is to have two windows located at the same address: one for read and the other one for write operations. This scheme, sometimes called dual overlapping windows, allows selecting both windows simultaneously. Once the source and destination windows are selected, the data block can be rapidly moved by means of a REP MOVSB instruction.

A second alternative to the two windows option is to locate the read and write windows at separate addresses. For example, a SuperVGA chipset can locate the write window at base address A000H and the read window at base address B000H. This would extend addressable memory to 128K and considerably simplify pixBlt operations. The objection to this approach is that a two-monitor system requires the B000H window for text operations; therefore this configuration would not be possible. Another solution is to cut the 64K window in half and provide separate 32K windows, one for read and the other one for write operations. The objection in this case is that normal display operation would require twice as many bank switches. Figure 13-3 is a schematic representation of the three possible windowing options.

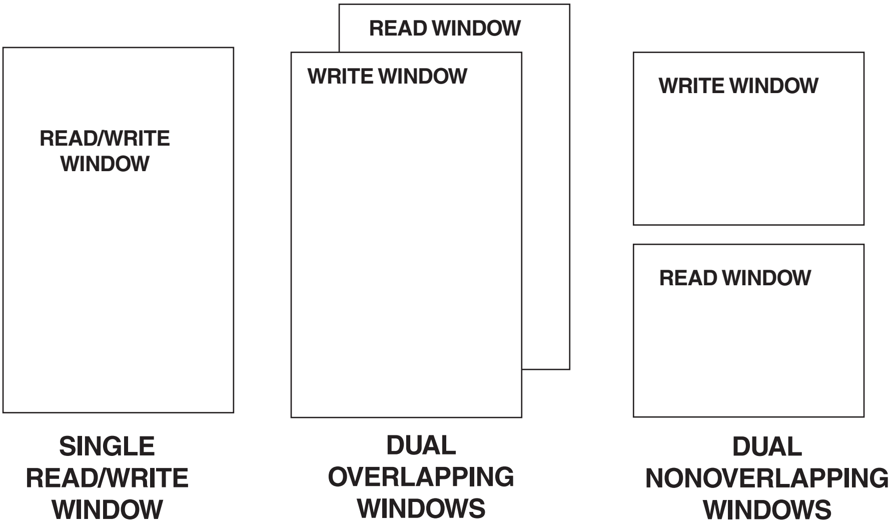


Figure 13-3 *VESA Window Types*

13.3 The VESA BIOS

The VESA BIOS has been designed to perform only those operations that are strictly necessary to achieve portability and hardware transparency of the SuperVGA system. The fundamental functions of the VESA BIOS, as used in SuperVGA programming, are the following:

1. Obtaining SuperVGA and mode information
2. Setting a standard VESA extended mode
3. Performing bank switching operations

The VESA BIOS does not provide graphics primitives. Furthermore, not even pixel setting and reading operations are included in the standard. Due to this design the software overhead is kept at a minimum. The actual function implementation of the functions are left to the chipset manufacturer, who also has the option of furnishing the BIOS in ROM, or as a TSR.

Of the functions provided by the VESA BIOS the bank switching operation is the most crucial in regards to display system performance. This is because bank switching is usually included in read and write loops and, therefore, in the program's critical path of execution. To provide the best possible performance the VESA BIOS allows access to the bank switching function directly, by means of a far call to the chipset manufacturer's own entry point to the service routine. This approach simplifies and accelerates access to the actual bank switching code. The result is that display routines that use VESA BIOS functions can perform bank switching operations almost as efficiently as routines that access the SuperVGA hardware directly.

13.3.1 VESA BIOS Services

The VESA BIOS is an extension of VGA BIOS video services located at interrupt 10H. Access to the VESA BIOS is by means of service number 79 (4FH). The sub-function refers to the specific VESA BIOS service. Eight VESA BIOS services have been implemented to date. These are shown in Table 13-2.

Table 13-2

VESA BIOS Sub-services to BIOS INT 10H

SUB-SERVICE	DESCRIPTION
00H	Return SuperVGA information
01H	Return SuperVGA mode information
02H	Set SuperVGA mode
03H	Return current video mode
04H	Save/restore SuperVGA video state
05H	Switch banks
06H	Set/get logical scan line length
07H	Set/get display start

The following code fragment is a general template for accessing the VESA BIOS sub-services

```
        MOV      AH,79          ; VESA BIOS service number
        MOV      AL,?           ; AL holds sub-service number
         .                      ; Other registers are loaded with
         .                      ; the values required by the
         .                      ; sub-service
        INT      10H
```

All VESA BIOS functions return the same error codes: AL = 79 (4FH) if the function is supported, AH = 0 if the call was successful.

Sub-service 0 - System Information

VESA BIOS sub-service number 0 provides general VESA information. The caller furnishes a pointer to a 256-byte data buffer which is filled by the VESA service. The following code fragment shows the set of variables and the register setup for this service.

```
DATA    SEGMENT
;**********************|
;    parameter block   |
;*********************|
VESA_BUFFER     DB      '    '  ; 'VESA' signature
VESA_VERSION    DW      ?       ; Version number
OEM_PTR_OFF     DW      ?       ; OEM string offset pointer
OEM_PTR_SEG     DW      ?       ; OEM string segment pointer
CAPABILITIES    DD      ?       ; Adapter capabilities
                                ; (first implemented in VESA
                                ; BIOS version 1.2)
MODES_PTR_OFF   DW      ?       ; Pointer to modes list, offset
MODES_PTR_SEG   DW      ?       ; Segment for idem
MEM_BLOCKS      DW      ?       ; Count of 64K memory blocks
                                ; (first implemented in VESA
                                ; BIOS version 1.1)
                DB      242 DUP (?) ; Remainder of block
;
DATA    ENDS
;
;
CODE    SEGMENT
        .
        .
        .

; Call VESA BIOS sub-service number 0 to obtain SuperVGA
; information
; Passed by caller:
;          DS:DI = pointer to 256-byte data buffer
; Returned by service:
;          AX = 004FH if no error
;          Data stored in the caller's buffer
;
;*********************|
;   setup registers   |
;*********************|
; Initialize entry registers
        LEA      DI,VESA_BUFFER ; Start of data buffer
; VESA BIOS sub-service number 0 uses ES as a segment base
        PUSH     ES             ; Caller's ES
        PUSH     DS             ; CAller's DS
        POP      ES             ; to ES
;*********************|
; get VESA information |
;*********************|
        MOV     AH,79           ; VESA BIOS service number
        MOV     AL,0            ; This sub-service
        INT     10H             ; BIOS video service
; At this point AX must hold 004FH if the call executed
        CMP     AX,004FH        ; Returned code
        JE      OK_VESA_0       ; Go if valid value
;*********************|
;     ERROR exit      |
;*********************|
; The programmer should code an error routine at this point
; to handle an invalid call to the VESA BIOS function
BAD_VESA:
        .
        .
        .
OK_VESA_0:
; Test buffer for a valid 'VESA' signature
```

```
        CMP     WORD PTR [DI],'EV'      ; First two letters
        JE      OK_VE           ; Go if matched
        JMP     BAD_VESA        ; Exit if not matched
OK_VE:
        CMP     WORD PTR [DI+2],'AS'    ; Last two letters
        JNE     BAD_VESA        ; Go if not matched
; At this point the VESA BIOS call to sub-service number 0
; was successful
        .
        .
        .
CODE    ENDS
```

The call to sub-service number 0 is usually made to determine if there is a VESA BIOS available, although the sub-service provides other information that could also be useful. Testing for a valid VESA BIOS is a two step process: first the code tests for the value 004FH in the AX register. This value corresponds to the standard VESA error codes mentioned at the beginning of this section. Once this first test is passed, the code makes certain that the four-character 'VESA' signature is stored at the start of the buffer. If these tests are satisfactory, execution can continue on the assumption that a valid VESA BIOS is present and that its functions are available to the software.

The data segment of the above code fragment shows the most important items returned by sub-service number 0. The field contents are as follows:

- VESA_BUFFER is the label that marks the start of the buffer. At this label the BIOS will store the word 'VESA' which serves as a string signature that identifies the BIOS.
- VESA_VERSION is a 2-byte field that encodes the current version of the VESA BIOS. The encoding is in fractional form, for example, the value 3131H corresponds to the ASCII digits 1,1 and represents version 1.1 of the VESA BIOS. An application can assume upward compatibility in the VESA BIOS.
- OEM_PTR_OFF and OEM_PTR_SEG are two word variables that encode the offset and segment values of a far pointer to an identification string supplied by the board manufacturer. Board-specific routines would use this string to check for compatible hardware.
- The CAPABILITIES label is a 4-byte field designed to hold a code that represents the general features of the SuperVGA environment. This field was not used until VESA BIOS version 1.2, released on October 22, 1991. At this time bit number 0 of this field was enabled to encode adapters with the possibility of storing extended primary color codes. In VESA BIOS version 1.2, and later, a value of 1 in bit 0 of the CAPABILITIES field indicates that the DAC registers can be programmed to hold more than 6-bit color codes. A value of 0 indicates that the DAC register is standard VGA, with 6-bits per primary color. Changing the bit width of the DAC registers is performed by calling sub-service number 8, discussed later in this section.
- MODES_PTR_OFF and MODES_PTR_SEG are two word variables that hold the offset and segment values of a far pointer to a list of implemented SuperVGA modes. Each mode occupies one word in the list. The code 0FFFFH serves as a list terminator. An application can examine the list of modes to make certain that a specific one is available or to select the best one among possible candidates.

- MEM_BLOCKS field encodes, in a word variable, the number of 64K blocks of memory installed in the adapter. Notice that this field was first implemented in VESA BIOS version 1.1.

Sub-service 1 - Mode Information

VESA BIOS sub-service number 1 provides information about a specific SuperVGA VESA mode. The caller furnishes a pointer to a 256-byte data buffer, which is filled by the VESA service, as well as the number of the desired mode. The following code fragment shows a possible set of data variables and register setup for this service.

```
DATA    SEGMENT
;
;**********************|
;  first field group   |
;**********************|
VESA_INFO       DW      ?       ; Mode attributes, mapped as
                        ; follows:
                        ; ..4 3 2 1 0 <= bits
                        ;   | | | | |__ 0 = mode not supported
                        ;   | | | |     1 = mode supported
                        ;   | | | |____ 0 = no extended mode info
                        ;   | | |       1 = extended mode info
                        ;   | | |_______0 = no output functions
                        ;   | |         1 = output functions
                        ;   | |________ 0 = monochrome mode
                        ;   |           1 = color mode
                        ;   |__________ 0 = text mode
                        ;               1 = graphics mode
                        ; 15..5 = RESERVED
WIN_A_ATTS      DB      ?       ; Window A attributes
WIN_B_ATTS      DB      ?       ; Window B attributes
WIN_GRAIN       DW      ?       ; Window granularity
WIN_SIZE        DW      ?       ; Window size
WIN_A_SEG       DW      ?       ; Segment address for window A
WIN_B_SEG       DW      ?       ; Segment address for window B
BANK_FUN        DD      ?       ; Far pointer to bank switch
                                ; function
BYTES_PER_ROW   DW      ?       ; Bytes per screen row
;**********************|
;  second field group  |
;**********************|
; Extended mode data. Optional until VESA BIOS version 1.2
X_RES           DW      ?       ; Horizontal resolution
Y_RES           DW      ?       ; Vertical resolution
X_CHAR_SIZE     DB      ?       ; Pixel width of character cell
Y_CHAR_SIZE     DB      ?       ; Pixel height of character cell
BIT_PLANES      DB      ?       ; Number of bit planes
BITS_PER_PIX    DB      ?       ; Bits per pixel in this mode
NUM_OF_BANKS    DB      ?       ; Number of video memory banks
MEM_MODEL       DB      ?       ; Memory model, as follows:
                                ; 00H = text mode
                                ; 01H = CGA graphics
                                ; 02H = Hercules graphics
                                ; 03H = 4-plane architecture
                                ; 04H = Packed pixel architecture
                                ; 05H = 256 color (unchained)
                                ; The following were defined
                                ; in VESA BIOS version 1.2:
```

```
                                ; 06H = Direct color
                                ; 07H = YUV color
                                ; 08H - 0FF = not yet defined
BANK_SIZE       DB      ?       ; Kilobytes per bank
PLANES          DB      ?       ; Number of planes:
                                ; 4 in 16 color modes, 1 in 256 color modes
                DB       1      ; Reserved for BIOS
;*********************|
;  third field group  |
;*********************|
; Direct color fields. Defined in VESA BIOS version 1.2
RED_MASK        DB      ?       ; Bit size of red mask
RED_POSITION    DB      ?       ; Red mask LSB position
GREEN_MASK      DB      ?       ; Bit size of green mask
GREEN_POSITION  DB      ?       ; Green mask LSB position
BLUE_MASK       DB      ?       ; Bit size of blue mask
BLUE_POSITION   DB      ?       ; Blue mask LSB position
RSVD_MASK       DB      ?       ; Bit size of reserved mask
RSVD_POSITION   DB      ?       ; Reserved mask LSB position
DC_INFO         DB      ?       ; Attributes of direct color
                                ; modes, as follows:
                                ; bit 0 = color ramp
                                ;       0 = fixed
                                ;       1 = programmable
                                ; bit 1 = Reserved field bits
                                ;       0 = not usable
                                ;       1 = usable
                DB      216 DUP (?) ; Remainder of block
DATA    ENDS
;
CODE    SEGMENT
        .
        .
;*********************|
;  get VESA mode info |
;*********************|
; Passed by caller:
; CX = mode number, as follows:
; GRAPHICS      number       resolution      colors
;                100H         640 by 400       256
;                101H         640 by 480       256
;                102H         800 by 600        16
;                103H         800 by 600       256
;                104H        1024 by 768        16
;                105H        1024 by 768       256
;                106H        1280 by 1224       16
;                107H        1280 by 1224      256
; TEXT           108H          80 by 60
;                109H        132 by 25
;                10AH        132 by 43
;                10BH        132 by 50
;                10CH        132 by 60
;          DS:DI = pointer to 256-byte data buffer
; Returned by service:
;          AX = 004FH if no error
;          Data stored in the caller's buffer
;*********************|
;    register setup   |
;*********************|
; CX to hold requested mode number
; DS:SI -> information block supplied by service
; Initialize entry registers
        LEA      DI,VESA_INFO   ; Start of data buffer
        MOV      CX,105H        ; Mode requested
; VESA BIOS sub-service number 1 uses ES as a segment base
        PUSH     ES             ; Caller's ES
        PUSH     DS             ; Caller's DS
        POP      ES             ; to ES
;**********************|
; get VESA information |
;**********************|
        MOV     AH,79           ; VESA BIOS service number
        MOV     AL,1            ; This sub-service
        INT     10H             ; BIOS video service
; At this point AX must hold 004FH if the call executed
        CMP     AX,004FH        ; Returned code
        JE      OK_MODE         ; Go if valid value
;**********************|
;     ERROR exit       |
;**********************|
; The programmer should code an error routine at this point
; to handle the case of an invalid VESA BIOS call
        .
        .
OK_MODE:
; At this point the VESA BIOS call to sub-service number 1
; was successful. However, the code cannot assume that the
; mode requested is implemented in the system
        .
        .
CODE    ENDS
```

The call to sub-service number 1 is usually made to determine if the desired mode is available in the hardware and, if so, to obtain certain fundamental parameters required by the program. If the call is successful, the code can examine the data at offset 0 in the data buffer in order to determine the mode's fundamental attributes. These mode attributes are shown in Figure 13-4.

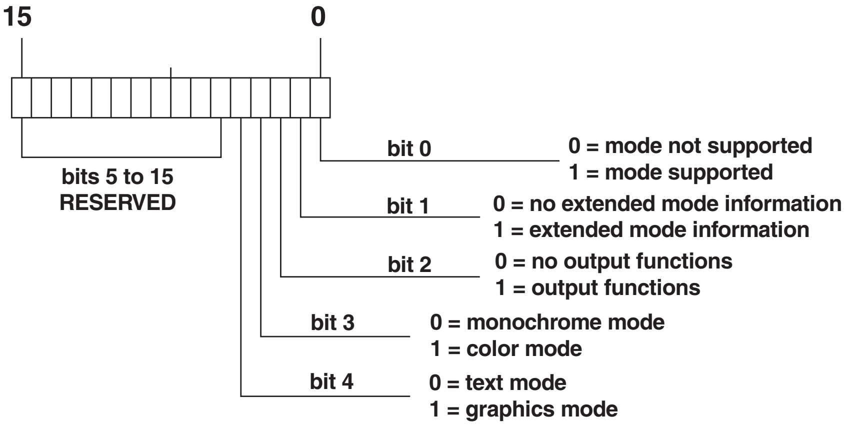


Figure 13-4 *VESA Mode Attribute Bitmap*

The data segment of the above code fragment shows the items returned by sub-service number 1. The data items are divided into three field groups. The contents of the variables in the first field group are as follows:

WIN_A_ATTS and WIN_B_ATTS are 2 bytes that encode the attributes of the two possible memory banks, or windows. Figure 13-5 is a bitmap of the window attribute bytes.

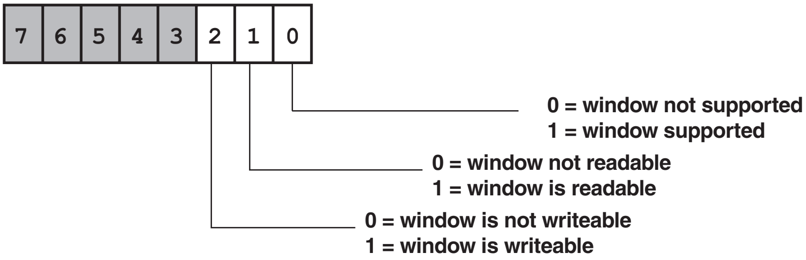


Figure 13-5 *Window Attributes Bitmap*

The code can inspect the window attribute bits to determine the window types used in the system (see Figure 13-3).

The WIN_GRAIN word specifies the granularity of each window. The granularity unit is 1 kilobyte. The value can be used to determine the minimum video memory boundary for the window.

The WIN_SIZE word specifies the size of the windows in kilobytes. This value can be used in tailoring bank switching operations to specific hardware configurations (see Section 13.3.1).

The word labeled WIN_A_SEG holds the segment base address for window A and the word labeled WIN_B_SEG the base address for window B. The base address in graphics modes is usually A000H, however, the code should not take this for granted.

The doubleword labeled BANK_FUN holds a far pointer to the bank shifting function in the BIOS. An application can shift memory banks using VESA BIOS sub-service number 5, described later in this section, or by means of a direct call to the service routine located at the address stored in this variable. The call can be coded with the instruction:

```
        CALL    DWORD PTR BANK_FUN
```

BYTES_PER_ROW is a word variable that encodes the number of bytes in each screen logical pixel row. Notice that this value can be larger than the number of pixels in a physical scan line.

The variables in the second field group are of optional nature. Bit number 1 of the mode attribute bitmap (see Figure 13-4) can be read to determine if this part of the data block is available. The contents of the various fields in the second group are described in the data segment of the preceding code fragment.

The direct color fields from the third field group. These fields were first implemented in VESA BIOS version 1.2 to support SuperVGA systems with color capabilities that extended beyond the 256 color modes. The contents of the various fields in the third group are described in the data segment of the preceding code fragment. Because, to date, very few SuperVGA adapters support the direct color modes, their programming is not considered in this book.

Sub-service 2 - Set Video Mode

VESA BIOS sub-service number 2 is used to initialize a video mode supported by the adapter. The VESA mode number is passed to the sub-service in the BX register. The high-order bit, which is sometimes called the clear memory flag, is set to request that video memory not be cleared. The following code fragment shows a call to this VESA BIOS service.

```
;*********************|
;    set video mode   |
;*********************|
; Select mode 105H using VESA BIOS sub-service number 2
        MOV     BX,105H         ; Mode number and high bit = 0
                                ; to request clear video
        MOV     AH,79           ; VESA BIOS service number
        MOV     AL,2            ; This sub-service
        INT     10H             ; BIOS video service
; Test for valid returned value
        CMP     AX,004FH        ; Status for no error
        JE      MODE_IS_SET     ; No error during mode set
;*********************|
;     ERROR exit      |
;*********************|
; The programmer should code an error routine at this point
; to handle the possibility of a mode setting error
        .
        .
        .
; At this label the mode was set satisfactorily
MODE_IS_SET:
        .
        .
        .
```

Sub-service 3 - Get Video Mode

VESA BIOS sub-service number 3 is used to obtain the current video mode. The VESA mode number is returned by the sub-service in the BX register. The following code fragment shows a call to this VESA BIOS service.

```
;*********************|
;    get video mode   |
;*********************|
; VESA BIOS sub-service number 3 to obtain current video mode
        MOV     AH,79           ; VESA BIOS service number
        MOV     AL,3            ; This sub-service
        INT     10H             ; BIOS video service
```

```
; Test for valid returned value
        CMP     AX,004FH        ; Status for no error
        JE      MODE_AVAILABLE  ; No error during mode set
;*********************|
;     ERROR exit      |
;*********************|
; The programmer should code an error routine at this point
; to handle the possibility of a mode reading error
        .
        .
        .
; At this label the mode was read satisfactorily. The BX
; register holds the mode number
MODE_AVAILABLE:
        .
        .
        .
```

Sub-service 4 - Save/Restore Video State

VESA BIOS sub-service number 4 is used to save and restore the state of the video system. This service, which is an extension of BIOS service number 28, is often used in a multitasking operating system to preserve the task states and by applications that manage two or more video environments. The sub-service can be requested in three different modes, passed to the VESA BIOS routine in the DL register.

Mode number 0 (DL = 0) of sub-service number 4 returns the size of the save/restore buffer. The 4 low bits of the CX register encode the machine state buffer to be reported. The bitmap for the various machine states is shown in Figure 13-6.

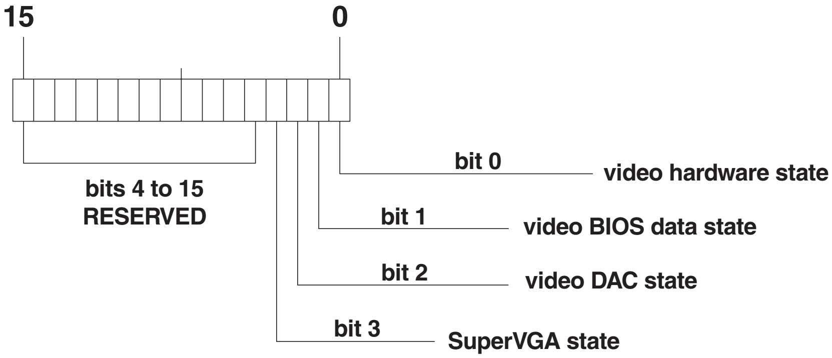


Figure 13-6 *VESA BIOS Machine State Bitmap*

The units of buffer size returned by mode number 0, of sub-service number 4, are 64-byte blocks. The block count is found in the BX register.

Mode number 1 (DL = 1), of sub-service number 4, saves the machine video state requested in the CX register (see Figure 13-6). The caller should provide a pointer to a buffer sufficiently large to hold the requested state data. The size of the buffer can be dynamically determined by means of a call using mode number 0, described above. The pointer to the buffer is passed in ES:BX.

Mode number 2 (DL = 2), of sub-service number 4, restores the machine video state requested in the CX register (see Figure 13-6). The caller should provide a pointer to the buffer that holds data obtained by means of a call using mode number 1 (see above).

Sub-service 5 - Switch Bank

VESA BIOS sub-service number 5 is used to switch memory banks in those modes that require it. Software should call sub-service number 1 to determine the size and address of the banks before calling this function. Two modes of this sub-service are implemented: one to switch to a desired bank and another one to request the number of the currently selected bank.

Mode number 0 (BH = 0) is the switch bank command. The BL register is used by the caller to encode window A (value = 0) or window B (value = 1). The bank number is passed in the DX register. The following code fragment shows the necessary processing:

```
; VESA BIOS sub-service number 5 register setup
        MOV     BX,0            ; Select bank in window A
                                ; and bank switch function
; BH = 0 to select bank
; BL = 0 to select window A
; DX = bank number
        MOV     AX,4F05H        ; Service and sub-service
        INT     10H
        .
        .
        .
```

Mode number 1 of sub-service (BH = 0) is used to obtain the number of the memory bank currently selected. The BL register is used by the caller to encode window A (value = 0) or window B (value = 1). The bank number is reported in the DX register.

Earlier in this section we mentioned that an application can also access the bank switching function in the BIOS by means of a far call to the service routine. The address of the service routine is placed in a far pointer variable by the successful execution of sub-service number 1. For the far call operation the register setup for BH, BL, and DX is the same as for using sub-service 5. However, in the far call version AH and AL need not be loaded, no meaningful information is returned, and AX and DX are destroyed.

Sub-service 6 - Set/Get Logical Scan Line

VESA BIOS sub-service number 6 is used to set or read the length of the logical scan line. Observe that the logical scan line can be wider than the physical scan line supported by the video hardware. This sub-service was first implemented in VESA BIOS version 1.1. For this reason it is not available in the BIOS functions of earlier adapters.

Sub-service 7 - Set/Get Display Start

VESA BIOS sub-service number 7 is used to set or read from the logical page data the pixel to be displayed in the top left screen corner. The sub-service is useful to applications that use a logical screen that is larger than the physical display in order to facilitate panning and screen scrolling effects. As is the case with sub-service number 6, this sub-service was first implemented in VESA BIOS version 1.1. For this reason it is not available in the BIOS functions of many adapters.

Sub-service 8 - Set/Get DAC Palette Control

VESA BIOS sub-service number 8 was designed to facilitate programming of SuperVGA systems with more than 6-bit fields in the primary color registers of the DAC. The sub-service contains two modes. Mode number 0 (BL = 0) is used to set a DAC color register width. The desired width value, in bits, is passed by the caller in the BH register. Mode number 1 (BL = 1) is used to obtain the current bit width for each primary color. The bit width is returned in the BH registers. The standard bit width for VGA systems is 6.

This sub-service was first implemented in version 1.2 of the VESA BIOS, released in October 22, 1991. Therefore it is not available in adapters with earlier versions of the VESA BIOS. Another feature introduced in VESA BIOS version 1.2 is the use of bit 0 of the CAPABILITIES field (see sub-service 0 earlier in this section) to encode the presence of DAC registers capable of storing color encodings of more than 6 bits. Applications that propose to use sub-service 8 should first test the low-order bit of the CAPABILITIES field to determine if the feature is implemented in the hardware.

13.4 Programming the SuperVGA System

Programming a particular SuperVGA chipset requires obtaining specific technical data from the manufacturer. The resulting code has little, if any, portability to other systems. This approach is used in coding hardware-specific drivers that take full advantage of the capabilities of the system. An alternative method that insures greater portability of the code at a small price in performance is the use of the VESA BIOS services described starting it Section 13.2.

It is theoretically possible to design a general-purpose graphics routine that operates in every SuperVGA chipset and display mode. However, this universality can only be achieved at a substantial price in performance, an element that is usually critical to graphics software. For this reason the design and coding of mode-specific graphics routines is generally considered a more efficient approach. By using VESA BIOS functions it is possible to design mode-specific routines that are compatible with most SuperVGA systems that support the particular mode.

In the examples that follow we have used VESA BIOS mode number 105H with a resolution of 1,024 by 768 pixels in 256 colors. We have selected this mode because it is compatible with modes used in 8514/A and XGA systems, and also because it is widely available in fully equipped SuperVGA adapters. The reader should be able to readily convert these routines to other SuperVGA graphics modes.

13.4.1 Address Calculations

Address calculations in a SuperVGA mode depend on the screen dimensions and the location of the video buffer in the system's memory space. In a mode-specific routine the number of pixels per row can be entered as a numeric value. In modes that require more than one memory bank the bank size must also enter into the address calculations. Most SuperVGA adapters use a bank size of 64K, which can be hard-coded in the address calculation routine. On the other hand, it is possible to use a memory variable that stores the number of pixels per row and the bank size parameters in order to design address calculation routines that will work in more than one mode. In the following code fragment we have assumed that the SuperVGA is in VESA mode 105H, with 1,024 pixels per scan line and that the bank size is 64K. The display routines assume that the base address of the video buffer is A000H.

```
; Calculate pixel address from the following coordinates:
;       CX = x coordinate of pixel
;       DX = y coordinate of pixel
; Code assumes:
;       1. SVGA is in a 1,024 by 768 pixel mode in 256 colors
;           (mode number 105H)
;       2. Bank size is 64K
; Get address in SVGA memory space
        CLC                     ; Clear carry flag
        PUSH    AX              ; Save color value
        MOV     AX,1024         ; Pixels per scan line
        MUL     DX              ; DX holds line count of address
        ADD     AX,CX           ; Add pixels in current line
        ADC     DX,0            ; Answer in DX:AX
                                ; DL = bank, AX = offset
        MOV     BX,AX           ; Offset to BX
        .
        .
        .
```

At this point BX holds the pixel offset and DX the bank number. Note that the pixel offset is the offset within the selected bank, and not the offset from the start of the screen as is often the case in VGA routines.

13.3.2 Bank Switching Operations

In a SuperVGA adapter set to VESA mode number 105H (resolution of 1,024 by 768 pixels in 256 colors) the number of video memory banks depends on the bank size. With a typical bank size of 64K the entire video memory space requires 12 memory banks, since:

$$\frac{1024 \times 768}{65535} = 12$$

In order to update the entire video screen the software has to perform 12 bank switches. This would be the case in performing a clear screen operation. Furthermore, many relatively small screen objects cross one or more bank boundaries. In fact, in VESA SuperVGA mode 105H any graphics object or window that exceeds 64 pixels in height will necessarily overflow one bank.

For these reasons bank switching operations should be optimized to perform their function as quickly as possible. The ideal solution would be to embed the hardware bank switching code within the address calculation routine. This is the method adopted for the XGA pixels display routine listed in Section 7.3.1. However, XGA software does not have to contend with variations in hardware. We have seen that in the SuperVGA environment to hard-code the bank switching operation would almost certainly make the routine not portable to other devices. An alternative solution is to perform bank switching by means of VESA BIOS service number 5, described in Section 13.2.1. The following code fragment shows the code for bank switching using the VESA BIOS service.

```
;*********************|
;    change banks     |
*********************|
; Select video bank using VESA BIOS sub-service number 5
; VESA BIOS sub-service number 5 register setup
; BH = 0 to select bank
; BL = 0 to select window A
; DX = bank number
        MOV     BX,0            ; Select bank in window A
        MOV     AX,4F05H        ; Service and sub-service
        INT     10H
        .
        .
        .
```

An alternative option that would improve performance of the bank switching operation is by means of a far call to the service routine, as mentioned in Section 13.2.1. The following code fragment shows bank switching using the far call method. The code assumes that the address of the service routine is stored in a doubleword variable named BANK_FUN. This address can be obtained by means of VESA BIOS sub-service number 1 (get mode information) discussed in Section 13.2.1.

```
;*********************|
;    change banks     |
;  by far call method |
;*********************|
; Select video bank by means of a far call to the bank switching
; routine provided by the chipset manufacturer
; Code assumes that the far address of the service routine is
; stored in a doubleword variable named BANK_FUN
; Register setup for far call method
; BH = 0 to select bank
; BL = 0 to select window A
; DX = bank number
        MOV     BX,0            ; Select bank in window A
        PUSH    AX              ; Preserve caller's context
        PUSH    DX
        CALL    DWORD PTR BANK_FUN
        POP     DX              ; Restore context
        POP     AX
        .
        .
        .
```

Observe that to use the far call method the doubleword variable that holds the address of the service routine must be reachable at the time of the call. Therefore, if the variable is in another segment, a segment override byte is required.

13.4.3 Setting and Reading a Pixel

Once the pixel address has been determined and the hardware has been switched to the corresponding video memory bank, setting the pixel is a simple write operation. For example, in VESA mode number 105H, once the address calculation routine in Section 13.3.1 and the bank switching routine in Section 13.3.2 have executed, the pixel can be set by means of the instruction

```
        MOV     BYTE PTR ES:[BX],AL
```

The code assumes that ES holds the base address of the video buffer, BX the offset within the bank, and AL the 8-bit color code. Note that since VESA mode number 105H is not a planar mode, no previous read operation is necessary to enable the latching mechanism (see Section 3.1.1).

Reading a pixel in a SuperVGA mode is usually based on the same address and bank switching operations as those required for setting a pixel. The actual read instruction is in the form

```
        MOV     AL,BYTE PTR ES:[BX]
```

The SVGA_PIX_105 procedure in the SVGA module of the GRAPHSOL library performs a pixel write operation while in SuperVGA mode number 105H. The procedure named SVGA_READ_105 can be used to read a screen pixel in this same mode.

13.4.4 VGA Code Compatibility

The SuperVGA enhanced graphics mode presents three basic differences in relation to VGA modes: multiple banks, non-planar architecture, and greater resolution. Once these factors are taken into account by the SuperVGA specific graphics read and write routines, many VGA calculations can be used directly in SuperVGA graphics. In the following section we describe the use, from SuperVGA modes, of several VGA routines in the VGA modules of the GRAPHSOL library. These include the VGA routines developed in Chapter 3 to access the LUT registers in the DAC, since most SuperVGA systems use the same color look-up table and DAC as VGA.

13.5 Using the SuperVGA Library

The GRAPHSOL library furnished with this book includes the module named SVGA which contains SuperVGA graphics routines. Many of these procedures were designed as mode-specific in order to optimize performance. The procedures in the SVGA module serve to initialize the SuperVGA system, to establish the presence of a VESA SuperVGA BIOS, to select a VESA mode number 105H, and to set and read individual screen pixels while in mode 105H.

In addition to the routines in the SVGA library, SuperVGA programs use several procedures in the VGA modules of GRAPHSOL.LIB. The use of the VGA procedures by a SuperVGA system requires a previous call to the SET_DEVICE routine in the VGA3 module. For SuperVGA systems this call is made with the AL register holding the ASCII character "S." The call sets a device-specific display switch to the VESA SuperVGA pixel display routine in the SVGA module. By enabling the SuperVGA display routine (named SVGA_PIX_105) the code makes possible the use of the geomet-rical procedures in the VGA3 module named BRESENHAM, LINE_BY_SLOPE, DISPLAY_LINE, CIRCLE, ELLIPSE, PARABOLA, and HYPERBOLA, and also the use of the text display procedures in the VGA2 module named FINE_TEXT, FINE_TEXTHP, and MULTITEXT, as well as the corresponding text display support routines FONT_TO_RAM and READ_HPFONT. Information regarding the VGA text display and geometrical routines can be found in Chapters 4 and 5 as well as in the source files VGA2.ASM and VGA3.ASM contained in the book's software.

Since most SuperVGA systems use the VGA LUT and DAC registers in the same architecture as VGA mode number 19, a SuperVGA program can use the color register procedures for VGA mode number 19 that appear in the VGA1 module of the GRAPHSOL library. These procedures are named TWO_BIT_IRGB, GRAY_256, SUM_TO_GRAY, SAVE_DAC, and RESTORE_DAC. The source file and program named SVGADEMO furnished in the diskette demonstrates the use of the SuperVGA library services in the SVGA module and the use of the compatible VGA services in the VGA modules of GRAPHSOL.LIB.

13.5.1 Procedures in the SVGA.ASM Module

SVGA_MODE

Call VESA BIOS sub-service number 0 to obtain SuperVGA and VESA information and sub-service number 1 to obtain mode-specific information.

```
Receives:
         1. word integer of VESA SuperVGA graphics mode number
            as follows:
               number       resolution      colors
  GRAPHICS      100H         640 by 400       256
  MODES         101H         640 by 480       256
                102H         800 by 600        16
                103H         800 by 600       256
                104H        1024 by 768        16
                105H        1024 by 768       256
                106H        1280 by 1224       16
                107H        1280 by 1224      256
  TEXT          108H          80 by 60
  MODES         109H        132 by 25
                10AH        132 by 43
                10BH        132 by 50
                10CH        132 by 60
  DIRECT COLOR  10DH        300 by 200          32K
  MODES         10EH        320 by 200          64K
                10FH        320 by 200       16.8Mb
                110H        640 by 480          32K
                111H        640 by 480          64K
                112H        640 by 480       16.8Mb
                113H        800 by 600          32K
                114H        800 by 600          64K
                115H        800 by 600       16.8Mb
                116H        1024 by 768         32K
                117H        1024 by 768         64K
                118H        1024 by 768      16.8Mb
                119H        1280 by 1024        32K
                11AH        1280 by 1024        64K
                11BH       1280 by 1024     16.8Mb

Returns:
         1. carry clear if no error, then
            ES:SI --> VESA_BUFFER, formatted as follows:
VESA_BUFFER     DB      '    '  ; VESA signature
VESA_VERSION    DW      ?       ; Version number
OEM_PTR_OFF     DW      ?       ; OEM string offset pointer
OEM_PTR_SEG     DW      ?       ; OEM string segment pointer
CAPABILITIES    DD      ?       ; System capabilities
MODES_PTR_OFF   DW      ?       ; Pointer to modes list, offset
MODES_PTR_SEG   DW      ?       ; Segment for idem
MEM_BLOCKS      DW      ?       ; Count of 64K memory blocks
                                ; (Only in June 2, 1990 revision)
                DB      242 DUP (0H)

            ES:DI --> VESA_INFO, formatted as follows:
VESA_INFO       DW      ?       ; Mode attribute bits
                        ; ..4 3 2 1 0 <= bits
                        ;   | | | | |__ 0 = mode not supported
                        ;   | | | |     1 = mode supported
                        ;   | | | |____ 0 = no extended mode info
                        ;   | | |       1 = extended mode info
                        ;   | | |_______0 = no output functions
                        ;   | |         1 = output functions
                        ;   | |________ 0 = monochrome mode
                        ;   |           1 = color mode
                        ;   |__________ 0 = text mode
                        ;               1 = graphics mode
                        ; 15..5 = RESERVED
WIN_A_ATTS      DB      ?       ; Window A attributes
WIN_B_ATTS      DB      ?       ; Window B attributes
WIN_GRAIN       DW      ?       ; Window granularity
WIN_SIZE        DW      ?       ; Window size
WIN_A_SEG       DW      ?       ; Segment address for window A
WIN_B_SEG       DW      ?       ; Segment address for window B
WIN_PTR         DD      ?       ; Far pointer to window function
BYTES_PER_ROW   DW      ?       ; Bytes per screen row

; Extended mode data. Optional until version 1.2
X_RES           DW      ?       ; Horizontal resolution
Y_RES           DW      ?       ; Vertical resolution
X_CHAR_SIZE     DB      ?       ; Pixel width of character cell
Y_CHAR_SIZE     DB      ?       ; Pixel height of character cell
BIT_PLANES      DB      ?       ; Number of bit planes
BITS_PER_PIX    DB      ?       ; Bits per pixel in this mode
NUM_OF_BANKS    DB      ?       ; Number of video memory banks
MEM_MODEL       DB      ?       ; Memory model
BANK_SIZE       DB      ?       ; Kb per bank
                DW      0       ; Padding
; Direct color fields. Defined in VESA BIOS version 1.2
RED_MASK        DB      ?       ; Bit size of red mask
RED_POSITION    DB      ?       ; Red mask LSB position
GREEN_MASK      DB      ?       ; Bit size of green mask
GREEN_POSITION  DB      ?       ; Green mask LSB position
BLUE_MASK       DB      ?       ; Bit size of blue mask
BLUE_POSITION   DB      ?       ; Blue mask LSB position
RSVD_MASK       DB      ?       ; Bit size of reserved mask
RSVD_POSITION   DB      ?       ; Reserved mask LSB position
DC_INFO         DB      ?       ; Attributes of direct color
```

```
                              ; modes, as follows:
                              ; bit 0 = color ramp
                              ;       0 = fixed
                              ;       1 = programmable
                              ; bit 1 = Reserved field bits
                              ;       0 = not usable
                              ;       1 = usable
                DB      216 DUP (?) ; Remainder of block
```

```
          2. Carry set if error
```

VESA_105

Set SuperVGA to VESA mode number 105H with a resolution of 1024 by 768 pixels in 256 colors.

```
Receives:

         Nothing

Assumes:

        That the data variables in the buffers VESA_BUFFER and VESA_INFO
have been filled by a previous call to the VESA_MODE procedure.

Returns:

         Carry clear if mode was set

         Carry set if error
```

SVGA_PIX_105

Write a screen pixel accessing SVGA memory directly and using a far call to the bank switching routine.

```
Receives:
         1. word variable of x pixel coordinate
         2. word variable of y pixel coordinate
         3. byte variable of 8-bit color code

Assumes:
         1. SVGA in VESA mode 105H (1,024 by 768 pixels in 256
            colors)
         2. Size of video bank is 64K
         3. ES holds base address of video buffer (A000H)

Returns:
         Nothing

Action:
        Pixel is set
```

SVGA_CLS_105

Clear video memory while in VESA mode number 105H.

```
Receives:
         1. byte integer of 8-bit color code

Assumes:
         1. SVGA in VESA mode 105H (1,024 by 768 pixels in 256
            colors)
         2. Size of video bank is 64K
         3. ES holds base address of video buffer (A000H)

Returns:
         Nothing

Action:
        Screen is initialized to requested color code.
```

SVGA_READ_105

Read a screen pixel accessing SVGA memory directly and using a far call to the bank switching routine.

```
Receives:
         1. word variable of x pixel coordinate
         2. word variable of y pixel coordinate

Assumes:
         1. SVGA in VESA mode 105H (1,024 by 768 pixels in 256
            colors)
         2. Size of video bank is 64K
         3. ES holds base address of video buffer (A000H)

Returns:
         1. byte integer of pixel color

Action:
        Pixel is read
```

Chapter 14

DOS Animation

Topics:

- Animation fundamantals
- User interaction in animation
- Image movement
- DOS imaging techniques

This chapter describes the principles and programming techniques of image animation in DOS. The chapter also covers mouse programming by means of the Microsoft mouse interface. The discussion includes image mapping, panning and geometrical transformations, as well as imaging techniques by looping, and by system timer and vertical retrace interrupts.

14.1 Graphics and Animation

Computer graphics animation is usually defined as the simulation of life-like qualities by digital manipulations of a computer-generated image. The concept is somewhat limiting since it excludes analog operations and assumes that the only objects that can be computer animated are images on the CRT. However, in the microcomputer environment animation is mostly about manipulating screen images so as to mimic life. This is often performed by moving images on the screen, but color and shapes can also be changed to create a life-like illusion.

Computer graphics animation can take place in a real- or a delayed-time frame. For example, a computer program can generate and store a series of consecutive images that simulate the movement of an object. The stored images can be recorded on storage devices, such as a video tape, and later played back at a faster rate than they were generated. In this case we can say that the computer animation took place in a delayed-time frame; the animated action was not visible until the images were played back on a television set. On the other hand, a computer program can simulate a ping-pong game on the screen. In this case the animation takes place in a real-time frame. Graphics animation in the microcomputer environment is, for the most part, image animation in real-time. For this reason in the present chapter we emphasize real-time operations. Delayed-frame is also known as frame-by-frame animation.

Animated screen images can be classified according to the user's interaction with the graphics object. When the object is directly controlled by the user of the software we speak of interactive animation. Screen objects that are animated independently of the user's action often move by means of a machine-generated time-pulse. In this sense we speak of time-pulse animation. The mouse is an input device closely related to interactive animation. For this reason we have incorporated mouse programming into the present chapter. Although not all mouse programming operations are related to animated screen objects, we have, for practical reasons, included all phases of mouse programming in the present treatment. Time-pulse animation is also discussed in some detail.

14.1.1 Physiology of Animation

The image of an object created by the human eye can persist in the brain for a brief period of time after the object no longer exists in the real world. This physiological phenomena is called visual retention. Although the biological mechanisms of retention are not fully understood, we do know that it involves the chemistry of the retina and the structure of cells and neurons in the eye. First cinematography, and more recently television, have taken advantage of visual retention to create the illusion of continuous movement. This is done by consecutively flashing still images at a faster rate than the period of visual retention. This technique, by which a new image replaces the old one before the period of retention has expired, creates in our minds the illusion of a smoothly moving object.

It has been determined experimentally that the critical image update rate for smooth animation is from 22 to 30 images per second. Modern day moving picture films are recorded and displayed at a rate of 24 images per second. Although the threshold for smooth animation varies with individuals, it is generally estimated at a rate of approximately 18 images per second. This means that if the consecutive images are projected at a rate slower than this threshold, the average individual is able to perceive a certain jerkiness. However, if the flashing rate exceeds the threshold, our brains merge the images together with no perception of the individual flashes. This threshold rate can be called the critical jerkiness frequency.

14.1.2 PC Animation

Animated graphics systems, such as the ones used in many electronic video games, are based on vector refresh technology. In these systems the movement of the electron beam is limited to the objects that must be redrawn during a refresh cycle. Therefore, vector refresh displays are more efficient in animating small objects than raster scan systems in which the entire screen area must be scanned by the electron gun or guns during each cycle.

PC graphics use raster scan technology. Animation on a raster scan computer is based on creating an illusion of movement by displaying successive images. The graphics object is typically stored in a dedicated buffer which is imaged on the CRT by the video hardware. The name frame buffer animation has often been used in this context. In VGA systems the frame buffer is the video memory itself. In XGA systems, in addition to video memory, there is a second, smaller, frame buffer dedicated to storing the sprite image. Image changes can be made by altering the contents of video memory or by changing the screen position at which the frame buffer is displayed.

Image size and critical jerkiness frequency are usually the limiting factors in frame buffer animation. For example, assume a VGA video system in mode number 18 (640 by 480 pixels in 16 colors). If to produce smooth animation the system must redraw the screen at a rate of 20 images per second, then the changes in the frame buffer must be performed in less than 1/20s. Furthermore consider that to animate a screen object its image must be erased from the current position before it is redrawn at a new position, otherwise the animation would leave a track of objects on the video display. Therefore the buffer update sequence is, in reality, a sequence of redraw, erase, redraw operations, which means that the critical jerkiness frequency is the time elapsed from redraw to redraw cycle. Consequently, the allotted time for the redraw-erase cycle becomes 1/48s.

Although the above example is a worse-case scenario it does show the constraints in which animation must be performed in a raster scan system. In the PC, in particular, graphics animation is a battle against time: the time in which the frame buffer must be updated before the entire screen is redrawn by the video hardware. Therefore the animation programmer must resort to every known trick and stratagem in order to squeeze the maximum performance while updating the frame buffer. But, in many cases, even the most efficient and imaginative programming is not able to overcome the system's limitations and the animated image is bumpy and coarse.

14.1.3 Software Support for Animation Routines

In previous chapters we provided software support mainly in the form of library routines that can be called by a graphics program. But most animation routines have extremely critical performance constraints. This determines that animation software be customized and optimized for a particular program design. Furthermore, animated programs are often designed with these hardware limitations in mind. To provide animation routines in the form of library procedures would introduce, in the first place, an unnecessary call-and-return overhead over on-line code. In addition, the procedures would have to be adaptable to the many varying circumstances of animated programs and, at the same time, optimized for maximum performance. Code that is simultaneously flexible and efficient is a programming contradiction.

For these reasons we have opted to provided code support for the animation techniques discussed in this chapter in the form of coding templates, rather than as library routines. The template files can also be found in the book's software package. The reader can use these templates to avoid having to re-code the routine manipulations in the various animation techniques. However, we have left blank lines in the templates (marked by ellipses) to indicate where the programmer must supply the customized code.

The software package furnished with this book contains a VGA animated program named MATCH. The reader should consult the README.NOW file in the MATCH directory before executing the program. The source files for the MATCH program demonstrate interactive and time-pulse animation in a VGA system.

14.2 Interactive Animation

Interactive animation refers to screen objects that are moved at will by the user. Typically the animated screen object is controlled by means of an input device, such as a mouse, puck, or graphics tablet (see Section 1.1.2). In the present section we discuss programming the mouse device as a means for animating an interactive screen object. Other interactive input devices are specialty tools used mostly in CAD software, therefore they are outside the scope of this book.

14.2.1 Programming the Mouse

The IBM BIOS, as documented in the IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference (see bibliography), describes a pointing device interface associated with service number 194 of INT 15H. However, there are several difficulties associated with this service. In the first place, the IBM documentation dealing with this mouse service is not sufficient for programming the device. Another consideration is that the services are not compatible with different mouse hardware. Then there is the problem that various non-IBM versions of the BIOS do not include this service. Finally, the service is not recognized in the DOS mode of OS/2.

If the BIOS mouse services of INT 15H were operational and compatible with standard mouse hardware, a program could use these functions much the same way as it uses the video, printer or the communications services in the BIOS. However, due to the difficulties mentioned in the preceding paragraph, most applications must find alternative ways of controlling mouse operation. But all alternative solutions have the disadvantage of requiring an installed mouse driver. To an application this leaves three alternatives: (1) the software must assume that the user has previously installed and loaded a compatible mouse driver, (2) the software must provide an installation routine that loads the driver, or (3) the code must include a low-level driver for the mouse device.

14.2.2 The Microsoft Mouse Interface

The mouse driver software that has achieved more general acceptance is the one by Microsoft Corporation. The Microsoft mouse control software is installed as a system driver or as a TSR program. The system version is usually stored in a disk file with the extension .SYS and the TSR version in a file with the extension .COM. The Microsoft mouse interface services are documented in the book Microsoft Mouse Programmer's Reference, published by Microsoft Press (see Bibliography).

Most manufacturers of mouse devices provide drivers that are compatible with the one by Microsoft. Therefore, the use of the Microsoft mouse interface is not limited to mouse devices manufactured by this company, but extends to all Microsoft-compatible hardware and software. The installation command for the mouse driver is usually included in the CONFIG.SYS or AUTOEXEC.BAT files. The

Microsoft mouse interface attaches itself to software interrupt 33H and provides a set of 36 sub-services. These mouse sub-services are accessible by means of an INT 33H instruction.

14.2.3 Checking Mouse Software Installation

We have mentioned that applications that use the mouse device must adopt one of three alternatives regarding the support software: assume that the driver was installed by the user, load a driver program, or provide the low-level services within its code. By far, most applications adopt the first option, that is, assume that the user has previously loaded the mouse driver software. Although the more refined programs that use a mouse device include an installation utility that selects the appropriate driver and creates or modifies a batch file in order to insure that the mouse driver is resident at the time of program execution.

In any case, the first operation usually performed by an application that plans to use the mouse control services in interrupt 33H is to test the successful installation of the driver program. Since the driver is vectored to interrupt 33H, this test consists simply of checking that the corresponding slot in the vector table is not a null value (0000:0000H) or an IRET operation code. Either one of these alternatives indicates that no mouse driver is presently available. The following coding template shows the required processing.

```
; Template file name: MOUSE1.TPL
; Code to check if mouse driver software is installed in the
; interrupt 33H vector. The check is performed by reading the
; interrupt 33H vector using MS-DOS service number 53,
; of INT 21H
        MOV     AH,53          ; MS_DOS service request
        MOV     AL,33H         ; Desired interrupt number
        INT     21H            ; MS-DOS service
; ES:BX holds address of interrupt handler, if installed
        MOV     AX,ES          ; Segment to AX
        OR      AX,BX          ; OR with offset
        JNZ     OK_INT33       ; Go if not zero
; Test for an IRET opcode in the vector
        CMP     BYTE PTR ES:[BX],0CFH   ; CFH is IRET opcode
        JNE     OK_INT33       ; Go if not IRET
; At this point the program should provide an error handler
; to exit execution or to load a mouse driver
        .
        .
        .
; Execution continues at this label if a valid address was found
; in the interrupt 33H vector
OK_INT33:
        .
        .
        .
```

14.2.4 Sub-services of Interrupt 33H

The Microsoft mouse interface was designed to provide control of the mouse device from high- and low-level languages. VGA alphanumeric programs can use the Microsoft mouse software by selecting one of two available text cursors. In the alpha modes the mouse driver manages the text cursor on a coarse grid of screen columns and rows, according to the active display mode. VGA programs that execute in graphics modes must provide their own cursor bitmap, which is installed by means of an interrupt 33H sub-service. However, since the graphics cursor operated by the driver is limited to a size of 16 by 16 pixels, many graphics programs create and manage their own cursor. In this case the driver services are used to detect mouse movements, but the actual cursor operation and display are handled directly by the application. This is also the case of XGA programs that use the sprite functions to manage a mouse cursor image

In addition to mouse cursor management and display, the sub-services of interrupt 33H include functions to set the mouse sensitivity and rate, to read button press information, to select video pages, and to initialize and install interrupt handlers that take control when the mouse is moved or when the mouse buttons are operated. However, some of the services in the interrupt 33H drivers reprogram the video hardware in ways that can conflict with an application. For this reason, we have limited our discussion to those mouse services that are not directly related to the video environment. These services can be used from any VGA, XGA, or SuperVGA graphics modes without interference. However, in this case, it is the application's responsibility to perform all video updates.

Sub-service 0 - Initialize Mouse

Sub-service number 0 of interrupt 33H is used to reset the mouse device and to obtain its status. An application usually calls this service to certify that the mouse driver is resident and to initialize the device parameters. The following coding template shows a call to this sub-service.

```
; Template file name: MOUSE2.TPL
; Initialize mouse by calling sub-service 0 of interrupt 33H
        MOV     AX,0            ; Reset mouse hardware and
                                ; software
        INT     33H             ; Mouse interrupt
        CMP     AX,0            ; Test for error during reset
        JNZ     OK_RESET        ; No problem
; At this point the program should provide an error routine to
; handle an invalid initialization call
        .
        .
        .
; Execution continues at this label if the mouse was initialized
OK_RESET:
        .
        .
        .
```

Sub-service 5 - Check Button Press Status

Programs that do not use interrupts can check mouse button press status by calling sub-service number 5 of the Microsoft mouse interface. The call is typically located in a polling loop. The calling program passes the button code in the BX register; the value of 0 corresponds to the left mouse button and a value of 1 to the right button. The call returns the button status in the AX register; bit 0 is mapped to the left mouse button and bit 1 to the right mouse button. A value of 1 indicates that the corresponding button is down. The BX register returns the number of button presses that have occurred since this call was last made or since a driver software reset (see sub-service 0 earlier in this section). The CX and DX registers hold the x and y cursor coordinates of the screen position where the last press occurred. The following coding template shows a call to this sub-service.

```
; Template file name: MOUSE3.TPL
;****************************************************************
;                   button action handler
;****************************************************************
; The following routine calls service 5 of interrupt 33H to
; detect mouse press action on the mouse device
; If the right button was pressed execution is directed to the
; label RIGHT_BUT, if the left button was pressed execution is
; directed to the label LEFT_BUT
;*********************|
;  check left button   |
;*********************|
;
        MOV     AX,5            ; Service request to read
                                ; mouse button status
        MOV     BX,0            ; First test left button
        INT     33H             ; Mouse interrupt
; Number of button presses is returned in the BX register
        CMP     BX,0            ; Test for no presses
        JE      TEST_RIGHT_BUT  ; Not pressed. Test right button
; Code at this point should take the program action corresponding
; to one or more presses of the left mouse button
        .
        .
        .
; Execution should be allowed to fall through to the right button
; test routine
;
;*********************|
;  check right button  |
;*********************|
TEST_RIGHT_BUT:
        MOV     AX,5            ; Service request to read
                                ; mouse button status
        MOV     BX,1            ; Test right button
        INT     33H             ; Mouse interrupt
; Number of button presses is returned in the BX register
        CMP     BX,0            ; Test for no presses
        JE      END_BUTTON_RTN  ; Not pressed. End of routine
; Code at this point should take the program action corresponding
; to one or more presses of the right mouse button
        .
        .
        .
; Button press status processing ends at this label
END_BUTTON_RTN:
        .
        .
        .
```

Sub-service 11 - Read Motion Counters

The actual movement of the mouse-controlled icon is dependent on the state of two counters maintained by the mouse interface software. The Microsoft mouse interface at interrupt 33H stores the motion parameters in 1/200-in units called mickeys. The changes in the motion counters represent values from the last time the function was called. Sub-service 11, of interrupt 33H, returns the values stored in the horizontal and vertical motion counters. The horizontal motion count is returned in the CX register and the vertical count in the DX register. The values are signed integers in two's complement form. A negative value in the horizontal motion counter indicates mouse movement to the left, while a negative value in the vertical motion counter indicates a movement in the upward direction. Both the vertical and the horizontal counters are automatically reset by the service routine.

We mentioned that the detection of mouse action can be by a polling loop or by interrupts. Polling loops are often used in reading the motion counters so as to keep interrupt processing times to a minimum, especially considering that the Microsoft mouse interface does not allow the installation of more than one service routine. The processing inside a polling loop or a service routine takes place in similar fashion. The following coding template shows the structure of a basic mouse movement handler.

```
; Template file name: MOUSE4.TPL
;****************************************************************
;                    mouse movement handler
;****************************************************************
; The following routine calls service 11 of interrupt 33H to
; detect horizontal or vertical movement of the mouse device
; If the movement is along the x axis (horizontal) execution is
; directed to the label H_MOVE, if the movement is along the y
; axis, execution is directed to the label Y_MOVE. If no change
; is detected in the motion counters, then execution is directed
; to the label NO_MOVE
;*********************|
;  service No. 11 of   |
;       INT 33H        |
;*********************|
        MOV     AX,11           ; Service request to read
                                ; motion counters
        INT     33H             ; Mouse interrupt
; CX = Horizontal mouse movement from last call to this service
; DX = vertical mouse movement from last call
        MOV     AL,CL           ; Horizontal counter to AL
        MOV     AH,DL           ; Vertical counter to AH
        CMP     AX,0            ; If AX is 0 then no mouse
        JNE     XORY_MOVE       ; Some movement detected
        JMP     NO_MOVE         ; Go if no movement
; At this point there is vertical or horizontal mouse movement
XORY_MOVE:
        CMP     CX,0            ; Test for no horizontal
        JE      Y_MOVE          ; Go to vertical movement test
;*********************|
;    horizontal move   |
;*********************|
; Code at this point moves the mouse icon according to the
; direction and magnitude of the value in the CX register
```

```
X_MOVE:
        PUSH    DX              ; Save vertical move counter
        .
        .
        .
        POP     DX              ; Restore vertical counter
; Once the horizontal movement is executed the code should fall
; through to the vertical movement routine. This takes care of
; the possibility of simultaneous movement along both axes
;**********************|
;     vertical move    |
;**********************|
; Code at this point moves the mouse icon according to the
; direction and magnitude of the value in the DX register
Y_MOVE:
        .
        .
        .
;**********************|
;    no movement       |
;**********************|
; This label is the routine's exit point
NO_MOVE:
        .
        .
        .
```

Sub-service 12 - Set Interrupt Routine

The user action on the mouse hardware can be monitored by polling or by interrupt generation, as is the case with most other input devices. Polling methods are based on querying the device status on a time lapse basis, therefore polling routines are usually coded as part of execution loops. In the case of the mouse hardware the polling routine can check the motion counter registers and the button press and release status registers that are maintained by the mouse interface software. The services to read these registers are described later in this section.

The second and often preferred method of monitoring user interaction with the mouse device, particularly mouse button action, is by means of hardware interrupts. In this technique the program enables the mouse hardware actions that generate interrupts and installs the corresponding interrupt handlers. Thereafter, user action on the enabled hardware sources in the mouse automatically transfers control to the handler code. This frees the software from polling frequency constraints and simplifies program design and coding.

A typical application enables mouse interrupts for one or more sources of user interaction. For example, a program that uses the mouse to perform menu selection would enable an interrupt for movement of the trackball (or other motion detector mechanism) and another interrupt for the action of pressing the left mouse button. If the mouse is moved, the interrupt handler linked to trackball movement changes the screen position of the marker or icon according to the direction and magnitude of the movement. If the left mouse button is pressed, the corresponding interrupt handler executes the selected menu option.

Another frequently used programming method is to poll the mouse motion counters that store trackball movement and to detect button action by means of interrupts. This design reduces execution time inside the interrupt handler, which can be an important consideration in time-critical applications. The MATCH demonstration program furnished in the book's software package uses a polling routine to move the mouse icon and an interrupt handler to detect button action.

In the mouse interface software, the hardware conditions that can be programmed to generate an interrupt are related to an integer value called the call mask. Figure 14.1 shows the call mask bitmap in the Microsoft mouse interface software. To enable a mouse interrupt condition the software sets the corresponding bit in the call mask. To disable a condition the call mask bit is cleared.

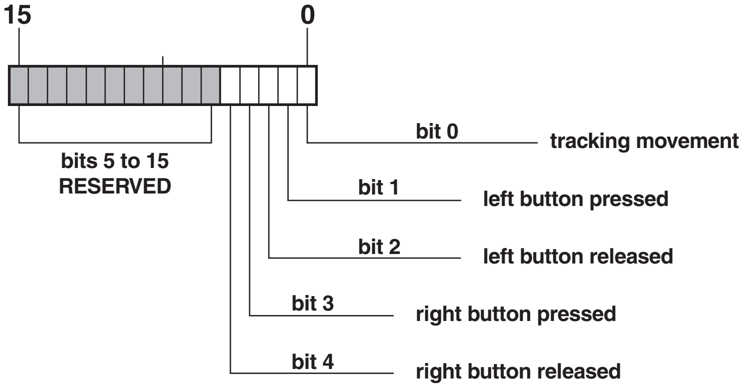


Figure 14-1 *Mouse Interrupt Call Mask*

Sub-service number 12 of the mouse interface at interrupt 33H provides a means for installing an interrupt handler and for selecting the action or actions that generate the interrupt. The following coding template shows the necessary processing for enabling mouse interrupts on right and left button pressed.

```
; Template file name: MOUSE5.TPL
; Select left mouse button pressed and right mouse button pressed
; as interrupt conditions and set address of service routine
; by means of mouse sub-service number 12, interrupt 33H
; The code assumes that the interrupt handler is located in the
; program's code segment, at the offset of the label named
; MOUSE_ACTION
      CLI                   ; Interrupts off
      PUSH    ES            ; Save video buffer segment
      PUSH    CS            ; Program's segment
      POP     ES            ; to ES
      MOV     AX,12         ; Mouse service number 12
; Interrupt mask bitmap:
;  15 ----------------------5 4 3 2 1 0
;  |-- these bits unused ---| | | | | |___ Tracking movement
;                             | | | |_____ Left button pressed
;                             | | |_______ Left button released
;                             | |_________ Right button pressed
;                             |___________ Right button released
```

```
MOV     CH,0            ; Unused bits
MOV     CL,00001010B    ; Interrupt on left button and
                        ; right button pressed
MOV     DX,OFFSET CS:MOUSE_ACTION ; Address of the
                        ; service routine
INT     33H             ; Mouse interrupt
POP     ES              ; Restore segment
STI                     ; Interrupts on
.
.
.
```

When the user's interrupt service routine receives control the mouse interface software passes a condition code in the AL register that matches the call mask bitmap (see Figure 14.1). In this manner the user's handler can determine which of the unmasked conditions actually generated the interrupt. An interrupt condition bit is set when the corresponding condition originated the interrupt. For example, if the conditions that originate the interrupt are the left or right mouse buttons pressed (as enabled by the previous coding template), then the program can test the state of bit number 1 (see Figure 14.1) to determine if the interrupt was caused by the left mouse button. If not, the code can assume that it was caused by the user pressing the right mouse button, since only these two conditions are active.

A characteristic of service number 12 or the Microsoft mouse interface is that only one interrupt handler can be installed. If two consecutive calls are made to this service, even if the call mask settings enable different bits, the address in the latest call replaces the previous one. Therefore it is not possible to install more than one service routine by means of this service. On the other hand, service number 24 allows the installation of more than one service routine, each one linked to a different interrupt cause. However, this service operates only when the Shift, Ctrl, or Alt keys are held down while the mouse action is performed. In addition, in several non-Microsoft versions of the mouse interface software the service does not perform as documented. For these reasons it is not considered in this book.

14.3 Image Animation

In the PC video animation usually consists of successively displaying images that vary in composition or in screen location according to a specific pattern of change. Notice that the concept of a pattern of change does not imply that this pattern be known beforehand to the software. For example, the image changes can be determined by user interaction or by the occurrence of random events. In this respect we can speak of the animation of object with predictable or unpredictable movements. The direction of movement of a mouse icon, for example, cannot be normally predicted by the software, therefore it falls in the second category. On the other hand, a graphics program could animate a screen object that moves in a predictable path across the screen. It is also possible for the movement of a screen object to contain both a predictable and an unpredictable element. For example, a mouse-controlled icon can be allowed to move inside a certain screen window, or the image of a planet that moves diagonally across the screen can exhibit random rotation on its own axis.

The combinations and variations of the predictable and unpredictable elements in the movement of screen objects can be quite complex. For example, the following screen image in an animated game could depend on screen objects with programmed movement, with random movement, and controlled by user interaction. The one common element to all three animated movements is the concept of a pattern of change, which means that the subsequent images of animated objects are somehow related to previous ones. The elements of this relationship are usually location, gradation (color hue), and object shape. In other words, to produce a realistically animated movement of a screen object the software must control the pattern of change. This usually implies restricting the transformations of location, gradation, and shape from one screen image of the object to the next one.

Many of the complexities of the theory and practice of computer image animation are beyond the scope of this book. In the bibliography we have listed some useful theoretical references in the field of computer graphics. However, computer animation in the PC is much more limited than in dedicated systems. The processing power of CPU and video hardware impose very restrictive limits on the number and size of objects that can be smoothly animated in this environment. The following discussion is also limited by these hardware limits.

14.3.1 Image Mapping and Panning

Image animation in raster scan systems is often based on manipulating a stored image map. This map can be located in a mechanical or optical device, in video memory, in ROM, or in the application's memory space. In previous chapters we have manipulated image maps contained in disk files, in ROM, in RAM, and in video memory. The storage location of the image map is often less important than its format. Bitmap formats and conventions are the subject of Chapter 10. Processing speed is usually an important consideration in image animation. Therefore the storage location for image maps is usually limited to the video memory and the applications's RAM space. The terms video buffer and image buffer are often used in this context.

Video and Image Buffers

While the video buffer is a physical device the concept of an image buffer is a logical one. Graphics systems use the concept of a virtual graphics device, which assumes an imaginary display of fictitious characteristics. Frequently, the attributes of the virtual machine exceed those of the physical one. Therefore, the capacity of the image buffer can exceed that of the video buffer. For example, a VGA system is equipped with a video buffer suitable for holding an image of 640-by-480 pixels in 16 colors. Yet a program running in the VGA environment may support an image buffer capable of storing 2000 by 1200 pixels in 512 colors.

We have made use of this concept in developing the calculation routines in the VGA libraries furnished with this book. In this manner the storage areas for screen coordinate points (named X_BUFFER and Y_BUFFER) in the VGA2 module are capable of storing 2048 values for each coordinate axis. This considerably exceeds the best available resolution in VGA systems, which is of 640 by 480 pixels. However, this additional storage space makes possible the use of the geometrical calculation routines in XGA and SuperVGA modes that have greater screen resolution (1,024 by

768 pixels) than the VGA. As far as the VGA calculation routines are concerned the limits of the video system are not those of the physical device (VGA, XGA, or SuperVGA) but those of an image buffer with a storage space for 2,048-by-2,048 pixels.

Viewport and Windows

The viewport is defined as the display area used for graphic operations. In IBM microcomputer graphics the entire display must be set for a chosen graphics or alphanumeric mode. Therefore the viewport is the entire display surface. In other words, the dimensions of the graphic viewport coincide with the those of the physical video display. A window is an area of the display surface, usually rectangular in shape. However, there is no reason for excluding windows of other shapes. In fact, circular and elliptical windows are visually pleasant and would serve to break the geometrical monotony of squares and rectangles.

A rectangular display window is usually defined by the coordinates of its start and end points. For example, on a 640-by-480 pixel display, a window filling the upper left quarter would have start coordinates (0,0) and end coordinates (320,240). Windows can also be defined descriptively; for example, we sometimes speak of the graphic window, the text window, and the menu window.

Panning

Image buffers, viewport, and windows are often used in producing a form of image animation called panning. In panning an image appears to move by changing the rectangular region of the image buffer that is mapped to the viewport or window. The elements of panning animation are shown in Figure 14.2.

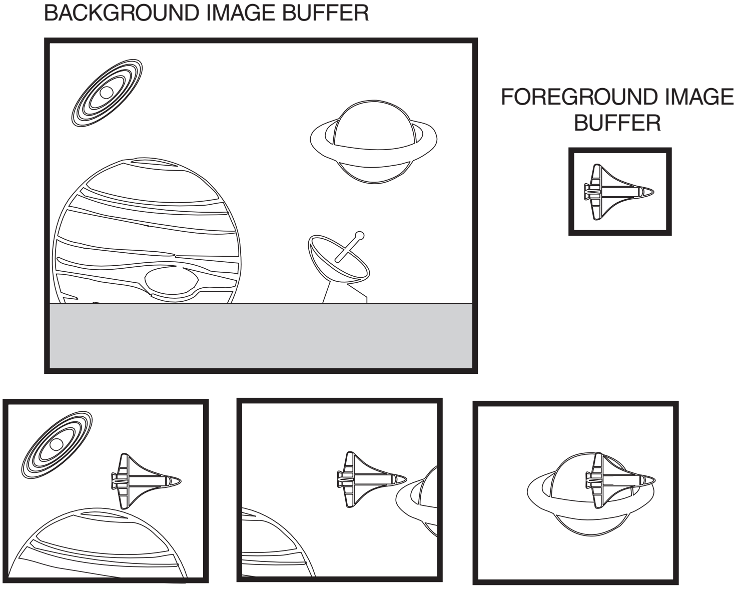


Figure 14-2 *Elements in Panning Animation*

In Figure 14.2 we can see that the viewport or window is smaller than the background image buffer. Therefore the display routine can show only a portion of the background image buffer at one time. A smooth panning effect can be produced on the video display by progressively changing the portion of the image buffer that is mapped to the viewport. An additional enhancement can be added in the form of a separate foreground screen object (in Figure 14.2 this object is a space shuttle). The foreground object is stored in its own image buffer (labeled the foreground image buffer in Figure 14.2). The panning effect can be further enhanced by changing the portion of the background image buffer mapped to the viewport, while the foreground object (in this example the shuttle image), remains in a fixed position. The resulting panning animation simulates the shuttle moving in space.

14.3.2 Geometrical Transformations

Graphical systems employ elaborate schemes for encoding image data. The purpose of these data structures is to facilitate image manipulation by hardware and software. The organization of graphical data is based, first, on identifying the fundamental image elements, such as lines, curves, arcs, polygons, and bitmaps. These primitive elements are stored in a logical structure called the display file. In turn, this display is composed of one or more modeling elements placed in structural levels sometimes called image files, image segments, and image descriptors. The design of graphical data storage devices and the manipulation of this data is a specialized field outside the scope of this book. The interested reader should consult a book on theoretical computer graphics (see Bibliography).

The subject of graphical data structures is related to animation by the fact that it is possible to transform a graphical image by performing logical and mathematical operations on the data structure that encodes it. In Chapter 5, starting in Section 5.3, we discussed geometrical transformations that are performed by manipulating image data. The most usual transformations are mirrowing, translation, rotation, scaling, and clipping. An animated effect can be achieved by performing and displaying progressive transformations of a graphical image. For example, a screen object can appear to be approaching the viewer by displaying a sequence of scaled images in which the object becomes progressively larger. Figure 14.3 shows how rotation and scaling transformations are used to simulate this effect.

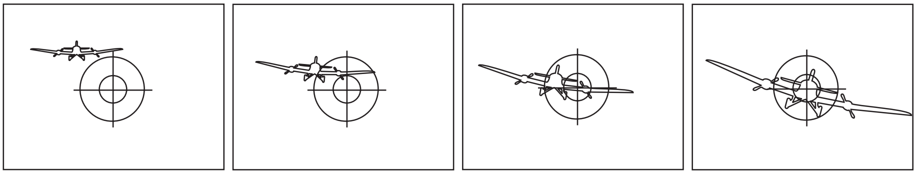


Figure 14-3 *Animation by Scaling and Rotation*

Notice, in Figure 14.3, that the simulation is enhanced by introducing a second, non-transformed object in the viewport (the reticle symbol). In any case of real-time animation by image transformation the quality of the simulation depends on the rate at which the successive images are displayed as well as on the rate of change between successive images. The faster the display rate and the slower the rate of image change, the more realistic the animation.

14.4 Imaging Techniques

We saw that computer animation often depends on the display of a series of images, called the image set. In some forms of animation the images themselves are progressively changed to form the image set. For example, panning animation is based on changing the portion of the image that is visible on the viewport. Other geometrical transformations can be used to generate the image set. In Figure 14.3 we see how scaling and rotation transformations are applied to a graphical object in order to simulate its approaching the viewer. In all cases, animation in real-time requires two separate programming steps: the creation of an image set and the sequential display of these images.

Many graphics and non-graphics techniques are used in the creation of an image set that follows a pre-defined pattern of change. We have mentioned how the image set can be generated by performing geometrical transformations on the display file. Hand-drawn or optically scanned bitmaps are also used to create the image set. Notice that the creation of the image set need not take place in real-time; it is its display that is time-critical. But whether the image set is in the form of geometrical display commands or encoded in consecutive bitmaps, the actual animation requires displaying these images consecutively, in real-time, and ideally, at a rate that is not less than the critical flicker frequency. In this section we discuss some programming methods used for displaying the animation image set in real-time.

14.4.1 Retention

We mentioned that the human visual organs retain, for a short time, the images of objects that no longer exist in the real world. This physiological phenomenon makes possible the creation of an illusion of animation by the frame-by-frame projection of a set of progressively changing images of a graphics object. We have referred to this collection of smoothly changing images as the animation image set. If the rate at which the individual images are shown on the video display is close to the critical rate of 22 images per second, then the animation appears smooth and pleasant. On the other hand, if the software cannot approximate this critical rate the user perceives a disturbing flicker and the animation appears coarse and bumpy to various degrees.

It is image retention which imposes performance requirements on real-time animated systems. If a computer animation program is to create a smooth and pleasant effect, all the manipulations and changes from image to image must be performed in less than 1/20 of a second. We mentioned that raster scan video systems, with bitmapped image buffers such as those in the PC, are not well suited for computer animation.

14.4.2 Interference

A raster scan display system is based on scanning each horizontal row of screen pixels with an electron beam. The pixel rows are usually scanned starting at the top-left screen corner and ending at the bottom-right corner. At the end of each pixel row, called a scan line, the electron beam is turned off while the gun is re-aimed to the start of the next scan line. When this row-by-row process reaches the bottom scan line, the beam is turned off while the gun is re-aimed to the top-left screen corner. The period of time required to re-aim the electron gun from the right-bottom of the screen to the top left corner is known as the vertical retrace or screen blanking cycle.

Some of the original graphics systems in the PC were prone to a form of display interference called snow. The direct cause for the interference was performing a buffer update during a screen refresh. Programmers soon discovered that on the CGA card this could be avoided or reduced by synchronizing the buffer updates with the period of time that the electron gun was turned off during vertical retrace. EGA and VGA systems were designed to avoid this form of interference when conventional methods of buffer update are used. However, the interference problem reapears when an EGA or VGA screen image has to be updated at short time intervals, as in animation.

The result is that, in order to avoid interference, the frequent screen updates required by most animation routines must be timed with the period during which the electron gun is off. This usually means synchronizing the buffer updated with the vertical retrace cycle of the CRT controller. This requirement, which applies to EGA, VGA, XGA, and SuperVGA systems, imposes a substantial burden on programs that perform animated graphics. For example, the screen refresh period in VGA graphics modes takes place at an approximate rate of 70 times per second. Since the individual images must be updated in the buffer while the electron gun is off, this gives the software 1/70th of a second to replace the old image with the new one. How much buffer update can be performed in 1/70 is the most limiting factor in programming smooth, real-time animation on IBM microcomputer video systems.

Notice that a screen refresh rate of approximately 1/70 considerably exceeds the critical jerkiness frequency of 1/24 used as the image refresh rate in motion picture technology (see Section 14.1.1). This difference is related to the time period required for the human eye to adjust to a light intensity change and detect flicker. We can speak of a critical flicker frequency, as different from the critical jerkiness frequency mentioned above. The motion picture projector contains a rotating diaphragm that blackens the screen only during the very short interval required to move the film to the next frame. This allows projection speeds to take place at the critical jerkiness rate rather than at the flicker rate. By the same token, a computer monitor must adjust the screen refresh cycle to this critical flicker frequency.

14.4.3 XOR Operations

In Section 14.1.2 we mentioned that in order to animate a screen object its image must be erased from current screen position before being redrawn at the new position. Otherwise the object's movement would leave an image track on the video display. The buffer update sequence takes the form: redraw, erase, redraw, erase, redraw. For example, in lateral translation, an object is made to appear to move across the screen, from left to right, by progressively redrawing and erasing its screen image at consecutively larger x coordinates. Notice that erasing the screen object is at least as time consuming as drawing it, since each pixel in the object must be changed to its previous state.

The are several ways of performing the redraw-erase cycle required in figure animation. The most obvious method is to save that portion of the screen image that is to be occupied by the object. The object can then be erased by re-displaying the saved image. The problem with this double pixBlt is that it requires a preliminary, and time-consuming, read operation to store the screen area that is to be occupied by the animated object. Therefore the redraw-erase cycle is performed by a video-to-RAM pixBlt (save screen), RAM-to-video pixBlt (display object), and another RAM-to-video pixBlt (restore screen).

A faster method of erasing and redrawing the screen is based on the properties of the logical exclusive or (XOR) operation. The action of the logical XOR is that a bit in the result is set if both operands contain opposite values. Consequently, XORing the same value twice restores the original contents, as in the following example:

```
          10000001B
XOR mask  10110011B
          ----------
          00110010B
XOR mask  10110011B
          ----------
          10000001B
```

Notice that the resulting bitmap (10000001B) is the same as the original one. The XOR method can be used in EGA, VGA, and SuperVGA systems because the Data Rotate register of the Graphics Controller can be programmed to write data normally, or to AND, OR, or XOR, the CPU data with the one in the latches. In XGA systems, mix mode number 06H produces a logical XOR of source and destination pixels (see Table 7-8).

The logical XOR operation provides a convenient and fast way for consecutively drawing and erasing a screen object. Its main advantage is that it does not require a previous read operation to store the original screen contents. This results in a faster and simpler read-erase cycle. The XOR method is particularly useful when more than one animated object can coincide on the same screen position since it insures that the original screen image is always restored.

The disadvantage of the XOR method is that the resulting image depends on the current screen contents. In other words, each individual pixel in the object displayed by means of a logical XOR operation is determined both by the XORed value and by the present pixel contents. For example, the following XOR operation produces a red object (in IRGB format) on a bright white screen background

```
              I R G B
background =  1 1 1 1  (bright white)
XOR mask   =  1 0 1 1
              -------
image      =  0 1 0 0  (red)
```

However, if the same XOR mask is used over a bright green background the resulting pixel is blue, as in the following example:

```
                  I R G B
background =      1 0 1 0  (bright green)
XOR mask   =      1 0 1 1
                  -------
image      =      0 0 0 1  (blue)
```

This characteristic of XOR operations, whereby an object's color changes as it moves over different backgrounds, can be an advantage or a disadvantage in graphics applications. For example, a marker symbol conventionally displayed will disappear as it moves over a background of its same color, while a marker displayed by means of a logical XOR can be designed to be visible over all possible backgrounds. On the other hand, the color of a graphics object might be such an important characteristic that any changes during display operations would be objectionable.

In conclusion, the peculiar effect of XOR operations on the object's color may not be objectionable, and even advantageous under some conditions, but in other applications it could make this technique unsuitable. More advanced video graphics systems include hardware support for animated imagery. In XGA, for example, the sprite mechanism allows for the display and movement of marker symbols or icons independently of the background. In this manner the XGA programmer can move the sprite symbol by defining its new coordinates. The XGA hardware takes care of erasing the old marker and restoring the underlaying image.

Programming the Function Select Bits

To make possible the XOR operation the software must manipulate the function select bits of the Graphics Controller Data Rotate register (see Section 2.2.4 and Table 2-6). The following code fragment shows the required processing.

```
; Set the Graphics Controller function select field of the Data
; Rotate register to the XOR mode
        MOV     DX,03CEH    ; Graphic controller port address
        MOV     AL,3        ; Select Data Rotate register
        OUT     DX,AL
        INC     DX          ; 03CFH register
        MOV     AL,00011000B    ; Set bits 3 and 4 for XOR
        OUT     DX,AL
```

Many conventional graphics operations, such as pixBlt and text display functions, require that the function select bits of the data rotate register be set for normal operation. The following code fragment shows the necessary processing.

```
; Set the Graphics Controller function select field of the Data
; Rotate register to the normal mode
        MOV     DX,03CEH    ; Graphic controller port address
        MOV     AL,3        ; Select Data Rotate register
        OUT     DX,AL
        INC     DX          ; 03CFH register
        MOV     AL,00000000B    ; Reset bits 3 and 4 for normal
        OUT     DX,AL
```

The procedure named LOGICAL_MODE in the VGA1 module of the GRAPHSOL library can be used to set the function select field of the Graphics Controller Data Rotate register to any one of four possible logical modes.

14.4.4 Time-Pulse Animation

Time-pulse animation is a real-time technique by which a screen object is successively displayed and erased at a certain rate. Ideally, the redraw rate in time-pulse animation should be higher than the critical jerkiness frequency of 20 images per second, although, in practice, the time pulse is often determined by the screen refresh rate.

Looping Techniques

The programmer has several methods of producing the timed pulse at which the animated image is updated. Which method is selected depends on the requirements of the application as well as on the characteristics of the video display hardware. The simplest method for updating the screen image of an animated object is by creating an execution loop to provide some form of timing device. But the loop must include not only the processing operations for updating the screen image, but also one or more polling routines. In addition, the loop's execution can be interrupted by hardware devices requiring processor attention. Another factor that can affect the precision of the loop timing is processor speed and and memory access facilities of the particular machine. The result is that an animation pulse created by loop methods is difficult to estimate, leading to non-uniform or unpredictable movement of the animated object.

The System Timer

Another time-pulse source available in the PC is the system's timer pulse. This pulse, which can be intercepted by an application, beats at the default rate of approximately 18.2 times per second. However, an application can reprogram the system timer to generate a faster rate. An interrupt intercept routine can be linked to the system timer so that the program receives control at every timer beat. If it were not for interference problems, the system timer intercept would be an ideal beat generator for use in animation routines.

The following coding template installs a system timer intercept routine. The installation routine accelerates the system timer from 18.2 to 54.6 beats per second, or three times the original rate.

```
; Template file name: ANIMATE1.TPL
;****************************************************************
;****************************************************************
;                    timer-driven pulse generator
;****************************************************************
;****************************************************************
;
; Changes performed during installation:
; 1. The BIOS system timer vector is stored in a code segment
;    variable
; 1. The timer hardware is made to run 3 times faster to ensure
;    a beat that is close to the critical flicker frequency
; 3. New service routine for INT 08H is installed in the
;    program's address space
;
; Operation:
```

```
; 3. The new interrupt handler at INT 08H gains control with
;    every bear of the system timer. The program maintains a
;    beat counter in the range 0 to 2. Every third beat
;    (counter = 2) execution is passed to the original INT 08H
;    handler in the BIOS in order to preserve the timer-dependent
;    services
;
CODE    SEGMENT

START:
        .
        .
        .
;****************************************************************
;            installation routine for INT 08H handler
;****************************************************************
; Operations:
;       1. Obtain vector for INT 08H and store in a CS variable
;          named OLD_VECTOR_08
;       2. Speed up system timer by a factor of 3
;       3. Set INT 08H vector to routine in this module
;****************************************************************
;**********************|
;   save old INT 08H   |
;**********************|
; Uses DOS service 53 of INT 21H
        MOV     AH,53           ; Service request number
        MOV     AL,08H          ; Code of vector desired
        INT     21H
; ES --> Segment address of installed interrupt handler
; BX --> Offset address of installed interrupt handler
        MOV     SI,OFFSET CS:OLD_VECTOR_08
        MOV     CS:[SI],BX      ; Save offset of original handler
        MOV     CS:[SI+2],ES    ; and segment
;**********************|
;   speed up system    |
;     timer by 3       |
;**********************|
; Original divisor is 65,536
; New divisor (65,536/3) = 21,845
;       CLI                     ; Interrupts off while write
                                ; LSB then MSM
                                ; xxxx 011x binary system
        OUT     43H,AL
        MOV     BX,21845        ; New divisor
        MOV     AL,BL
        OUT     40H,AL          ; Send LSB
        MOV     AL,BH
        OUT     40H,AL          ; Send MSB
;**********************|
; set new INT 08H in   |
;     vector table     |
;**********************|
; Mask off all interrupts while changing INT 08H vector
        CLI
; Save mask in stack
        IN      AL,21H          ; Read 8259 mask register
        PUSH    AX              ; Save in stack
        MOV     AL,0FFH         ; Mask off IRQ0 to IRQ7
        OUT     21H,AL          ; Write to 8259 mask register
```

```
; Install new interrupt vector
        MOV     AH,25H
        MOV     AL,08H          ; Interrupt code
        MOV     DX,OFFSET HEX08_INT
        INT     21H
; Restore original interrupt mask
        POP     AX              ; Recover mask from stack
        OUT     21H,AL          ; Write to 8259 mask register
        STI                     ; Set 80x86 interrupt flag
; At this point the graphics program continues execution
        .
        .
        .
;****************************************************************
;                        exit routine
;****************************************************************
; Before the program returns control to the operating system
; it must restore the hardware to its original state. This
; requires resetting the time speed to 18.2 beats per second
; and re-installing the BIOS interrupt handler in the vector
; table
;*********************|
;  reset system timer  |
;*********************|
; Original divisor is 65,536
        CLI                     ; Interrupts off while write
                                ; LSB then MSM
                                ; xxxx 011x binary system
        OUT     43H,AL
        MOV     BX,65535        ; Default divisor
        MOV     AL,BL
        OUT     40H,AL          ; Send LSB
        MOV     AL,BH
        OUT     40H,AL          ; Send MSB
;*********************|
;   restore INT 0AH    |
;*********************|
        PUSH    DS              ; Save program's DS
        MOV     SI,OFFSET CS:OLD_VECTOR_08
; Set DS:DX to original segment and offset of keyboard interrupt
        MOV     DX,CS:[SI]      ; DX --> offset
        MOV     AX,CS:[SI+2]    ; AX --> segment
        MOV     DS,AX           ; Segment to DS
        MOV     AH,25H          ; DOS service request
        MOV     AL,08H          ; Interrupt number
        INT     21H
        POP     DS
        STI                     ; Interrupts on again
; At this point the exiting program usually resets the video
; hardware to text mode and returns control to the operating
; system
        .
        .
        .
;****************************************************************
;                      new INT 08H handler
;****************************************************************
; The handler is designed so that a new timer tick cannot take
; place during execution. This is ensured by not sending the 8259
; end-of-interrupt code until the routine's processing is
; complete
;****************************************************************
HEX08_INT:
        STI                     ; Interrupts on
        PUSH    AX              ; Save registers used by routine
        PUSH    BX
        PUSH    CX              ; Other registers can be pushed
        PUSH    DX              ; if necessary
        PUSH    DS
; User video image update routine is coded at this point
        .
        .
        .
; The intercept routine maintains a code segment variable named
; TIMER_COUNT which stores a system timer pulse count. This
; variable is used to return control to the system timer
; interrupt every third timer beat, thus maintaining the
; original rate of 18.2 beats per second
        DEC     CS:TIMER_COUNT
        JZ      TIME_OF_DAY     ; Exit through time_of_day
;*********************|
;     direct exit     |
;*********************|
        MOV     AL,20H          ; Send end-of-interrupt code
        OUT     20H,AL          ; to 8259 interrupt controller
        POP     DS              ; Restore registers
        POP     DX
        POP     BX
        POP     AX
        IRET                    ; Return from interrupt
;*********************|
;   pass to original  |
;   INT 08H handler   |
;*********************|
TIME_OF_DAY:
        MOV     CS:TIMER_COUNT,2    ; Reset counter variable
        POP     DS
        POP     DX
        POP     BX
        POP     AX
        STC                     ; Continue processing
        JMP     DWORD PTR CS:OLD_VECTOR_08
        IRET
;*********************|
;  code segment data  |
;*********************|
TIMER_COUNT     DB      2       ; Timer counter
OLD_VECTOR_08   DD      0       ; Far pointer to original INT 08H
        .
        .
        .
;
CODE    ENDS
```

Interference Problems

PC software that uses the system timer to produce a pulse for animation routines encounter interference problems. At least two methods are available to avoid or minimize display interference: to turn-off the CRT while the buffer is being changed or to time the buffer updates with the vertical retrace cycle of the CRT controller.

Neither method is a panacea; as we have already mentioned it is not always possible to produce smooth real-time animation in an IBM microcomputer. Applications can try either or both methods and select the better option. The following coding template fragment shows the processing necessary to turn off the VGA video display system.

```
; Template file name: ANIMATE2.TPL
; Screen is turned off by setting the Clocking Mode register bit
; number 5 of the VGA Sequencer Group
        MOV     DX,03C4H        ; Sequencer group
        MOV     AL,01H          ; Clocking Mode register
        OUT     DX,AL           ; Select this register
        JMP     SHORT $+2       ; I/O delay
        INC     DX              ; To data port 3C5H
        IN      AL,DX           ; Read Clocking Mode register
        OR      AL,00100000B    ; Set bit 5, preserve others
        OUT     DX,AL           ; Write back to port
; At this point the VGA video display function is OFF
        .
        .
        .
```

The reverse process is necessary to turn on the VGA video display system.

```
; Template file name: ANIMATE3.TPL
; Screen is turned on by clearing the Clocking Mode register bit
; number 5 of the VGA Sequencer Group
        MOV     DX,03C4H        ; Sequencer group
        MOV     AL,01H          ; Clocking Mode register
        OUT     DX,AL           ; Select this register
        JMP     SHORT $ + 2     ; I/O delay
        INC     DX              ; To data port 3C5H
        IN      AL,DX           ; Read Clocking Mode register
        AND     AL,11011111B    ; Clear bit 5, preserve others
        OUT     DX,AL           ; Write back to port
; At this point the VGA video display function is ON
        .
        .
        .
```

The second method for reducing interference is to synchronize the video buffer update with the vertical retrace cycle of the CRT controller. In the following section we will see how, in some systems, we can enable an interrupt that occurs on the vertical retrace cycle. But whether the vertical retrace interrupt is available or not, it is possible to detect the start of the vertical retrace cycle in order to perform the buffer update operations while the CRT controller is turned off. The following coding template shows the processing necessary to detect the start of the vertical retrace in VGA systems.

```
; Template file name: ANIMATE4.TPL
; Test for start of the vertical retrace cycle of the CRT
; controller. Bit 3 of the Input Status Register 1 is set if a
; vertical cycle is in progress
        MOV     DX,3DAH         ; VGA Input Status register 1
VRC_CLEAR:
        IN      AL,DX           ; Read byte at port
        TEST    AL,00001000B    ; Is bit 3 set?
        JNZ     VRC_CLEAR       ; Wait until bit clear
```

```
; At this point the vertical retrace ended. Wait for it to
; restart
VRC_START:
        IN      AL,DX           ; Read byte at port
        TEST    AL,00001000B    ; Is bit 3 set?
        JZ      VRC_START       ; Wait until bit set
; At this point a vertical retrace cycle has just started
; The code can now proceed to update the video image
        .
        .
        .
```

Figure 7.7 is a bitmap of the Input Status register 0 and 1 of the VGA General Register Group. Notice that bit 7 of the Input Status register 0 can be used to detect the vertical retrace cycle only if the vertical retrace interrupt is enabled. If not, we must use bit 3 of Input Status register 1, as in the above code fragment.

14.4.5 The Vertical Retrace Interrupt

For many PC graphics applications the most satisfactory method for obtaining a timed pulse is by programming the CRT controller to generate an interrupt at the start of the vertical retrace cycle. The EGA, VGA, and XGA screen refresh rate, which is 70 cycles per second, is more than sufficient to produce smooth animation. In fact, the most important objection to this method is that it leaves very little time in which to perform image or data processing operations between timed pulses. Another consideration is that not all IBM and IBM-compatible video systems support a vertical retrace interrupt. For example, the IBM VGA Adapter is not documented to support the vertical retrace interrupt. The same applies to many VGA cards by third party vendors. Therefore VGA programs that use the vertical retrace interrupt may not be portable to these systems.

One advantage of using the vertical retrace interrupt as a time-pulse generator is that, since screen updates take place while the video system is turned off, interference is automatically avoided. The typical method of operation is to synchronize the screen update with the beginning of the vertical retrace cycle of the CRT controller. How much processing can be done while the CRT is off depends on the system hardware. In VGA systems this depends mainly on the type and speed of the CPU and the memory access facilities. XGA systems have their own graphics coprocessor and, for this reason, can execute considerably more processing during the vertical retrace cycle. Notice that in IBM XGA documentation the vertical retrace cycle is called the screen blanking period.

VGA Vertical Retrace Interrupt

In VGA systems the smooth animation of relatively small screen objects can be executed satisfactorily by vertical retrace synchronization. As the screen objects get larger it is more difficult to update the video buffer in the short time lapse of the vertical retrace cycle. Since so many performance factors enter into the equation it is practically impossible to give exact limits or guidelines for satisfactory animation. For example, the demonstration program, MATCH, furnished with the book's software package uses the vertical retrace interrupt to animate a running boar target. At the same time, the user interactively animates by mouse controls the image of a crosshair symbol. Both simultaneous animation operations used in the MATCH program tax

VGA and system performance to the maximum. For this reason the program requires an IBM microcomputer equipped with a 80386 or 486 processor to perform satisfactorily. A certain bumpiness is noticeable in the MATCH animation when the program executes in a 80286 or slower machine.

It is often possible to program around the limitations of vertical retrace timing. In the first place, the image update operation can be split into two or more vertical retrace cycles. This is possible because the jerkiness frequency of 20 cycles per second is considerably less than the typical vertical retrace pulse of 70 cycles per second. However, splitting the update operations introduces programming complications, as well as an additional overhead in keeping track of which portion of the image is to be updated in each cycle. This method should be considered only if no simpler solution is available.

We mentioned that in VGA the vertical retrace cycle of the CRT controller takes place at a rate of approximately 70 times per second. In VGA systems that support the vertical retrace interrupt, software can enable it as a pulse generator and install a routine that receives control on every vertical retrace cycle. The following coding template contains the program elements necessary for the installation and operation of a vertical retrace intercept in a VGA system.

```
; Template file name: ANIMATE5.TPL
;****************************************************************
;****************************************************************
;           vertical retrace interrupt pulse generator
;                       for VGA systems
;****************************************************************
;****************************************************************
;
; Operations performed during installation:
; 1. The VGA port base address is stored in a code segment
;    variable named CRT_PORT and the default contents of the
;    Vertical Retrace End register are stored in a variable
;    named OLD_VRE
; 2. The address of the interrupt 0AH handler is saved in a
;    far pointer variable named OLD_VECTOR_0A
; 3. A new handler for interrupt 0AH is installed at the label
;    HEX0A_INT.
; 4. The IRQ2 bit is enabled in the 8259 (or equivalent)
;    interrupt controller mask register
; 5. The vertical retrace interrupt is activated
;
; Operation:
;    The new interrupt handler at INT 0AH gains control with
;    every vertical retrace cycle of the CRT controller.
;    The software can perform limited buffer update operations
;    at this time without causing video interference
;****************************************************************
;                    Installation routine for
;                 the vertical retrace interrupt
;****************************************************************
; The following code enables the vertical retrace interrupt on
; a VGA system and intercepts INT 0AH (IRQ2 vector)
;*********************|
;    save parameters   |
;*********************|
; System port address is saved in CS variables
        CLI                     ; Interrupts off
        MOV     AX,0H           ; Clear AX
        MOV     ES,AX           ; and ES
        MOV     DX,ES:[0463H]   ; Get CRT controller base address
                                ; from BIOS data area
        MOV     CS:CRT_PORT,DX  ; Save address in memory variable
        MOV     AL,11H          ; Offset of Vertical Retrace End
                                ; register in the CRTC
        OUT     DX,AL           ; Select this register
; Value stored in port's data register is saved in a code segment
; variable for later use by the software
        INC     DX              ; Point to Data register
        IN      AL,DX           ; Read default value in register
        JMP     SHORT $+2       ; I/O delay
        MOV     CS:OLD_VRE,AL   ; Save value in variable
;*********************|
;   save old INT 0AH  |
;*********************|
; Uses DOS service 53 of INT 21H to store the address of the
; original INT 0AH handler in a code segment variable
        MOV     AH,53           ; Service request number
        MOV     AL,0AH          ; Code of vector desired
        INT     21H
; ES --> Segment address of installed interrupt handler
; BX --> Offset address of installed interrupt handler
        MOV     SI,OFFSET CS:OLD_VECTOR_0A
        MOV     CS:[SI],BX      ; Save offset of original handler
        MOV     CS:[SI+2],ES    ; and segment
;*********************|
; install this INT 0AH |
;        handler      |
;*********************|
; Uses DOS service 37 of INT 21H to install the present handler
; in the vector table
        MOV     AH,37           ; Service request number
        MOV     AL,0AH          ; Interrupt code
        PUSH    DS              ; Save data segment
        PUSH    CS
        POP     DS              ; Set DS to CS for DOS service
        MOV     DX,OFFSET CS:HEX0A_INT
        INT     21H
        POP     DS              ; Restore local data
;*********************|
;    enable IRQ2      |
;*********************|
; Clear bit 2 of the 8259 Mask register to enable the IRQ2 line
        CLI                     ; Make sure interrupts are off
        MOV     DX,21H          ; Port address of 8259 Mask
                                ; register
        IN      AL,DX           ; Read byte at port
        AND     AL,11111011B    ; Mask for bit 2
        OUT     DX,AL           ; Back to 8259 port
;*********************|
;  activate vertical  |
;  retrace interrupt  |
;*********************|
        MOV     DX,CS:CRT_PORT  ; Recover CRT base address
        MOV     AL,11H          ; Offset of Vertical Retrace End
                                ; register in the CRTC
        MOV     AH,CS:OLD_VRE   ; Default value in Vertical
                                ; Retrace End register
        AND     AH,11001111B    ; Clear bits 4 and 5 in VRE
                                ; Bit 4 = clear vertical
                                ; interrupt
                                ; Bit 5 = enable vertical retrace
        OUT     DX,AX           ; To port
        OR      AH,00010000B    ; Mask to set bit 4 to re-enable
        OUT     DX,AX
        STI                     ; Enable interrupts
; At this point the vertical retrace interrupt is active
; Program code to follow
        .
        .
;****************************************************************
;                        exit routine
;****************************************************************
; Before the program returns control to the operating system
; it must restore the hardware to its original state. This
; requires disabling the vertical retrace interrupt and restoring
; the original INT 0AH handler in the vector table
;**********************|
;   disable vertical   |
;      interrupts      |
;**********************|
; Code assumes that on program entry the vertical retrace
; was disabled
        MOV     DX,CS:CRT_PORT  ; Recover CRT base address
        MOV     AL,11H          ; Offset of Vertical Retrace End
                                ; register in the CRTC
        MOV     AH,CS:OLD_VRE   ; Default value in Vertical
                                ; Retrace End register
        OUT     DX,AX           ; To port
;**********************|
;   restore original   |
;   INT 0AH handler    |
;**********************|
        MOV     SI,OFFSET CS:OLD_VECTOR_0A
; Set DS:DX to original segment and offset of keyboard interrupt
        MOV     DX,CS:[SI]      ; DX --> offset
        MOV     AX,CS:[SI+2]    ; AX --> segment
        MOV     DS,AX           ; segment to DS
        MOV     AH,25H          ; DOS service request
        MOV     AL,0AH          ; IRQ2
        INT     21H
; At this point the exiting program usually resets the video
; hardware to a text mode and returns control to the operating
; system
        .
        .
;****************************************************************
;              VGA vertical retrace interrupt handler
;****************************************************************
; The following routine gains control with every vertical retrace
; interrupt (approximately 70 times per second)
; The code can now perform limited video buffer update operations
; without interference
; The vertical retrace interrupt is not re-enabled until the
; routine has concluded to avoid re-entrancy
;****************************************************************
```

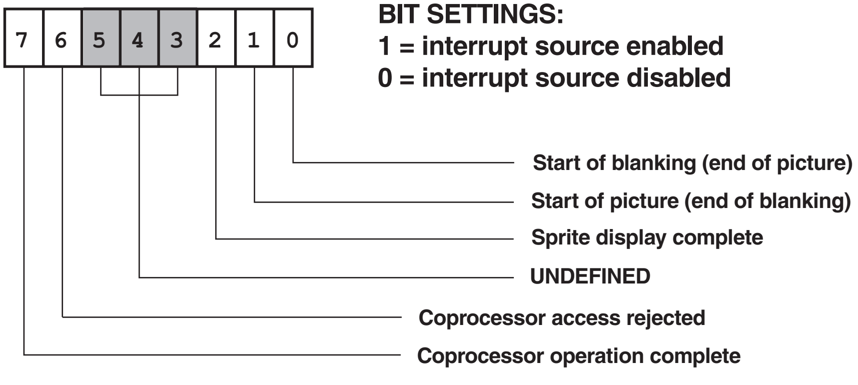
```
HEX0A_INT:
        CLI                     ; Interrupts off
; Save registers
        PUSH    AX              ; Save context at interrupt time
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    ES
;**********************|
;  test for vertical   |
;  retrace interrupt   |
;**********************|
; Since several hardware interrupts can be located at IRQ2 the
; software must make sure that it was the vertical retrace that
; originated this action. This is done by testing bit 7 of the
; Input Status Register 0, which will be set if a vertical
; retrace interrupt has occurred
        MOV     DX,3C2H         ; Input Status Register 0
        IN      AL,DX           ; Read byte at port
        TEST    AL,10000000B    ; Is bit 7 set
        JNE     VRI_CAUSE       ; Go if vertical retrace
;**********************|
; chain to next handler|
;**********************|
; At this point the interrupt was not due to a vertical retrace
; Execution is returned to the IRQ2 handler
        POP     ES              ; Restore context
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        STC                     ; Continue processing
        JMP     DWORD PTR CS:OLD_VECTOR_0A
;**********************|
; animation operations |
;**********************|
VRI_CAUSE:
; At this point the handler contains the graphics operations
; necessary to perform the animation function
        .
        .
;**********************|
; service routine exit |
;**********************|
; Enable 8259 interrupt controller to receive other interrupts
        MOV     AL,20H          ; Port address
        OUT     20H,AL          ; Send EOI code
; Re-enable vertical retrace interrupt by clearing bits 4 and 5
; of the Vertical Retrace End register and then setting bit 5
; so that the interrupt is not held active
        MOV     DX,CS:CRT_PORT  ; Recover CRT base address
        MOV     AL,11H          ; Offset of Vertical Retrace End
                                ; register in the CRTC
        MOV     AH,CS:OLD_VRE   ; Default value in VRE register
        AND     AH,11001111B    ; Clear bits 4 and 5
                                ; 4 = clear vertical interrupt
                                ; 5 = enable vertical retrace
        OUT     DX,AX           ; To port
        OR      AH,00010000B    ; Set bit 4 to reset flip-flop
        OUT     DX,AX           ; To port
```

```
;*********************|
;   restore context    |
;*********************|
; Registers used by the service routine are restored from the
; stack
        POP     ES
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        STI                     ; Re-enable interrupts
        IRET
;****************************************************************
;                          code segment data
;****************************************************************
OLD_VECTOR_0A   DD      0       ; Pointer to original INT 0AH
                                ; interrupt
CRT_PORT        DW      0       ; Address of CRT controller
OLD_VRE         DB      0       ; Original contents of VRE
                                ; register
        .
        .
        .
```

Applications can extend the screen update time by locating the animated image as close as possible to the bottom of the video screen. In this manner the interference-free period includes not only the time lapse during which the beam is being diagonally re-aimed, but also the period during which the screen lines above the image are being scanned. This technique is used in the MATCH program included in the book's software package.

XGA Screen Blanking Interrupt

The XGA documentation refers to the vertical retrace cycle as the screen blanking period. Two interrupts sources are related to the blanking period: the start of picture interrupt and the start of blanking interrupt. The start of picture coincides with the end of the blanking period. Both interrupts are enabled in the XGA Interrupt Enable register (offset 21x4H). Figure 14.4 shows a bitmap of the XGA Interrupt Enable register.



7
6
5
4
3
2
1
0
BIT SETTINGS:
1 = interrupt source enabled
0 = interrupt source disabled
Start of blanking (end of picture)
Start of picture (end of blanking)
Sprite display complete
UNDEFINED
Coprocessor access rejected
Coprocessor operation complete

Figure 14-4 *XGA Interrupt Enable Register Bitmap*

Like the VGA interrupts, the XGA video interrupts are vectored to the IRQ2 line of the 8259/A (or compatible) interrupt controller chip, which is mapped to the 0AH vector. By testing the bits in the Interrupt Status register (at offset 21x5H) an XGA program can determine the cause of an interrupt on this line. Figure 14.5 shows a bitmap of the XGA Interrupt Status register.

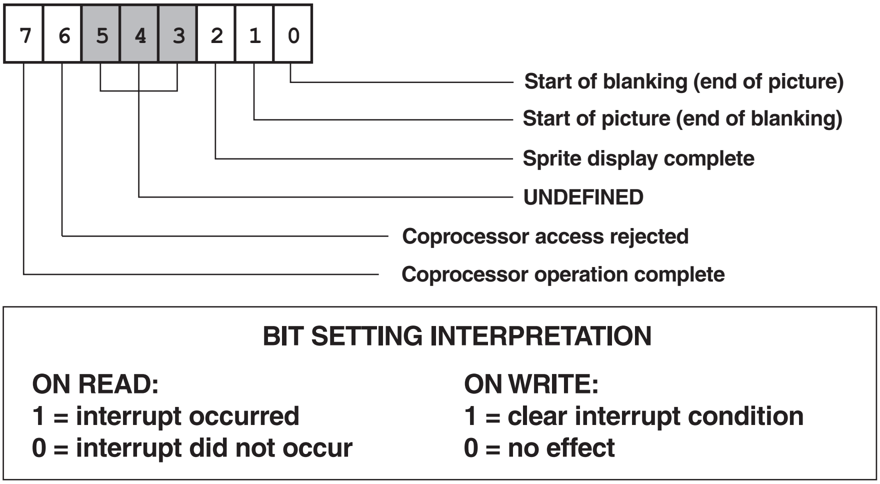

7 6 5 4 3 2 1 0
Start of blanking (end of picture)
Start of picture (end of blanking)
Sprite display complete
UNDEFINED
Coprocessor access rejected
Coprocessor operation complete
BIT SETTING INTERPRETATION
ON READ:
1 = interrupt occurred
0 = interrupt did not occur
ON WRITE:
1 = clear interrupt condition
0 = no effect

Figure 14-5 *XGA Interrupt Status Register Bitmap*

The XGA Interrupt Status register is also used to clear an interrupt condition. This operation is performed by the handler in order to reset the interrupt origin. The following template contains the program elements necessary for the installation and operation of a vertical retrace intercept in an XGA system.

```
; Template file name: ANIMATE6.TPL
;****************************************************************
;****************************************************************
;           screen blanking interrupt pulse generator
;                       for XGA systems
;****************************************************************
;****************************************************************
;
; Operations performed during installation:
; 1. The XGA port base address is stored in a code segment
;    variable named XGA_BASE
; 2. The address of the interrupt 0AH handler is saved in a
;    far pointer variable named OLD_VECTOR_0A
; 3. A new handler for interrupt 0AH is installed at the label
;    XGA_0A_INT.
; 4. The IRQ2 bit is enabled in the 8259 (or equivalent)
;    Interrupt Controller Mask register
; 5. The XGA screen blanking interrupt is enabled
;
; Operation:
; 3. The new interrupt handler at INT 0AH gains control with
;    every vertical retrace cycle of the CRT controller.
;    The software can perform limited buffer update operations
;    at this time without causing video interference
```

```
;
;****************************************************************
;                    Installation routine for
;                the XGA screen blanking interrupt
;****************************************************************
; The following code enables the screen blanking interrupt on
; a XGA system and intercepts INT 0AH (IRQ2 vector)
;**********************|
;      init XGA        |
;**********************|
; XGA initialization is performed by means of the services in the
; XGA1 and XGA2 modules of the GRAPHSOL library
        CALL    OPEN_AI      ; Open Adapter Interface for use
        CALL    INIT_XGA      ; Initialize XGA hardware
; The INIT_XGA procedure returns the address of the XGA register
; base in the BX register. The code stores this value in a code
; segment variable named XGA_BASE
        MOV     CS:XGA_BASE,BX ; Store in code segment variable
        MOV     AL,2           ; Select mode XGA mode number 2
                               ; 1024 by 768 pixels in 256 colors
        CALL    XGA_MODE       ; Mode setting procedure
;**********************|
;   save old INT 0AH   |
;**********************|
; Uses DOS service 53 of INT 21H to store the address of the
; original INT 0AH handler in a code segment variable
        MOV     AH,53           ; Service request number
        MOV     AL,0AH          ; Code of vector desired
        INT     21H
; ES --> Segment address of installed interrupt handler
; BX --> Offset address of installed interrupt handler
        MOV     SI,OFFSET CS:OLD_VECTOR_0A
        MOV     CS:[SI],BX      ; Save offset of original handler
        MOV     CS:[SI+2],ES    ; and segment
;**********************|
; install this INT 0AH |
;        handler       |
;**********************|
; Uses DOS service 37 of INT 21H to install the present handler
; in the vector table
        MOV     AH,37           ; Service request number
        MOV     AL,0AH          ; Interrupt code
        PUSH    DS              ; Save data segment
        PUSH    CS
        POP     DS              ; Set DS to CS for DOS service
        MOV     DX,OFFSET CS:XGA_0A_INT
        INT     21H
        POP     DS              ; Restore local data
;**********************|
;    enable IRQ2       |
;**********************|
; Clear bit 2 of the 8259 Mask register to enable the IRQ2 line
        CLI                     ; Make sure interrupts are off
        MOV     DX,21H          ; Port address of 8259 Mask
                                ; register
        IN      AL,DX           ; Read byte at port
        AND     AL,11111011B    ; Mask for bit 2
        OUT     DX,AL           ; Back to 8259 port
;**********************|
;  activate XGA screen |
;  blanking interrupt  |
;*********************|
; Reset all interrupts in the Status register
        MOV     DX,CS:XGA_BASE  ; Base address of XGA video
        ADD     DX,05H          ; Interrupt Status register
        MOV     AL,0C7H         ; All ones
        OUT     DX,AL           ; Reset all bits
; Enable the start of blanking cycle interrupt source (bit 0)
        MOV     DX,CS:XGA_BASE  ; XGA base address
        ADD     DX,04H          ; Interrupt Enable register
        IN      AL,DX           ; Read register contents
        OR      AL,00000001B    ; Make sure bit 0 is set
        OUT     DX,AL           ; Back to Interrupt Enable
                                ; register
        STI                     ; Interrupts ON
; At this point the XGA start of blanking interrupt is active
; Program code to follow
        .
        .
        .
;****************************************************************
;                        exit routine
;****************************************************************
; Before the program returns control to the operating system
; it must restore the hardware to its original state. This
; requires disabling the XGA screen blanking interrupt and
; restoring the original INT 0AH handler in the vector table
;*********************|
;  disable XGA screen  |
;  blanking interrupt  |
;*********************|
        MOV     DX,CS:XGA_BASE  ; XGA base address
        ADD     DX,04H          ; Interrupt Enable register
        IN      AL,DX           ; Read register contents
        AND     AL,11111110B    ; Make sure bit 0 is clear
        OUT     DX,AL           ; Back to Interrupt Enable
                                ; register
;*********************|
;   restore original   |
;   INT 0AH handler    |
;*********************|
        MOV     SI,OFFSET CS:OLD_VECTOR_0A
; Set DS:DX to original segment and offset of keyboard interrupt
        MOV     DX,CS:[SI]      ; DX --> offset
        MOV     AX,CS:[SI+2]    ; AX --> segment
        MOV     DS,AX           ; segment to DS
        MOV     AH,25H          ; DOS service request
        MOV     AL,0AH          ; IRQ2
        INT     21H
; At this point the exiting program usually resets the video
; hardware to a text mode and returns control to the operating
; system
        .
        .
        .
;****************************************************************
;              XGA screen blanking interrupt handler
;****************************************************************
; The following routine gains control with every vertical retrace
; interrupt (approximately 70 times per second)
; The code can now perform limited video buffer update operations
; without interference
; In order to avoid interrupt re-entrancy, the screen blanking
; interrupt is not re-enabled until the routine has concluded
;****************************************************************
XGA_0A_INT:
        CLI                     ; Interrupts off
; Save registers
        PUSH    AX              ; Save context at interrupt time
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    ES
;**********************|
;   test for screen    |
;  blanking interrupt  |
;**********************|
; Since several hardware interrupts can be located at IRQ2 the
; software must make sure that it was screen blanking that
; originated this action. This can be done by testing bit 0 of
; the XGA Interrupt Status register
        MOV     DX,CS:XGA_BASE  ; XGA base address
        ADD     DX,05H          ; Interrupt Status register
        IN      AL,DX           ; Read register contents
        TEST    AL,00000001B    ; Test start of blanking bit
        JNZ     BLK_CAUSE       ; Go if bit set
;**********************|
; chain to next handler|
;   if not blanking    |
;**********************|
; At this point the interrupt was not due to an XGA screen
; blanking interrupt. Execution is returned to the IRQ2 handler
        POP     ES              ; Restore context
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        STC                     ; Continue processing
        JMP     DWORD PTR CS:OLD_VECTOR_0A
;**********************|
; animation operations |
;**********************|
BLK_CAUSE:
; At this point the handler contains the graphics operations
; necessary to perform the animation function
        .
        .
        .
;**********************|
; service routine exit |
;**********************|
; Enable 8259 interrupt controller to receive other interrupts
        MOV     AL,20H          ; Port address
        OUT     20H,AL          ; Send EOI code
; The handler must reset bit 0 of the XGA Interrupt Status
; register to clear the interrupt condition
        MOV     DX,CS:XGA_BASE  ; Display controller base address
        ADD     DX,05H          ; Interrupt Status register
        IN      AL,DX           ; Read status
        OR      AL,00000001B    ; Set bit 0, preserve others
        OUT     DX,AL           ; Reset start of blanking
;**********************|
;   restore context    |
;**********************|
; Registers used by the service routine are restored from the
; stack
        POP     ES
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        STI                     ; Re-enable interrupts
        IRET
;****************************************************************
;                         code segment data
;****************************************************************
OLD_VECTOR_0A   DD      0       ; Pointer to original INT 0AH
                                ; interrupt
XGA_BASE        DW      0       ; Address of CRT controller
        .
        .
        .
```

The comparatively high performance of the XGA system makes possible the smooth animation of images much larger and elaborate than those that can be animated in VGA. Whenever possible the animation routine should use direct coprocessor programming (see Chapter 12) in order to minimize execution time. The system memory to video RAM pixBlt operation discussed in Section 12.5.3 can often be used in XGA animation.

Chapter 15

DOS Bitmapped Graphics

Topics:

- Image file encoding
- GIF file format
- LZW compression
- TIFF file format
- TIFF packBits compression
- PCL format for bitmapped fonts

This chapter describes the various techniques and standards used in encoding computer graphics images into units of memory storage. It includes a discussion of three popular image data storage formats: GIF, TIFF format, and PCL bitmapped fonts, also of the various data compression methods used in reducing the size of image data files, such as PackBits and LZW.

15.1 Image File Encoding

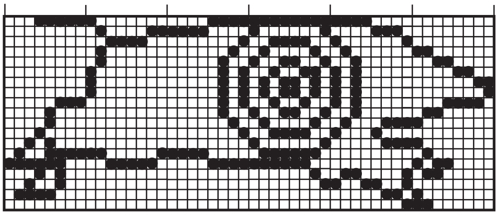
Bitmapping is the graphics technique whereby a memory bit represents the attribute of a screen pixel. In previous chapters we created and manipulated bitmapped image in an intuitive and almost primitive manner. The encodings were tailored to the specific video hardware; for example, in 16-color modes we used a 4-bit image code in IRGB format, and in 256-color modes, a double-bit format based on an IIRRGGBB encoding. In all cases the encodings we so far used have contained little more than the image's pixel-by-pixel color for a particular display system setup.

However, a graphics image can be encoded in more a complete and efficient structure than is offered by a pixel-by-pixel attribute list. A limitation of a raw pixel color list is that in most IBM graphics systems the pixel attribute is not a color code in itself, but an index into a color look-up table. For example, in XGA 256-color modes the pixel value 00001100B is displayed as bright red if the LUT registers are in the default setting, but the same code corresponds to a light shade of green if the LUT is changed to the IIRRGGBB encoding (see the XGALUT program in the book's software package). This means that the actual pixel code is meaningless if the image encoding does not offer information about the LUT register setting. LUT register data can be furnished implicitly, by designating a conventional format, such as IRGB, or explicitly, as a list of values to be loaded into the DAC registers.

The movement towards the standardization of image file encodings in IBM microcomputers originated with commercial software developers in need of methods for storing and displaying graphics images. At the present time there are over 20 different image file encodings in frequent use. It is common for a graphics application import or export service to present the user with over a dozen image file formats. Although some of these commercial encodings have gained more popularity than others, very little has been achieved in standardizing image file encodings for IBM microcomputers. In this chapter we have selected the image file formats that we believe are more useful and that have gained more widespread acceptance in the IBM microcomputer field. This selection does not imply that we endorse these particular encodings or approve of their design or operation.

15.1.1 Raw Image Data

We mentioned that the simplest possible image data encoding is a bare list of pixel attributes. This simple encoding, called the raw image data, is often all that is required by a graphics application. For example, the monochrome bitmap of a running boar target is encoded in the MATCH program (see book's software package) as raw image data. Figure 15-1 shows the bitmap and pixel list.



```
 1.  1FH 80H 0FH FFH F0H 00H      11. 08H 00H 02H 42H 47H 80H
 2.  00H 43H F0H 81H 0EH 00H      12. 10H 00H 01H 3CH 88H 00H
 3.  00H 3CH 01H 3CH 81H 00H      13. 28H 00H 00H 81H 07H 80H
 4.  00H 40H 02H 42H 40H C0H      14. 5FH C1H F0H 3FH 00H 40H
 5.  00H 40H 04H 99H 20H 30H      15. FCH 3EH 0FH FCH 00H B0H
 6.  00H 80H 05H 24H A0H 0CH      16. 14H 00H 00H 02H 61H 60H
 7.  00H 80H 05H 5AH A0H 03H      17. 24H 00H 00H 01H 99H 00H
 8.  00H 80H 05H 5AH A0H 01H      18. 78H 00H 00H 00H 06H 80H
 9.  07H 00H 05H 24H A0H 1EH      19. 00H 00H 00H 00H 01H C0H
10.  08H 00H 04H 99H 20H 60H
```

Figure 15-1 *Raw Image Data for a Monochrome Bitmap*

Since the image in Figure 15-1 is displayed in monochrome, the encoding is based on a bit per pixel scheme; a 1-bit in the attribute list indicates that the screen pixel is set, a 0-bit indicates that it remains in the background attribute. The reader can match the first line of the encoding (1FH 80H 0FH FFH F0H 00H) with the pixels on the top image row. The first value on the list (1FH = 00011111B) corresponds to the first eight image pixels, the second value on the list (80H = 10000000B) corresponds to the next eight image pixels, and so forth to the last value on the list.

But a display routine usually requires more data that can be encoded in a pixel attribute list. For example, the procedure named MONO_MAP_18 in the VGA2 module of the GRAPHSOL library requires the x and y screen coordinates, the color attribute, and the number of pixel rows and columns in the bitmap. This data is furnished to the MONO_MAP_18 procedure in a preamble data block that precedes the pixel attribute list. The following code fragment corresponds to the image block for the left-hand running boar target used in the MATCH program (see the MATCHC.ASM module in the book's software package).

```
;***********************|
; left-to-right boar    |
;***********************|
; Block control area:                      Displacement -->
LPIG_X  DW      4           ; Present x coordinate         0
LPIG_Y  DW      440         ; y coordinate                 2
        DB      19          ; Horizontal rows in block     4
        DB      6           ; Number of bytes per row      5
; Pixel attribute list for the left-hand running boar target
        DB      01FH,080H,00FH,0FFH,0F0H,000H ; 1
        DB      000H,043H,0F0H,081H,00EH,000H ; 2
        DB      000H,03CH,001H,03CH,081H,000H ; 3
        DB      000H,040H,002H,042H,040H,0C0H ; 4
        DB      000H,040H,004H,099H,020H,030H ; 5
        DB      000H,080H,005H,024H,0A0H,00CH ; 6
        DB      000H,080H,005H,05AH,0A0H,003H ; 7
        DB      000H,080H,005H,05AH,0A0H,001H ; 8
        DB      007H,000H,005H,024H,0A0H,01EH ; 9
        DB      008H,000H,004H,099H,020H,060H ; 10
        DB      008H,000H,002H,042H,047H,080H ; 11
        DB      010H,000H,001H,03CH,088H,000H ; 12
        DB      028H,000H,000H,081H,007H,080H ; 13
        DB      05FH,0C1H,0F0H,03FH,000H,040H ; 14
        DB      0FCH,03EH,00FH,0FCH,000H,0B0H ; 15
        DB      014H,000H,000H,002H,061H,060H ; 16
        DB      024H,000H,000H,001H,099H,000H ; 17
        DB      078H,000H,000H,000H,006H,080H ; 18
        DB      000H,000H,000H,000H,001H,0C0H ; 19
        DW      0000H                       ; padding
;
BOAR_COLOR      DB      00000100B       ; Red bit set
```

Notice that the pixel attribute list in the above code fragment corresponds to the raw data in Figure 15-1, and also that the display color is encoded in a separate variable (named BOAR_COLOR) whose address is passed to the MONO_MAP_18 display routine in the BX register. The block format in the above image is customized to store the data necessary to the MONO_MAP_18 display routine. The advantage of this method is that only the necessary data for the display manipulations is encoded with the raw pixel attribute list. This provides a compact data structure which can be used in optimizing the code. On the other hand, this customized encoding would almost certainly not be portable to any other graphics application.

The program designer must often decide whether to use a customized format that usually includes only the data that is strictly necessary for the display routine, or to represent the image in one of the general purpose formats that are recognized by other graphics applications. The basis for this decision is usually one of image portability. A stand-alone program (such as MATCH) which has no need to communicate graphics data to other applications, can use a raw data format whenever it is convenient. On the other hand, an application that must exchange image data with other graphics programs could benefit from adopting one of the existing image data formats described later in this chapter.

15.1.2 Bitmaps in Monochrome and Color

Etymologically, the term monochrome means "of one color;" however, in computer jargon, it is often interpreted as black-and-white. This equivalency is certainly untrue in bitmapped graphics, because a monochrome bitmap can be displayed in any available color or attribute. Furthermore, it is possible to combine several monochrome bitmaps to form a multicolor image on the screen. For example, several of the color images used in the MATCH program (furnished in the book's software package) are composites formed by overlaying separate monochrome bitmaps. The image of the rifle in the initial MATCH screen is formed by overlaying the monochrome bitmaps shown in Figure 15-2.

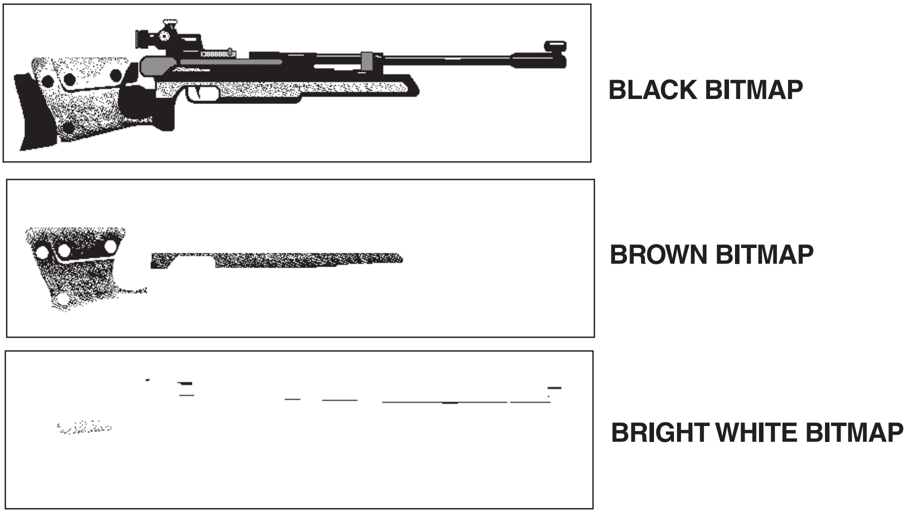


Figure 15-2 *Monochrome Overlays to Form a Color Image*

The original image of the rifle used in the first screen of the MATCH program was scanned from a black-and-white catalog illustration into a bitmap editing program. The three color overlays in Figure 15-2 were created by editing the original scan. The overlays were then saved into disk based image files in the TIFF format (discussed later in this chapter). The MATCH program successively reads and displays the three monochrome bitmaps and superimposes them to form a multicolor image. Notice that the order in which the bitmaps are displayed is important, because if two overlays contain a common pixel, this pixel is shown in the attribute of the last bitmap displayed.

A color image can also be stored in a single bitmap in which each pixel is represented in any of the available colors. The result is a more compact image file and a faster display operation. In fact, the only reasons for using several monochrome bitmaps in the creation of a color image are convenience and limited resources. The raw pixel data format for a color image often matches the characteristics of the video system for which it is intended. In VGA, SuperVGA, and XGA systems color images are typically stored in 16 or 256 colors. We already mentioned that, in IBM microcomputers, the pixel color data is an index into a look-up table (LUT) and the actual pixel color is determined by the setting of the DAC registers.

15.1.3 Image Data Compression

Bitmapped image data takes up considerable memory space. For example, the raw image data for a full screen, in an XGA or SuperVGA mode of 1,024 by 768 pixels resolution in 256 colors, requires approximately 768K. This is three-fourths of the total memory space available in an IBM microcomputer under MS-DOS. Consequently, several data compression schemes have been devised to reduce the memory space required for storing pixel-coded images. However, image data compression is achieved at a price: the additional processing time required for packing and unpacking the image data. In microcomputer graphics, performance is usually such a critical factor that this overhead is an important consideration in adopting a compressed data format.

Many of the compression methods used for alphanumeric data are not adaptable for image data. In the first place, all of the irreversible techniques used in character data compaction cannot be used for graphics images, since image data must be restored integrally. The same applies to many semantic-dependent techniques of various degrees of effectiveness. On the other hand, some general principles of data compression are applicable to graphics and packed bits encoding schemes can be used to compress pixel color data. For example, the IRGB encoding used in VGA 16-color graphics modes can be packed into two codes per byte, saving one half the storage space required for unpacked data.

Run-length Encoding

The principles of run-length encoding are particularly useful in compacting graphics data. The method is based on the suppression of repeated character codes, according to the principle that if a character is repeated three times or more, then the data can be more compactly represented in coded form. Run-length encoding is a simple and efficient graphics data compression scheme based on the assumption that image data often contains entire areas of repeated pixel values. Notice that approximately two-thirds of the bitmaps shown in Figure 15-2 consist of NULL pixels (white background color). Furthermore, even the images themselves contain substantial areas of black and of uniform shades of gray. In this case a simple compression scheme could be used to pack the data in the white, black, and gray areas so as to save considerable image storage space.

The Kermit protocol, well known in computer data transmission, uses a run-length encoding based on three data elements. The first code element indicates that a compression follows, the second character is the repetition code, and the third one represents the repetition count. The PackBits compression algorithm, which originated in the Macintosh computers, is an even more efficient run-length encoding scheme for graphics image data. The TIFF image file format discussed later in this chapter uses PackBits compression encoding.

Facsimile Compression Methods

Facsimile machines and methods (FAX) are often used in transmitting alphanumeric characters and graphics image data over telephone lines. Several compression protocols have been devised for facsimile transmission. The International Telegraph and Telephone Consultative Committee (CCITT), based in Geneva, Switzerland, has standardized several data compression protocols for use in facsimile equipment. The TIFF convention has adapted the CCITT standards to the storage of image data in computer systems. Notice that the actual compression algorithm used in CCITT is a variation of a method known developed by David A. Huffman in the 1950s. The CCITT method, which is quite efficient for monochrome scanned and dithered images, is elaborate and difficult to implement.

LZW Compression

LZW is a compression technique suited to color image data. The method is named after Abraham Lempel, Jabob Ziv, and Terry Welch. The algorithm, also known as Ziv-Lempel compression, was first published in 1977 in an article by Ziv and Lempel in the *IEEE Transactions on Information Theory*. The compression technique was refined by Welch in an article titled "A Technique for High-Performance Data Compression" that appeared in *Computer*, in 1984 (see bibliography). LZW compression is based on converting raw data into a reversible encoding in which the data repetitions are tokenized and stored in compressed form. LZW compression is used in many popular data and image compression programs, including the Compuserve GIF image data encoding format and in some versions of the TIFF standard. Notice that LZW compression has been patented by Unisys Corporation. Therefore its commercial use requires a license from the patent holders. The following statement is inserted at the request of Unisys Corporation:

> *The LZW data compression algorithm is said to be covered by U.S. Patent 4,558,302 (the "Welch Patent"). The Welch Patent is owned by Unisys Corportation. Unisys has a significant number of licensees of the patent and is comitted to licensing the Welch Patent on reasonable non-discriminatory terms and conditions. For further information, contact Unisys Welch Licensing Department, P.O. Box 500, Blue Bell, PA 19424, M/S C1SW19.*

LZW algorithm is explained later in this chapter.

15.1.4 Encoders and Decoders

An encoder is a program or routine used to convert raw image data into a standard format. We speak of a GIF encoder as a program or routine used to store a graphics image in a file structured in the GIF format. A decoder program or routine performs the reverse operation, that is, it reproduces the graphics image or the raw data from the information stored in an encoded image file. In the more conventional sense, a GIF decoder displays on the screen an image file stored in the Compuserve GIF format. Therefore the fundamental tool-kit for operating with a given image data format consists of encoder and decoder code. Notice that with some compressed image formats the processing required in encoders and decoders can be quite elaborate.

15.2 The Graphics Interchange Format (GIF)

The Graphics Interchange Format (GIF) originated in the Compuserve computer information service. The first description of the GIF protocol, which appeared on the Compuserve Picture Support Forum on May 28, 1987, was identified with the code letters GIF87a, while the current version is labeled GIF89a. GIF is the only graphics image storage format in use today that is not associated with any software company. Although the GIF standard is copyrighted, Compuserve grants royalty-free adoption rights to anyone wishing to use it. This means that, according to Compuserve, software developers are free to use the GIF encodings by accepting the terms of the Compuserve licensing agreement, which basically states that all changes to the standard must be made by the copyright holders and that the software utilizing GIF must acknowledge Compuserve's ownership. The agreement can be obtained form the Compuserve Graphics Technology Department or from the graphics forum files.

GIF was conceived as a compact and efficient storage and transmission format for computer imagery. The GIF87a specification supports multiple images with a maximum of 16,000 by 16,000 pixels resolutions in 256 colors. This format is quite suited to the maximum resolution available today in SuperVGA and XGA systems, although it seems that the 256-color modes will soon require expansion.

The advantages of the GIF standard are related to its being compact, powerful, portable, and, presumably, public, and also the fact that there is an extensive collection of public domain images in GIF format which can be found in the Compuserve graphics forums and in many bulletin board services. The programmer should keep in mind that images of recognizable individuals often require the person's release before the image can be legally used commercially. This is true even if the image file is publicly available.

The major disadvantage of the GIF standard is that many commercial programs do not support it. Consequently, users of popular graphics programs often discover that GIF is not included in the relatively extensive catalog of file formats which the application can import and export. This limitation can often be solved by means of a conversion utility that translates a format recognized by the particular application into a GIF encoding. Several of these format conversion utilities are available on the Compuserve graphics forums.

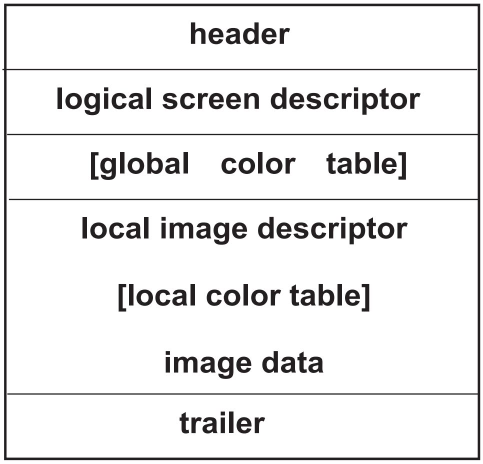
15.2.1 GIF Sources

The main sources of information about the GIF standard are the graphics forums on the Compuserve Information Service. The specifications of GIF89a are available in the file GIF89A.DOC found in library number 14 of the Compuserve Graphics Support forum. Image files in the GIF format are plentiful on the Compuserve Graphics Support libraries as well as in many BBS's. In this book's software package we have included several public domain image files in the GIF format. Also in the book's software package is a Shareware GIF file display program named Compushow.

15.2.2 The GIF File Structure

The two versions of the GIF standard at the time of this writing are labeled GIF87a and GIF89a. Version 89a is an extension of version 87a which adds several features to the original GIF protocol, namely: the display of text messages, comments, and application and graphics control data. The detailed description of the GIF protocol is found in the file GIF89A.DOC mentioned in the previous paragraph. The following description is limited to the features common to both the GIF87a and GIF89a specifications.

The GIF87a format is defined as a series of blocks and sub-blocks containing the data necessary for the storage and reproduction of a computer graphics image. A GIF data stream contains the data stored in these blocks and sub-blocks in the order defined by the GIF protocol. The first block in the data stream is the header and the last one is the trailer. Image data and other information is encoded between the header and trailer blocks. These can include a logical screen descriptor block and a global color table, as well as one or more local image descriptors, local color tables, and compressed image data. The GIF89a protocol allows graphics control and rendering blocks, plain text blocks, and an application data block. Figure 15-3 shows the elements of the GIF87a data stream.



Note: optional items are enclosed in braces

Figure 15-3 *Elements of the GIF Data Stream*

Header

The first item in the GIF data stream is the header. It consists of six ASCII characters. The first three characters, called the signature, are the letters "GIF." The following three characters encode the GIF version number. The value "87a" in this field refers to the version of the GIF protocol approved in May 1987, while the value "89a" refers to the GIF version dated July 1989. Figure 15-4 shows the elements of the GIF header.

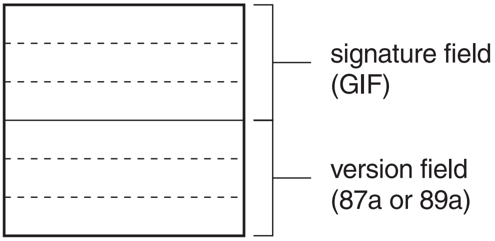


Figure 15-4 *GIF Header*

One header must be present in each GIF data stream. A GIF encoder must initialize all six characters in the GIF header. The version number field should correspond with the earliest GIF version that defines all the blocks in the actual data stream. In other words, a GIF file that uses only the elements of the GIF87a protocol should contain the characters 87a in the version field of the GIF header, even if the file was created after the implementation of the GIF89a protocol. The GIF decoder uses the information in the header block to certify that the file is encoded in the GIF format and to determine version compatibility.

Logical Screen Descriptor

The block immediately following the header is named the logical screen descriptor. This block contains the information about the display device or mode compatible with the image. One logical screen descriptor block must be present in each GIF data stream. Figure 15-5 shows the elements of the logical screen descriptor block.

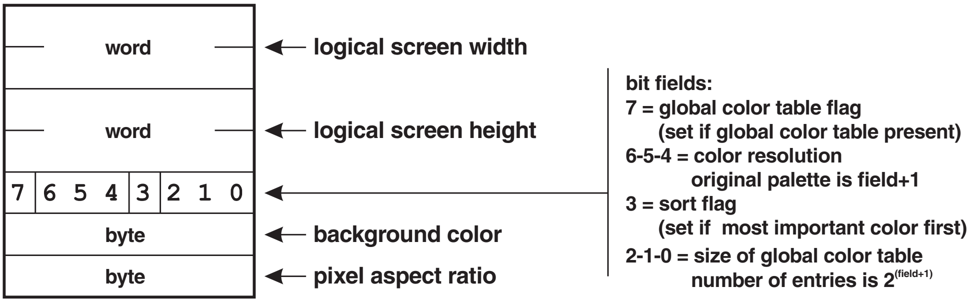

word
logical screen width
word
logical screen height
7 6 5 4 3 2 1 0
byte
background color
byte
pixel aspect ratio
bit fields:
7 = global color table flag
(set if global color table present)
6-5-4 = color resolution
original palette is field+1
3 = sort flag
(set if most important color first)
2-1-0 = size of global color table
number of entries is 2^(field+1)

Figure 15-5 *GIF Logical Screen Descriptor*

The fields of the GIF logical screen descriptor are formatted as follows:

1. The words at offset 0 and 2, labeled logical screen width and logical screen height, encode the pixel dimensions of the logical screen to be used by the display device. In IBM microcomputers this value usually coincides with the selected display mode.
2. The byte at offset 4 is divided into 4 bit fields. Bit 7, labeled the global color table flag, serves to indicate if a global color table is present in the data stream that follows. The global color table is discussed later in this section. Bits 6, 5, and 4 are the color resolution field. This value represents the number of palette bits for the selected mode, plus

one. For example, a 16-color VGA palette (4 bits encoding) would be represented by the bit value 011 (decimal 3). Bit 3, labeled the sort flag, is used to signal that the global color table (if present) is sorted starting with the most important colors. This information can be used by the software if the display device has fewer colors available than those used in the image. Finally, the field formed by bits 2, 1, and 0 determines the size of the global color table (if one is present). The value is encoded as a power of 2, diminished by 1. Therefore, to restore the original exponent it is necessary to add 1 to the value encoded in the bit field. For example, a bit value of 011 (3 decimal) corresponds to a global color table representing 24, or 16 colors. Notice that this value corresponds with the number of color in the global color table, not with its byte length (discussed later in this section). The maximum representable value in a 3-bit field is 7, which limits the number of colors in the global color table to 28, or 256 colors.

3. The field at offset 5, labeled background color in Figure 15-4, is used to represent the color of those pixels located outside of the defined image or images. The value is an offset into the global color table.

4. The field at offset 6, labeled the pixel aspect ratio in Figure 15-4, is used to compensate for non-proportional x and y dimensions of the display device (see Section 11.4.1). This field should be set to zero for systems with a symmetrical pixel density, such as the most used modes in VGA and XGA systems.

Global Color Table

The global color table is an optional GIF block used to encode a general color palette for displaying images in data streams without a local color table. The global color table serves as a default palette for the entire stream. Recall that the GIF data stream can contain multiple images. The presence of a global color table and its size is determined from the data furnished in the logical screen descriptor block (see Figure 15-4). Only one global color table can be present in the data stream. Figure 15-6 shows the structure of a global color table.

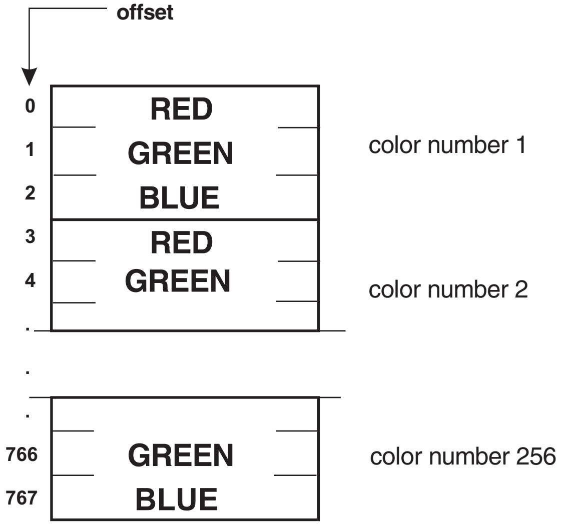


Figure 15-6 *GIF Global Color Table Block*

The entries in the global color table consist of values for the red, green, and blue palette registers. Each component color takes up 1 byte in the table, therefore each palette color consists of 3 bytes in the global color table. The number of entries in the global color table can be determined by reading bits 0, 1, and 2 of the global color size field in the logical screen descriptor block (see Figure 15-4). The byte length of the table is three times the number of entries. The maximum number of palette colors is 256. In this case the global color table takes up 768 bytes (see Figure 15-6).

Image Descriptor

Each image in the GIF data stream is defined by an image descriptor, an optional local color table, and one or more blocks of compressed image data. The image descriptor block contains the information for decoding and displaying the image. Figure 15-7 shows the elements of the image descriptor block.

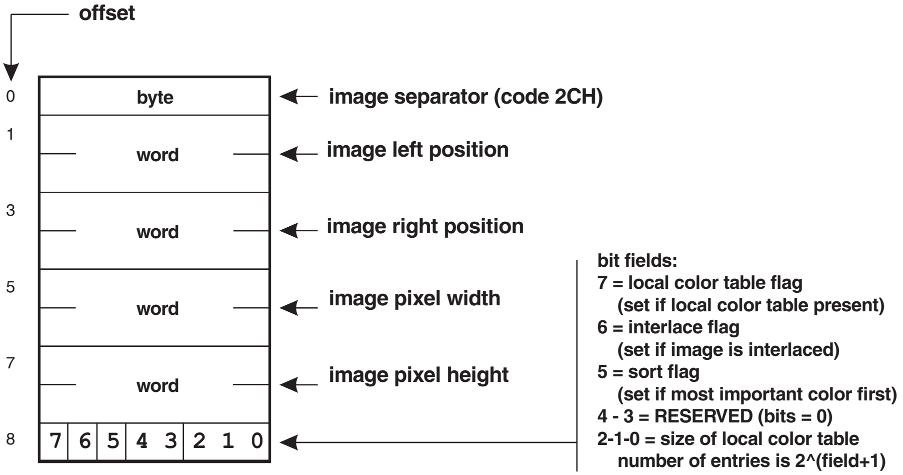

offset
0
1
3
5
7
8
byte
word
word
word
word
7 6 5 4 3 2 1 0
image separator (code 2CH)
image left position
image right position
image pixel width
image pixel height
bit fields:
7 = local color table flag
(set if local color table present)
6 = interlace flag
(set if image is interlaced)
5 = sort flag
(set if most important color first)
4 - 3 = RESERVED (bits = 0)
2-1-0 = size of local color table
number of entries is 2^(field+1)

Figure 15-7 *GIF Image Descriptor*

The fields of the GIF image descriptor are formatted as follows:

1. The byte at offset 0, labeled image separator in Figure 15-7, must be the code 2CH.
2. The words at offset 1 and 3, labeled image left position and image right position, respectively (see Figure 15-7), encode the screen column and row coordinates of the image's top left corner. This location is an offset within the logical screen defined in the logical screen descriptor block (see Figure 15-4).
3. The words at offset 5 and 7, labeled image pixel width and image pixel height, respectively (see Figure 15-7), encode the size of the image, measured in screen pixels.
4. The byte at offset 8 in Figure 15-7 is divided into 5 bit fields. Bit 7, labeled the local color table flag, serves to indicate if a local color table follows the image descriptor block. If a local color table is present in the data stream it is used for displaying the image represented in the corresponding descriptor block. Bit 6, labeled interlace flag, encodes if the image is interlaced, that is, if its rows are not arranged in consecutive order. In IBM microcomputers interlaced images are used in some CGA and EGA display modes, but

not in the proprietary VGA and XGA modes. Bit 5, labeled the sort flag, is used to signal that the local color table (if present) is sorted starting with the most important colors. This information can be used by the software if the display device has fewer available colors than those in the table. The field formed by bits 2, 1, and 0 determines the size of the local color table (if one is present). The value is encoded as a power of 2, diminished by 1. Therefore, to restore the original exponent it is necessary to add 1 to the value encoded in the bit field. For example, a bit value of 011 (3 decimal) corresponds to a global color table representing 24, or 16 colors. Notice that this value corresponds to the number of colors in the local color table, not with its byte length (refer to the previous discussion about the global color table).

Local Color Table

The local color table is an optional GIF block that encodes the color palette used in displaying the image corresponding to the preceding image descriptor block. If no local color table is furnished, the image is displayed using the values in the global color table. If neither table is present, it shall be displayed using the current setting of the DAC registers. The GIF data stream can contain multiple images, with each one having its own local color table. The structure of the local color table is identical to the one described for the global color table (see Figure 15-6).

Compressed Image Data

The image itself follows the local color table, if one is furnished, or the image descriptor block if the data stream does not include a local color table. The GIF standard sets no limit to the number of images contained in the data stream. Image data is divided into sub-blocks; each sub-block can have at the most 255 bytes. The data values in the image are offsets into the current color palette. For example, if the palette is set to standard IRGB code, a pixel value of 1100B (decimal 12) corresponds to the 12th palette entry, which, in this case, encodes the LUT register settings for bright red.

Preceding the image data blocks is a byte value that holds the code size used for the LZW compression of the image data in the stream. This data item normally matches the number of bits used to encode the pixel color. For example, an image intended for VGA mode number 18, in 16 colors, has an LZW code size of 4, while an image for VGA mode number 19, in 256 colors, has an LZW code size of 8. Figure 15-8 shows the format of the GIF data blocks.

The image data sub-blocks contain the image data in compressed form. The LZW compression algorithm used in the GIF protocol is discussed in Section 15-3.2. Each data sub-block starts with a block-size byte, which encodes the byte length of the data stored in the rest of the sub-block. The count, which does not include the count byte itself, can be in the range 0 to 255. The compressed data stream ends with a sub-block with a zero byte count (see Figure 15-8).

Trailer

The simplest GIF block is named the trailer. This block consists of a single byte containing the GIF special code 3BH. Every GIF data stream must end with the trailer block. The GIF trailer is shown in Figure 15-9.

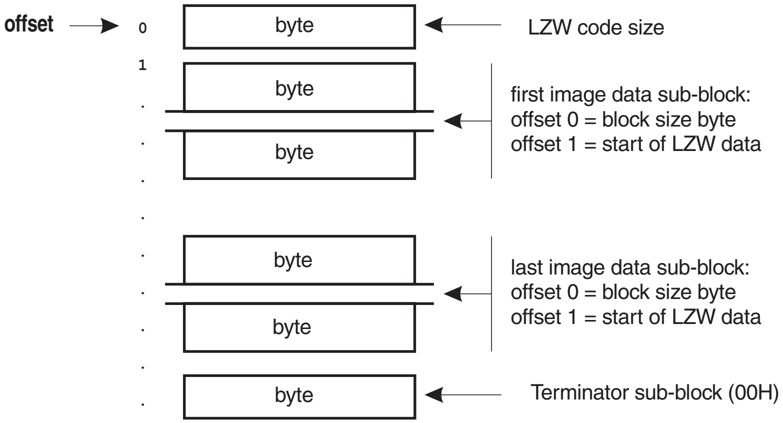

offset
0
1
byte
byte
byte
byte
byte
byte
LZW code size
first image data sub-block:
offset 0 = block size byte
offset 1 = start of LZW data
last image data sub-block:
offset 0 = block size byte
offset 1 = start of LZW data
Terminator sub-block (00H)

Figure 15-8 *GIF Image Data Blocks*

GIF89a Extensions

We mentioned that GIF version 89a contains several features that are not present in version 87a. These features include the following new blocks:

1. A graphics control extension refers to a graphics rendering block, also a new feature introduced in version 89a. The graphics control extension contains information on displaying the rendering block. This information includes instructions about the disposing of the currently displayed image, handling the background color, action on user input, time delay during the display operation, and image transparency.
2. The graphics rendering blocks can be an image descriptor block, as described for GIF version 87a, or a new plain text extension. The plain text extension contains ASCII data to be displayed in a coarse grid of character cells determined in the block. Also in the plain text block are the foreground and background colors, the coordinates of the start position, and the text message itself.
3. The applications extension is an extension block in GIF version 89a that contains application-specific information. The block includes an 8-byte application identifier field intended for an ASCII string that identifies the particular piece of software. A 3-byte authentication code follows the identifier. Application data follows the authentication code field.

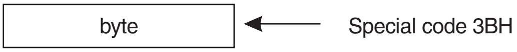


Figure 15-9 *GIF Trailer*

15.2.3 GIF Implementation of LZW Compression

One operation in creating a GIF image data file is the formatting of the various blocks according to the specifications described in the standard (see 15.2.1). This operation is quite simple and presents no programming complications. However, the image data in a GIF file must be stored in compressed form; the GIF standard offers no alternative. The compression algorithm adopted by GIF is the method originally devised by Lempel and Ziv and later improved by Welch (see Section 15.3.3). The implementation of this compression algorithm, often designated LZW (Lempel-Ziv-Welch) compression, is the most difficult programming operation in developing a GIF encoder or decoder program or routine.

LZW Concepts

The original concept of LZW compression is based on the assumption that the data to be compressed presents patterns of repetition. These repetitions can be in the form of the vowel-consonant patterns of all modern languages, in the words of a text file, or in the pixel repetition pattern of a graphics image. For this reason LZW compression has been successfully used in compressing both text and image data. Many well-known compression programs found in Web sites, such as PAK, PKARK, PKZIP, and PKUNZIP, use LZW compression. In the graphics field LZW compression is used in GIF, TIFF, and other image file storage formats.

The programmer must consider that LZW is an algorithm, not a standard. This means that each particular implementor of a data compression scheme based on LZW feels free to adapt the algorithm to meet specific needs. In this manner LZW compression as used in the GIF standard is different from LZW compression as used in TIFF or in other data storage conventions, in spite of the fact that the actual compression methods are quite similar in all LZW implementations. Once understood, LZW compression can be easily applied to match the requirements of any specific application.

The central idea of LZW compression is to replace repeated characters with individual symbols. In text compression this translates to encoding strings with single codes. In graphics compression the method consists of detecting repeated pixel patterns and representing them with a single value. LZW does not search the data for repetitions, but stores them as they are encountered in the data stream. The adverse consequences of this form of operation is that some significant patterns of repetition can be missed during the encoding, and that repeated patterns are often encoded more than once. The advantage of this "compress as you find them" technique is that the decoder can reconstruct the repetitions from the information in the data stream, making it unnecessary to transmit tables of patterns or other general decoding information.

The General LZW Algorithm

The LZW compression algorithm requires a basic-table of codes representing each item in the data stream. For example, an alphanumeric implementation of LZW can be based on the IBM extended character set, which consists of 256 character codes (see Table 1-2). In this case the basic-table contains 256 entries, one for each possible data code in the stream. On the other hand, an LZW implementation for pixel data in the IRGB format would require only a basic-table with 16 entries, one for each possible IRGB combination in the data stream.

The LZW compression codes start after the basic table. In the GIF implementation two special codes (discussed later in this section) are added at the end of the basic-table. However, in the present discussion we assume that the compression codes start immediately after the basic-table. For example, if the LZW implementation is based on 256 alphanumeric character codes, in the range 0 to 255, the first available compression code would be the value 256. The highest compression code in LZW is preset to the value 4095. Therefore, in this example, the compression codes would be values in the range 256 to 4095. In LZW compression, the part of the table that stores the repeated patterns is often called the string-table.

The compression algorithm assumes that information is received in a continuous data stream and that the software has some means of detecting the end of this data stream. In our first example of LZW compression we assume, for the sake of simplicity, that the data stream consists of character bytes in the range 0 to 255. Therefore the basic-table can contain codes in this range, and the string-table starts at the value 256. Let us assume that the data stream consists of a series of monetary values separated by the slash symbol, as follows:

```
/$15.00/$22.00/$12.10/$222.00<EOI>
```

In the above data sample the expression <EOI> indicates the presence of an "end of information" code in the data stream. The compression algorithm requires a scratchpad data structure which is sometimes called the current string. In the following description we arbitrarily designate the current string with the @ symbol. Compression takes place in the following steps:

STEP 1: Initialize the basic-table with all the code combinations that can be present in the data stream. The string-table codes start after the last code in the basic-table.

STEP 2: Initialize the current string (scratchpad element) to a NULL string. Designate the current string as @.

STEP 3: Read character from the data stream. Designate the current character as C. If C = <EOI> then end execution.

STEP 4: Concatenate the current string (@) and the character (C) to form @+C.

STEP 5: If @+C is in the basic-table or in the string-table perform the following operations:

a. @ = @+C

b. Go to STEP 3

STEP 6: If @+C is not in the basic-table or in the string-table perform the following operations:

a @+C in the string-table

b. send @ to the output stream

c. @ = C

d. go to STEP 3

The above description assumes that the data stream does not overflow the total number of allowed entries in the string-table. Later in this section we will present a working sample of GIF LZW compression that takes this possibility into account. Table 15-1 shows the LZW compression of the string listed above.

Table 15-1

LZW Compression Example

ITERATION	INPUT	STRING TABLE	OUTPUT	CURRENT STRING (@)		
NUMBER	STREAM	ENTRY	STREAM	INITIAL	@C	FINAL
1	'/'	NONE	--	NULL	'/'	'/'
2	'$'	256 = '/$’	'/'	'/'	'/$'	'$'
3	'1'	257 = '$1'	'$'	'$'	'$1'	'1'
4	'0'	258 = '10'	'1'	'1'	'10'	'0'
5	'.'	259 = '0.'	'0'	'0'	'0.'	'.'
6	'0'	260 = '.0'	'.'	'.'	'.0'	'0'
7	'0'	261 = '00'	'0'	'0'	'00'	'0'
8	'/'	262 = '0/'	'0'	'0'	'0/'	'/'
9	'$'	NONE	--	'/'	'/$'	'/$'
10	'2'	263 = '/$2'	<256>	'/$'	'/$2'	'2'
11	'2'	264 = '22'	'2'	'2'	'22'	'2'
12	'.'	265 = '2.'	'2'	'2'	'2.'	'.'
13	'0'	NONE	--	'.'	'.0'	'.0'
14	'0'	266 = '.00'	<260>	'.0'	'.00'	'0'
15	'/'	NONE	--	'0'	'0/'	'0/'
16	‘$'	267 = '0/$'	<262>	'0/'	'0/$'	'$'
17	'1'	NONE	--	'$'	'$1'	'$1'
18	'2'	268 = '$12'	<257>	'$1'	'$12'	'2'
19	'.'	NONE	--	'2'	'2.'	'2.'
20	'1'	269 = '2.1'	<265>	'2.'	'2.1'	'1'
21	'0'	NONE	--	'1'	'10'	'10'
22	'/'	270 = '10/'	<258>	'10'	'10/'	'/'
23	'$'	NONE	--	'/'	'/$'	'/$
24	'2'	NONE	--	'/$'	'/$2'	'/$2'
25	'2'	271 = '/$22'	<263>	'/$2'	'/$22'	'2'
26	'2'	NONE	--	'2'	'22'	'22'
27	'.'	272 = '22.'	<264>	'22'	'22.'	'.'
28	'0'	NONE	--	'.'	'.0'	'.0'
29	'0'	NONE	--	'.0'	'.00'	'.00'
30	<EOI>	NONE	<266>			

String: /$10.00/$22.00/$12.10/$222.00<EOI>

In the compression of the string in Table 15-1 notice the following interesting points:

1. On iteration number 1 the current string is initialized to a NULL string. Since the input character '/' is in the basic-table, algorithm STEP 5 executes. Therefore @ = '/' at the conclusion of this iteration.

2. On iteration number 2 the current string (@) contains the initial value of '/' (previous character input). @+C becomes '/$', which is not in the basic-table or the string-table (the string-table is empty at this time). Therefore algorithm STEP 6 executes and '/$' is the first entry in the string-table, which is numbered 256.

3. On iteration number 3 the current string (@+C) contains '$1' which is not in the string-table. Therefore STEP 6 executes again. In this case the '$1' is entry number 257 in the string-table.

4. The iterations during which there is no entry in the string-table (labeled NONE in Table 15-1) are those in which algorithm STEP 5 executes. Notice that no output takes place in this case.
5. Every iteration that produces an entry in the string-table also generates output to the character stream (algorithm STEP 6). The output is the contents of the current string (@), which can be a single character or a string. The string corresponds to an entry in the string-table and is represented by its number.
6. Compression concludes when the "end of information" code is detected in the input stream. This situation takes place in iteration number 30 of Table 15-1.

Notice several important features of the LZW compression algorithm:

1. The compression codes are of variable length.
2. The decoder program is able to reproduce the string-table from the input data. This table is identical to the one used by the encoder.
3. The use of variable-length codes results in greater compression efficiency than if the information were conveyed on fixed-size data packets.
4. The self-reproducing string-table saves having to transmit conversion or character tables to the decoder.

The GIF Implementation

The implementation of LZW compression in the GIF protocol closely matches the original algorithm as described by Lempel, Ziv, and Welch. Two variations are introduced in the GIF implementation: a special code that serves to signal to the decoder that the string-table must be cleared, and another one to signal the end of the compressed data. The code to clear the string-table is often represented with the letters <CC> and the code to end the compressed data stream is identified as <EOI> (end of information).

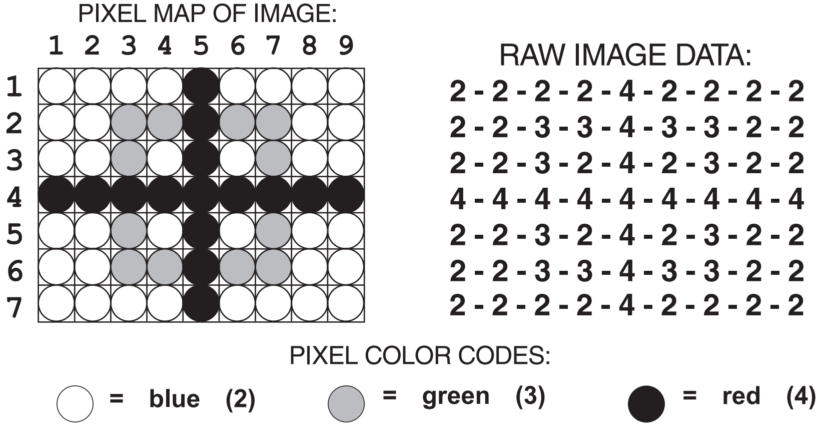
These two special codes, <CC> and <EOI>, are added to the basic-table. Since the GIF implementation is applied to graphics data, the basic-table for GIF LZW compression consists of all the pixel codes used in the image, plus the "clear string-table" code <CC> and the "end of information" code <EOI>. For example, in encoding a video image for VGA mode number 18, with 16 possible colors, the basic-table would have the codes 0 to 15. In this case the clear code <CC> would be assigned code number 16, and the <EOI> code would be assigned number 17. Therefore the first entry in the string-table would correspond to code number 18. Since the LZW string-table can extend to code number 4,095, the range in this case would be from 18 to 4,095.

LZW Code Size

We saw (Figure 15-8) that in the GIF encoding the compressed data in the first image data sub-block must be preceded with a byte that encodes the LZW code size. This value coincides with the bit-size of the elements in the basic-table. In the example mentioned above, in which the image is encoded for VGA mode number 18, in 16 colors, the LZW code size is 4. By the same token, the LZW code size would be 8 for an image encoded in 256 colors.

The GIF Image File

Perhaps the easiest way to understand the GIF encoding and its implementation of LZW compression is by an example. Figure 15-10 shows the pixel map of a simple graphics image in three colors.



2 - 2 - 2 - 2 - 4 - 2 - 2 - 2 - 2
2 - 2 - 3 - 3 - 4 - 3 - 3 - 2 - 2
2 - 2 - 3 - 2 - 4 - 2 - 3 - 2 - 2
4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4
2 - 2 - 3 - 2 - 4 - 2 - 3 - 2 - 2
2 - 2 - 3 - 3 - 4 - 3 - 3 - 2 - 2
2 - 2 - 2 - 2 - 4 - 2 - 2 - 2 - 2

Figure 15-10 *Sample Image for GIF LZW Compression*

The following code fragment shows the data structures necessary for encoding the image in Figure 15-10 in GIF format. In order to create a disk image of the GIF file we must first assemble the source and then strip off the object-file header appended by the assembler program. This can be easily done by means of the "write" command of a debugger program (such as Microsoft Debug or Symdeb) or of a disk file editing utility.

```
DATA    SEGMENT
;
;**********************|
;  GIF file header     |
;**********************|
; The 6-byte header block includes the GIF signature and version
; fields (see Figure 15-4)
       DB      'GIF87a'
;**********************|
;    logical screen    |
;      descriptor      |
;**********************|
; The logical screen descriptor block contains the information
; listed in Figure 15-4. In this example we have adopted a VGA
; resolution of 640 by 480 pixels in 8 colors
        DW      640             ; Logical screen width
        DW      480             ; Logical screen height
        DB      10100010B       ; Global flag
                        ; Global flag bitmap:
                        ; 0 0 1 1 0 0 0 0
                        ; 7 6 5 4 3 2 1 0 <= bits
                        ; | | | | | |_|_|_ size of global color
                        ; | | | | |        table (2 ^(field+1))
                        ; | | | | |___ sort flag
                        ; | | | |      1 = most important color
                        ; | | | |          first
                        ; | |_|_|__ color resolution original
                        ; |         palette is (field + 1)
                        ; |________ global color table
                        ;           1 = table present
                        ;           0 = no global table
        DB      0               ; Background color index
                                ; (meaningless in this case)
        DB      0               ; Pixel aspect ration
                                ; (symmetrical in VGA systems)
;**********************|
;  global color table  |
;**********************|
; The code furnishes an 8-entry global color table. Each entry
; consists of 3 bytes encoding the red, green, and blue values.
; Notice that only colors number 2, 3, and 4 are required by the
; image (see Figure 15-6)
;                R    G    B      Color       color number
        DB      000H,000H,000H  ; Black         0
        DB      0BBH,0BBH,0BBH  ; White         1
        DB      000H,000H,0AAH  ; Blue          2
        DB      000H,0AAH,000H  ; Green         3
        DB      0AAH,000H,000H  ; Red           4
        DB      080H,080H,0AAH  ; Light blue    5
        DB      080H,0AAH,080H  ; Light green   6
        DB      0AAH,080H,080H  ; Light red     7
;**********************|
;  image descriptor    |
;**********************|
; This block contains the information listed in Figure 15-7
        DB      2CH             ; GIF image separator code
        DW      10              ; x coordinate for image
        DW      10              ; y coordinate
        DW      9               ; Image width (in pixels)
        DW      7               ; Image height (in pixels)
        DB      00000000B       ; Local flag
                        ; Local flag bitmap:
                        ; 1 0 0 0 0 0 1 0
                        ; 7 6 5 4 3 2 1 0 <= bits
                        ; | | | | | |_|_|_ size of local color
                        ; | | | | |        table
                        ; | | | | |        value is 2 ^(field+1)
                        ; | | | |_|_______ RESERVED
                        ; | | |_______ sort flag
                        ; | |          1 = most important color
                        ; | |              first
                        ; | |______ interlace flag
                        ; |         1 = image is interlaced
                        ; |         0 = image is not interlaced
                        ; |________ local color table flag
                        ;           1 = table present
                        ;           0 = no local table
;**********************|
;     image data       |
;  (LZW compression)   |
;**********************|
; Follows image data compressed according to the GIF
; implementation of the LZW algorithm (see Figure 15-8)
        DB      3               ; LZW code size
        DB      20              ; Image size (in bytes)
        DB      028H,02AH,0B4H,03BH,083H,040H,037H,098H,0A8H,08CH
        DB      0E8H,0ADH,055H,06DH,098H,017H,04DH,08EH,0B5H,024H
; Block terminator
        DB      0
;**********************|
;      trailer         |
;**********************|
; The trailer is a single-byte block that marks the end of a GIF
; data stream. The required terminator code is 3BH (Figure 15-9)
        DB      3BH             ; GIF terminator
;
DATA    ENDS
        END
```

Although only three colors are necessary for the image in Figure 15-10, we have added white and black to the palette. The monochrome colors are often added so as to allow displaying a color image in a black-and-white system. In Section 15.2.1 we mentioned that the number of colors in the GIF global and local color tables must coincide with powers of 2, therefore, 2, 4, 8, 16, 32, 64, 128, and 256 entries can be chosen for the palette. This example requires 5 colors, hence an 8-color palette is selected. Palette entry number 0 corresponds to the color black and entry number 1 to the color white. The colors corresponding to palette entries 2, 3, and 4 are shown in Figure 15-15. In actual programming we can either zero the remaining palette entries (5, 6, and 7) or set them to any given color value. However, the memory space must be reserved for the total number of palette entries. In the previous code sample palette entries number 5, 6, and 7 have been initialized to light blue, light green, and light red respectively.

The image descriptor block in the previous code sample contains the x and y coordinates for image display. Notice that we have placed the image at 10 pixels from the screen's top left corner, and also that the image dimensions are 9 horizontal pixels by 7 pixel rows, as in Figure 15-10.

GIF LZW Encoding

In the previous code fragment we saw that the image data consists of the LZW code size byte, a block count byte, 20 image code bytes, and the block terminator code 00H. The process of obtaining the compressed codes is shown in Table 15-2.

Notice, in Table 15-2, that the raw data from the image in Figure 15-10 is used as an input stream for GIF LZW compression, and that the first code output to the stream is the clear string-table command <CC> which is assigned the value 8. Notice also that the output stream ends in the end-of-information code <EOI>, which is number 9 in this case. The rest of the output stream is generated following the LZW algorithm as described in the general example in Table 15-1.

The asterisks in Table 15-2 mark the first characters of each image row (see Figure 15-10). Also notice that the string-table entries in the output stream are enclosed with angle brackets to differentiate them from the basic-table entries. Table 15-3 shows the processing operations required to obtain the compressed data encoding from the output stream in Table 15-2.

Table 15-2

GIF LZW Compression Example

ITERATION	INPUT	STRING TABLE	OUTPUT	CURRENT STRING (@)		
NUMBER	STREAM	ENTRY	STREAM	INITIAL	@C	FINAL
1	---	NONE	8			
2	2 *	NONE	--	NULL	2	2
3	2	10 = 22	2	2	22	2
4	2	NONE	--	2	22	22
5	2	11 = 222	<10>	22	222	2
6	4	12 = 24	2	2	24	4
7	2	13 = 42	4	4	42	2
8	2	NONE	--	2	22	22
9	2	NONE	--	22	222	222
10	2	14 = 2222	<11>	222	2222	2
11	2 *	NONE	--	2	22	22
12	2	NONE	--	22	222	222
13	3	15 = 2223	<11>	222	2223	3
14	3	16 = 33	3	3	33	3
15	4	17 = 34	3	3	34	4
16	3	18 = 43	4	4	43	3
17	3	NONE	--	3	33	33
18	2	19 = 332	<16>	33	332	2
19	2	NONE	--	2	22	22
20	2 *	NONE	--	22	222	222
21	2	NONE	--	222	2222	2222
22	3	20 = 22223	<14>	2222	22223	3
23	2	21 = 32	3	3	32	2
24	4	NONE	--	2	24	24
25	2	22 = 242	<12>	24	242	2
26	3	23 = 23	2	2	23	3
27	2	NONE	--	3	32	32
28	2	24 = 322	<21>	32	322	2
29	4 *	NONE	--	2	24	24
30	4	25 = 244	<12>	24	244	4
31	4	26 = 44	4	4	44	4
32	4	NONE	--	4	44	44
33	4	27 = 444	<26>	44	444	4
34	4	NONE	--	4	44	44
35	4	NONE	--	44	444	444
36	4	28 = 4444	<27>	444	4444	4
37	4	NONE	--	4	44	44
38	2 *	29 = 442	<26>	44	442	2
39	2	NONE	--	2	22	22
40	3	30 = 223	<10>	22	223	3
41	2	NONE	--	3	32	32
42	4	31 = 324	<21>	32	324	4
43	2	NONE	--	4	42	42
44	3	32 = 423	<13>	42	423	3
44	2	NONE	--	3	32	32
45	2	NONE	--	32	322	322
46	2 *	33 = 3222	<24>	322	3222	2
47	2	NONE	--	2	22	22
48	3	NONE	--	22	223	223
49	3	34 = 2233	<30>	223	2233	3

(continues)

Table 15-2

GIF LZW Compression Example (continued)

ITERATION	INPUT	STRING TABLE	OUTPUT	CURRENT STRING (@)		
NUMBER	STREAM	ENTRY	STREAM	INITIAL	@C	FINAL
50	4	NONE	--	3	34	34
51	3	35 = 343	<17>	34	343	3
52	3	NONE	--	3	33	33
53	2	NONE	--	33	332	332
54	2	36 = 3322	<19>	332	3322	2
55	2 *	NONE	--	2	22	22
56	2	NONE	--	22	222	222
57	2	NONE	--	222	2222	2222
58	2	37 = 22222	<14>	222	22222	2
59	4	NONE	--	2	24	24
60	2	NONE	--	24	242	242
61	2	38 = 2422	<22>	242	2422	2
62	2	NONE	--	2	22	22
63	2	NONE	--	22	222	222
64	<EOI>	NONE	--	<11>		
65			9			

Basic table: 0 7 = colors 8 = <CC> 9 = <EOI> String table: 10 4095

Table 15-3

GIF LZW Compression Data Processing

TABLE ENTRY	OUTPUT DECIMAL	(FROM TABLE 15-2) BINARY	BLOCKED BINARY OUTPUT	HEXADECIMAL VALUE
	8	1000	00101000	28
10	2	0010		
11	10	1010	00101010	2A
12	2	0010		
13	4	0100	10110100	B4
14	11	1011		
15	11	1011	00111011	3B
16 <==	3	0011		
17	3	00011	10000011	83
18	4	00100	01000000	40
19	16	10000		
20	14	01110	00110111	37
21	3	00011	10011000	98
22	12	01100		
23	2	00010	10101000	A8
24	21	10101		
25	12	01100	10001100	8C
26	4	00100	11101000	E8
27	26	11010		
28	27	11011	10101101	AD
29	26	11010	01010101	55
30	10	01010		
31	21	10101	01101101	6D
32 <==	13	01101		

(continues)

Table 15-3

GIF LZW Compression Data Processing ***(continued)***

TABLE ENTRY	OUTPUT (FROM TABLE 15-2) DECIMAL	 BINARY	BLOCKED BINARY OUTPUT	HEXADECIMAL VALUE
33	24	011000	10011000	98
34	30	011110	00010111	17
35	17	010001	01001101	4D
36	19	010011		
37	14	001110	10001110	8E
38	22	010110	10110101	B5
	11	001011	00100100	24
	9	001001		

We mentioned that an important characteristic of the LZW compression algorithm is the variable-length of the encoded data. In Table 15-3 we can see that the binary column of compression codes starts at 4 bits width, then changes to 5 bits, and later to 6 bits wide. Notice that the variable width of the output codes results from the increasing values of the string-table entry numbers, since the entries from the basic-table are always limited to the initial range. In the example in Table 15-2 the first string-table entry is number 10, which is representable in 4 bits, but the last entry is number 38, which requires 6 bits.

The arrows in Table 15-3 signal the string-table entry numbers 16 and 32. The value 16 is the first one requiring a 5-bit encoding and the value 32 is the first requiring a 6-bit encoding. Therefore, as soon as table entry number 16 is generated, the representation of the output codes is increased by 1-bit. Another 1-bit increase takes place immediately after table entry number 32. The width increases take place automatically after the table entry is created (not as wider codes are required in the output stream) because the decoding software must be able to predict the code-length changes. Figure 15-11, on the following page, is a flowchart of LZW compression as implemented in the GIF standard.

GIF encoder software must block the variable-length binary output codes that result from the compression process into groups of 8 bits so that they can be stored in byte-size memory cells or transmitted through the communications lines. The blocking operation consists of packing these bits right-to-left as shown in Table 15-3. Observe that the last column of this table, labeled "hexadecimal value", coincides with the image data listed in the GIF image code fragment.

GIF LZW Decoding

GIF decoding software obtains system and image information from the standard data blocks in the file. The first operation performed by the decoder is to make certain the GIF signature is present at the start of the file and the processing software is compatible with the version field of this block. The GIF standard recommends that if the decoder encounters a version with which it is not familiar, the software should post a warning message and process the file as best it can.

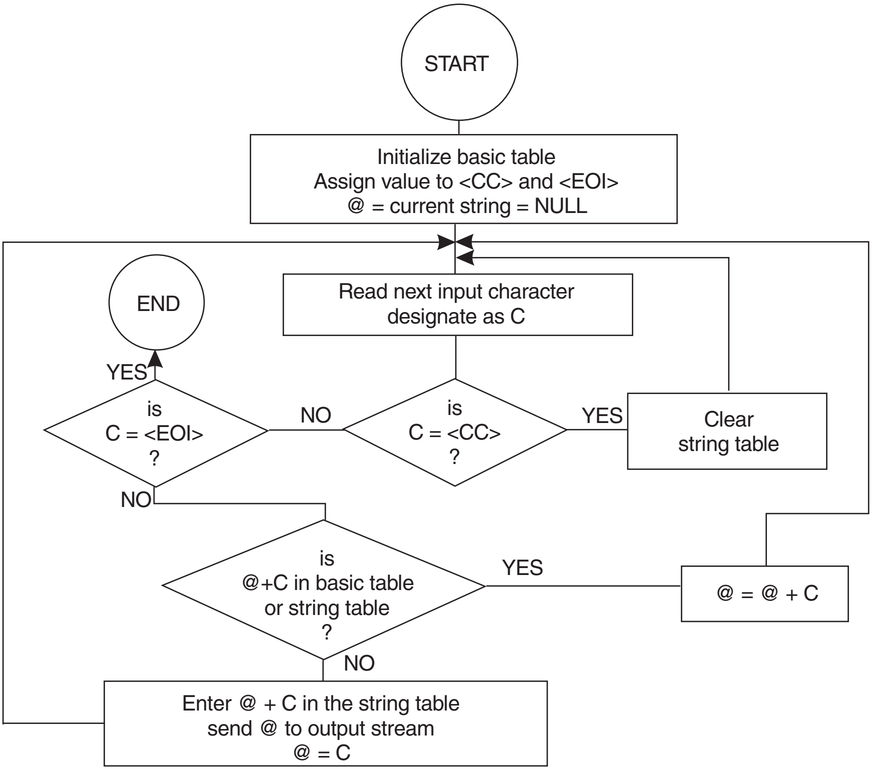


Figure 15-11 *GIF LCW Compression Flowchart*

As is the case with the encoder, the most elaborate operation to be performed by the decoder software is regarding the LZW compressed data. GIF LZW decompression follows the reverse process as the compression previously described. To the decoder the compression codes form the input stream. The initial bit width is calculated by adding 1 to the value in the LZW code-size field of the image block. Table 15-4 is a LZW decompression example that uses as input the compressed string generated in Table 15-1.

The decompression algorithm, as described below uses a variable to temporarily store the previous input value. This variable is placed in the column labeled OLD CODE in Table 15-4 and designated with the % symbol. The decompression process can be described as follows:

STEP 1: Initialize the basic-table with all the code combinations that can be present in the data stream. The string-table codes start after the last code in the basic-table.

STEP 2: Create a variable named OLD CODE (%) to hold the previous input. Initialize % to NULL. Designate the first character of the current input value as C.

STEP 3: Read first character from the data stream. If C = <EOI> then end execution. If C = <CC> then re-initialize string-table. If not, then output the first character.

STEP 4: If C is a character in the basic-table perform the following operations:

a. output C

b. % = C

c. create a new string-table entry with the value % + C

d. go to STEP 4

STEP 5: If C is a compression code perform the following operations:

a. look up compression code in string-table and output value

b. % = C

c. C = first character in compression string

d. create a new string-table entry with the value % + C

e. go to STEP 4

Table 15-4

LZW Decompression Example

ITERATION	INPUT STREAM	OLD CODE (%)	CHARACTER (C)	OUTPUT STREAM	STRING TABLE (% C)
1	'/'	--	'/'	'/'	
2	'$'	'/'	'$'	'$'	256 = '/$'
3	'1'	'$'	'1'	'1'	257 = '$1'
4	'0'	'1'	'0'	'0'	258 = '10'
5	'.'	'0'	'.'	'.'	259 = '0.'
6	'0'	'.'	'0'	'0'	260 = '.0'
7	'0'	'0'	'0'	'0'	261 = '00'
8	<256>	'0'	'/'	'/$'	262 = '0/'
9	'2'	<256>	'2'	'2'	263 = '/$2'
10	'2'	'2'	'2'	'2'	264 = '22'
11	<260>	'2'	'.'	'.0'	265 = '2.'
12	<262>	<260>	'0'	'0/'	266 = '.00'
13	<257>	<262>	'$'	'$1'	267 = '0/$'
14	<265>	<257>	'2'	'2.'	268 = '$12'
15	<258>	<265>	'1'	'10'	269 = '2.1'
16	<263>	<258>	'/'	'/$2'	270 = '10/'
17	<264>	<263>	'2'	'22'	271 = '/$22'
18	<266>	<264>	'.'	'.00'	272 = '22.'

Basic table: ASCCI codes in range 0 to 255

Notice that in performing the read-operation the software must keep track of bit boundaries in the input data. Also that the algorithm assumes that the first element in the input stream is a character and handles this case independently (STEP 3).

There are less iterations in LZW decompression than in compression. For example, there are 30 iterations in the compression process shown in Table 15-1 while there are only 18 in the example in Table 15-4. Notice that in the decompression process a string-table entry results in each iteration after the first one. This is a consequence of the mechanics of the compression process, in which an output is generated only when an entry is made in the string-table (see Table 15-1). Also notice that the string-table that results from the decompression (Table 15-4) is identical to the one generated during compression (Table 15-1).

15.3 The Tag Image File Format (TIFF)

The tag image file format (TIFF) was developed by ALDUS Corporation with the support of several other companies, including Hewlett-Packard and Microsoft. The standard is an effort at providing a flexible file-storage format for raster images. Its origin is related to scanner hardware and software for microcomputers. The first version of TIFF was published in the fall of 1986. The present update, designated as TIFF Revision 6.0, was released in June 1992. TIFF is a non-proprietary standard which can be used without license or previous royalty agreement. Technical information about TIFF can be obtained from the Aldus Developer's Desk at Aldus Corporation, Seattle, Washington, or from the Aldus forum on Compuserve (GO ALDSVC).

The purpose of the TIFF standard is to provide an image storage convention with maximum flexibility and portability. TIFF is not intended for any particular computer, operating system, or application program. Consistent with this idea, the files in TIFF format have no version number or other update identification code. A typical TIFF reader searches for the data necessary to its own purposes and ignores all other information contained in the file. The format supports both the Intel and the Motorola data ordering schemes but hardware-specific features are not documented in the TIFF file. Which mode, resolution, or color range used in displaying a TIFF file is left entirely to the software.

The TIFF standard supports monochrome, grayscale, and color images of various specifications. The original TIFF documents classified the various image types into four classes. Class B was used for binary (black-and-white) images, class G for grayscale images, class P for palette color images (8-bits per pixel color), and class R for full-color images (24-bits per pixel color). A TIFF application need not provide support for all TIFF image types. For example, a VGA TIFF reader could exclude class R images since the system's maximum color range is 8 bits per pixel (256 colors). By the same token, a routine or application that reads monochrome scanned images could limit its support to the class B category. The image class designations by letter codes was dropped in TIFF revision 6.0; however, the image classification into bilevel, grayscale, RGB, and palette types was preserved.

TIFF originally supported uncompressed images as well as compressed data according to several compression schemes, namely, PackBits, CCITT, and LZW (see Section 15-3.3). LZW compression support was dropped in TIFF version 6.0; because the compression algorithm is patented by Unysis Corporation (see Section 15.3.3). Notice that, in the TIFF standard, compression methods are usually associated with the particular file classes mentioned in the preceding paragraph.

15.3.1 The TIFF File Structure

The TIFF standard is an image file protocol. A file in the TIFF format is divided into three areas: the header, the image file directory, and the actual image data. These elements are described separately in the following paragraphs.

The notion of tags is the feature that identifies files in the TIFF format. A TIFF tag is a word integer that serves to identify the file structure that follows. For example, the tag value 103H indicates that the structure that follows contains data compression information. TIFF file processing software can search for this tag in order to determine which, if any, compression scheme was used in encoding the image data. TIFF tags are discussed in greater detail later in this section.

The TIFF Header

An image file in TIFF format must start with an 8-byte block called the header. Figure 15-12 shows the structure of the TIFF image file header.

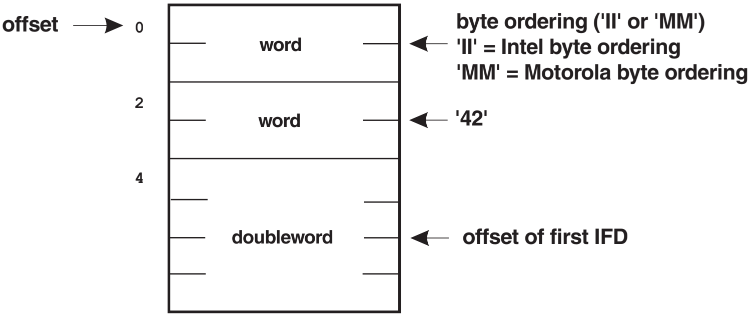

offset
0
word
byte ordering ('II' or 'MM')
'II' = Intel byte ordering
'MM' = Motorola byte ordering
2
word
'42'
4
doubleword
offset of first IFD

Figure 15-12 *TIFF File Header*

The word at offset 0 of the TIFF file header consists of the ASCII characters 'II' or 'MM'. The 'II' code identifies a file in the Intel byte ordering scheme, that is, word and doubleword entries appear with the least significant byte in the lowest numbered memory address. This data ordering format is sometimes known as the "little-endian" scheme. The 'MM' code identifies a file in the Motorola byte ordering order, that is, with the least significant byte of word and doubleword entries in the highest numbered memory address. This format is known as the "big-endian" scheme. The ASCII number '42' found at the word at offset 2 of the header serves to further identify a file in TIFF format. The numbers themselves have no documented significance. The ASCII code '42' has sometimes been called the TIFF version number, although it is not described as such in the standard. The doubleword at offset 4 of the header block contains the offset, in the TIFF file, of the first image file directory (IFD).

The file header block is the only TIFF file structure that must be located at a predetermined offset from the start of the file. The remaining structures can be located anywhere in the TIFF file. TIFF file processing code reads the data in the header block to certify that the file is in TIFF format and to make decisions regarding the data ordering scheme. A sophisticated application could be capable of making adjustments in order to read data both in the Intel and in Motorola orders, while another one could require data in a specific format.

The TIFF Image File Directory (IFD)

Once the code determines that the file is in TIFF format and that it is encoded in a valid ordering scheme, it uses the doubleword at offset 4 of the header (see Figure 15-12) in order to determine the location of the first image file directory (IFD). Notice that a TIFF file can contain more than one image. If so, each image in the file is associated with its own IFD. However, by far the more common situation is that a TIFF file contains a single image. This assumption is made in the code and examples for manipulating TIFF files. The structure of the IFD is shown in Figure 15-13.

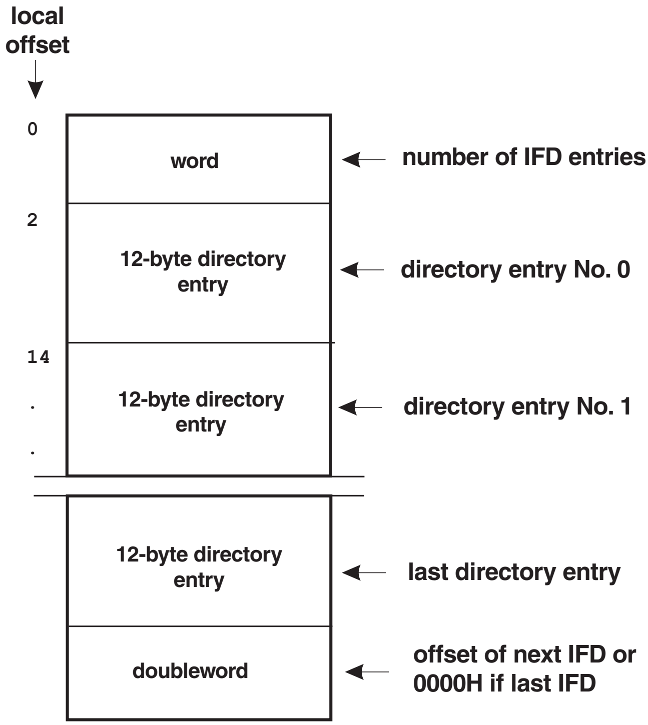


Figure 15-13 *TIFF Image File Directory (IFD)*

Observe that the offset values in the leftmost column of Figure 15-13 (labeled "local offset") refer to offsets within the IFD block. This must be so because the IFD itself can be located anywhere within the TIFF file. The word at local offset 0 of the IFD is a count of the number of directory entries. Recall that the number of directory entries is unlimited in the TIFF standard. The last directory entry is followed by a doubleword field which contains the offset of the next IFD, if one exists. If not, this doubleword contains the value 0000H (see Figure 15-13). Each entry in the IFD takes up 12 bytes. The structure of each IFD entry is shown in Figure 15-14.

The tag code is located at local offset 0 in the directory entry field. TIFF requires that the entry fields be sorted by increasing order of the tag codes, therefore, a lower numbered tag code always precedes a higher numbered one. This simplifies searching for a particular tag code since the search terminates when one with a higher numbered tag is encountered. The type code is located at local offset 2 within the directory entry field. Table 15-5 shows the type code values according to TIFF version 6.0. Be aware that code numbers 6 and higher were introduced in Version 6.0 and are not documented in previous versions of the standard.

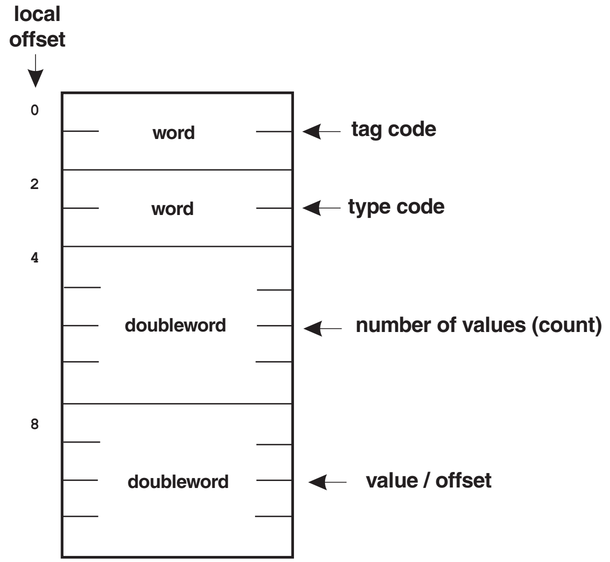


Figure 15-14 *TIFF Directory Entry*

Table 15-5

TIFF Version 6.0 Field Type Codes

TYPE CODE	STORAGE UNIT	FIELD CONTENTS
1	byte	8-bit unsigned integer
2	ASCII character	Offset of ASCII string terminated in NULL byte
3	word	16-bit unsigned integer
4	doubleword	32-bit unsigned integer
5	quadword	Rational number; the first doubleword is the numerator of a fraction and the last doubleword the denominator
6	byte	8-bit signed integer
7	byte	Undefined; can be used at will by the software
8	word	16-bit signed integer in 2's complement form
9	doubleword	32-bit signed integer in 2's complement form
10	quadword	Rational number; the first doubleword is the signed numerator of a fraction and the last doubleword the signed denominator
11	doubleword	Single precision floating point number in IEEE format
12	quadword	Double precision floating point number in IEEE format

The count field is a doubleword at offset 4 of the directory entry. This field, which was named the length field in previous versions of TIFF, encodes the number of data repetitions in the current directory entry. Notice that this value does not encode the number of bytes, but the number of data units. For example, if the field type code is 3 (word unit) then the count field would represent the number of data words of information that are associated with the entry.

The value/offset field is designated in this manner because it contains either a direct value or an offset into the TIFF file. The general rule is that if the encoded data fits into a doubleword storage (4 bytes) then the data is entered directly in the doubleword at local offset 8 of the directory entry (see Figure 15-14). This design saves coding space and simplifies processing. However, some TIFF tags, such as the StripOffset tag mentioned later in this section, always contain offset data in this field. The software determines if the data in the value/offset field is either a value or an offset by means of the tag, the field type code, and the data item count.

If the tag contains either a value or an offset, the program must first examine the field type codes (see Table 15-4). In this case data corresponding to field type codes 1, 3, 4, 5, 6, 7, 8, 9, and 11 (see Table 15-4), are contained in a doubleword storage unit and are therefore entered as values. By the same token, field types 2, 5, 10, and 12 encode an offset in the value/offset field of the directory entry. Once determined that an individual data item fits in the 4 bytes allocated to the value/offset field then the software must examine the number of values associated with the directory entry. If the total number of values exceeds the allocated space (4 bytes) then the value/offset field contains an offset. In this case the type code and the count fields are multiplied in order to determined the number of items supplied.

15.3.2 TIFF Tags for Bilevel Images

Over 50 tags have been defined in the TIFF standard; however, only a handful are used in most TIFF images. A complete description of all the TIFF tags can be found in the TIFF Revision 6.0 specification available, at no charge, from Aldus Corporation (see Section 15.3). The TIFF tags mentioned in the following discussion are those that would be commonly found in monochrome (bilevel in TIFF terminology) scanned images. These are also the tags decoded by the TIFFSHOW program found in the /TIFF directory of books' software package.

OldSubFileType (tag code 00FFH)

This tag, originally called the SubFileType, has been replaced by the NewSubFileType tag mentioned below; however, many older TIFF programs still use this tag. The tag provides information about the bitmap associated with the IFD. The tag can take the following values:

Value = 1 indicates that the image is in full-resolution format.

Value = 2 indicates the image data is in reduced-resolution format.

Value = 3 indicates that the image data is a single page of a multi-page image.

NewSubFileType (00FEH)

This tag, which replaces OldSubFileType, describes the kind of data in the IFD. The tag is made up of a doubleword integer with the following significant bits:

Bit 0 is set if the image is a reduced-resolution version of another image.

Bit 1 is set of the image is a single-page of a multi-page image.

Bit 2 is set if the image is a transparency mask (see the PhotometricInterpretation tag later in this section.)

ImageWidth (tag code 0100H)

This tag encodes the number of pixel columns in the image.

ImageLength (tag code 0101H)

This tag encodes the number of pixel rows in the image.

BitsPerSample (tag code 0102H)

This tag encodes the number of bits required to represent each pixel sample. The value of this tag is 1 for bilevel images, 4 for 16-color palette images, and 8 for 256-color palette images. In IBM video graphics systems the number of bits per sample is usually the same as the number of bits per pixel color. Regarding images encoded in RGB format (as used in some Macintosh systems and in the XGA Direct Color mode) the number of bits per sample refers to each individual color. In this case the SamplesPerPixel tag (described below) encodes the number of pixel colors (three colors in RGB encoding), and the BitsPerSample tag the number of bits assigned to each color. For example, if 6 bits are assigned to the red sample, 8 bits to the green, and 6 bits to the blue, the total number of bits per pixel would be 20.

Compression (tag code 0103H)

This tag encodes the compression scheme used in the image data. The tag can take the following values:

Value = 1 indicates that the image data is not compressed. Pixel information is packed at the byte level, as tightly as possible. Uncompressed data has the disadvantage over compressed data that it takes up more memory space. On the other hand, it has the advantage that it can be manipulated faster by the display routines.

Value = 2 indicates that image data is compressed according to CCITT Group 3 (Modified Huffman) run-length encoding.

Value = 32,773 (8005H) indicates the data is compressed according to the PackBits scheme described in detail later in this section.

PhotometricInterpretation (tag code 0106H)

This tag describes how to interpret the color encoding in the bitmap. The tag can take the following values:

Value = 0 is used in bilevel and grayscale images to indicate that a bit value of 0 represents the white color.

Value = 1 is used in bilevel and grayscale images to indicate that a bit value of 0 represents the black color.

Value = 2 is used to indicate an encoding in RGB format.

Value = 3 is used to indicate palette color format. In this case a ColorMap tag must be included to hold the LUT values.

Value = 4 indicates that the image is a transparency mask used to define an irregularly shaped region of another image.

Threshholding (tag code 0107H)

This tag describes the technique used for representing the gray scale in a black-and-white image. The tag can have the following values:

Value = 1 indicates that the image contains no dithering or halftoning. Bilevel images use this value.

Value = 2 indicates that the image has been dithered or halftoned.

Value = 3 indicates that a randomized process, such as the error diffusion algorithm, has been applied to the image data.

StripsOffset (tag code 0111H)

This tag provides the information necessary for the software to locate the image data within the TIFF file. By definition, the value in this tag is always an offset from the beginning of the TIFF file. The structure of the TIFF image data as well as the use of this tag is discussed in Section 15.3.3.

SamplesPerPixel (tag code 0115H)

This tag encodes the number of color components for each screen pixel. The value of this tag is 1 for bilevel, grayscale, and palette color images, and 3 for images in RGB format.

RowsPerStrip (tag code 0116H)

This tag determines the number of rows in each strip. Image encoding in the TIFF standard is discussed in Section 15.3.3.

StripByteCounts (tag code 0117H)

This tag determines the number of bytes in each strip, after compression. Image encoding in the TIFF standard is discussed in Section 15.3.3.

XResolution (tag code 011AH)

This tag provides information about the x-axis resolution at which the original image was created or scanned. The data is important to software that must reproduce the image exactly as it was originally produced. This is a critical factor in the reproduction of dithered images, which do not allow scaling.

YResolution (tag code 011BH)

This tag provides information about the y-axis resolution at which the original image was created or scanned. See the text in the XResolution tab.

PlanarConfiguration (tag code 011CH)

This tag provides information regarding the organization of color pixel data. It is relevant only for color images in RGB format (more than 1 samples per pixel). The tag can have the following values:

Value = 1 indicates that RGB data is stored in the order of the color components, that is, in a repeating sequence of RED, GREEN, and BLUE values. This organization is called the chunky format in TIFF documentation.

Value = 2 indicates that RGB data is stored by bit planes, that is, the red color components are stored first, followed by the green, and then by the blue. This organization is called the planar format in TIFF documentation.

ResolutionUnit (tag code 128H)

This tag determines the unit of measurement used in the parameters contained in XResolution and YResolution tags. Many TIFF programs do not use this tag, but it is recommended by the standard. The tag can have the following values:

Value = 1 indicate no unit of resolution.

Value = 2 indicates inches.

Value = 3 indicates centimeters.

15.3.3 Locating TIFF Image Data

Although TIFF file processing software often ignores many tags and makes assumptions regarding others, one necessary manipulation in an image display operation is the locating and decoding of the image bitmap.

TIFF Image data can be located almost anywhere in the file. This is true of both uncompressed and compressed data. Furthermore, the TIFF standard allows dividing an image into several areas, called strips. The idea is to facilitate data input and output in machines limited to a 64K segment size. This is the case of Intel processors operating in MS DOS or Windows systems. The data for each individual strip is represented by a separate tag.

When the image is divided into strips, three tags participate in locating the image data: RowsPerStrip, StripOffsets, and StripByteCounts. The first operation is for the software to calculate the number of strips into which the image data is divided. This value, which is not encoded in any particular tag, can be obtained from the number of values field of the StripOffsets tag (see Figure 15-14). The following code fragment shows the processing necessary to determine if a TIFF image is encoded in a single strip or in multiple strips.

```
; The number of strips in the image is obtained from the length
; field of the StripOffsets tag
; Code assumes that the SI register points to the start of the
; first IFD in the TIFF file
        MOV     AX,0111H        ; Tag for strip offsets
        CALL    FIND_TAG
        JNC     OK_OFFSETS      ; Go if tag found
;**********************|
;     ERROR handler    |
;**********************|
; At this point the code should contain an error routine to
; handle the case of a TIFF file with no StripOffsets tag.
        .
        .
        .
; At this label the processing has located the StripOffsets tag.
; Image can be encoded in one or more strips. The number of
; strips is stored in the length field of the StripOffsets tag
; Unpacking and display of multi-strip images requires the number
; of rows per strip and the number of bytes in each strip row. ; These
parameters are not necessary if the image is encoded in a
; single strip
OK_OFFSETS:
        MOV     AX,WORD PTR [SI+4]        ; Get number of strips
        CMP     AX,1            ; Test for single strip
        JNE     MULTI_STRIP     ; Go if not a single strip
        JMP     ONE_STRIP
;**********************|
;  multi-strip image   |
;**********************|
MULTI_STRIP:
; Multi-strip image processing routine
        .
        .
        .
;**********************|
;  single strip image  |
;**********************|
ONE_STRIP:
; Single strip processing routine
        .
        .
        .
FIND_TAG        PROC    NEAR
; Find a specific tag code in the Image File Directory
; On entry:
;          AX = desired tag code
;          SI ==> start of Image File Directory (IFD)
; On exit:
;          Carry clear if tag code found
;          SI ==> first tag field (code)
;
;          Carry set code not present in IFD
;
TEST_TAG_CODE:
        MOV     BX,WORD PTR [SI]        ; Get tag code
        CMP     BX,0            ; Test for last IFD
        JE      END_OF_IFD      ; Go if last
        CMP     AX,BX           ; Compare with one desired
        JNE     NEXT_TAG_CODE   ; Index if not
; At this point desired tag code has been found
        CLC                     ; Tag found return code
        RET
NEXT_TAG_CODE:
        ADD     SI,12           ; Index to next tag
; Test for last tag
        JMP     TEST_TAG_CODE   ; Continue
END_OF_IFD:
        STC                     ; Tag not found
        RET
FIND_TAG        ENDP
        .
        .
        .
```

Notice that the FIND_TAG procedure in the previous code fragment provides a convenient tool for indexing into the IFD in search of any particular tag code. Such a procedure would be called repeatedly by a TIFF image processing routine. The procedure named FIND_TIFF_TAG in the BITIO module of the GRAPHSOL library performs this operation.

Locating the image data in a single strip image consists of adding the value in the StripOffsets tag to the start of the TIFF file. In this case the image size (in bytes) is obtained by reading the value in the ImageWidth tag (which is the number of pixels per row), dividing it by 8 to determine the number of data bytes per pixel row, and multiplying this value by the number of pixel rows stored in the ImageLength tag. The processing operations can be seen in the TIFFSHOW.ASM file in the book's software package.

If the image data consists of multiple strips, then each strip is handled separately by the software. In this case the number of bytes in each strip, after compression, is obtained from the corresponding entry in the StripByteCounts tag. The display routine obtains the number of pixel rows encoded in each strip from the value stored in the RowsPerStrip tag. However, if the total number of rows, as stored in the ImageLength tag, is not an exact multiple of the RowsPerStrip value, then the last strip could contain less rows than the value in the RowsPerStrip tag. TIFF software is expected to detect and handle this special case.

15.3.4 Processing TIFF Image Data

Once the start of the TIFF image data is located within the TIFF file, the code must determine if the data is stored in compressed or uncompressed format and proceed accordingly. This information is found in the Compression tag previously mentioned. In TIFF Version 5.0 the Compression tag could hold one of six values. Value number 1 correspondes to no compression, values 2, 3, and 4 corresponded to three modes of CCITT compression, and value 5 to LZW compression, finally value 32,773 in the Compression tag indicates PackBits compression.

We mentioned that several of these compression schemes were dropped in Version 6.0 of the TIFF standard (see Section 15-3). In the present TIFF implementation, values 3, 4, and 5 for the Compression tag are no longer supported. Since there are substantial reasons to favor the LZW algorithm for the compression of color images (which was dropped in TIFF Version 6.0 because of patent rights considerations) we have limited the discussion on TIFF image decoding to the case of PackBits compression. Hopefully, a future TIFF version will again support LZW compression methods.

TIFF PackBits Compression

The PackBits compression algorithm was originally developed on the Macintosh computer. The MacPaint program uses a version of PackBits compression for its image files. Macintosh users have available compression and decompression utilities for files in this format. The compression scheme is simple to implement and often offers satisfactory results with monochrome and scanned images.

PackBits, as implemented in TIFF, is a byte-level, simplified run-length compression scheme. The encoding is based on the value of the first byte of each compressed data unit, often designated as the "n" byte. The decompression logic can be described in the following steps.

```
STEP 1:  If end-of-information code then end decompression.
STEP 2:  Read next source byte. Designate as n (n is an unsigned integer).
STEP 3:  if n is in the range 0 to 127 (inclusive) perform the following operations:
            a. read the next n+1 bytes literally from the source file into the output stream.
            b. go to STEP 1.
STEP 4:  if n is in the range 129 to 255 (inclusive) perform the following operations:
            a. negate n (n = -n).
            b. copy the next byte n+1 times to the output stream.
            c. go to STEP 1.
STEP 5:  Goto STEP 1.
```

Notice that in the above description we assume that n is an unsigned integer. This convention, which facilitates coding in 80x86 assembly language, differs from other descriptions of the algorithm in which n is a signed value. Figure 15-15 is a flowchart of this decompression logic.

Observe that in the TIFF implementation of PackBits no action is taken if n = 128. If n = 0 then 1 byte is copied literally from source to output. The maximum number of bytes in a compression run is 128. In addition, the TIFF implementation of PackBits compression adopted the following special rules:

1. Each pixel row is compressed separately. Compressed data cannot cross pixel row boundaries.
2. The number of uncompressed bytes per row is defined as the value in the ImageWidth tag, plus 7, divided by 8. If the resulting image map has an even number of bytes per row, the decompression buffer should be word-aligned.

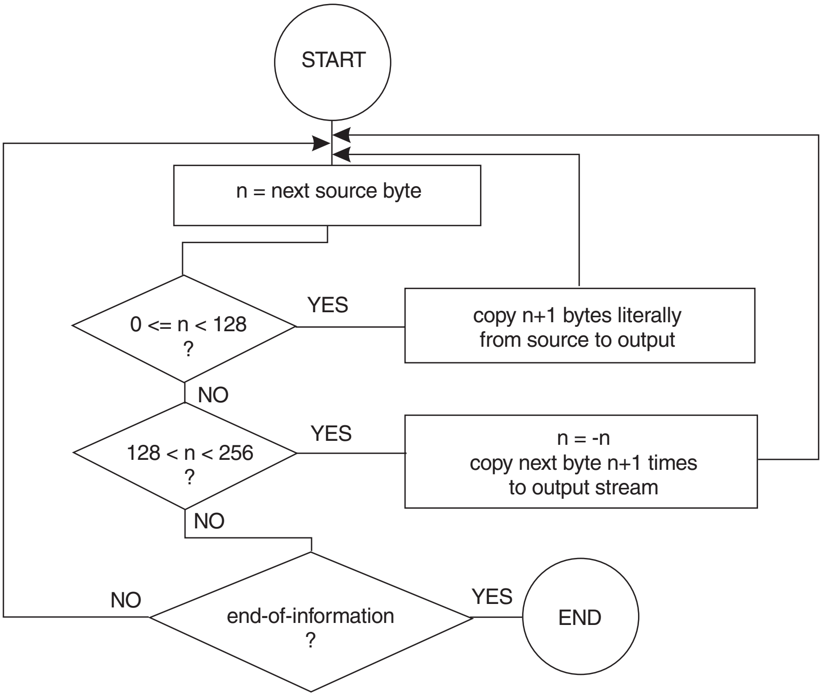


Figure 15-15 *TIFF PackBits Decompression*

The following code fragment shows the processing required for unpacking a TIFF file compressed as a single strip, using the PackBits method.

```
; Unpacking logic for TIFF PackBits scheme
; PackBits packages consist of 2 bytes. The first byte (n)
; encodes the following options:
;       1. if n is in the range 0 to 127 then the next n+1 bytes
;          are to be interpreted as literal values
;       2. if n is in the range -127 to -1 then the following
;          byte is repeated -n+1 times
;       3. if n = 128 then no operation is executed
; Code assumes:
;       1. SI --> start of the compressed image (1 strip)
;       2. DI --> storage buffer for decompressed image
;       3. the variable IMAGE_SIZE holds the byte size of the
;          uncompressed image. In a single strip image this
;          value is obtained by dividing ImageWidth (number
;          of pixels per row) by 8 and multiplying by ImageLength
;
; Note: the routine keeps track of the number of bytes in the
;       decompressed bitmap in the variable EXP_COUNT. This
;       value is compared to the IMAGE_SIZE variable to determine
;       the end-of-information point
;*********************|
;   test value of n   |
;*********************|
TEST_N_BYTE:
        MOV     AL,[SI]         ; Get n byte
        CMP     AL,128          ; Code for NOP
        JB      LITERAL_CODE    ; Go if in the literal range
        JA      REPEAT_CODE     ; Go if in repeat range
; At this point n = 128. No operation is performed
        INC     SI              ; Skip NOP code
        JMP     NEXT_PACK_CODE  ; Continue
;**********************|
;    0 <= n < 128      |
;  (literal expansion) |
;**********************|
LITERAL_CODE:
        MOV     CL,AL           ; Counter to CL
        MOV     CH,0            ; Clear high byte of counter
        INC     CX              ; Add 1
        INC     SI              ; Skip n byte
        ADD     EXP_COUNT,CX    ; Add bytes to counter
LIT_MOVE:
        MOV     AL,[SI]         ; Get literal byte
        MOV     [DI],AL         ; Place in bitmap
        INC     DI              ; Bump pointers
        INC     SI
        LOOP    LIT_MOVE
        JMP     NEXT_PACK_CODE
;**********************|
;    128 < n < 256     |
; (repeated expansion) |
;**********************|
REPEAT_CODE:
        NEG     AL              ; Negate to convert to 2's
                                ; complement representation
        MOV     CL,AL           ; Counter to CL
        MOV     CH,0            ; Clear high byte of counter
        INC     CX              ; Add 1
        INC     SI              ; Skip n byte
        ADD     EXP_COUNT,CX    ; Add bytes to counter
                                ; to keep track of decompressed
                                ; bytes
        MOV     AL,[SI]         ; Get byte to repeat
        INC     SI              ; Skip to next n byte
EXP_MOVE:
        MOV     [DI],AL         ; Place byte in buffer
        INC     DI              ; Bump bitmap pointer
        LOOP    EXP_MOVE
;**********************|
;     test for <EOI>   |
;**********************|
; EXP_COUNT holds the byte count in bitmap at this point
; IMAGE_SIZE holds the total bytes in the expanded bitmap
NEXT_PACK_CODE:
        MOV     AX,EXP_COUNT    ; Bytes now in bitmap
        CMP     AX,IMAGE_SIZE   ; Compare with map size
        JAE     EOI_FOUND       ; Go if at end of image
        JMP     TEST_N_BYTE
; Decompression has concluded at this label
EOI_FOUND:
        .
        .
        .
```

15.3.5 TIFF Software Samples

The book's software package includes several software items related to TIFF file operations. In the first place we have furnished source and executable files for a rudimentary TIFF reader program named TIFFSHOW. Notice that the code is limited to the analysis, decompression, and display of small, bilevel TIFF files. The data in the source TIFF encoding can be either uncompressed or compressed by means of the PackBits option. The code also requires that the data be located in a single strip. TIFFSHOW can be used to examine the TIFF format files (extension .TIF) that are part of the MATCH program. For this reason TIFFSHOW is included in the \MATCH directory of the book's software package.

In addition to the TIFFSHOW program, we have also furnished several TIFF procedures as part of the GRAPHSOL library. These procedures are located in the BITIO.ASM module. The procedure named SHOW_TIFF can be used to display a bitmap encoded in TIFF bilevel format. This procedure requires that the user pass a formatted data block, as shown in the header. The SHOW_TIFF procedures calls the procedure named LOAD_TIFF, also in the BITIO module, which decompresses and loads the encoded image. One advantage of using these library procedures is that they place the TIFF file and the image bitmap in a separate data segment, therefore freeing the caller's code from having to devote storage space to TIFF data.

15.4 The Hewlett-Packard Bitmapped Fonts

The LaserJet line of printers, manufactured by Hewlett-Packard Corporation, has gained considerable popularity in the microcomputer world. For use in these printers Hewlett-Packard developed a standard for encoding text characters, sometimes known as the Hewlett-Packard Printer Control Language (PCL) bitmap convention. Fonts in PCL format are widely available as disk files (soft fonts) from Hewlett-Packard and other companies. Although these fonts are primarily designed for use in laser printers that recognized the PCL printer language (discussed in Chapter 11), they can also be put to less conventional uses. For example, in the MATCH program (furnished in the book's software) we have used PCL soft fonts to display text message in larger letters than those available in the VGA system.

In the present section we discuss the structure and design of the PCL soft fonts. However, the PCL bitmap format is a refined and elaborate one. We believe that the information presented here is sufficient to make the PCL bitmap technology accessible to the graphics programmer. On the other hand, the design of new soft fonts in PCL bitmap format requires knowledge of typography and character graphics as well as a high degree of familiarity with the PCL encoding. Hewlett-Packard has published several technical reference manuals for their LaserJet printers that include detailed description of the PCL bitmap fonts. These titles (listed in the Bibliography) can be obtained directly from Hewlett-Packard or through one of their dealers.

Notice that PCL commercial fonts are usually copyright by the font developers or vendors. The programmer should investigate the legality of the intended use before distributing or modifying the font files.

15.4.1 PCL Character Encoding

Two technologies are commonly used for encoding text characters: bitmaps and vector graphics. We encountered vector fonts in the short stroke vector characters used in 8514/A and XGA systems (see Section 11.3.5). Some printers of the Hewlett-Packard LaserJet family are equipped with vector fonts supplied in the form of scalable character sets; Hewlett-Packard has adopted the scalable font technology developed by Agfa Corporation, designated as the Font Access and Interchange Format (FAIS).

Bitmapped fonts in the PCL format are compatible with all HP PCL laser printers. In addition, several commercial programs are available to generate and edit font files in PCL format. One advantage of fixed-size bitmapped fonts if that their display quality is often judged to be better than the one obtained from scalable fonts.

A PCL-format soft font disk file contains the following elements:

1. One font descriptor field that encodes the general characteristics of the font.
2. One or more character descriptors fields that encodes the data pertaining to each individual character as well as the character bitmap. The bitmap is the binary raster data that defines the character's shape.
3. Several command strings in PCL language.

The PCL command strings are unrelated to the font definition, although they are sometimes used to locate specific data areas within the file. These command strings are provided to facilitate programming of LaserJet printers and compatible devices, a subject discussed in detail in Chapter 11.

Font Descriptor

The first element we encounter in a font file in PCL format is a PCL language command string. The initial command in the font file is the one used to download the font descriptor field into a PCL printer. The command can be generically represented as follows:

```
1BH's???W'
```

The value 1BH is the escape code that precedes all PCL commands (see Chapter 11). The character string 's???W' represent a generic command in which the question mark ('?') takes the place of one to three ASCII characters that encode the byte length of the descriptor field. Table 15-6 is a partial screen snapshot of the Hewlett-Packard font file TR140RPN.UPS

The data elements in the PCL bitmap font descriptor field are those that apply to the entire font. There are 33 data entries in the font descriptor field of PCL level 5, although software and devices often ignore many of these entries. The font descriptor field starts after the end of the download command string. In Table 15-6 the command string takes the form

```
1BH )s257W
```

Table 15-6

Hexadecimal and ASCII Dump of the HP PCL Font File TR140RPN.UPS

```
                            HEXADECIMAL DUMP
OFFSET     0  1  2  3  4  5  6  7    8  9  A  B  C  D  E  F       ASCII DUMP

 0000      1B 29 73 32 35 37 57 00   40 00 00 00 00 00 2B 00     .)s257W.........
 0010      3C 00 3E 00 01 00 15 00   3C 00 E9 00 6C 00 00 00     ................
 0020      05 00 06 00 00 F1 04 01   18 00 6D 00 20 00 7F 00     ................
 0030      55 00 00 00 00 00 00 54   6D 73 52 6D 6E 20 20 20     .......TmsRmn
 0040      20 20 20 20 20 20 20 00   BF 01 28 43 29 20 43 6F     (C) Co
 0050      . . .                                                 pyright Hewlett
 0060                                                            Packard Company,
 0070                                                            1986. All right
 0080                                                            s reserved.  Rep
 0090                                                            roduction, adapt
 00A0                                                            ation or distrib
 00B0                                                            ution of copies
 00C0                                                            of this font is
 00D0                                                            prohibited, exce
 00E0                                                            pt as allowed un
 00F0                                               . . .        der the copyrigh
 0100      74 20 6C 61 77 73 2E 20   1B 2A 63 33 33 45 1B 28     t laws. .*c33E.(
 0110      73 35 37 57 04 00 0E 01   00 00 00 06 00 27 00 07     s57W............
 0120      00 29 00 4C 38 7C FE FE   FE FE FE FE FE 7C FE 7C     ................
 0130      7C 7C 7C 7C 7C 7C 38 7C   38 38 38 38 38 38 38 10     ................
 0140      38 10 00 00 00 00 08 7C   FE FE FE 7C 38 1B 2A 63     ..............*c
 0150      33 34 45 1B 28 73 34 36   57 04 00 0E 01 00 00 00     34E.(e46W.......
```

The ASCII characters '257' in the sample of Table 15-6 encodes the number of bytes in the font descriptor field, not including the command string. Notice, in Table 15-6, that the end of the font descriptor field coincides with the command string 1BH *c33E (see dump offset 0108H) discussed later in this section.

Font descriptor data starts at offset 0007H of the dump shown in Table 15-6. The size of the binary data section of the font descriptor is 64 bytes (40H). The remainder of the font descriptor, from the byte at offset 0048H to the end of the field, contains an optional ASCII-coded copyright message preceded by a character count (BFH). Table 15-7, on the following page, shows the elements in the bitmap font descriptor field according to PCL level 5.

Data in the font descriptor field is stored according to the big-endian scheme, that is, the low-order element of word and doubleword entries are located at the highest memory location. For example, at offset 0007H of the dump of Table 15-6 is the font descriptor size word, which, in this case, has the value 00 40. If we were to load this word value into a register of a processors that follows the little-endian storage scheme (such as the ones used in IBM microcomputers) the high- and low-order elements would be inverted. In this case the code can exchange the low- and high-bytes in order to correct this situation. The processing operations can be followed in the source code for the CSETHP program furnished in the book's software package.

Table 15-7

PCL Bitmap Font Descriptor Field

OFFSET	STORAGE UNIT	VALUE RANGE	CONTENTS
0	word	64	Font descriptor size
2	byte	0	Descriptor format (0 = bitmap)
3	byte	0/1/2	Font type: 0 = 7 bit (96 characters) 1 = 8 bits (192 characters) 2 = 8 bits (256 characters)
4	byte		Style, MSB (see offset 23)
5	byte		RESERVED
6	word		Baseline distance (in PCL dots)
8	word		Cell width (in PCL dots)
10	word		Cell height (in PCL dots)
12	byte	0/1/2/3	Orientation: 0 = portrait 1 = landscape 2 = reverse portrait 3 = reverse landscape
13	byte	0/1	Spacing (fixed or proportional) 0 = fixed spacing 1 = proportional spacing
14	word		Symbol set
16	word		Pitch (in PCL quarterdots)
18	word		Height (in PCL quarterdots)
20	word		xheight (in PCL quarterdots)
22	byte		Width type (code)
23	byte		Style, LSB (see offset 4)
24	byte		Stroke weight (code)
25	byte		Typeface family, LSB
26	byte		Typeface family, MSB
27	byte		Serif style (code)
28	byte	0/1/2	Quality (code): 0 = data processing 1 = near letter quality 2 = letter quality
29	byte		Placement (code)
30	byte		Underline distance (in PCL dots)
31	byte		Underline height (in PCL dots
32	word		Text height (in PCL quarterdots)
34	word		Text width (in PCL quarterdots)
36	word		First printable character
38	word		Last printable character
40	byte		Pitch field extension
41	byte		Height field extension
42	word		Cap height (percent of em)
44	doubleword		Font number (code)
48	15 bytes		ASCII font name
64	---		Start of optional copyright notice

note: data is in Motorola storage format (big-endian scheme)

Many of the data entries in the font descriptor field would be of interest only to the font designer or graphics text specialist. Other entries contain information that would be required only in developing sophisticated text management functions, such as those expected of a typesetting or desktop publishing program. Figure 15-16 shows the fundamental information furnished by the font descriptor field.

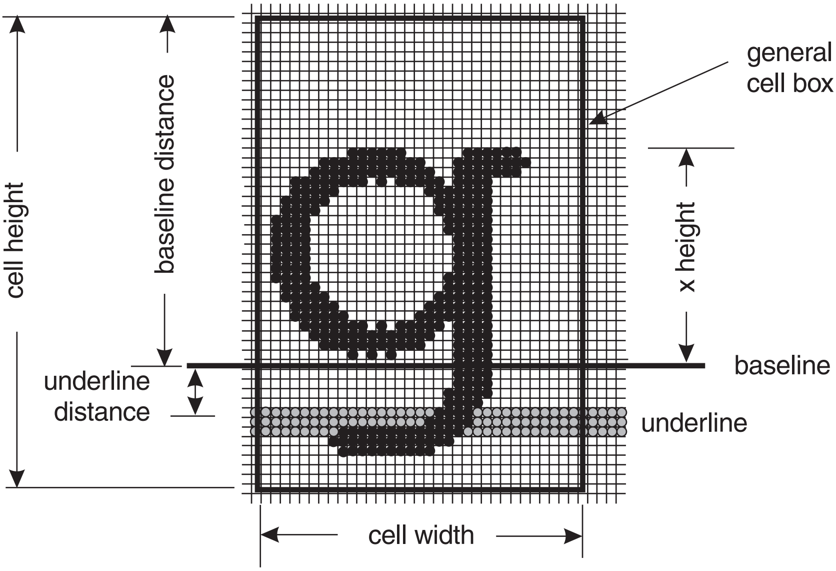


Figure 15-16 *PCL Bitmap Character Cell*

In Figure 15-16 notice that the dimensions labeled "cell width" and "cell height" correspond to the entries at offset 6 and 8 of the font descriptor (see Table 15-7), while the "baseline distance" is found at offset 6, the "underline distance" at offset 30, and "x-height" dimension at offset 20.

Character Descriptor

The PCL format is optimized so that each bitmap takes up the minimum storage space. In this respect the cell height and cell width parameters of Figure 15-16 refer merely to a "general cell box" that is required to enclose all the characters in the font. The character descriptor field contains the data elements that define the individual character.

Like the font descriptor field, the disk image of the character descriptor field starts with a PCL command string. In fact, in the case of individual characters two command strings are necessary: the first one, known as the character code command, is used to inform the device of the decimal code for the character that follows. At offset 0108H, in Table 15-6, we can see the first PCL command string, which is

```
1BH 'c33E'
```

In this case the decimal value '33' identifies the '!' symbol, which is located at this position in the character table (see Table 1-2). The second command string, known as the character descriptor and data command is used to download to the PCL device the information associated with the particular character as well as its bitmap. At offset 010EH of the dump at Table 15-6 we can see this command string:

```
1BH 's57W'
```

In this case the sub-string '57' encodes, in ASCII, the byte length of the data descriptor plus its corresponding bitmap. Immediately following this PCL command string we find a 16-byte block which is called the character header area of the character descriptor field. Table 15-8 shows the data elements in the character descriptor header.

Table 15-8

PCL Bitmap Character Descriptor Header

OFFSET	STORAGE UNIT	VALUE RANGE	CONTENTS
0	byte	4/10	Data format: 4 = LaserJet family 10 = Intelifont scalable
1	byte	0/1	Continuation: 0 = bitmap is a character block 1 = bitmap is a continuation of another block
2	byte	14/2	Size of character descriptor header: 14 byte for LaserJet family 2 for Intelifont scalable
3	byte	1/2/3/4	Bitmap class: 1 = uncompressed bitmap 2 = compressed bitmap 3 = Intelifont scalable 4 = Compound contour (Intelifont)
4	byte	0/1/2/3	Orientation: 0 = portrait 1 = landscape 2 = reverse portrait 3 = reverse landscape
5	byte		RESERVED
6	word		Left offset (in PCL dots)
8	word		Top offset (in PCL dots)
10	word		Character width (in PCL dots)
12	word		Character height (in PCL dots)
14	word		Delta x (in PCL quarter dots)
16	----		Start of character bitmap

Notice that the character header field has a total length of 16 bytes, which is consistent with the value 14 decimal stored at offset 2 of the character descriptor header, since this last value refers to the "remaining" portion of the header field.

The continuation entry, at offset 1 of the header block, is related to the limit of 32,767 bytes imposed by the PCL language on the character bitmap. If a character bitmap exceeds this limit it has to be divided into two or more sections. In this case the continuation entry is set to indicate that the associated bitmap (which follows this byte) is a continuation of the previous one.

The entry at offset 3 indicates the bitmap class. Most PCL character bitmaps in commercial use are in the uncompressed format (class 1). Compressed bitmaps use a run-length compression scheme. This variation, introduced in PCL level 5, is not compatible with level 4 devices, such as the LaserJet series II and compatible printers. For this reason we will not discuss the compressed encodings any further.

The entry at offset 4 indicates the orientation of the character. The word "portrait" is used in this context in reference to a character that takes up a vertical rectangle, such as the one in Figure 15-16. By the same token, characters located in a horizontal rectangle are referred to as being of "landscape" orientation. Notice that this use of the words "portrait" and "landscape" is related to photographic terminology.

The remaining entries in the character descriptor header refer to the character's dimensions. Figure 15-17 shows the locations of these dimensions in a sample character.

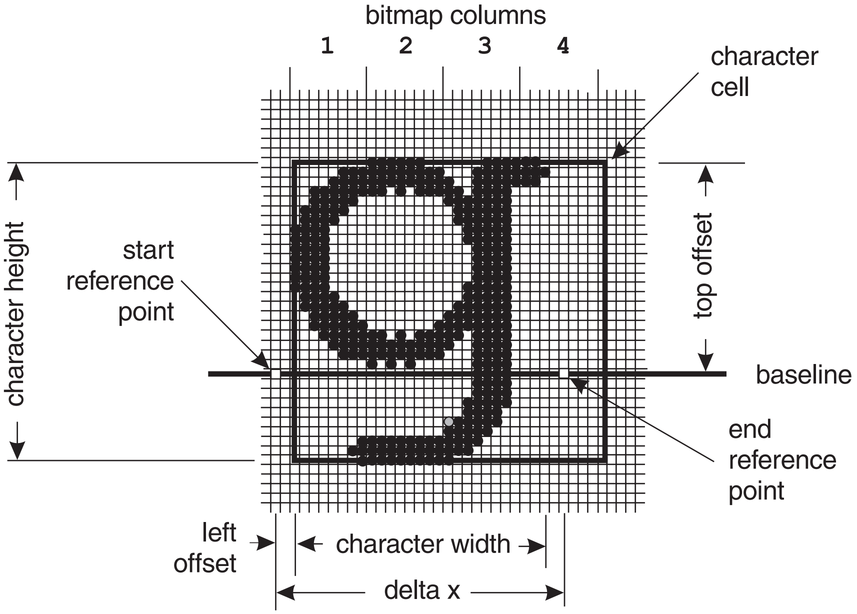


Figure 15-17 *PCL Character Dimensions*

Notice the two reference points along the baseline of the character in Figure 15-17. The start reference point can be thought of as the cursor position at the start of character display. The end reference point marks the cursor position once the character is displayed. The character reference points are used by software in implementing typesetting operations such as kerning and proportional spacing (see Glossary).

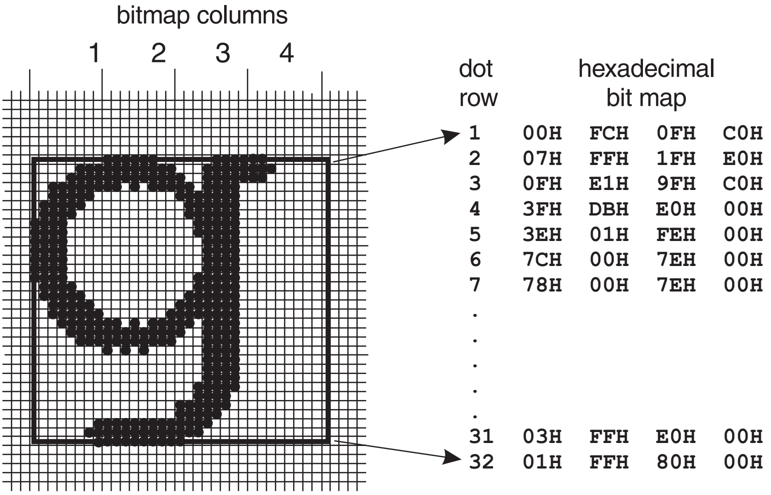
The word entry at offset 6 in Table 15-8 indicates the character's left offset (see Figure 15-17). This dimension is the distance, expressed in PCL dots, from the character pattern to the start reference point. The word at offset 7 refers to the character's top offset, which is the distance from the reference points to the top of the character pattern.

The character width entry, located at offset 10 in Table 15-8, determines the character's dot width. The dimension extends from the leftmost dot to the rightmost one. Notice that the actual bitmap often requires padding so that it can be encoded in byte-size storage units. Therefore the character width may not coincide with the width of the bitmap, as is the case in the character shown in Figure 15-17. The character height, located at offset 12 in Table 15-8, is the measurement of the number of vertical dots in the character map (see Figure 15-17). The delta x dimension, located at offset 14 in Table 15-8, is the distance, measured in PCL quarter dots, from the start reference point to the end reference point (see Figure 15-17).

The PCL Bitmap

The bitmap for each particular character starts at offset 16 of the character descriptor header. Software can obtain the bitmap dimensions from the character width and character height entries in the header. For example, in Table 15-6 we find the character width for the first character in the set at offset 011EH. The value in this case is 00 07 (7 decimal) which indicates that the character map is 7 dots wide. Since storage must be in byte units, the bitmap takes up 1 horizontal byte, in which the low-order bit is padded with zero. The bitmap height is obtained from the character height dimensions at offset 120H, which stores the value 00 29 (41 decimal). Therefore we calculate that the character bitmap is 1 byte wide and 41 bytes high, which means that it occupies 41 bytes of memory space.

Notice that the first character represented in Table 15-6 corresponds with the decimal value 33. The font uses the conventional US symbol set, therefore we can refer to Table 1-2 and find that the value 33 (21H) corresponds to the exclamation point symbol. This means that the 1-by-41 bitmap mentioned in the preceding paragraph represents the exclamation point symbols in the Hewlett-Packard TmsRmn, 14 point, normal density, portrait font encoded in the file named TR140RPN.USP. Figure 15-18 shows the bitmap for the lowercase letter "q" used in the previous illustrations.



dot row	hexadecimal bit map			
1	00H	FCH	0FH	C0H
2	07H	FFH	1FH	E0H
3	0FH	E1H	9FH	C0H
4	3FH	DBH	E0H	00H
5	3EH	01H	FEH	00H
6	7CH	00H	7EH	00H
7	78H	00H	7EH	00H
.				
.				
.				
.				
.				
31	03H	FFH	E0H	00H
32	01H	FFH	80H	00H

Figure 15-18 *Character Dot Drawing and Bitmap*

15.4.2 PCL Bitmap Support Software

The book's software package includes several software items related to PCL bitmap operations. The VGA2 module of the GRAPHSOL library includes two procedures which allow the screen display of a Hewlett-Packard printer font in PCL format. One procedure, named READ_HPFONT, allows loading a PCL soft font into RAM. The second procedure, named FINE_TEXTHP, allows displaying a text message using a previously loaded PCL font. The MATCH program (in the book's software package) uses PCL fonts to display large screen text. It is also possible to display screen text message using PCL fonts while in XGA and SuperVGA modes. In this case it is first necessary to call the SET_DEVICE procedure in the VGA3 module in order to enable XGA or SuperVGA display operations.

In addition to the library routines, the book's software package contains a program named CSETHP which displays all the characters in a PCL disk file. The program uses a VGA graphics mode.

Part III

Windows API Graphics

Chapter 16

Graphics Programming in Windows

Topics:

- Using Developer Studio wizards
- Elements of a Windows program
- WinMain()
- The Windows procedure
- Using program resources
- The HelloWindows program

This chapter is a brief review of the basic techniques used in Windows API programming. The book assumes a basic level of Windows programming skills, therefore it is not a intended to teach Windows programming, but to serve as a review and a refresher. Furthermore, we need to agree upon a code base for the chapters that follow. Here we establish the code structures and the coding style for the rest of the book.

16.1 Windows at the API Level

Our approach to Windows programming is to avoid class libraries or other wrappers, such as The Microsoft Foundation Classes (MFC). At an initial level of Windows programming the use of pre-canned interfaces may have some attraction; however, in high-performance graphics these packages are, at best, a nuisance and more often a major hindrance. On the other hand, we do take advantage of the editing and code generating facilities provided by Developer Studio and use the program-generating wizards, since there is no control or performance price to be paid in this case.

Before we can create a major graphics application we must be able to construct the Windows code framework that supports it. Fabricating a program requires not only knowledge of the programming language, but also skills in using the development environment. For example, to create an icon for your program's title bar you need to know about the API services that are used in defining and loading the icon, but you also need to have skills in using the icon editor that is part of Developer Studio. Even after the icon has been created and stored in a file, you need to follow a series of steps that make this resource available to the program.

16.1.1 The Program Project

We assume that you have already installed one of the supported software development products. The text is compatible with Microsoft Visual C++ Version 5.0 and later. We used Visual C++ Version 6.0 in creating the sample programs for this book. The following section describes the steps in creating a new project in Microsoft Developer Studio, inserting a source code template into the project, modifying and saving the template with a new name, and compiling the resulting file into a Windows executable.

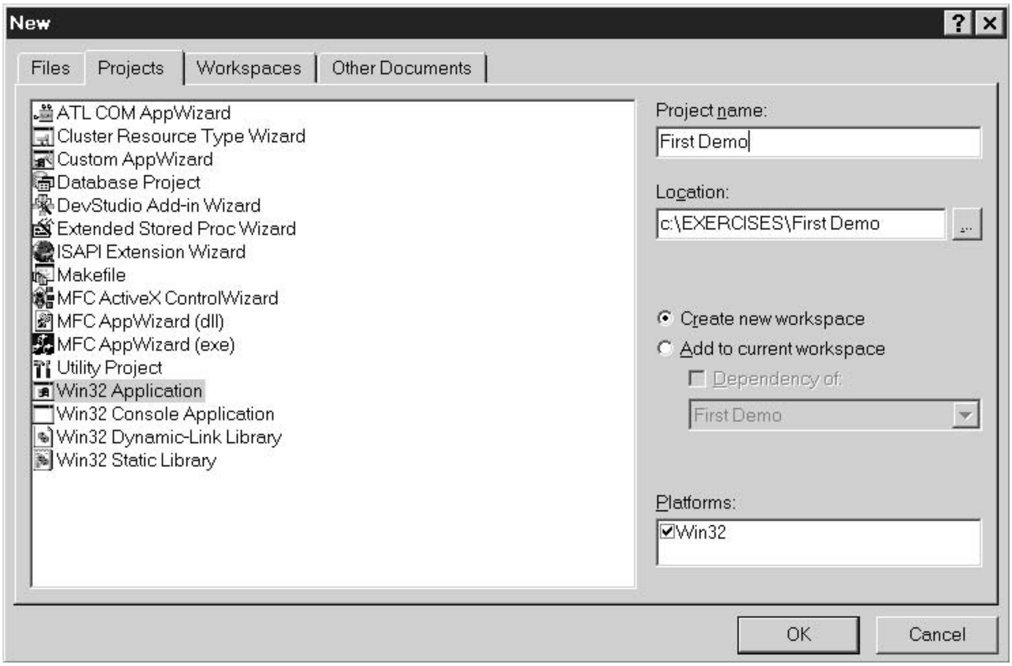
Creating a Project

You start Developer Studio by double-clicking on the program icon on the desktop, or selecting it from the Microsoft Visual C++ program group. The initial screen varies with the program version, the Windows configuration, the options selected when Developer Studio was last executed, and the project under development. Version 5.0 introduced the notion of a project workspace, also called a workspace, as a container for several related projects. In version 5 the extension .mdp, used previously for project files, was changed to .dsw, which now refers to a workspace. The dialog boxes for creating workspaces, projects, and files were also changed. The workspace/project structure and the basic interface are also used in Visual C++ Version 6.0.

We start by creating a project from a template file. The walkthrough is intended to familiarize the reader with the Developer Studio environment. Later in this chapter you will learn about the different parts of a Windows program and develop a sample application. We call this first project Program Zero Demo, for the lack of a better name. The project files are found in the Program Zero project folder in the book's software package.

A project is located in a workspace, which can include several projects. Project and workspace can be located in the same folder or subfolder or in different ones, and can have the same or different names. In the examples and demonstration programs used in this book we use the same folder for the project and the workspace. The result of this approach is that the workspace disappears as a separate entity, simplifying the creation process.

A new project is started by selecting the New command from the Developer Studio File menu. Once the New dialog box is displayed, click on the Project tab option and select a project type from the displayed list. In this case our project is Win32 Application. Make sure that the project location entry corresponds to the desired drive and folder. If not, click the button to the right of the location text box and select another one. Next, enter a project name in the corresponding text box at the upper right of the form. The name of the project is the same one used by Development Studio to create a project folder. In this example we create a project named Program Zero Demo which is located in a folder named 3DB_PROJECTS. You can use these same names or create ones of your liking. Note that as you type the project name it is added to the path shown in the location text box. At this point the New dialog box appears as in Figure 16-1.



Project name: First Demo

Location: c:\EXERCISES\First Demo

● Create new workspace

○ Add to current workspace

□ Dependency of: First Demo

Platforms: ☑ Win32

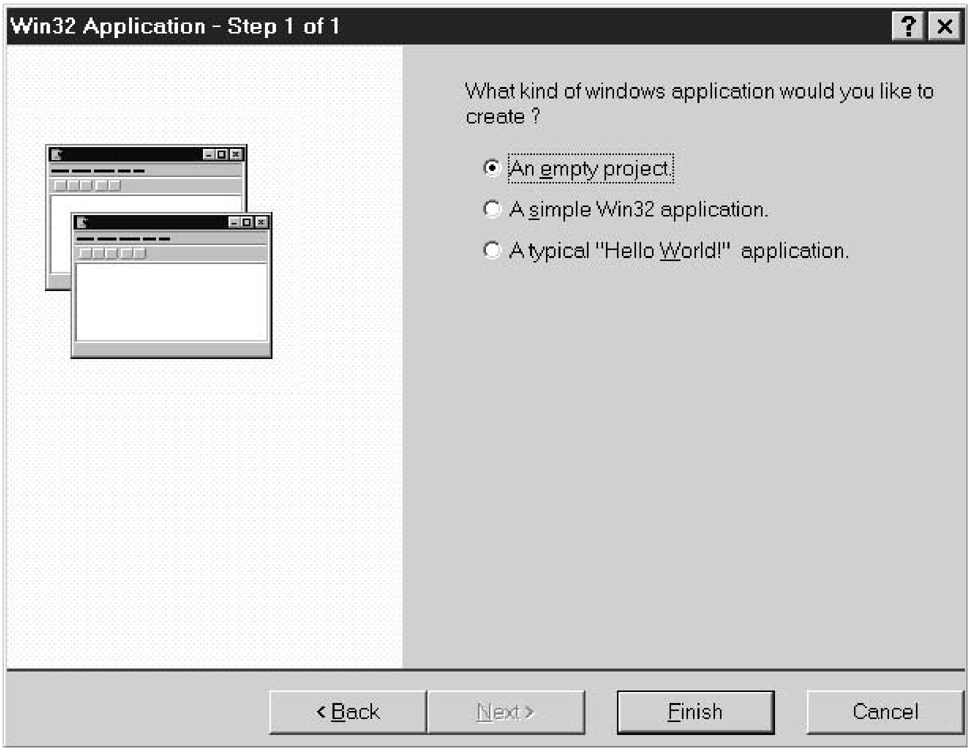
Figure 16-1 *Using the New Command in Developer Studio File Menu*

Make sure that the radio button labeled Create new workspace is selected so that clicking the OK button on the dialog box creates both the project and the workspace. At this point, you have created a project, as well as a workspace of the same name, but there are no program files in it yet. How you proceed from here depends on whether you are using another source file as a base or template or starting from scratch.

If you wish to start a source file from scratch, click on Developer Studio Project menu and select Add To Project and New commands. This action displays the same dialog box as when creating a project, but now the Files tab is open. In the case of a source file, select the C++ Source File option from the displayed list and type a file name in the corresponding text box. The dialog appears as shown in Figure 16-2, on the following page.

The development method we use in this book is based on using source code templates. To use a template as a base, or another source file, you have to follow a different series of steps. Assuming the you have created a project, the next step is to select and load the program template or source file. We use the template named Templ01.cpp. If you have installed the book's software in your system, the template file is in the path 3DB/Templates.

To load the source file into your current project, open Developer Studio Project menu and select Add To Project item and then the Files commands. This action displays an Insert Files into Project dialog box. Use the buttons to the right of the Look in text box to navigate into the desired drive and folder until the desired file is selected. Figure 16-3 shows the file Templ01.cpp highlighted and ready for inserting into the project.



What kind of windows application would you like to create ?

● An empty project.

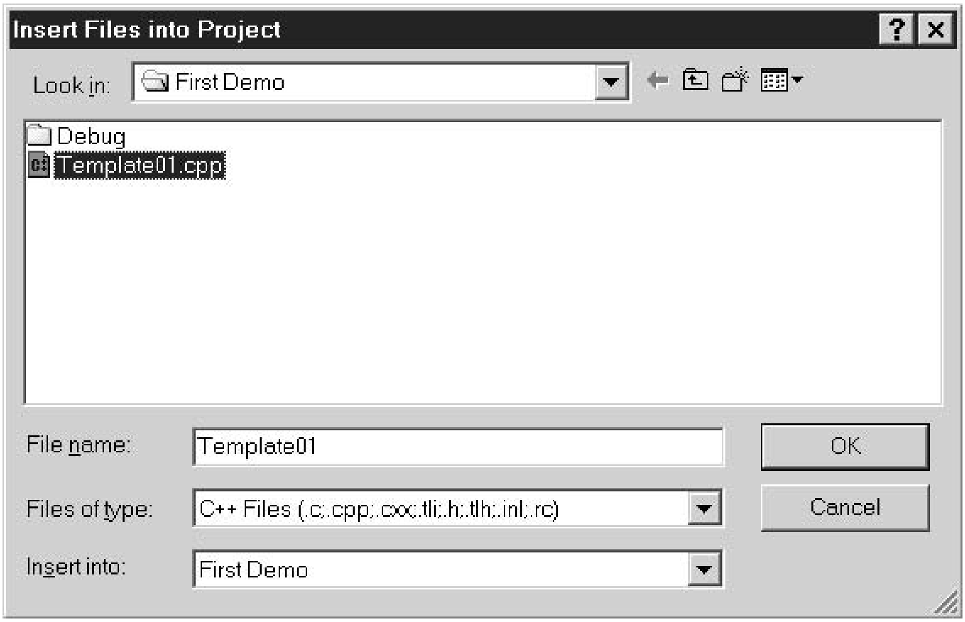
○ A simple Win32 application.

○ A typical "Hello World!" application.

Figure 16-2 *Creating a New Source File In Developer Studio*

When using a template file to start a new project you must be careful not to destroy or change the original source. The template file is usually renamed once it is inserted into the project. It is possible to insert a template file in a project, rename it, delete it from the project, and then reinsert the renamed file. However, it is easier to rename a copy of the template file before it is inserted into the project. The following sequence of operations are used:

1. Click the File menu and select the Open command. Navigate through the directory structure to locate the file to be used as a template. In this case the file Templ01.cpp is located in 3DB/Templates folder.
2. With the cursor still in Developer Studio editor pane, open the File menu and click on the Save As command. Navigate through the directory structure again until you reach the 3DB_PROJECTS\Program Zero Demo folder. Save the file using the name Prog_zero.cpp.
3. Click on the Project menu and select the commands Add to Project and Files. Locate the file named Prog_Zero.cpp in the Insert Files into Project dialog box, select it, and click the OK button.



Insert Files into Project

Look in: First Demo

Debug
Template01.cpp

File name: Template01

Files of type: C++ Files (.c;.cpp;.cxx;.tli;.h;.tlh;.inl;.rc)

Insert into: First Demo

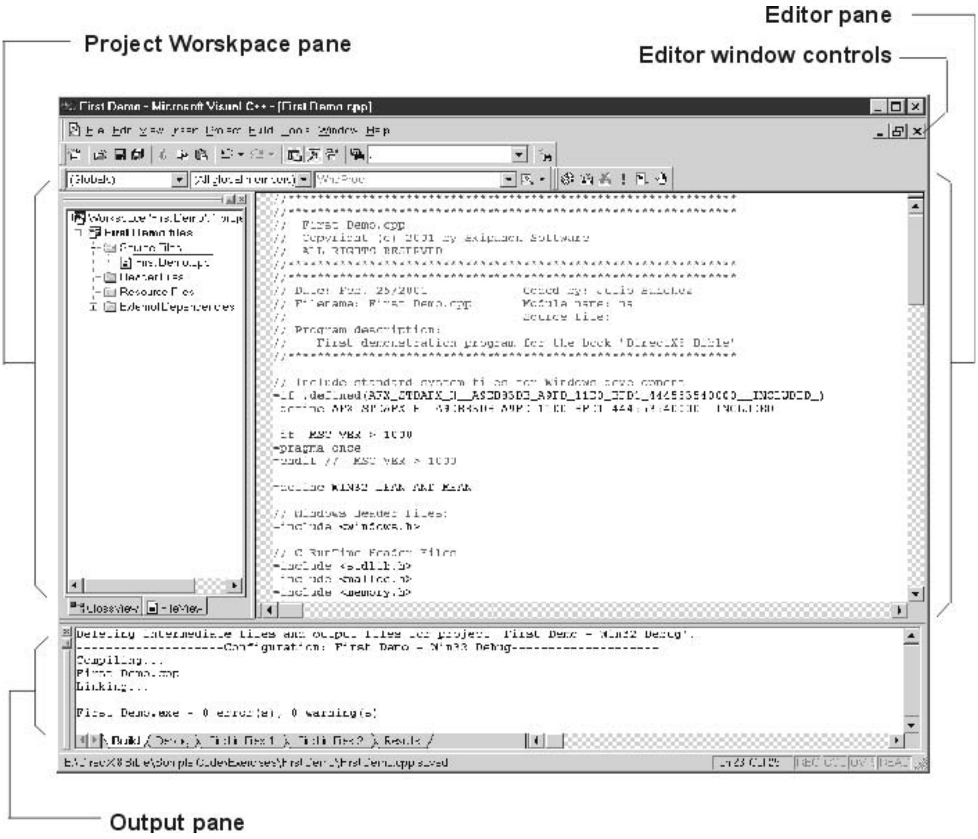
Figure 16-3 *Inserting an Existing Source File Into a Project*

The file Prog_zero.cpp now appears in the Program Zero Demo file list in Developer Studio workspace pane. It is also displayed in the Editor window.

The Developer Studio main screen is configurable by the user. Furthermore, the size of its display areas is determined by the system resolution. For this reason, it is impossible to depict a Developer Studio screen display that matches the one that every user will see. In the illustrations and screen snapshots throughout this book we have used a resolution of 1152-by-854 pixels in 16-bit color with large fonts. However, our screen format may not exactly match yours. Figure 16-4, on the following page, shows a full screen display of Developer Studio with the file Prog_zero.cpp loaded in the Editor area.

The Project Workspace pane of Developer Studio was introduced in Version 4.0. It has four possible views: Class View, File View, Info View, and Resource View. The Resource View is not visible in Figure 6-4. In order to display the source file in the editor pane, you must first select File View tab and double-click on the Prog_zero.cpp filename.

At this point, you can proceed to develop the new project using the renamed template file as the main source. The first step is to make sure that the development software is working correctly. To do this open the Developer Studio Build menu and click the Rebuild All command. Developer Studio compiles and builds your program, which is at this stage nothing more than the renamed template file. The results are shown in the Output area. If compilation and linking took place without error, reopen the Build menu and select the Execute Prog_zero.exe command button. If everything is in order, a do-nothing program executes in your system.



```
//****************************************************************
//****************************************************************
//  First Demo.cpp
//  Copyright (c) 2001 by [illegible] Software
//  ALL RIGHTS RESERVED
//****************************************************************
//****************************************************************
// Date: Feb. 25/2001            Coded by: Julio Sanchez
// Filename: First Demo.cpp      Module name: [illegible]
//                               Source file:
// Program description:
//    First demonstration program for the book 'DirectX8 Bible'
//****************************************************************

// Include standard system files for Windows apps support
#if !defined(AFX_STDAFX_H__A5CD95D3_A9TD_11D3_B7D1_444553540000__INCLUDED_)
#define AFX_STDAFX_H__A5CD95D3_A9TD_11D3_B7D1_444553540000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define WIN32_LEAN_AND_MEAN

// Windows header files:
#include <windows.h>

// C RunTime Header Files
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
```

```
Deleting intermediate files and output files for project 'First Demo - Win32 Debug'.
--------------------Configuration: First Demo - Win32 Debug--------------------
Compiling...
First Demo.cpp
Linking...

First Demo.exe - 0 error(s), 0 warning(s)
```

Figure 16-4 *Developer Studio Project Workspace, Editor, and Output Panes*

Now click the Save command on the File menu to make sure that all project files are saved on your hard drive.

16.2 Elements of a Windows Program

The template file Templ01.cpp, which we used and renamed in the previous example, is a bare bones windows program with no functionality except to display a window on the screen. Before proceeding to edit this template into a useful program, you should become acquainted with its fundamental elements. In this section, we take apart the template file Templ01.cpp for a detailed look into each of its components. The program contains two fundamental components: WinMain() and the Windows procedure.

16.2.1 WinMain()

All Windows GUI applications must have a WinMain() function. WinMain() is to a Windows GUI program what main() is to a DOS application. It is usually said that WinMain() is the program's entry point, but this is not exactly true. C/C++ compilers generate a startup code that calls WinMain(), so it is the startup code and not WinMain() that is actually called by Windows. The WinMain() header line is as follows:

```
 |------------------ Return type
 |     |------------- One of the standard calling conventions
 |     |              defined in windows.h
 |     |      |------- Function name
 |     |      |
 |     |      |      [         parameter list ....
--- ------ -------  ---------------------------------------------
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow) {
```

WINAPI is a macro defined in the windows.h header file which translates the function call to the appropriate calling convention. Recall that calling conventions refer to how the function arguments are placed in the stack at call time, and if the caller or the called routine is responsible for restoring stack integrity after the call. Microsoft Basic, FORTRAN, and Pascal push the parameters onto the stack in the same order in which they are declared. In these languages the stack must be restored by the caller. In C and C++, the parameters are pushed in reverse order, and the stack is restored automatically after the call returns. For historical reasons (and to take advantage of hardware features of the Intel processors) Windows requires the Pascal calling convention. In previous versions of Windows the calling convention for WinMain() was PASCAL or FAR PASCAL. You can still replace WINAPI for FAR PASCAL and the program will compile and link correctly, but the use of the WINAPI macro makes your program more portable.

Parameters

Most often parameters are passed to WinMain() by Windows, but some can be passed by whatever program executes your application. Your code can inspect these parameters to obtain information about the conditions in which the program executes. Four parameters are passed to WinMain():

- HINSTANCE is a handle-type identifier. The variable hInstance is an integer that identifies the instance of the program. Consider that in a multitasking environment there can be several copies (instances) of the same program running simultaneously. Windows sets this value and passes it to your code. Your program needs to access this parameter to enter it in the WNDCLASSEX structure; also when calling the CreateWindow() function. Because the handle to the instance is required outside of WinMain() by many functions of the Windows API, the template file stores it in a public variable, named pInstance. In general, the use of public variables is undesirable in Windows programming, but this case is one of the valid exceptions to the rule.
- The variable hPrevInstance is also of type HINSTANCE. This parameter is included in the call for compatibility with previous versions of Windows, which used a single copy of the code to run more than one program instance. In 16-bit Windows the first instance had a special role in the management of resources. Therefore, an application needed to know if it was the first instance. hPrevInstance held the handle of the previous instance. In Windows 95/98/NT this parameter is unused and its value is set to NULL.
- PSTR szCmdLine. This is a pointer to a string that contains the command tail entered by the user when the program is executed. It works only when the program name is entered from the DOS command line or from the Run dialog box. For this reason, it is rarely used by code.

- int iCmdShow. This parameter determines how the window is to be initially displayed. The program that executes your application (normally Windows) assigns a value to this parameter, as shown in Table 16-1.

Table 16-1

WinMain() Display Mode Parameters

VALUE	MEANING
SW_HIDE	Hides the window and activates another window
SW_MINIMIZE	Minimizes the specified window and activates the top-level window in the system's list
SW_RESTORE	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as SW_SHOWNORMAL)
SW_SHOW	Activates a window and displays it in its current size and position
SW_SHOWMAXIMIZED	Activates a window and displays it as a maximized window
SW_SHOWMINIMIZED	Activates a window and displays it as an icon
SW_SHOWMINNOACTIVE	Displays a window as an icon. The active window remains active
SW_SHOWNA	Displays a window in its current state. The active window remains active
SW_SHOWNOACTIVATE	Displays a window in its most recent size and position. The active window remains active
SW_SHOWNORMAL	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as SW_RESTORE)

16.2.2 Data Variables

The program file Templ01.cpp defines several variables. One of them, the handle to the program's main window, is defined globally. The other ones are local to WinMain() or the windows procedure. The variable defined globally is:

```
HWND        hwnd;
```

HWND is a 16-bit unsigned integer which serves as a handle to a window. The variable hwnd refers to the actual program window. The variable is initialized when we make the call to CreateWindow() service, described later in this section.

The variables defined in WinMain() are as follows:

```
   static char szClassName[] = "MainClass" ; // Class name
   MSG         msg ;
```

The first one is and array of char that shows the application's class name. In the template it is given the name MainClass, which you can replace for a more meaningful one. The application class name must be the same one used in the WNDCLASSEX structure.

MSG is a message-type structure of which msg is a variable. The MSG structure is defined in the Windows header files as follows:

```
typedef struct tagMSG {      // msg
    HWND   hwnd;      // Handle to window receiving message
    UINT   message;  // message number
    WPARAM wParam;   // Context-dependent additional information
    LPARAM lParam;   // about the message
    DWORD  time;     // Time at which message was posted
    POINT  pt;       // Cursor position when message was posted
} MSG;
```

The comments to the structure members show that the variable holds information that is important to the executing code. The values of the message variable are reloaded every time a new message is received.

16.2.3 WNDCLASSEX Structure

This structure is defined in the windows header files, as follows:

```
typedef struct tagWNDCLASSEX {
UINT       cbSize;
UINT       style;
WNDPROC    lpfnWndProc;
int        cbClsExtra;
int        cbWndExtra;
HINSTANCE  hInstance;
HICON      hIcon;
HCURSOR    hCursor;
HBRUSH     hbrBackground;
LPCSTR     lpszMenuName;
LPCSTR     lpszClassName;
HICON      hIconSm;
} WNDCLASSEX;
```

The WNDCLASSEX structure contains window class information. It is used with the RegisterClassEx() and GetClassInfoEx() functions. The structure is similar to the WNDCLASS structure used in 16-bit Windows. The differences between the two structures is that WNDCLASSEX has a cbSize member, which specifies the size of the structure, and the hIconSm member, which contains a handle to a small icon associated with the window class. In the template file Templ01.cpp the structure is declared and the variable initialized as follows:

```
// Creating a WNDCLASSEX structure
WNDCLASSEX  wndclass ;
wndclass.cbSize        = sizeof (WNDCLASSEX) ;
wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
wndclass.lpfnWndProc   = WndProc ;
wndclass.cbClsExtra    = 0 ;
wndclass.cbWndExtra    = 0 ;
wndclass.hInstance     = hInstance ;
wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
wndclass.hbrBackground = (HBRUSH) GetStockObject
                           (WHITE_BRUSH) ;
```

```
wndclass.lpszMenuName  = NULL ;
wndclass.lpszClassName = szClassName ;
wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;
```

The window class is a template that defines the characteristics of a particular window, such as the type of cursor and the background color. The class also specifies the address of the windows procedure that carries out the work for the window. The structures variables define the window class, as follows:

cbSize specifies the size, in bytes, of the structure. The member is set using the sizeof operator in the statement:

```
sizeof(WNDCLASSEX);
```

style specifies the class style or styles. Two or more styles can be combined by means of the C bitwise OR (|) operator. This member can be any combination of the values in Table 16-2.

Table 16-2

Summary of Window Class Styles

SYMBOLIC CONSTANT	ACTION
CS_BYTEALIGNCLIENT	Aligns the window's client area on the byte boundary (in the x direction) to enhance performance during drawing operations. This style affects the width of the window and its horizontal position on the display.
CS_BYTEALIGNWINDOW	Aligns a window on a byte boundary (in the direction) to enhance performance during operations that involve moving or sizing the window. This style affects the width of the window and its horizontal position on the display.
CS_CLASSDC	Allocates one device context to be shared by all windows in the class. Window classes are process specific; therefore, different threads can create windows of the same class.
CS_DBLCLKS	Sends double-click messages to the window procedure when the user double-clicks the mouse while the cursor is within a window belonging to the class.
CS_GLOBALCLASS	Allows an application to create a window of the class regardless of the value of the hInstance parameter passed to the CreateWindowEx() function. If you do not specify this style, the hInstance parameter passed to CreateWindowEx() function must be the same as the one passed to the RegisterClass() function.
CS_HREDRAW	Redraws the entire window if a movement or size adjustment changes the width of the client area.
CS_NOCLOSE	Disables the Close command on the System menu.

(continues)

Table 16-2

Summary of Window Class Styles (continued)

SYMBOLIC CONSTANT	ACTION
CS_OWNDC	Allocates a unique device context for each window in the class.
CS_PARENTDC	Specifies that child windows inherit their parent window's device context. Specifying CS_PARENTDC enhances an application's performance.
CS_SAVEBITS	Saves, as a bitmap, the portion of the screen image obscured by a window. Windows uses the saved bitmap to recreate the screen image when the window is removed. This style is useful for small windows (such as menus or dialog boxes) that are displayed briefly and then removed before other screen activity takes place.
CS_VREDRAW	Redraws the entire window if a movement or size adjustment changes the height of the client area.

Of these, the styles CS_HREDRAW and CS_VREDRAW are the ones most commonly used. They can be ORed to produce a window that is automatically redrawn if it is resized vertically or horizontally, as implemented in the Templ01.cpp code.

lpfnWndProc is a pointer to the window procedure, described later in this chapter. In the template Templ01.cpp it is initialized to the name of the Windows procedure, as follows:

```
wndclass.lpfnWndProc    = WndProc;
```

cbClsExtra is a count of the number of extra bytes to be allocated following the window-class structure. The operating system initializes the bytes to zero. In the template this member is set to zero.

cbWndExtra is a count of the number of extra bytes to allocate following the window instance. The operating system initializes the bytes to zero. In the template this member is set to zero.

hInstance is a handle to the instance of the window procedure.

hIcon is a handle to the class icon. If this member is NULL, an application must draw an icon whenever the user minimizes the application's window. In the template this member is initialized by calling the LoadIcon() function.

hCursor is a handle to the class cursor. If this member is NULL, an application must explicitly set the cursor shape whenever the mouse moves into the application's window. In the template this member is initialized by calling the LoadCursor() function.

hbrBackground is a background brush. This member can be a handle to the physical brush to be used for painting the background, or it can be a color value. If it is a color value, then it must be one of the standard system colors listed in Table 16-3.

Table 16-3

Common Windows Standard System Colors

SYMBOLIC CONSTANT	MEANING
COLOR_ACTIVEBORDER	Border color of the active window
COLOR_ACTIVECAPTION	Caption color of the active window
COLOR_APPWORKSPACE	Window background of MDI clients
COLOR_BACKGROUND	Desktop color
COLOR_BTNFACE	Face color for buttons
COLOR_BTNSHADOW	Shadow color for buttons
COLOR_BTNTEXT	Text color on buttons
COLOR_CAPTIONTEXT	Text color for captions, size boxes, and scroll bar boxes
COLOR_GRAYTEXT	Color for dissabled text
COLOR_HIGHLIGHT	Color of a selected item
COLOR_HIGHLIGHTTEXT	Text color of a selected item
COLOR_INACTIVEBORDER	Border color of inactive window
COLOR_INACTIVECAPTION	Caption color of an inactive window
COLOR_MENU	Background color of a menu
COLOR_MENUTEXT	Text color of a menu
COLOR_Scroll bar	Color of a scroll bar's gray area
COLOR_WINDOW	Background color of a window
COLOR_WINDOWFRAME	Frame color of a window
COLOR_WINDOWTEXT	Text color of a window

When this member is NULL, an application must paint its own background whenever it is required to paint its client area. In the template this member is initialized by calling the GetStockObject() function.

lpszMenuName is a pointer to a null-terminated character string that specifies the resource name of the class menu, as it appears in the resource file. If you use an integer to identify the menu, then you must use the MAKEINTRESOURCE macro. If this member is NULL, the windows belonging to this class have no default menu, as is the case in the template file.

lpszClassName is a pointer to a null-terminated string or it is an atom. If this parameter is an atom, it must be a global atom created by a previous call to the GlobalAddAtom() function. The atom, a 16-bit value, must be in the low-order word of lpszClassName; the high-order word must be zero. If lpszClassName is a string, it specifies the window class name. In Templ01.cpp this member is set to the szClassName[] array.

In Windows 95/98 hIconSm is a handle to a small icon that is associated with the window class. This is the icon shown in dialog boxes that list filenames and by Windows Explorer. A Windows 95/98 application can use a predefined icon in this case, using the LoadIcon function with the same parameters as for the hIcon member. In Windows NT this member is not used and should be set to NULL. Windows 95/98 applications that set the small icon to NULL still have the default small icon displayed on the task bar.

In most cases it is better to create both the large and the small icon than to let Windows create the small one from the large bitmap. Later in this chapter we describe how to create both icons as a program resource and how to make these resources available to the application.

Contrary to what has sometimes been stated, the LoadIcon() function cannot be used to load both large and small icons from the same resource. For example, if the icon resource is named IDI_ICON1, and we proceed as follows:

```
wndclass.hicon     = LoadIcon (hInstance,
                       MAKEINTRESOURCE(IDI_ICON1);
.
.
.
wndclass.hiconSm  = LoadIcon (hInstance,
                     MAKEINTRESOURCE(IDI_ICON1);
```

the result is that the large icon is loaded from the resource file, but not the small one. This happens even if the resource file contains both images. Instead, you must use the LoadImage() function, as follows:

```
wndclass.hIcon         = (HICON)LoadImage(hInstance,
                           MAKEINTRESOURCE(IDI_ICON1),
                           IMAGE_ICON,        // Type
                           32, 32,            // Pixel size
                           LR_DEFAULTCOLOR) ;
.
.
.
wndclass.hIconSm       = (HICON)LoadImage(hInstance,
                           MAKEINTRESOURCE(IDI_ICON1),
                           IMAGE_ICON,         // Type
                           16, 16,             // Pixel size
                           LR_DEFAULTCOLOR) ;
```

Now both the large and the small icon resources are loaded correctly and are used as required. Also notice that the value returned by LoadImage() is typecast into HICON. This manipulation became necessary starting with version 6 of Microsoft Visual C++ due to changes made to the compiler in order to improve compatibility with the ANSI C++ standard.

16.2.4 Registering the Windows Class

Once your code has declared the WNDCLASSEX structure and initialized its member variables, it has defined a window class that encompasses all the structure attributes. The most important ones are the window style (wndclass.style), the pointer to the Windows procedure (wndclass.lpfnWndProc), and the window class name (wndclass. lpszClassName). The RegisterClassEx() function is used to notify Windows of the existence of a particular window class, as defined in the WNDCLASSEX structure variable. The address-of operator is used to reference the location of the specific structure variable, as in the following statement:

```
RegisterClassEx (&wndclass) ;
```

The RegisterClassEx() function returns an atom (16-bit integer). This value is non-zero if the class is successfully registered. Code should check for a successful registration since you cannot create a window otherwise. The following construct ensures that execution does not proceed if the function fails.

```
    if(!RegisterClassEx (&wndclass))
      return(0);
```

This coding style is the one used in the template Templ01.cpp.

16.2.5 Creating the Window

A window class is a general classification. Other data must be provided at the time the actual windows is created. The CreateWindowEx() function receives the additional information as parameters. CreateWindowEx() is a Windows 95 version of the CreateWindow() function. The only difference between them is that the new version supports an extended window style passed as its first parameter.

The CreateWindowEx() function is very rich in arguments, many of which apply only to special windows styles. For example, buttons, combo boxes, list boxes, edit boxes, and static controls can all be created with a CreateWindowEx() call. At this time, we refer only to the most important function parameters that relate to the a program's main window.

```
In the file Templ01.cpp the call to CreateWindowEx is coded as follows:
   hwnd = CreateWindowEx (
          WS_EX_LEFT,                // left aligned (default)
          szClassName,               // pointer to class name
          "Window Caption",          // window caption (title bar)
          WS_OVERLAPPEDWINDOW,       // window style
          CW_USEDEFAULT,             // initial x position
          CW_USEDEFAULT,             // initial y position
          CW_USEDEFAULT,             // initial x size
          CW_USEDEFAULT,             // initial y size
          NULL,                      // parent window handle
          NULL,                      // window menu handle
          hInstance,                 // program instance handle
          NULL) ;                   // creation parameters
```

The first parameter passed to the CreateWindowEx() function is the extended window style introduced in the Win32 API. The one used in the file Templ01.cpp, WS_EX_LEFT, acts as a placeholder for others that you may want to select, since it is actually the default value. Table 16-4 lists some of the most common extended styles.

The second parameter passed to the CreateWindowEx() function call is either a pointer to a string with the name of the window type, a string enclosed in double quotation marks, or a predefined name for a control class.

In the template file, szClassName is a pointer to the string defined at the start of WinMain(), with the text "MainClass." You can edit this string in your own applications so that the class name is more meaningful. For example, if you were coding an editor program you may rename the application class as "TextEdClass." However, this is merely a name used by Windows to associate a window with its class; it is not displayed as a caption or used otherwise.

Table 16-4

Most Commonly Used Windows Extended Styles

SYMBOLIC CONSTANT	MEANING
WS_EX_ACCEPTFILES	The window created with this style accepts drag-drop files.
WS_EX_APPWINDOW	A top-level window is forced onto the application taskbar when the window is minimized.
WS_EX_CLIENTEDGE	Window has a border with a sunken edge.
WS_EX_CONTEXTHELP	The title bar includes a question mark. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, it receives a WM_HELP message.
WS_EX_CONTROLPARENT	Allows the user to navigate among the child windows of the window by using the TAB key.
WS_EX_DLGMODALFRAME	Window that has a double border. Optionally the window can be created with a title bar by specifying the WS_CAPTION style in the dwStyle parameter.
WS_EX_LEFT	Window has generic "left-aligned" properties. This is the default.
WS_EX_MDICHILD	Creates an MDI child window.
WS_EX_NOPARENTNOTIFY	Specifies that a child window created with this style does not send the WM_PARENTNOTIFY message to its parent window when it is created or destroyed.
WS_EX_OVERLAPPEDWINDOW	Combines the WS_EX_CLIENTEDGE and WS_EX_WINDOWEDGE styles.
WS_EX_PALETTEWINDOW	Combines the WS_EX_WINDOWEDGE, WS_EX_TOOLWINDOW, and WS_EX_TOPMOST styles.
WS_EX_RIGHTSCROLLBAR	Scroll bar Vertical scroll bar (if present) is to the right of the client area. This is the default.
WS_EX_STATICEDGE	Creates a window with a three-dimensional border style intended to be used for items that do not accept user input.
WS_EX_TOOLWINDOW	Creates a tool window. This type of window is intended to be used as a floating toolbar.
WS_EX_TOPMOST	A window created with this style should be placed above all non-topmost windows and should stay above them, even when the window is deactivated.
WS_EX_TRANSPARENT	A window created with this style is transparent. That is, any windows that are beneath it are not obscured by it.
WS_EX_WINDOWEDGE	Window has a border with a raised edge.

Control classes can also be used as a window class name. These classes are the symbolic constants BUTTON, Combo box, EDIT, List box, MDICLIENT, Scroll bar, and STATIC.

The third parameter can be a pointer to a string or a string enclosed in double quotation marks entered directly as a parameter. In either case, this string is used as the caption to the program window and is displayed in the program's title bar. Often this caption coincides with the name of the program. You should edit this string to suit your own program.

The fourth parameter is the window style. Over 25 styles are defined as symbolic constants. The most used ones are listed in Table 16-5.

Table 16-5

Window Styles

SYMBOLIC CONSTANT	MEANING
WS_BORDER	Window that has a thin-line border.
WS_CAPTION	Window that has a title bar (includes the WS_BORDER style).
WS_CHILD	Child window. This style cannot be used with the WS_POPUP style.
WS_CLIPCHILDREN	Excludes the area occupied by child windows when drawing occurs within the parent window.
WS_CLIPSIBLINGS	Clips child windows relative to each other. When a particular child window receives a WM_PAINT message, this style clips all other overlapping child windows out of the region of the child window to be updated. If WS_CLIPSIBLINGS is not specified and child windows overlap, it is possible to draw within the client area of a neighboring child window.
WS_DISABLED	Window is initially disabled. A disabled window cannot receive input from the user.
WS_DLGFRAME	Window has a border of a style typically used with dialog boxes. The window does not have a title bar.
WS_HSCROLL	Window that has a horizontal scroll bar.
WS_ICONIC	Window is initially minimized. Same as the WS_MINIMIZE style.
WS_MAXIMIZE	Window is initially maximized.
WS_MAXIMIZEBOX	Window that has a Maximize button. Cannot be combined with the WS_EX_CONTEXTHELP style.
WS_MINIMIZE	Window is initially minimized. Same as the WS_ICONIC style.
WS_MINIMIZEBOX	Window has a Minimize button. Cannot be combined with the WS_EX_CONTEXTHELP style.
WS_OVERLAPPED	Overlapped window. Has a title bar and a border.
WS_OVERLAPPEDWINDOW	Overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_TILEDWINDOW style.
WS_POPUP	Pop-up window. Cannot be used with the WS_CHILD style.

(continues)

Table 16-5

Window Styles (continued)

SYMBOLIC CONSTANT	MEANING
WS_POPUPWINDOW	Pop-up window with WS_BORDER, WS_POPUP, and WS_SYSMENU styles. The WS_CAPTION and WS_POPUPWINDOW styles must be combined to make the System menu visible.
WS_SIZEBOX	Window that has a sizing border. Same as the WS_THICKFRAME style.
WS_SYSMENU	Window that has a System-menu box in its title bar. The WS_CAPTION style must also be specified.
WS_TILED	Overlapped window. Has a title bar and a border. Same as the WS_OVERLAPPED style.
WS_TILEDWINDOW	Overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_OVERLAPPEDWINDOW style
WS_VISIBLE	Window is initially visible.
WS_VSCROLL	Window that has a vertical scroll bar.

The style defined in the template file Templ01.ccp is WS_OVERLAPPEDWINDOW. This style creates a window that has the styles WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX. It is the most common style of windows.

The fifth parameter to the CreateWindowEx() service defines the initial horizontal position of the window. The value CS_USERDEFAULT (0x80000000) determines the use of the default position. The template file uses the same CS_USERDEFAULT symbolic constant for the y position, and the windows x and y size.

Parameters nine and ten are set to NULL since this window has no parent and no default menu.

The eleventh parameter, hInstance, is a the handle to the instance that was passed to WinMain() by Windows.

The last entry, called the creation parameters, can be used to pass data to a program. A CREATESTRUCT-type structure is used to store the initialization parameters passed to the windows procedure of an application. The data can include an instance handle, a new menu, the window's size and location, the style, the window's name and class name, and the extended style. Since no creation parameters are passed, the field is set to NULL.

The CreateWindowEx() function returns a handle to the window of type HWND. The template file Templ01.cpp stores this handle in a global variable named hwnd. The reason for this is that many functions in the Windows API require this handle. By storing it in a global variable we make it visible throughout the code.

If CreateWindowsEx() fails, it returns NULL. Code in WinMain() can test for this error condition with the statement:

```
    if(!hwnd)
      return(0);
```

We do not use this test in the template file Templ01.cpp because it is usually not necessary. If WinMain() fails, you may use the debugger to inspect the value of hwnd after CreateWindowEx() in order to make sure that a valid handle was returned.

16.2.6 Displaying the Window

CreateWindowEx() creates the window internally but does not display it. To display the window your code must call two other functions: ShowWindow() and UpdateWindow(). ShowWindow() sets the window's show state and UpdateWindow() updates the window's client area. In the case of the program's main window, ShowWindow() must be called once, using as a parameter the iCmdShow value passed by Windows to WinMain(). In the template file the call is coded as follows:

```
    ShowWindow (hwnd, iCmdShow) ;
```

The first parameter to ShowWindow() is the handle to the window returned by CreateWindowEx(). The second parameter is the window's display mode parameter, which determines how the window must be initially displayed. The display mode parameters are listed in Table 16-1, but in this first call to ShowWindow() you must use the value received by WinMain().

UpdateWindow() actually instructs the window to paint itself by sending a WM_PAINT message to the windows procedure. The processing of the WM_PAINT message is described later in this chapter. The actual code in the template file is as follows:

```
    UpdateWindow (hwnd) ;
```

If all has gone well, at this point your program is displayed on the screen. It is now time to implement the message passing mechanisms that are at the heart of event-driven programming.

16.2.7 The Message Loop

In an event-driven environment there can be no guarantee that messages are processed faster than they originate. For this reason Windows maintains two message queues. The first type of queue, called the system queue, is used to store messages that originate in hardware devices, such as the keyboard and the mouse. In addition, every thread of execution has its own message queue. The message handling mechanism can be described with a simplified example: when a keyboard event occurs, the device driver software places a message in the system queue. Windows uses information about the input focus to decide which thread should handle the message. It then moves the message from the system queue into the corresponding thread queue.

A simple block of code, called the message loop, removes a messages from the thread queue and dispatches it to the function or routine which must handle it. When a special message is received, the message loop terminates, and so does the thread. The message loop in Templ01.cpp is coded as follows:

```
   while (GetMessage (&msg, NULL, 0, 0))
```

```
        {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
        }
    return msg.wParam ;
```

The while statement calls the function GetMessage(). The first parameter to GetMessage() is a variable of the structure type MSG, described in Section 16.2.2. The structure variable is filled with information about the message in the queue, if there is one. If no message is waiting in the queue, Windows suspends the application and assigns its time slice to other threads of execution. In an event-driven environment, programs act only in response to events. No event, no message, no action.

The second parameter to GetMessage() is the handle to a window for which to retrieve a message. Most applications set this parameter to NULL, which signals that all messages for windows that belong to the application making the call should be retrieved. The third and the fourth parameter to GetMessage() are the lowest and the highest message numbers to be retrieved. Threads that only retrieve messages within a particular range can use these parameters as a filter. When the special value 0 is assigned to both of these parameters (as is the case in our message loop) then no filtering is performed and all messages are passed to the application.

There are two functions inside the message loop. TranslateMessage() is a keyboard processing function that converts keystrokes into characters. The characters are then posted to the message queue. If the message is not a keystroke that needs translation, then no special action is taken. The DispatchMessage() function sends the message to the windows procedure, where it is further processed and either acted upon, or ignored. The windows procedure is discussed in the following section. GetMessage() returns 0 when a message labeled WM_QUIT is received. This signals the end of the message loop; at this point execution returns from WinMain(), and the application terminates.

16.3 The Window Procedure

At this moment in a program's execution the window class has been registered, the window has been created and displayed, and all messages are being routed to your code. The windows procedure, sometimes called the window function, is where you write code to handle the messages received from the message loop. It is in the windows procedure where you respond to the events that pertain to your program.

Every window must have a window procedure. Although the name WinProc() is commonly used, you can use any other name for the windows procedure provided that it appears in the procedure header, the prototype, in the corresponding entry of the WNDCLASSEX structure, and that it does not conflict with another name in your application. Also, a Windows program can have more than one windows procedure. The program's main window is usually registered in WinMain() but others can be registered elsewhere in an application. Here again, each windows procedure corresponds to a window class, has its own WNDCLASSEX structure, as well as a unique name.

In the template, the windows procedure is coded as follows:

```
    |------------------------ Return type, equivalent to a long type
    |       |---------------- Same as FAR PASCAL calling convention.
    |       |                 Used in windows and dialog procedures.
    |       |        |------- Procedure name
    |       |        |     [ parameter list ...                ]
------- -------- -------   ----------------------------------------
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                          LPARAM lParam) {
```

The windows procedure is of callback type. The CALLBACK symbol was first introduced in Windows 3.1 and is equivalent to FAR PASCAL, and also to WINAPI, since all of them currently correspond to the __stdcall calling convention. Although it is possible to substitute __stdcall for CALLBACK in the function header, it is not advisable, since this could compromise the application's portatibility to other platforms or to future versions of the operating system.

The return value of a windows procedure is of type LRESULT, which is a 32-bit integer. The actual value depends on the message, but it is rarely used by application code. However, there are a few messages for which the windows procedure is expected to return a specific value. It is a good idea to check the Windows documentation when in doubt.

16.3.1 Windows Procedure Parameters

The four parameters to the windows procedure are the first four fields in the MSG structure. The MSG structure is discussed earlier in this chapter. Since the windows procedure is called by Windows, the parameters are provided by the operating system at call time, as follows:

- hwnd is the handle to the window receiving the message. This is the same handle returned by CreateWindow().
- iMsg is a 32-bit unsigned integer (UINT) that identifies each particular message. The constants for the various messages are defined in the windows header files. They all start with the letters WM_, which stand for window message.
- wParam and lParam are called the message parameters. They provide additional information about the message. Both values are specific to each message.

The last two members of the message structure, which correspond to the message's time of posting and cursor position, are not passed to the windows procedure. However, application code can use the functions GetMessageTime() and GetMessagePos() to retrieve these values.

16.3.2 Windows Procedure Variables

The implementation of the windows procedure in Templ01.cpp starts by declaring a scalar of type HDC and two structure variables of type HWND and MSG respectively. The variables are as follows:

- hdc is a handle to the device context. A device context is a data structure maintained by Windows which is used in defining the graphics objects and their attributes, as well as their associated graphics modes. Devices such as the video display, printers, and plotters, must be accessed through a handle to their device contexts, which is obtained from Windows.

- ps is a PAINTSTRUCT variable. The structure is defined by Windows as follows:

```
 typedef struct tagPAINTSTRUCT {
   HDC    hdc;           // identifies display device
   BOOL   fErase;        // not-zero if background must be erased
   RECT   rcPaint;       // Rectangle structure in which painting is
                         // requested
   BOOL   fRestore;      // RESERVED
   BOOL   fIncUpdate;    // RESERVED
   BYTE   rgbReserved[32]; // RESERVED
} PAINTSTRUCT;
```

  The structure contains information that is used by the application to paint its own client area.

- rect is a RECT structure variable. The RECT structure is also defined by Windows:

```
 typdef struct _RECT {
   LONG   left;      // x coordinate of upper-left corner
   LONG   top;       // y of upper-left corner
   LONG   right;     // x coordinate of bottom-right corner
   LONG   bottom;    // y of bottom-right
 } RECT;
```

The RECT structure is used to define the corners of a rectangle, in this case of the application's display area, which is also called the client area.

16.3.3 Message Processing

The windows procedure receives and processes messages. The message can originate as follows:

- Some messages are dispatched by WinMain(). In this group are the messages placed in the thread's message queue by the DispatchMessage() function in the message loop. Messages handled in this manner are referred to as queued messages. Queued messages originate in keystrokes, mouse movements, mouse button clicks, the system timer, and in orders to redraw the window.
- All other messages come directly from Windows. These are called nonqueued messages.

The windows procedure examines each message, queue or nonqueued, and either takes action or passes the message back for default processing. In the template file Templ01.cpp the message processing skeleton is coded as follows:

```
switch (iMsg)
    {
// Windows message processing
     // Preliminary operations
     case WM_CREATE:
          return (0);

     // Redraw window
     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;
          GetClientRect (hwnd, &rect) ;
     // Initial display operations here
          EndPaint (hwnd, &ps) ;
          return 0 ;
     // End of program execution
     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
```

Messages are identified by uppercase symbolic constants that start with the characters WM_ (window message). Over two hundred message constants are defined in Windows. Three messages are processed in the template file: WM_CREATE, WM_PAINT and WM_DESTROY.

When the Windows procedure processes a message it must return 0. If it does not process a particular message, then the function DefWindowsProc() is called to provide a default action.

WM_CREATE Message Processing

The WM_CREATE message is sent to an application as a result of the CreateWindowEx() function in WinMain(). This message gives the application a chance to perform preliminary initialization, such as displaying a greeting screen, or playing a sound file. In the template, the WM_CREATE processing routine does nothing. It serves as a placeholder where the programmer can inserts the appropriate code.

WM_PAINT Message Processing

The WM_PAINT message informs the program that all or part of the client window must be repainted. This happens when the user minimizes, overlaps, or resizes the client window area. Recall that the style of the program's main window is defined in the template with the statement:

```
    wndclass.style    = CS_HREDRAW | CS_VREDRAW ;
```

This style determines that the screen is redrawn if it is resized vertically or horizontally.

In WM_PAINT, processing begins with the BeginPaint() function. BeginPaint() serves to prepare the window for a paint operation by filling a variable of type PAINTSTRUCT, previously discussed. The call to BeginPaint() requires the hwnd variable, which is the handle to the window that is to be painted. Also a variable ps, of a structure of type PAINTSTRUCT, which is filled by the call. During BeginPaint() Windows erases the background using the currently defined brush.

The call to GetClientRect() requires two parameters. The first one is the handle to the window (hwnd), which is passed to the windows procedure as a parameter. In the template file this value is also stored in a public variable. The second parameter is the address of a structure variable of type RECT, where Windows places the coordinates of the rectangle that defines the client area. The left and top values are always set to zero.

Processing ends with EndPaint(). EndPaint() notifies Windows that the paint operation has concluded. The parameters passed to EndPaint() are the same ones passed to BeginPaint(): the handle to the window and the address of the structure variable of type PAINTSTRUCT.

WM_DESTROY Message Processing

The WM_DESTROY message is received by the windows procedure when the user takes an action to destroy the window, usually clicking the Close button or selecting the Close or Exit commands from the File or the System menus. The standard processing performed in WM_DESTROY is:

```
PostQuitMessage (0) ;
```

The PostQuitMessage() function inserts a WM_QUIT message in the message queue, thus terminating the GetMessage loop and ending the program.

16.3.4 The Default Windows Procedure

The code in the template file contains a return statement for each of the messages that it handles. For example:

```
case WM_PAINT :
     hdc = BeginPaint (hwnd, &ps) ;
     GetClientRect (hwnd, &rect) ;
// Initial display operations here
     EndPaint (hwnd, &ps) ;
     return 0 ;
```

The last statement in this routine returns a value of zero to Windows. The Windows documentation states that zero must be returned when an application processes the WM_PAINT message. Some Windows messages, not many, require a return value other than zero.

Many of the messages received from Windows, or retrieved from the message queue, are of no interest to your application. In this case, code must provide a default action for those messages that it does not handle. Windows contains a function, named DefWindowsProc(), that ensures this default action. DefWindowsProc() provides specific processing for those messages that require it, thus implementing a default behavior. For those messages that can be ignored, DefWindowsProc() returns zero. Your application uses the return value of DefWindowsProc() as its own return value from the Windows procedure. This action is coded as follows in the template file:

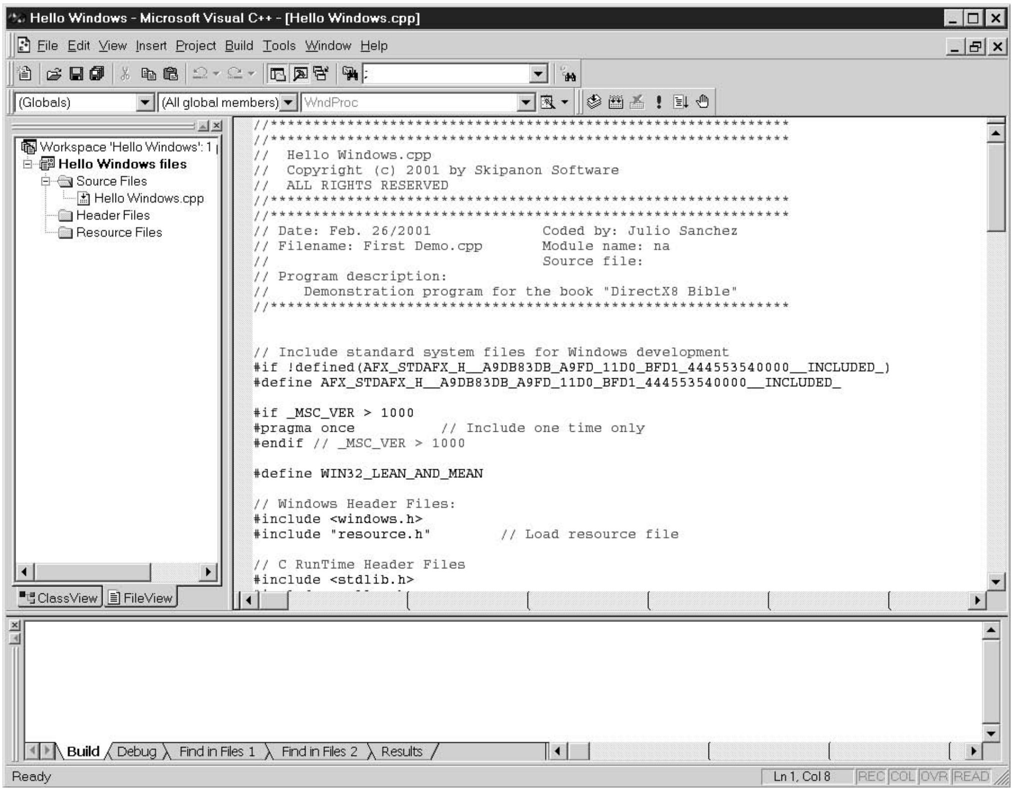
```
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
```

The parameters passed to DefWindowsProc() are the same message parameters received by your windows procedure from the operating system.

16.4 The WinHello Program

In the first walkthrough, at the beginning of this chapter, we used the template file Templ01.cpp to create a new project, which we named Program Zero Demo. Program Zero Demo resulted in a do-nothing program since no modifications were made to the template file at that time. In the present walkthrough we proceed to make modifications to the template file in order to create a Windows program different from the template. This project, which we named Hello Windows, is a Windows version of the classic "Hello World" program.

We first create a new project and use the template file Templ01.cpp as the source code base for it. In order to do this we must follow all the steps in the first walkthrough, except that the project name is now Hello Windows and the name template file Templ01.cpp is copied and renamed WinHello.cpp. After you have finished all the steps in the walkthrough you will have a project named Hello Windows and the source file named WinHello.cpp listed in the Project Workspace and displayed in the Editor Window. After the source file is renamed, you should edit the header block to reflect the file's new name and the program's purpose. Figure 16-5 shows the Developer Studio screen at this point.



```
//**************************************************************
//**************************************************************
//  Hello Windows.cpp
//  Copyright (c) 2001 by Skipanon Software
//  ALL RIGHTS RESERVED
//**************************************************************
//**************************************************************
// Date: Feb. 26/2001             Coded by: Julio Sanchez
// Filename: First Demo.cpp       Module name: na
//                                Source file:
// Program description:
//    Demonstration program for the book "DirectX8 Bible"
//**************************************************************


// Include standard system files for Windows development
#if !defined(AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_)
#define AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_

#if _MSC_VER > 1000
#pragma once          // Include one time only
#endif // _MSC_VER > 1000

#define WIN32_LEAN_AND_MEAN

// Windows Header Files:
#include <windows.h>
#include "resource.h"        // Load resource file

// C RunTime Header Files
#include <stdlib.h>
```

Figure 16-5 *The Hello Windows Project and Source File*

The project Hello Windows, which we are about to code, has the following features:

- The caption displayed on the program title bar is changed to "Hello Windows."
- When the program executes it displays a greeting message on the center of its client area.

- The program now contains a customized icon. A small version of the icon is displayed in the title bar and a larger one is used when the program's executable is represented by a shortcut on the Windows desktop.

Once you have created the project named Hello Windows and included in it the source file WinHello.cpp, you are ready to start making modifications to the source and inserting new elements into the project.

16.4.1 Modifying the Program Caption

The first modification that we make to the source is to change the caption that is displayed on the title bar when the program executes. This requires editing the third parameter passed to the CreateWindowsEx() function in WinMain(). The parameter now reads "Hello Windows." Throughout this book we use the project's name, or a variation of it, as the title bar caption. Our reason for this is to make it easy to find the project files from a screen snapshot of the executable.

16.4.2 Displaying Text in the Client Area

The second modification requires entering a call to the DrawText() API function in the case WM_PAINT processing routine. The routine now is:

```
case WM_PAINT :
     hdc = BeginPaint (hwnd, &ps) ;
     GetClientRect (hwnd, &rect) ;

     // Display message in the client area
        DrawText (hdc,
                 "Hello World from Windows",
                 -1,
                 &rect,
                 DT_SINGLELINE | DT_CENTER | DT_VCENTER);

      EndPaint (hwnd, &ps) ;
      return 0 ;
```

The call to DrawText() requires five parameters. When calls require several parameters, we can improve the readability of the source by devoting a separate text line to each parameter, or to several associated parameters, as in the previous listing.

- The first parameter to DrawText() is the handle to the device context. This value was returned by the call to BeginPaint(), described previously in this chapter.
- The second parameter to DrawText() points to the string to be displayed. The string can also be enclosed in double quotation marks, as in the previous listing.
- The third parameter is –1 if the string defined in the second parameter terminates in NULL. If not, then the third parameter is the count of the number of characters in the string.
- The fourth parameter is the address of a structure of type RECT which contains the logical coordinates of the area in which the string is to be displayed. The call to GetClientRect(), made in the WM_PAINT message intercept, filled the members of the rect structure variable.

- The fifth parameter are the text formatting options. Table 16-6 lists the most used of these controls.

Table 16-6

Symbolic Constant in DrawText() Function

SYMBOLIC CONSTANT	MEANING
DT_BOTTOM	Bottom-justifies text. Must be combined with DT_SINGLELINE.
DT_CALCRECT	This constant is used to determine the width and height of the rectangle. If there are multiple lines of text, DrawText uses the width of the rectangle in the RECT structure variable supplied in the call and extends the base of the rectangle to bound the last line of text. If there is only one line of text, DrawText modifies the right side of the rectangle so that Text is not drawn.
DT_CENTER	Centers text horizontally.
DT_EXPANDTABS	Expands tab characters. The default number of characters per tab is eight.
DT_EXTERNALLEADING	Includes the font external leading in line height. Normally, it is not included.
DT_LEFT	Aligns text to the left.
DT_NOCLIP	Draws without clipping. The function executes somewhat faster when DT_NOCLIP is used.
DT_NOPREFIX	DrawText interprets the control character & as a command to underscore the character that follows. The control characters && prints a single &. By specifying DT_NOPREFIX, this processing is turned off.
DT_RIGHT	Aligns text to the right.
DT_SINGLELINE	Displays text on a single line only. Carriage returns and linefeeds are ignored.
DT_TOP	Top-justifies text (single line only).
DT_VCENTER	Centers text vertically (single line only).
DT_WORDBREAK	Breaks words. Lines are automatically broken between words if a word extends past the edge of the rectangle specified by the lpRect parameter. A carriage return-linefeed sequence also breaks the line.

16.4.3 Creating a Program Resource

The last customization that you have to perform on the template file is to create two customized icons, which are associated with the program window. The icons correspond to the hIcon and hIconSm members of the WNDCLASSEX structure described previously and listed in Appendix A. hIcon is the window's standard icon. Its default size is 32-by-32 pixels, although Windows automatically resizes this icon as required. The standard icon is used on the Windows desktop when a shortcut is created and in some file listing modes of utilities like Windows Explorer. The small icon is 16-by-16 pixels, which makes it one-fourth the size of the large one. This is the icon shown in dialog boxes that list filenames, by Windows Explorer, and in the program's title bar. Windows NT uses a scaled version of the standard icon when a smaller one is required.

An icon is a resource. Resources are stored in read-only, binary data files, that the application can access by means of a handle. We introduce icons at this time because other program resources such as cursors, menus, dialog boxes, bitmaps, and fonts are handled similarly. The icons that we create in this walkthrough are considered an application-defined resource.

The most convenient way of creating and using resources is to take advantage of the facilities in the development environment. Visual C++ provides several resource editors, and Developer Studio facilitates the creation and manipulation of the support files required for using resources. Graphics programmers often want to retain the highest possible control over their code; however, the use of these facilities in creating and managing resources does not compromise this principle. The files created by the development environment are both visible and editable. As you gain confidence and knowledge about them you can progressively take over some or all of the operations performed by the development software. In this book we sometimes let the development environment generate one or more of the program files and then proceed to edit them so that it better suits our purpose.

The convenience of using the automated functions of the development environment is made evident by the fact that a simple resource often requires several software elements. For example, a program icon requires the following components:

- A bitmap that graphically encodes the icon. If the operating system and the application supports the small icon, then two bitmaps are required.
- A script file (also called a resource definition file) that lists all the resources in the application and may describe some of them in detail. The resource script can also reference other files and may include comments and preprocessor directives. The resource compiler (RC.EXE) compiles the script file into a binary file with the extension .RES. This binary file is referenced at link time. The resource file has the extension .RC.
- The script file uses a resource header file, with the default filename "resource.h", which contains preprocessor directives related to the resources used by the application. The application must reference this file with an #include statement.

16.4.4 Creating the Icon Bitmap

Developer Studio provides support for the following resources: dialog boxes, menus, cursors, icons, bitmaps, toolbars, accelerators, string tables, and version controls.

Each resource has either a graphics editor or a wizard that helps create the resource. In this discussion we refer to either one of them as a resource editor.

Resource editors can be activated by clicking on the Resource command in the Insert menu. At this time Developer Studio displays a dialog box with an entry for each type of resource. Alternatively, you can access the resource editors faster by displaying the Resource toolbar. In Visual C++ 4 and later this is accomplished by clicking on the Toolbars command in the View menu, and then selecting the checkbox for the Resource option. In Versions 5 and 6 select the Customize command in the Tools menu, open the Toolbars tab in the Customize dialog box and select the checkbox for the Resource option. The Graphics and Colors boxes should also be checked to display the normal controls in the resource editors. The resulting toolbar is identical in both cases. Once the Resource toolbar is displayed, you can drag it into the toolbar area or to any other convenient screen location. The Insert Resource dialog screen and the resource toolbar are shown in Figure 16-6.

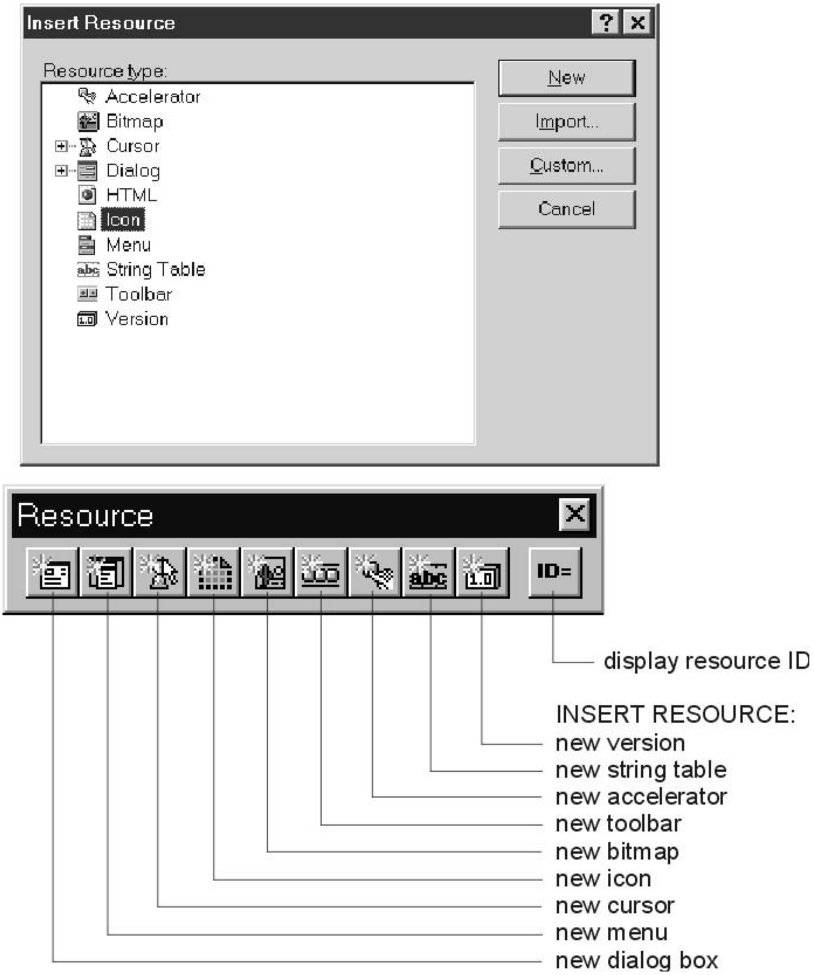


Figure 16-6 *Developer Studio Insert Resource Dialog Screen and Toolbar*

You can activate the icon editor either by selecting the icon option in the Resource dialog box or by clicking the appropriate button on the toolbar. The icon editor is simple to use and serves well in most cases. It allows creating the bitmap for several sizes of icons. Although the interface to the icon editor is simple, it is also powerful and flexible. You should experiment with the icon editor, as well as with the other resource editors, until you have mastered all their options and modes. Figure 16-7 shows the icon editor in Developer Studio.

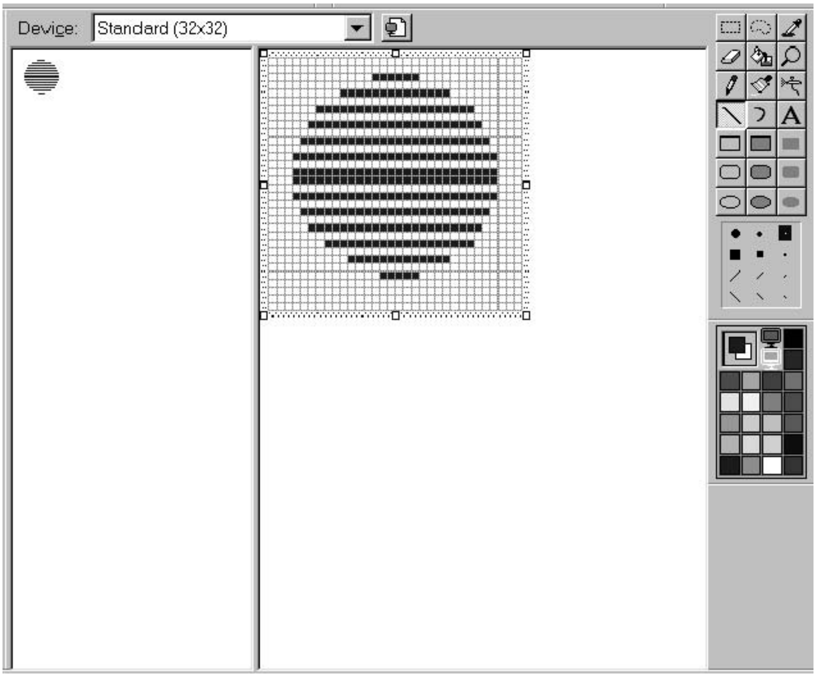


Figure 16-7 *Creating An Icon Resource with Developer Studio Icon Editor*

The toolbar on the right of the icon editor is similar to the one used in the Windows Paint utility and in other popular graphics programs. There are several tools that allow drawing lines, curves, and geometrical figures in outline or filled form. Also, there is a palette box from which colors for foreground and background can be selected.

Developer Studio makes possible the creation of a large and a small icon in the same resource. To request the small icon, click on the New Device Image button and then select the 16-by-16 icon. The two icons, 32 by 32 pixels and 16 by 16 pixels, can be developed alternatively by selecting one of them in the Open Device Image scroll box in the icon editor. Windows automatically uses the large and the small icon as required.

In the WinHello program the WNDCLASSEX structure is edited to support user-created large and small icons, as follows:

```
//   The program icon is loaded in the hIcon and hIconSm
//   structure members
WNDCLASSEX  wndclass ;
wndclass.hIcon          = (HICON) LoadImage(hInstance,
                           MAKEINTRESOURCE(IDI_ICON1),
                           IMAGE_ICON,       // Type
                           32, 32,           // Pixel size
                           LR_DEFAULTCOLOR) ;
.
.
.
wndclass.hIconSm        = (HICON) LoadImage(hInstance,
                           MAKEINTRESOURCE(IDI_ICON1),
                           IMAGE_ICON,        // Type
                           16, 16,            // Pixel size
                           LR_DEFAULTCOLOR) ;
```

The MAKEINTRESOURCE macro is used to convert an integer value into a resource. Although resources can also be referenced by their string names, Microsoft recommends the use of the integer value. The name of the icon resource, IDI_ICON1, can be obtained from the resource script file. However, an easier way of finding the resource name is to click the Resource Symbols button on the Resource toolbar (labeled ID=) or select the Resource Symbols command in the View menu. Either the symbolic name or the numerical value for the icon resource that is shown on the Resource Symbols screen can also be used in the MAKEINTRESOURCE macro.

In the process of creating an icon bitmap, Developer Studio also creates a new script file, or adds the information to an existing one. However, when working outside of the MFC, you must manually insert the script file into the project. This is done by selecting the Add to Project command in the Project menu and then clicking on the Files option. In the Insert Files into Project dialog box, select the script file, which in this case is the one named Script1.rc, and then press the OK button. The script file now appears on the Source Files list in the Files View window of the Project Workspace.

In addition to the script file, Developer Studio also creates a header file for resources. The default name of this file is resource.h. In order for resources to be available to the code you must enter an #include statement in the main source file, as follows:

```
#include "resource.h"
```

Notice that the double quotation marks surrounding the filename indicate that it is in the current folder.

At this point, all that is left to do is to compile the resources, the source files, and link the program into an executable. This is done by selecting the Rebuild All command in the Build menu. Figure 16-8 shows the screen display of the WinHello program.

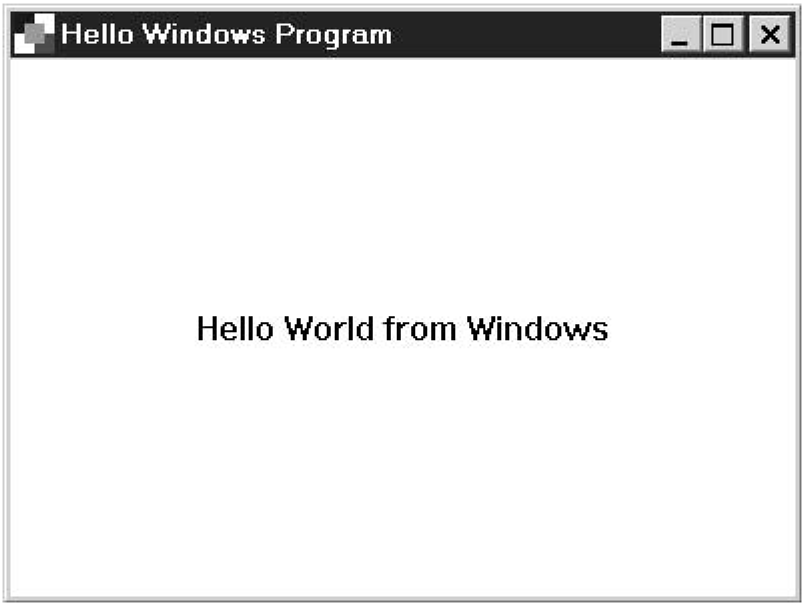


Figure 16-8 *Screen Snapshot of the WinHello Program*

16.5 WinHello Program Listing

The following is a listing of the WinHello cpp source file that is part of the Hello Windows project .

```
//**********************************************************************
//  PROJECT: Hello Windows
//  Source: WinHello.cpp
//  Chapter reference: 16
//**********************************************************************
//  Description:
//    A Hello Windows demonstration program
//  Topics:
//       1. Create a program icon
//       2. Display a text message in the client area
//**********************************************************************

#include <windows.h>        // Standard Windows header
#include "resource.h"        // Load resource file for icon

// Predeclaration of the window procedure
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

//********************************************************************
//                             WinMain
//********************************************************************
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static char szAppName[] = "Demo" ; // Class name
     HWND        hwnd ;
     MSG         msg ;

     // Defining a structure of type WNDCLASSEX
     //   The program icon is loaded in the hIcon and hIconSm
     //   structure members
     WNDCLASSEX  wndclass ;
     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = (HICON)LoadImage(hInstance,
                                     MAKEINTRESOURCE(IDI_ICON1),
                                     IMAGE_ICON,
                                     32, 32,
                                     LR_DEFAULTCOLOR) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject
                              (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = (HICON)LoadImage(hInstance,
                                     MAKEINTRESOURCE(IDI_ICON1),
                                     IMAGE_ICON,
                                     16, 16,
                                     LR_DEFAULTCOLOR) ;
     // Registering the structure wmdclass
     RegisterClassEx (&wndclass) ;

     // CreateWindow()
     hwnd = CreateWindowEx (
            WS_EX_LEFT,              // Left aligned (default)
            szAppName,                 // pointer to class name
           "Hello Windows",          // window caption
            WS_OVERLAPPEDWINDOW,     // window style
            CW_USEDEFAULT,           // initial x position
            CW_USEDEFAULT,           // initial y position
            CW_USEDEFAULT,           // initial x size
            CW_USEDEFAULT,           // initial y size
            NULL,                    // parent window handle
            NULL,                    // window menu handle
            hInstance,               // program instance handle
            NULL) ;                  // creation parameters

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

     // Message loop
     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
}
//****************************
//    Windows Procedure
//****************************
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
LPARAM lParam)
{

     PAINTSTRUCT ps ;
     RECT        rect ;
     HDC         hdc;

     switch (iMsg)
         {
     // Windows message processing
          case WM_CREATE:
               return 0;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;
               GetClientRect (hwnd, &rect) ;

               // Display message in the client area
               DrawText (hdc,
                        "Hello World from Windows",
                         -1,
                         &rect,
                         DT_SINGLELINE | DT_CENTER | DT_VCENTER);


               EndPaint (hwnd, &ps) ;
               return 0 ;

          // End of program execution
          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

Chapter 17

Text Graphics

Topics:

- Text in Windows applications
- The client area and the display context
- Mapping modes
- Text as a graphics object
- Drawing with text

In this chapter we discuss a field of Windows programming that is not conventionally considered as part of computer graphics, mainly text display. Windows is a graphics environment; all Windows programming is, in a sense, graphics programming. A natural line of demarcation between graphics and non-graphics services does not exist in the GDI. Text can be considered a graphics resource, since displaying and manipulating text characters is not different than any other graphics object.

Furthermore, discussing text programming at this point serves as an introduction into Windows application development. Understanding text programming requires knowledge of the fundamental concepts of Windows programming. These are the client area, the Windows coordinate system, the display context, and the mapping mode, which are also central elements of Windows graphics.

17.1 Text in Windows

Computer systems, including the PC, have historically differentiated between text and graphics. The original notion was that programs could either execute in textual form, by displaying messages composed of alphabetical and numeric characters, or they could use pictures and images to convey information. When the VGA (Video Graphics Array) video standard was released in 1987, it defined both text and graphics modes, with entirely different features and programming. Even in Windows, which is a graphics environment by design, there is a distinction between console-based applications and graphics-based applications. In console-based applications, Windows refers to a Console User Interface, or CUI, and in graphics-based applications, to a Graphics

User Interface, or GUI. When you select the New command in the Developer Studio File menu, the Projects tab contains an option for creating a Win32 Console Application.

In fact, in the Windows environment, the distinction between text and graphics programs is not clear. The text-related functions in the API, which are more than 20, are actually part of the GDI (Graphics Device Interface). In Windows text is a another graphics resource.

Here we consider Windows text operations as related to GUI programming. Console-based applications are not discussed in this book. In addition, text manipulations and programming provide an introduction to topics related to client area access and control, which are at the core of Windows programming.

17.1.1 The Client Area

The part of the window in which a program can draw is called the client area. The client area does not include the title bar, the sizing border, nor any of the optional elements such as the menu, toolbar, status bar, and scroll bars. The client area is the part of the program window that you access to convey information to the user and on which your application displays child windows and program controls.

DOS programmers own the device, whether working on graphics or on text modes. Once a DOS text program has set a video mode, it knows how many characters can be displayed in each text line, and how many text lines fit on the screen. By the same token, a DOS graphics program knows how many pixel rows and columns are in its domain. Some Windows programs use a fixed-size window, but in most cases, a Windows application cannot make assumptions regarding the size of its client area. Normally, the user is free to resize the screen vertically, horizontally, or in both directions simultaneously. There is practically no limit to how small it can be made, and it can be as large as the entire Windows application area. Writing code that can reasonably accommodate the material displayed to any size of the client area is one of the challenges of Windows programming.

17.2 Device and Display Contexts

The notion of a device context is that of a Windows data structure that stores information about a particular display device, such as the video display or a printer. All Windows functions that access the GDI require a handle to the device context as a parameter. The device context is the link between your application, the GDI, and the device-dependent driver that executes the graphics command on the installed hardware. Figure 17-1 is a schematic diagram of this relationship.

In Figure 17-1 we see that the Windows application uses one of several available operations to obtain a device context. The call to BeginPaint(), used in TEMPL01.CPP and in the WinHello program listed in Chapter 16, returns the handle to the device context. BeginPaint() is the conventional way of obtaining the handle to the device context in a WM_PAINT handler. The GetDC() function is often used to obtain the handle to the device context outside of WM_PAINT. In either case, from now on, a particular device context data structure is associated with the application.

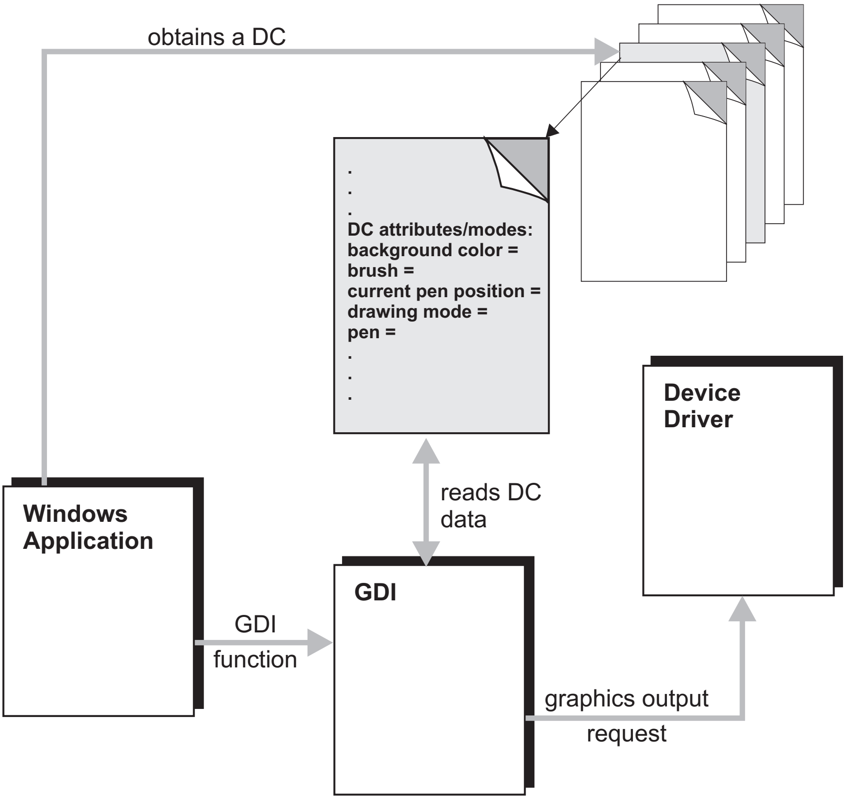


Figure 17-1 *The Device Context, Application, GDI, and Device Driver*

Once a device context has been obtained, GDI calls examine the device context attributes to determine how to perform a drawing operation. In Figure 4-1 we see some of the DC attributes: the background color, the brush, and the current position of the drawing pen. There are many attributes associated with a common display context. For example, the default stock pen is defined as BLACK_PEN in the device context. If this stock pen is not changed, the GDI uses it in all drawing operations. The application can, however, change the stock pen in the device context to NULL_PEN or WHITE_PEN by calling SelectPen().

17.2.1 The Display Context

The video display is a device that requires most careful handling in a multitasking environment. Several applications, as well as the system itself, usually share the display device. The notions of child and parent windows, client and non-client areas, desktop windows, and of applications area, all relate to this topic. The display context is a special device context for a display device.

The principal difference between a device context and the display context is that a device context allows access to the entire device, while the display context limits access to the output area of its associated window. A display context usually refers to one of the following areas:

- The window's client area
- The window's entire surface, including the non-client area
- The entire desktop surface

Application output is usually limited to the client area, therefore, this is the default display context.

Since the display context is a specialization of the term device context, it is correct to refer to the display context as a device context. The reverse, however, is not always true. For example, the printer device context is not a display context. In Chapter 3 we referred to the display context as a device context, which is acceptable. Windows documentation does not always use these terms rigorously. This has been the cause of some misunderstanding. The fact that Windows documentation sometimes uses the term display device context as equivalent to display context has added to the confusion.

17.2.2 Display Context Types

According to the application's needs, there are four possible classes of display contexts: common DC, single DC, private DC, and parent DC. The type of display context for a window is defined in the WNDCLASSEX structure. During the call to RegisterClassEx() we establish the type of display context for the windows class. This is determined by the value entered in the wndclass.style member of WNDCLASSEX.

In Table 16-2 there are three constants that refer to the display context types: CS_OWNDC, CS_CLASSDC, and CS_PARENTDC. When no display type constant is entered in the wndclass.style, then the display context type is common, which is the default. In the case of a common display context, Windows resets all attributes to the default values each time the handle is retrieved. This requires the application to reset each attribute that is different from the default settings.

The class display context is enabled with the CS_CLASSDC constant at the time of registering the window class. In this case, Windows initializes the display context attributes once, for all windows of the class. When the display context is retrieved, Windows resets the device origin and the clipping region, but not the other attributes. All windows of this class obtain the same attributes with the handle to the display context. One disadvantage of a class display context is that if one window makes changes to the display context, these changes remain in effect for all subsequent windows that use it.

The parent display context is enabled by entering the CS_PARENTDC constant in the WNDCLASSEX structure. In this case, Windows creates a common display context and sets its clipping region to the same as that of the parent. The result is that a child window can draw to its parent's client area. The most common use of a parent display context is in drawing controls inside dialog boxes. Round-off errors that result from calculating the bounding box for dialog boxes sometimes cause controls that are clipped at display time. Using a parent display context solves this problem.

The private display context is associated with a window when the CS_OWNDC constant is used in the wndclass.style member of WNDCLASSEX. At registration time, each window created from the class is given a private display context. Because each window has its own display context permanently associated, it need be retrieved only once. All attributes assigned to a private display context are retained until they are explicitly changed. In some types of applications the use of a private display context minimizes coding and improves performance.

Applications that often make changes to the client area, as is the case with many graphics programs, can often profit from a private display context. In order to accomplish this, several changes have to be made to the TEMPL01.CPP program file. In the first place, an OR operation must be performed between the CS_OWNDC constant and the other values in the wndclass.style member of WNDCLASSEX, as follows:

```
// Defining a structure of type WNDCLASSEX
  WNDCLASSEX  wndclass ;
  wndclass.cbSize        = sizeof (wndclass) ;
  wndclass.style         = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
     .
     .
     .
```

The remaining changes take place in the Windows procedure. In the first place, you must declare a variable of type HDC. This variable must have static scope so that its value is preserved between reentries of the windows procedure. The display context can be obtained during WM_CREATE processing, which executes at the time the window is created. This is possible because the display context is private. In this case, you can use the GetDC() function to obtain the handle to the display context, as in the following code fragment:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                          LPARAM lParam) {
     // Local variables
     PAINTSTRUCT        ps ;
     RECT               rect ;
     static HDC         hdc;      // Handle to private DC

     switch (iMsg)
         {
     // Windows message processing
          case WM_CREATE:
               hdc = GetDC(hwnd);    // Obtain handle to
                                     // private DC
               return 0;
     .
     .
     .
```

The private display context is available during WM_PAINT message intercept, and need not be retrieved during each iteration. Therefore, the return value from BeginPaint() can be discarded and the EndPaint() function becomes unnecessary, as in the following code fragment:

```
case WM_PAINT :
     BeginPaint (hwnd, &ps) ;
     GetClientRect (hwnd, &rect) ;
     // Display message in the client area
     DrawText (hdc,
              "Demo program using a private DC",
              -1,
              &rect,
               DT_SINGLELINE | DT_CENTER | DT_VCENTER);
     return 0 ;
        .
        .
        .
```

The project named Private DC Demo, in the book's software package, contains the full source for a private DC demonstration. You can use the source file TEMPL02.CPP as a template for creating applications that use a private display context.

17.2.3 Window Display Context

Applications sometimes wish to draw not only on the client area, but elsewhere in the window. Normally, areas such as the title bar, menus, status bar, and scroll bars are inaccessible to code that uses one of the display context types previously mentioned. You can, however, retrieve a window-level display context. In this case, the display context's origin is not at the top-left corner of the client area, but at the top-left corner of the window. The GetWindowDC() function is used to obtain the handle to the window-level display context and the ReleaseDC() function to release it. In general, drawing outside of the client area should be avoided, since it can create problems to the application and to Windows.

17.3 Mapping Modes

One of the most important attributes of the display context is the mapping mode, since it affects practically all drawing operations. The mapping mode is actually the algorithm that defines how logical units of measurement are translated into physical units. To understand mapping modes we must start with logical and device coordinates.

The programmer specifies GDI operations in terms of logical coordinates, or logical units. The GDI sends commands to the device driver in physical units, also called device coordinates. The mapping mode defines the logical units and establishes the methods for translating them into device coordinates. This translation can be described as a mapping operation. In regards to the display device, as well as in most printers, device coordinates are expressed in pixels. Logical coordinates depend on the selected mapping mode. Windows defines six fixed-size mapping modes, as shown in Table 17-1.

Two other mapping modes, not listed in Table 17-1, are MM_ANISOTROPIC and MM_ISOTROPIC. These modes can be used for shrinking and expanding graphics by manipulating the coordinate system. These two scalable mapping modes, useful for very powerful graphics manipulation, are discussed in Chapter 19.

Table 17-1

Windows Fixed-Size Mapping Modes

MAPPING MODE	LOGICAL UNITS	X-AXIS	Y-AXIS
MM_TEXT	pixel	right	down
MM_LOWMETRIC	0.1 mm	right	up
MM_HIGHMETRIC	0.01 mm	right	up
MM_LOENGLISH	0.01 inch	right	up
MM_HIENGLISH	0.001 inch	right	up
MM_TWIPS	1/1440 inch	right	up

The default mapping mode, MM_TEXT, is also the most used one. In MM_TEXT, the logical coordinates coincide with the device coordinates. Programmers who learned graphics in the DOS environment usually feel very comfortable with this mapping mode. Note that the name MM_TEXT refers to how we normally read text in the Western languages: from left-to-right and top-to-bottom. The name is unrelated to text display.

The selection of a mapping mode depends on the needs and purpose of the application. Two of the mapping modes, MM_LOMETRIC and MM_HIMETRIC, are based on the metric system (millimeters). MM_LOENGLISH and MM_HIENGLISH are based on the English system of measurement (inches). MM_TWIPS is based on a unit of measurement used in typography called the twip, which is equivalent to 1/20th of a point, or 1/1440 inch. An application that deals with architectural or technical drawings, in which dimensions are usually in inches or millimeters, can use one of the mapping modes based on metric or English units of measurement. A graphics design program, or a desktop publishing application, would probably use the MM_TWIPS mapping mode.

The SetMapMode() function is used to change the mapping mode in the device context. One of the parameters in the call is the handle to the device context; the other parameter is one of the predefined mapping mode constants. For example, to change the mapping mode to LO_METRIC, you would code:

```
static int     oldMapMode;
    .
    .
    .
 oldMapMode = SetMapMode (hdc, LO_METRIC);
```

The function returns the previous mapping mode, which can be stored in an integer variable. Later on, the original mapping mode can be restored as follows:

```
SetMapMode (hdc, oldMapMode);
```

SetMapMode() returns zero if the function call fails.

17.3.1 Screen and Client Area

Windows uses several coordinate systems. The basic unit of measurement is the pixel, also called a device unit. Horizontal values increase from left to right and vertical values from top to bottom. The origin of the coordinate system is the top-left corner of the drawing surface. Three different extents are used in relation to the device area: screen, client area, and window coordinate systems.

The screen coordinate system refers to the entire display area. This coordinate system is used when location and size information refer to the entire video display. The call to CreateWindowEx(), in the program WINHELLO.CPP and most of the template files, uses the symbolic constant CW_USEDEFAULT. This constant lets Windows select a position and size for the program's window. Alternatively, we could have specified the window's location and size in device units. For example, the following call to CreateWindowEx() locates the window at 20 by 20 pixels from the screen's upper-left corner and forces a dimension of 400 by 500 pixels:

```
// CreateWindow()
hwnd = CreateWindowEx (
       WS_EX_LEFT,               // Left aligned (default)
       szClassName,              // pointer to class name
       "WinHello Program",       // window caption
       WS_OVERLAPPEDWINDOW,      // window style
       20,                       // initial x position
       20,                       // initial y position
       400,                      // initial x size
       500,                      // initial y size
       NULL,                     // parent window handle
       NULL,                     // window menu handle
       hInstance,                // program instance handle
       NULL) ;                   // creation parameters
```

Other Windows functions, such as those that return the mouse cursor position, the location of a message box, or the location and size of the windows rectangle, also use screen coordinates.

Client area coordinates are relative to the upper-left corner of the client area, not to the video display. The default unit of measurement is the pixel. The function ClientToScreen() can be used to obtain the screen coordinates of a point in the client area. ScreenToClient() obtains the client area coordinates of a point defined by its screen coordinates. In either function, the x and y coordinates are passed and returned in a structure of type POINT.

Window coordinates refer to the top-left corner of the window itself, not to the client area. Applications that use the window display context, mentioned earlier in this chapter, use windows coordinates.

17.3.2 Viewport and Window

The terms viewport and window, when used in relation to logical and device coordinates, can be the source of some confusion. In the first place, Windows documentation uses the term viewport in a way that does not coincide with its most accepted meaning. In graphics terminology, a viewport is a specific screen area set aside for a particular graphics function. In this sense, the notion of a viewport implies a region within the application's window.

In Windows, the viewport is often equated with the client area, the screen area, or the application area, according to the bounds of the device context. The one characteristic element of the viewport is that it is expressed in device units, which are pixels. The window, on the other hand, is expressed in terms of logical coordinates. Therefore, the unit of measurement of a window can be inches, millimeters, twips, or pixels in the six fixed-sized mapping modes, or one defined by the application in the two scalable mapping modes.

In regards to viewports and windows, there are two specific boundaries that must be considered: the origin and the extent. The origin refers to the location of the window or viewport, and the extent to its width and height. The origin of a window and a viewport can be set to different values in any of the mapping modes. Function calls to set the window and the viewport extent are ignored when any one of the six fixed-sized mapping modes is selected in the device context. However, in the two scalable mapping modes, MM_ISOTROPIC and MM_ANISOTROPIC, both the origin and the extent of the viewport and the window can be set separately.

A source of confusion is that both the viewport and the window coincide in the default mapping mode (MM_TEXT). In the fixed-size mapping modes, the extent of the viewport and the window cannot be changed, as mentioned in the preceding paragraph. This should not be interpreted to mean that they have the same value. Actually, the measurement in units of length of the viewport and the window extent is meaningless. It is the ratio between the extent that is useful. For example, if the viewport extent is 20 units and the window extent is 10 units, then the ratio of viewport to window extent is of 20/10, or 2. This value is used as a multiplier when converting between window and device coordinates. Other factors that must be taken into account in these conversions are the location of the point, the origin of the viewport, and the origin of the window. Figure 17-2, on the following page, is a simplified, schematic representation of the concepts of viewport and window.

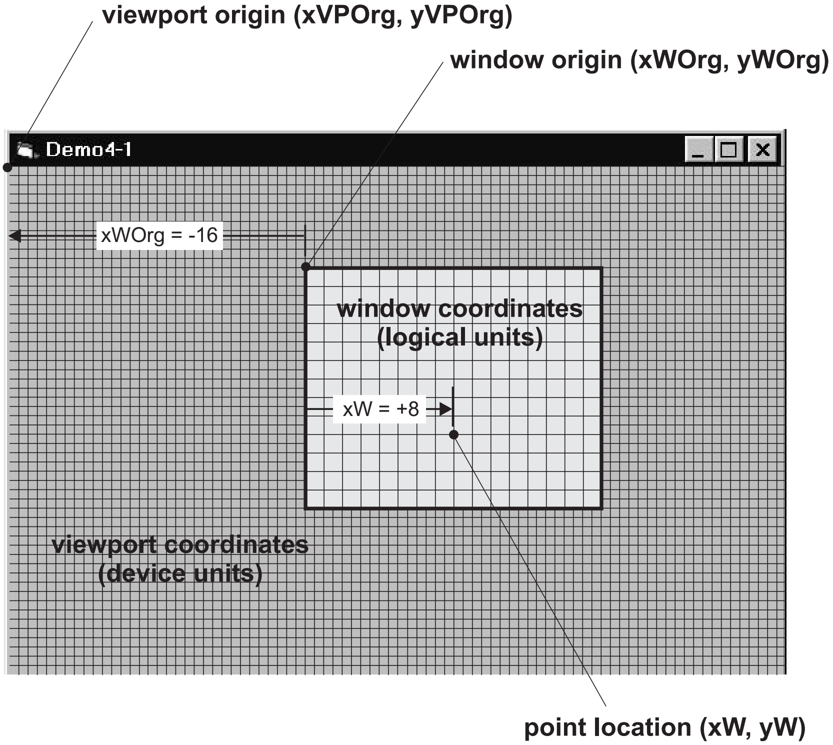
In Figure 17-2, the dimension of the logical units is twice that of the device units, in both axes. Therefore, the ratio between the window extension and the device extension (xVPExt / xWExt and yVPExt / yWExt) equals 2. The point located at xW, yW is at window coordinates xW = 8, yW = 9, as shown in the illustration. To convert to device coordinates, we apply the corresponding formulas. In calculating the x-axis viewport coordinate of the point xW, yW, we proceed as follows:

```
xVP = (xW - xWOrg) x (xVPExt / xWExt) + xVPOrg
xVP = (8 -(- 16)) x 2 + 0
xVP = 48
```

This means that in the example in Figure 17-2, the point at window coordinates x = 8, y = 9, located in a window whose origin has been displaced 16 logical units on the x-axis, and 5.5 logical units in the y-axis, is mapped to viewport coordinates xVP = 48, yVP = 25. Note that the sample calculations do not include the y-coordinate.

17.4 Programming Text Operations

Text operations in console-based applications are usually a simple task. The text characters are displayed using whatever font is selected at the system level, and at the screen line and column where the cursor is currently positioned. In analogy with the old Teletype machines, this form of text output programming is said to be based on the model of a "glass TTY." But even when the program takes control of the display area, the matter of text output is no more complicated than selecting a screen line and a column position.



CONVERSION FORMULAS: (all mapping modes)

xVP = (xW - xWOrg) x (xVPExt / xWExt) + xVPOrg
yVP = (yW - yWOrg) x (yVPExt / yWExt) + yVPOrg

xW = (xVP- xVPOrg) x (xWExt / xVPExt) + xWOrg
yW = (yVP- yVPOrg) x (yWExt / yVPExt) + yWOrg

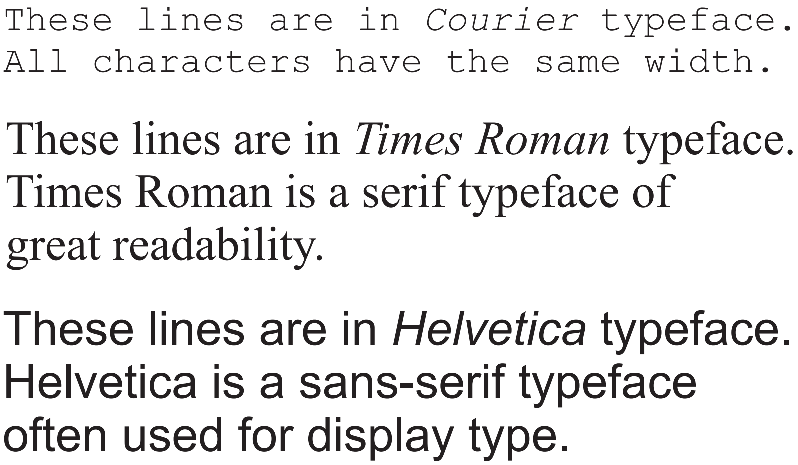
Figure 17-2 *Viewport and Window Coordinates*

In graphics programming, and particularly in Windows graphics, the coding of text operations often becomes a major task, to the point that Windows text programming is considered a specialty field. In this sense, it is possible to speak of bitmapped graphics, of vector graphics, and of text graphics. Developing a GDI-based text-processing application, such as a Windows word processing or desktop publishing program, involves a great amount of technical complexity. In addition to programming skills, it requires extensive knowledge of typography, digital composition, and graphics arts. At present, we are concerned with text graphics in a non-specialized context. That is, text display is one of the functionality that is normally necessary in implementing a Windows application. But even in this more general sense, text programming in Windows is not without some complications.

17.4.1 Typefaces and Fonts

A collection of characters of the same design is called a typeface. Courier, Times Roman, and Helvetica are typefaces. Courier is a monospaced typeface that originated in typewriter technology. The characters in the Courier typeface all have the same width. Times Roman is a typeface developed in the nineteenth century by an English newspaper with the purpose of making small type readable when printed on newspaper stock. Times Roman uses short, horizontal lines of a different thickness. To some, these elements resemble hooks; for which the typeface is called serif (hook, in French). On the other hand, the characters in the Helvetica typeface have the same thickness; therefore, it is called a sans-serif typeface (without hooks).

Times Roman and Helvetica are proportionally spaced fonts; that is, each character is designed to an ideal width. In a proportionally spaced font, the letter "w" is wider than the letter "i." In Windows, proportionally spaced fonts are sometimes called variable pitch fonts. They are more pleasant and easier to read than monospaced fonts, but digits displayed in proportionally spaced fonts do not align in columns. Figure 17-3 shows text in Courier, Times Roman, and Helvetica typefaces.



These lines are in *Courier* typeface.
All characters have the same width.

These lines are in *Times Roman* typeface.
Times Roman is a serif typeface of
great readability.

These lines are in *Helvetica* typeface.
Helvetica is a sans-serif typeface
often used for display type.

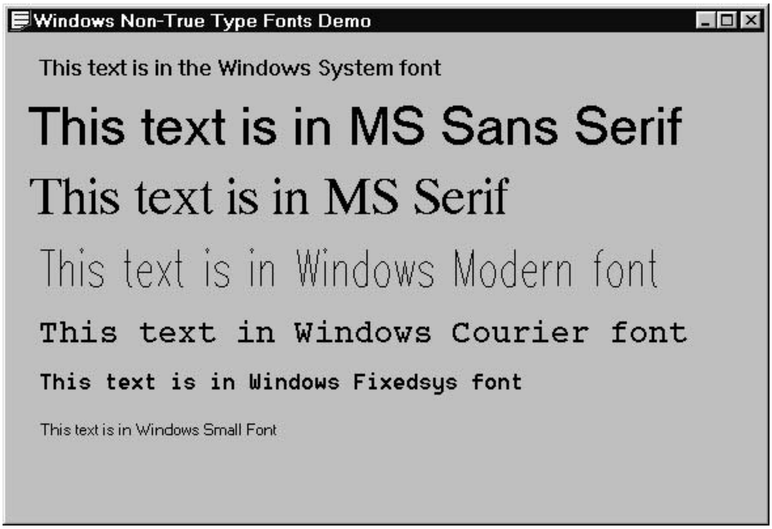
Figure 17-3 *Courier, Times Roman, and Helvetica Typefaces.*

A group of related typefaces is called a typeface family; for example, Helvetica Bold and Helvetica Oblique are typeface families. A font is a collection of characters of the same typeface and size. In this sense you can speak of the Times Roman 12-point font. Type style is a term used somewhat loosely in reference to specific attributes applied to characters in a font. Boldface (dark), roman (straight up), and italics (slanted towards the right) are common type styles.

Historically, Windows fonts have been of three different types: raster, vector, and TrueType. Raster fonts are stored as bitmaps. Vector fonts, sometimes called stroke fonts, consist of a set of drawing orders required to produce each letter. TrueType fonts, introduced in Windows 95, are similar to PostScript fonts. They are defined as lines and curves, can be scaled to any size, and rotated at will. TrueType fonts are more versatile and have the same appearance on the screen as when printed. TrueType fonts also assure portability between applications. Programmers working in Windows 95 and NT deal mostly with TrueType fonts.

For reasons related to copyright and trademark laws, some Windows fonts have names that differ from the traditional typefaces. For example, Times New Roman is the Windows equivalent of Times Roman, and the Helvetica typeface is closely approximated by the Windows versions called Arial, Swiss, and Switzerland.

The default Windows font is named the system font. In current versions of Windows, the system font is a proportionally spaced font. It is also a raster font, therefore, the characters are defined as individual bitmaps. Figure 17-4 is a screen snapshot of a Windows program that demonstrates the screen appearance of the various non-TrueType fonts.



Windows Non-True Type Fonts Demo

This text is in the Windows System font

This text is in MS Sans Serif

This text is in MS Serif

This text is in Windows Modern font

This text in Windows Courier font

This text is in Windows Fixedsys font

This text is in Windows Small Font

Figure 17-4 *Windows Non-TrueType Fonts*

17.4.2 Text Formatting

In order to display text in a graphics, multitasking environment (one in which the screen can be resized at any time), code must be able to obtain character sizes at run time. For example, in order to display several lines of text you must know the height of the characters so that the lines are shown at a reasonable vertical distance from each other. By the same token, you also need to know the width of each character, as well as the width of the client area, in order to handle the end of each text line.

The GetTextMetrics() function provides information about the font currently selected in the display context. GetTextMetrics() requires two parameters: the handle to the device context and the address of a structure variable of type TEXTMETRICS. Table 17-2 lists the members of the TEXTMETRIC structure:

Table 17-2

TEXTMETRIC structure

TYPE	MEMBER	CONTENTS
LONG	tmHeight	Character height (ascent + descent)
LONG	tmAscent	Height above the baseline
LONG	tmDescent	Height below the baseline
LONG	tmInternalLeading	Internal leading
LONG	tmExternalLeading	External leading
LONG	tmAveCharWidth	Width of the lowercase letter "x"
LONG	tmMaxCharWidth	Width of widest letter in font
LONG	tmWeight	Font weight
LONG	tmOverhang	Extra width per string added to some synthesized fonts
LONG	tmDigitizedAspectX	Device horizontal aspect
LONG	tmDigitizedAspectY	Device vertical aspect. The ratio of tmDigitizedAspectX / tmDigitizedAspectY members is the aspect ratio of the device for which the font was designed.
BCHAR	tmFirstChar	First character in the font
BCHAR	tmLastChar	Last character in the font
BCHAR	tmDefaultChar	Character used as a substitute for Those not implemented in the font
BCHAR	tmBreakChar	Character used as a word break in Text justification
BYTE	tmItalic	Nonzero if font is italic
BYTE	tmUnderlined	Nonzero if font is underlined
BYTE	tmStruckOut	Nonzero if font is strikeout
BYTE	tmPitchAndFamily	Contains information about the font family in the four low-order bits of the following constants: CONSTANT BIT MEANING TMPF_FIXED PITCH 0 fixed pitch font TMPF_VECTOR 1 vector font TMPF_TRUETYPE 2 True Type font TMPF_DEVICE 3 device font
BYTE	tmCharSet	Specifies the font's character set

Notice that in printing and display technology, the baseline is an imaginary horizontal line that aligns the base of the characters, excluding descenders. The term leading (pronounced "led-ing") refers to the space between lines of type, usually measured from the baseline of one line to the baseline of the next one. Figure 17-5 shows the vertical character dimensions represented by the corresponding members of the TEXTMETRIC structure.

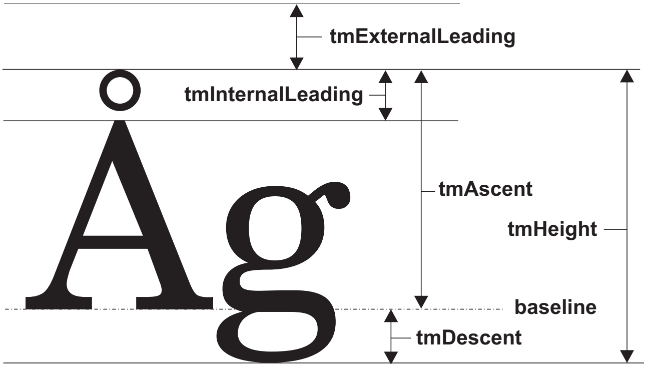


Figure 17-5 *Vertical Character Dimensions in the TEXTMETRIC Structure*

Text metric values are determined by the font installed in the device context. For this reason, where and how an application obtains data about text dimensions depend on the font and on how the device context is handled. An application that uses the system font, and no other, need only obtain text metric values once in each session. Since the system font does not change during a Windows session, these values are valid throughout the program's lifetime. However, if an application changes device contexts or fonts during execution, then the text metric values may also change.

In the simplest case, a text processing application can obtain text metric data while processing the WM_CREATE message. Usually, the minimal data required for basic text manipulations is the character height and width. The height is calculated by adding the values in the tmHeight and tmExternalLeading members of the TEXTMETRIC structure for the current display context (see Figure 17-5). The width of the lowercase characters can be obtained from the tmAveCharWidth member.

The calculation of the average width of uppercase characters is somewhat more complicated. If the font currently selected in the display context is monospaced (fixed pitch, in Windows terminology), then the width of the uppercase characters is the same as the lowercase ones. However, if the current font is proportionally spaced (sometimes called a variable pitch font in Windows), then you can obtain an approximation of the width of the uppercase characters by calculating 150 percent of the width of the lowercase ones. We have seen that the tmPitchAndFamily member of TEXTMETRIC has the low-order bit set if the font is monospaced. We can logically AND this value with a binary 1 in order to test if the font is monospaced. Assuming that a TEXTMETRIC structure variable is named tm, and that the width of the lowercase characters is stored in an integer variable named cxChar, the code would be as follows:

```
int cxCaps;     // Storage for width of uppercase characters
 if(tm.tmPitchAndFamily & 0x1)
   cxCaps = (3 * cxChar) / 2;    // 150 percent
else
   cxCaps = cxChar;              // 100 percent
```

More compact coding results from using the ? operator, as follows:

```
cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2;
```

The values can be stored in static variables for future use. The following code fragment shows the usual processing in this case:

```
static int cxChar;     // Storage for lowercase character width
static int cxCaps;     // Storage for uppercase character width
static int cyChar;     // Storage for character height plus
                       // leading
  .
  .
  .
   case WM_CREATE :
     hdc = GetDC (hwnd) ;
     GetTextMetrics (hdc, &tm) ;

     cxChar = tm.tmAveCharWidth ;
```

```
    cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2;
    cyChar = tm.tmHeight + tm.tmExternalLeading ;

    ReleaseDC (hwnd, hdc) ;
    return 0 ;
```

In addition to information about text dimensions, text processing applications also need to know the size of the client area. The problem in this case is that in most applications, the size of the client area can change at any time. If the window was created with the style WM_HREDRAW and WM_VREDRAW, a WM_SIZE message is sent to the Windows procedure whenever the client area size changes vertically or horizontally. A WM_PAINT message automatically follows. The application can intercept the WM_SIZE message and store, in a static variable, the vertical and horizontal dimensions of the client area. The size of the client area can be retrieved from these variables whenever you need to redraw to the window. Traditionally, the variables named cxClient and cyClient are used to store these values. The low word of the lParam value, passed to the Windows procedure during WM_SIZE, contains the width of the client area, and the high word contains the height. The code can be as follows:

```
static int cxClient;    // client area width
  static int cyClient;    // client area height
  .
  .
  .
  case WM_SIZE:
    cxClient = LOWORD (lParam);
    cyClient = HIWORD (lParam);
    return 0;
```

17.4.3 Paragraph Formatting

The logic needed for text formatting at the paragraph level is as follows: First, we determine the character dimensions by calling the GetTextMetric() and then reading the corresponding members of a TEXTMETRIC structure. Next, we obtain the size of the client area during WM_SIZE processing by means of the high- and low-word of the lParam argument. This information is sufficient for performing exact calculation on a monospaced font. In the case of a proportionally spaced font, we are forced to deal in approximations, since what we have obtained is the average width of lower-case characters and an estimate of the width of the upper-case ones.

GetTextExtentPoint32(), a function that has suffered several transformations in the various versions of Windows, computes the exact width and height of a character string. The function takes as a parameter the handle to the device context, since the string size calculated is based on the currently installed font. Other parameters are the address of the string, its length in characters, and the address of a structure of type SIZE where information is returned to the caller. The SIZE structure contains only two members: one for the x dimension and another one for the y dimension. The value returned by GetTextExtentPoint32() is in logical units.

Putting it all together: suppose you have a rather long string, one that requires more than one screen line, stored in a static or public array, and you want to display this string breaking the screen lines only at the end of words. Since in Windows the length of each line in the client area can be changed at any time by the user, the code would have to dynamically adjust for this fact. Placing the processing in a WM_PAINT message handler ensures that the display is updated when the client area changes in size. This also requires that we intercept the WM_SIZE message to recalculate the size of the client area, as discussed previously. The processing logic in WM_PAINT could be as follows:

1. Step through the string, pausing at each space, and calculate the string length using GetTextExtentPoint32(). Keep count of the number of characters to the previous space, or the beginning of the text string in the case of the first word.
2. If the length of the string is larger than could fit in the client area, then backtrack to the previous space and display the string to that point. Reset the string pointer so that the new string starts at the last character displayed. Continue at step 1.
3. If the end of the string has been reached, display the string starting at the last space and exit the routine.

The actual implementation requires a few other processing details. For example, you may want to leave a margin of a couple of characters on the left and right sides of the display area. In addition, the code would need to manipulate pointers and counters to keep track of the string positions and the number of characters to the previous space. One possible algorithm is reminiscent of the classic case of a circular buffer with two pointers: one to the buffer head and another one to the tail. Figure 17-6 graphically shows the code elements in one of many possible implementations.

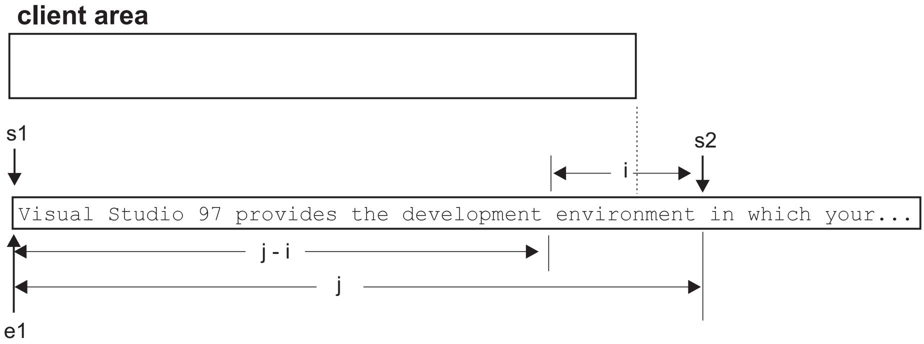

client area
s1
s2
i
Visual Studio 97 provides the development environment in which your...
j - i
j
e1

Figure 17-6 *Processing Operations for Multiple Text Lines*

In Figure 17-6, the pointer that signals the start of the string is designated with the letter s and the one for the end of the string with the letter e. s1 and e1 is the start position for both pointers. The variable i is a counter that holds the number of characters since the preceding space, and j holds the number of characters in the current substring. The code steps along the string looking for spaces. At each space, it measures the length of the string and compares it to the horizontal dimension of the client area. When the s pointer reaches location s2, the substring is longer than the display space available in the client area. The variable i is then used to reset the pointer to the preceding space and to decrement the j counter. The substring is displayed starting at e1, for a character count of j. Pointers and counters are then reset to the new sub-string and processing continues until the end of the string is found.

A demonstration program named TEX1_DEMO is furnished in the book's software package. The message to be displayed is stored in a public string, as follows:

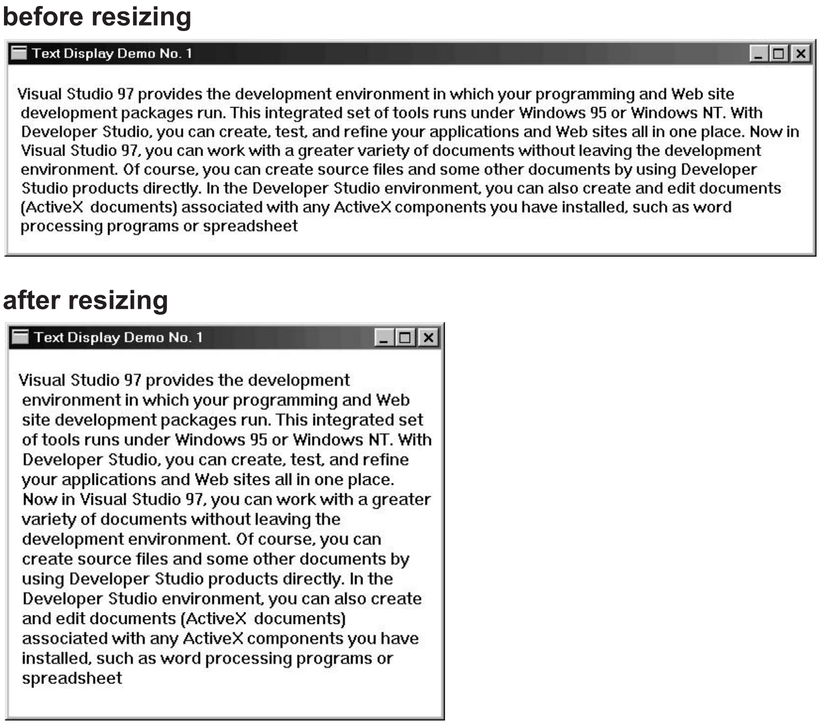
```
// Public string for text display demonstration
char TextMsg[] = {"Visual Studio 97 provides the development "
"environment in which your programming and Web site "
"development packages run. This integrated set of tools runs"
 .
 .
 .
"spreadsheet programs." };
```

The processing operations, located in the Windows procedure, are coded as follows:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                          LPARAM lParam) {
 static int  cxChar, cxCaps, cyChar ;// Character dimensions
 static int  cxClient, cyClient;  // Client area parameters
     HDC         hdc ;            // handle to device context
     int j;                       // Offset into string
     int i;                       // characters since last
                                  // space
     char *startptr, *endptr;     // String pointers
     int cyScreen;                // Screen row holder

 // Structures
     PAINTSTRUCT ps;
     TEXTMETRIC  tm;
     SIZE        textsize; // Test string size

 switch (iMsg)
          {
      case WM_CREATE :
           hdc = GetDC (hwnd) ;
           GetTextMetrics (hdc, &tm) ;

           // Calculate and store character dimensions
           cxChar = tm.tmAveCharWidth ;
           cxCaps = ((tm.tmPitchAndFamily & 1) ? 3 : 2) *\
                     cxChar / 2 ;
           cyChar = tm.tmHeight + tm.tmExternalLeading ;

           ReleaseDC (hwnd, hdc) ;
           return 0 ;

      case WM_SIZE:
           // Determine and store size of client area
           cxClient = LOWORD(lParam);
           cyClient = HIWORD(lParam);
           return 0;

      case WM_PAINT :
           hdc = BeginPaint (hwnd, &ps) ;

           // Initialize variables
           cyScreen = cyChar;      // screen row counter
           startptr = TextMsg;     // start position pointer
```

```
        endptr = TextMsg;       // end position pointer
        j = 0;                  // length of string
        i = 0;                  // characters since last
                                // space

        // Text line display loop
        // INVARIANT:
        //        i = characters since last space
        //        j = length of current string
        // startptr = pointer to substring start
        //   endptr = pointer to substring end

        while(*startptr) {
          if(*startptr == 0x20){    // if character is
                                    // space
            GetTextExtentPoint32 (hdc, endptr, j,\
                                  &textsize);

   // ASSERT:
   //       textsize.cx is the current length of the
   //       string
   //       cxClient is the abscissa of the client area
   //       (both in logical units)
   // Test for line overflow condition. If so, adjust
   // substring to preceding space and display
      if(cxClient - (2 * cxChar) < textsize.cx) {
         j = j - i;
         startptr = startptr - i;
         TextOut (hdc, cxChar, cyScreen, endptr, j);
         cyScreen = cyScreen + cyChar;
         endptr = startptr;
         j = 0;
         }
   // End of space character processing.
   // Reset chars-to-previous-space counter, whether
   // or not string was displayed
         i = 0;
          }
   // End of processing for any text character
   // Update substring pointer and counters
       startptr++;
       j++;
       i++;
       }
   // End of while loop
   // Display last text substring
       j = j - i;
       TextOut (hdc, cxChar, cyScreen, endptr, j);
       EndPaint (hwnd, &ps);
       return 0 ;

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

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

In Figure 17-7 there are two screen snapshots of the TEX1_DEMO program in the Text Demo No 1 project folder. The first one shows the text line as originally displayed in our system. The second one shows them after the client area has been resized.



before resizing

Text Display Demo No. 1

Visual Studio 97 provides the development environment in which your programming and Web site development packages run. This integrated set of tools runs under Windows 95 or Windows NT. With Developer Studio, you can create, test, and refine your applications and Web sites all in one place. Now in Visual Studio 97, you can work with a greater variety of documents without leaving the development environment. Of course, you can create source files and some other documents by using Developer Studio products directly. In the Developer Studio environment, you can also create and edit documents (ActiveX documents) associated with any ActiveX components you have installed, such as word processing programs or spreadsheet

after resizing

Text Display Demo No. 1

Visual Studio 97 provides the development environment in which your programming and Web site development packages run. This integrated set of tools runs under Windows 95 or Windows NT. With Developer Studio, you can create, test, and refine your applications and Web sites all in one place. Now in Visual Studio 97, you can work with a greater variety of documents without leaving the development environment. Of course, you can create source files and some other documents by using Developer Studio products directly. In the Developer Studio environment, you can also create and edit documents (ActiveX documents) associated with any ActiveX components you have installed, such as word processing programs or spreadsheet

Figure 17-7 *Two Screen Snapshots of the TEX1_DEMO Program*

Notice that the TEX1_DEMO program uses a variable (j) to store the total size of the substring (see Figure 17-6). In C++ it is valid to subtract two pointers in order to determine the number of elements between them. The code in the TEX1_DEMO program could have calculated the number of elements in the substring by performing pointer subtraction.

17.4.4 The DrawText() Function

Another useful text display function in the Windows API is DrawText(). This function is of a higher level than TextOut() and, in many cases, text display operations are easier to implement with DrawText(). DrawText() uses a rectangular screen area that defines where the text is to be displayed. In addition, it recognizes some control characters embedded in the text string as well as a rather extensive collection of format controls, which are represented by predefined constants. The following are the general forms for TextOut() and DrawText()

```
TextOut (hdc, nXStart, nYStart, lpString, cbString);
DrawText (hdc, lpString, nCount, &rect, uFormat);
```

In both cases, hdc is the handle to the device context and lpString is a pointer to the string to be displayed. In TextOut() the second and third parameters (xXstart and nYStart) are the logical coordinates of the start point in the client area, and the last parameter is the string length. In DrawText() the third parameter (nCount) is the string length in characters. If this parameter is set to –1 then Windows assumes that the string is zero terminated. The positioning of the string in DrawText() is by means of a rectangle structure (type RECT) described in Chapter 3 and listed in Appendix A. This structure contains four members, two for the rectangle's top-left coordinates, and two for its bottom-right coordinates. The values are in logical units. The last parameter (uFormat) is any combination of 19 format strings defined by the constants listed in Table 17-3.

Table 17-3

String Formatting Constants in DrawText()

SYMBOLIC CONSTANT	MEANING
DT_BOTTOM	Specifies bottom-justified text. Must be combined with DT_SINGLELINE.
DT_CALCRECT	Returns width and height of the rectangle. In the case of multiple text lines, DrawText() uses the width of the rectangle pointed to by lpRect and extends its base to enclose the last line of text. In the case of a single text line, then DrawText() modifies the right side of the rectangle so that it encloses the last character. In either case, DrawText() returns the height of the formatted text, but does not draw the text.
DT_CENTER	Text is centered horizontally.
DT_EXPANDTABS	Expands tab characters. The default number of characters per tab is eight.
DT_EXTERNALLEADING	ncludes the font's external leading in the line height. Normally, external leading is not included in the height of a line of text.
DT_LEFT	Specifies text that is aligned flush-left.
DT_NOCLIP	Draws without clipping. This improves performance.
DT_NOPREFIX	Turns off processing of prefix characters. Normally, DrawText() interprets the ampersand (&) mnemonic-prefix character as an order to underscore the character that follows. The double ampersands (&&) is an order to print a single ampersand symbol. This function is turned off by DT_NOPREFIX.
DT_RIGHT	Specifies text that is aligned flush-right.
DT_SINGLELINE	Specifies single line only. Carriage returns and linefeed are ignored.
DT_TABSTOP	Sets tab stops. The high-order byte of nFormat is the number of characters for each tab. The default number of characters per tab is eight.
DT_TOP	Specifies top-justified text (single line only).
DT_VCENTER	Specifies vertically centered text (single line only).
DT_WORDBREAK	Enables word-breaking. Lines are automatically broken between words if a word woul extend past the edge of the rectangle specified by lpRect. A carriage return (\n) or linefeed code (\r) also breaks the line.

The program TEX2_DEMO, located in the Text Demo No 2 project folder on the book's software package, is a demonstration of text display using the DrawText() function. Following are the excerpts from the program code:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                          LPARAM lParam) {

 static int  cxChar, cyChar ;       // Character dimensions
 static int  cxClient, cyClient;    // Client area parameters
 HDC     hdc ;                      // handle to device context

 // Structures
 PAINTSTRUCT ps;
 TEXTMETRIC  tm;
 RECT        textRect;

 switch (iMsg) {
    case WM_CREATE :
         hdc = GetDC (hwnd) ;
         GetTextMetrics (hdc, &tm) ;

         // Calculate and store character dimensions
         cxChar = tm.tmAveCharWidth ;
         cyChar = tm.tmHeight + tm.tmExternalLeading ;

         ReleaseDC (hwnd, hdc) ;
         return 0 ;

    case WM_SIZE:
         // Determine and store size of client area
         cxClient = LOWORD(lParam);
         cyClient = HIWORD(lParam);
         return 0;

    case WM_PAINT :
         hdc = BeginPaint (hwnd, &ps) ;

         // Initialize variables
           SetRect (&textRect,       // address of structure
                    2 * cxChar,              // x for start
                    cyChar,                  // y for start
                    cxClient -(2 * cxChar), // x for end
                    cyClient);               // y for end

         // Call display function using left-aligned and
         //wordbreak controls
         DrawText( hdc, TextStr, -1, &textRect,
                   DT_LEFT | DT_WORDBREAK);

         EndPaint (hwnd, &ps);
         return 0 ;

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

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

17.5 Text Graphics

Comparing the listed processing operations with those used in the TEX1_DEMO program (previously in this chapter) you can see that the processing required to achieve the same functionality is simpler using DrawText() than TextOut(). This observation, however, should not mislead you into thinking that DrawText() should always be preferred. The interpretation of the reference point at which the text string is displayed when using TextOut() depends on the text-alignment mode set in the device context. The GetTextAlign() and SetTextAlign() functions can be used to retrieve and change the eleven text alignment flags. This feature of TextOut() (and its newer version TextOutExt()) allow the programmer to change the alignment of the text-bounding rectangle and even to change the reading order to conform to that of the Hebrew and Arabic languages.

Windows NT and Windows 95 GDI supports the notion of paths. Paths are discussed in detail in Chapter 20. For the moment, we define a path, rather imprecisely, as the outline produced by drawing a set of graphical objects. One powerful feature of TextOut(), which is not available with DrawText(), is that when it is used with a TrueType font, the system generates a path for each character and its bounding box. This can be used to display text transparently inside other graphics objects, to display character outlines (called stroked text), and to fill the text characters with other graphics objects. The resulting effects are often powerful.

17.5.1 Selecting a Font

The one limitation of text display on paths is that the font must be TrueType. Therefore, before getting into fancy text graphics, you must be able to select a TrueType font into the device context. Font manipulations in Windows are based on the notion of a logical font. A logical font is a description of a font by means of its characteristics. Windows uses this description to select the best matching font among those available.

Two API functions allow the creation of a logical font. CreateFont() requires a long series of parameters that describe the font characteristics. CreateFontIndirect() uses a structure in which the font characteristics are stored. Applications that use a single font are probably better off using CreateFont(), while programs that change fonts during execution usually prefer CreateFontIndirect(). Note that the item list used in the description of a logical font is the same in both functions. Therefore, storing font data in structure variables is an advantage only if the structure can be reused. The description that follows refers to the parameters used in the call to CreateFont(), which are identical to the ones used in the structure passed by CreateFontIndirect().

The CreateFont() function has one of the longest parameter lists in the Windows API: fourteen in all. Its general form is as follows:

```
HFONT CreateFont( nHeight, nWidth, nEscapement, int nOrientation,
                  fnWeight, fdwItalic, fdwUnderline, fdwStrikeOut,
                  fdwCharSet, fdwOutputPrecision, fdwClipPrecision,
                  fdwQuality, fdwPitchAndFamily,
                  LPCTSTR lpszFace);
```

Following are brief descriptions of the function parameters.

- nHeight (int) specifies the character height in logical units. The value does not include the internal leading, so it is not equal to the tmHeight value in the TEXTMETRIC structure. Also note that the character height does not correspond to the point size of a font. If the MM_TEXT mapping mode is selected in the device context, it is possible to convert the font's point size into device units by means of the following formula:
- hHeight = (point_size * pixels_per_inch) / 72
- The pixels per inch can be obtained by reading the LOGPIXELSY index in the device context, which can be obtained by the call to GetDeviceCaps(). For example, to obtain the height in logical units of a 50-point font we can use the following expression:
- 50 * GetDeviceCaps (hdc, LOGPIXELSY) / 72
- nWidth (int) specifies the logical width of the font characters. If set to zero, the Windows font mapper uses the width that best matches the font height.
- nEscapement (int) specifies the angle between an escapement vector, defined to be parallel to the baseline of the text line, and the drawn characters. A value of 900 (90 degrees) specifies characters that go upward from the baseline. Usually this parameter is set to zero.
- nOrientation (int) defines the angle, in tenths of a degree, between the character's base line and the x-axis of the device. In Windows NT the value of the character's escapement and orientation angles can be different. In Windows 95 they must be the same.
- fnWeight (int) specifies the font weight. The constants listed in Table 4-4 are defined for convenience:

Table 17-4

Character Weight Constants

WEIGHT	CONSTANT
FW_DONTCARE	= 0
FW_THIN	= 100
FW_EXTRALIGHT	= 200
FW_ULTRALIGHT	= 200
FW_LIGHT	= 300
FW_NORMAL	= 400
FW_REGULAR	= 400
FW_MEDIUM	= 500
FW_SEMIBOLD	= 600
FW_DEMIBOLD	= 600
FW_BOLD	= 700
FW_EXTRABOLD	= 800
FW_ULTRABOLD	= 800
FW_HEAVY	= 900
FW_BLACK	= 900

- fdwItalic (DWORD) is set to 1 if font is italic.
- fdwUnderline (DWORD) is set to 1 if font is underlined.
- fdwStrikeOut (DWORD) is set to 1 if font is strikeout.
- fdwCharSet (DWORD) defines the font's character set. The following are predefined character set constants:

```
ANSI_CHARSET
DEFAULT_CHARSET
SYMBOL_CHARSET
SHIFTJIS_CHARSET
GB2312_CHARSET
HANGEUL_CHARSET
CHINESEBIG5_CHARSET
OEM_CHARSET
```

Windows 95 only:

```
JOHAB_CHARSET
HEBREW_CHARSET
ARABIC_CHARSET
GREEK_CHARSET
TURKISH_CHARSET
THAI_CHARSET
EASTEUROPE_CHARSET
RUSSIAN_CHARSET
MAC_CHARSET
BALTIC_CHARSET
```

The DEFAULT_CHARSET constant allows the name and size of a font to fully describe it. If the font does not exist, another character set can be substituted. For this reason, this field should be used carefully. A specific character set should always be defined to ensure consistent results.

fdwOutputPrecision (DWORD) determines how closely the font must match the values entered in the fields that define its height, width, escapement, orientation, pitch, and font type. Table 17-5 lists the constants associated with this parameter.

Table 17-5

Predefined Constants for Output Precision

PREDEFINED CONSTANT	MEANING
OUT_CHARACTER_PRECIS	Not used.
OUT_DEFAULT_PRECIS	Specifies the default font mapper behavior.
OUT_DEVICE_PRECIS	Instructs the font mapper to choose a Device font when the system contains multiple fonts with the same name.
OUT_OUTLINE_PRECIS	Windows NT: This value instructs the font mapper to choose from TrueType and other outline-based fonts. Not used in Windows 95.
OUT_RASTER_PRECIS	Instructs the font mapper to choose a raster font when the system contains multiple fonts with the same name.

(continues)

Table 17-5

Predefined Constants for Output Precision (continued)

PREDEFINED CONSTANT	MEANING
OUT_STRING_PRECIS	This value is not used by the font mapper, but it is returned when raster fonts are enumerated.
OUT_STROKE_PRECIS	Windows NT: This value is not used by the font mapper, but it is returned when TrueType, other outline-based fonts, and vector fonts are enumerated. Windows 95: This value is used to map vector fonts, and is returned when TrueType or vector fonts are enumerated.
OUT_TT_ONLY_PRECIS	Instructs the font mapper to choose from only TrueType fonts. If there are no TrueType fonts installed in the system, the font mapper returns to default behavior.
OUT_TT_PRECIS	Instructs the font mapper to choose a TrueType font when the system contains multiple fonts with the same name.

If there is more than one font with a specified name, you can use the OUT_DEVICE_PRECIS, OUT_RASTER_PRECIS, and OUT_TT_PRECIS constants to control which one is chosen by the font mapper. For example, if there is a font named Symbol in raster and TrueType form, specifying OUT_TT_PRECIS forces the font mapper to choose the TrueType version. OUT_TT_ONLY_PRECIS forces the font mapper to choose a TrueType font, even if it must substitute one of another name.

fdwClipPrecision (DWORD) specifies the clipping precision. This refers to how to clip characters that are partially outside the clipping region. The constants in Table 17-6 are recognized by the call.

Table 17-6

Predefined Constants for Clipping Precision

PREDEFINED CONSTANT	MEANING
CLIP_DEFAULT_PRECIS	Default clipping behavior.
CLIP_CHARACTER_PRECIS	Not used.
CLIP_STROKE_PRECIS	Not used by the font mapper, but is returned when raster, vector, or TrueType fonts are enumerated. Windows NT: For compatibility, this value is always returned when enumerating fonts.
CLIP_MASK	Not used.
CLIP_EMBEDDED	Specify this flag to use an embedded read-only font.
CLIP_LH_ANGLES	The rotation for all fonts depends on whether the orientation of the coordinate system is left- or right-handed. If not used, device fonts always rotate counterclockwise.
CLIP_TT_ALWAYS	Not used.

fdwQuality (DWORD) specifies the output quality. This value defines how carefully GDI must attempt to match the logical font attributes to those of an actual physical font. The constants in Table 17-7 are recognized by CreateFont().

Table 17-7

Predefined Constants for Output Precision

PREDEFINED CONSTANT	MEANING
DEFAULT_QUALITY	Appearance of the font does not matter.
DRAFT_QUALITY	Appearance of the font is less important than when the PROOF_QUALITY value is used.
PROOF_QUALITY	Character quality of the font is more important than exact matching of the logical-font attributes. When PROOF_QUALITY is used, the quality of the font is high and there is no distortion of appearance.

fdwPitchAndFamily (DWORD) defines the pitch and the family of the font. The 2 low-order bits specify the pitch, and the 2 high-order bits specify the family. Usually, the 2 bit fields use a logical OR for this parameter. Table 17-8 lists the symbolic constants recognized by CreateFont() for the font pitch and the family values.

Table 17-8

Pitch and Family Predefined Constants

TYPE	VALUE	MEANING
PITCH:		
	DEFAULT_PITCH	
	FIXED_PITCH	
	VARIABLE_PITCH	
FAMILY:		
	FF_DECORATIVE	Novelty fonts (such as Old English)
	FF_DONTCARE	Don't care or don't know.
	FF_MODERN	Fonts with constant stroke width, with or without serifs, such as Pica, Elite, and Courier New.
	FF_ROMAN	Fonts with variable stroke width and with serifs, such as MS Serif.
	FF_SCRIPT	Fonts designed to look like handwriting, such as Script and Cursive.
	FF_SWISS	Fonts with variable stroke width and without serifs, such as MS Sans Serif.

lpszFace (LPCTSTR) points to a null-terminated string that contains the name of the font's typeface. Alternatively, the typeface name can be entered directly inside double quotation marks. If the requested typeface is not available in the system, the font mapper substitutes with an approximate one. If NULL is entered in this field, a default typeface is used. Example typefaces are Palatino, Times New Roman, and Arial. The following code fragment shows a call to the CreateFont() API for a 50-point, normal weight, high quality, italic font using the Times New Roman typeface.

```
HFONT      hFont;       // handle to a font
// Create a logical font
hFont = CreateFont (
    50 * GetDeviceCaps (hdc, LOGPIXELSY) / 72, //height
    0,                        // width
```

```
        0,                          // escapement angle
        0,                          // orientation angle
        FW_NORMAL,                  // weight
        1,                          // italics
        0,                          // not underlined
        0,                          // not strikeout
        DEFAULT_CHARSET,            // character set
        OUT_DEFAULT_PRECIS,         // precision
        CLIP_DEFAULT_PRECIS,        // clipping precision
        PROOF_QUALITY,              // quality
        DEFAULT_PITCH | FF_DONTCARE, // pitch and family
        "Times New Roman");          // typeface name

    // Select font into the display context
    SelectObject (hdc, hFont);
```

17.5.2 Drawing with Text

Once a TrueType font is selected in the display context, you can execute several manipulations that treat text characters as graphics objects. One of them is related to the notion of a path, introduced in Windows NT and also supported by Windows 95 and later. A path is the outline generated by one or more graphics objects drawn between the BeginPath() and EndPath() functions. Paths are related to regions and to clipping, topics covered in detail in Chapter 20.

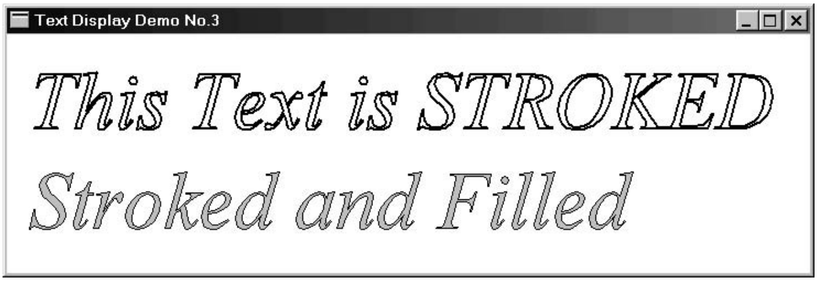
The TextOut() function has a unique property among the text display functions: it generates a path. For this to work, a TrueType font must first be selected into the display context. Path drawing operations are not immediately displayed on the screen but are stored internally. Windows provides no handles to paths, and there is only one path for each display context. Three functions are available to display graphics in a path: StrokePath() shows the path outline, FillPath() fills and displays the path's interior, and StrokeAndFillPath() performs both functions. You may question the need for a FillAndStrokePath() function since it seems that you could use StrokePath() and FillPath() consecutively to obtain the same effect. This is not the case. All three path-drawing APIs automatically destroy the path. Therefore, if two of these functions are called consecutively, the second one has no effect.

The path itself has a background mix mode, which is delimited by the rectangle that contains the graphics functions in the path. The background mix mode is a display context attribute that affects the display of text, as well as the output of hatched brushes and nonsolid pens. Code can set the background mix mode to transparent by means of the SetBkMode() function. This isolates the text from the background. The program TEX3_DEMO, located in the Text Demo No 3 folder in the book's software package, is a demonstration of text display inside paths. One of the text lines is stroked and the other one is stroked and filled. The program first creates a logical font and then selects it into the display context. Processing is as follows:

```
case WM_PAINT :
    hdc = BeginPaint (hwnd, &ps) ;
    .
    .
    .
    // Start a path for stroked text
    // Set background mix to TRANSPARENT mode
    BeginPath (hdc);
    SetBkMode(hdc, TRANSPARENT);        // background mix
    TextOut(hdc, 20, 20, "This Text is STROKED", 20);
    EndPath(hdc);
    // Create a custom black pen, 2 pixels wide
    aPen = CreatePen(PS_SOLID, 2, 0);
    SelectObject(hdc, aPen);            // select it into DC
    StrokePath (hdc);                   // Stroke the path

    // Second path for stroked and filled text
    BeginPath (hdc);
    SetBkMode(hdc, TRANSPARENT);
    TextOut(hdc, 20, 110, "Stroked and Filled", 18);
    EndPath(hdc);
    // Get and select a stock pen and brush
    aPen = GetStockObject(BLACK_PEN);
    aBrush = GetStockObject(LTGRAY_BRUSH);
    SelectObject(hdc, aPen);
    SelectObject(hdc, aBrush);
    StrokeAndFillPath (hdc);          // Stroke and fill path

    // Clean-up and end WM_PAINT processing
    DeleteObject(hFont);
    EndPaint (hwnd, &ps);
```

Figure 17-8 is a screen snapshot of the TEXTDEM3 program.

Figure 17-8 *Screen Snapshot of the TEXTDEM3 Program*



This Text is STROKED

Stroked and Filled

Chapter 18

Keyboard and Mouse Programming

Topics:

- Keyboard input and input focus
- Keystroke processing
- The caret
- Mouse programming
- Mouse messages
- The cursor

Most applications require user input and control operations. The most common input devices are the keyboard and the mouse. In this chapter we discuss keyboard and mouse programming in Windows.

18.1 Keyboard Input

Since the first days of computing, typing on a typewriter-like keyboard has been an effective way of interacting with the system. Although typical Windows programs rely heavily on the mouse device, the keyboard is the most common way to enter text characters into an application.

The mechanisms by which Windows monitors and handles keyboard input are based on its message-driven architecture. When the user presses or releases a key, the low-level driver generates an interrupt to inform Windows of this action. Windows then retrieves the keystroke from a hardware register, stores it in the system message queue, and proceeds to examine it. The action taken by the operating system depends on the type of keystroke, and on which application currently holds the keyboard foreground, called the input focus. The keystroke is dispatched to the corresponding application by means of a message to its Windows procedure.

The particular way by which Windows handles keystrokes is determined by its multitasking nature. At any given time, several programs can be executing simultaneously, and any one of these programs can have more than one thread of execution.

One of the possible results of a keystroke (or a keystroke sequence) is to change the thread that holds the input focus, perhaps to a different application. This is the reason why Windows cannot directly send keyboard input to any specific thread.

It is the message loop in the WinMain() function of an application that retrieves keyboard messages from the system queue. In fact, all system messages are posted to the application's message queue. The process makes the following assumptions: first, that the thread's queue is empty; second, that the thread holds the input focus; and third, that a keystroke is available at the system level. In other words, it is the application that asks Windows for keystrokes; Windows does not send unsolicited keystroke data.

The abundance of keyboard functions and keyboard messages makes it appear that Windows keyboard programming is difficult or complicated. The fact is that applications do not need to process all keyboard messages, and hardly ever do so. Two messages, WM_CHAR and WM_KEYDOWN, usually provide code with all the necessary data regarding user keyboard input. Many keystrokes can be ignored, since Windows generates other messages that are more easily handled. For example, applications can usually disregard the fact that the user selected a menu item by means of a keystroke, since Windows sends a message to the application as if the menu item had been selected by a mouse click. If the application code contains processing for menu selection by mouse clicks, then the equivalent keyboard action is handled automatically.

18.1.1 Input Focus

The application that holds the input focus is the one that gets notified of the user's keystrokes. A user can visually tell which window has the input focus since it is the one whose title bar is highlighted. This applies to the parent window as well as to child windows, such as an input or dialog box. The application can tell if a window has the input focus by calling the GetFocus() function, which returns the handle to the window with the input focus.

The Windows message WM_SETFOCUS is sent to the window at the time that it receives the input focus, and WM_KILLFOCUS at the time it loses it. Applications can intercept these messages to take notice of any change in the input focus. However, these messages are mere notifications; application code cannot intercept these messages to prevent losing the input focus.

Keyboard data is available to code holding the input focus at two levels. The lower level, usually called keystroke data, contains raw information about the key being pressed. Keystroke data allows code to determine whether the keystroke message was generated by the user pressing a key or by releasing it, and whether the keystroke resulted from a normal press-and-release action or from the key being held down (called typematic action). Higher-level keyboard data relates to the character code associated with the key. An application can intercept low-level or character-level keystroke messages generated by Windows.

18.1.2 Keystroke Processing

Four Windows messages inform application code of keystroke data: WM_KEYDOWN, WM_SYSKEYDOWN, WM_KEYUP, and WM_SYSKEYUP. The keydown-type messages are generated when a key is pressed, sometimes called the make action. The keyup-type messages are generated when a key is released, called the break action. Applications usually ignore the keyup-type message. The "sys-type" messages, WM_SYSKEYDOWN and WM_SYSKEYUP, relate to system keys. A system keystroke is one generated while the Alt key is held down.

When any one of these four messages takes place, Windows puts the keystroke data in the lParam and wParam passed to the window procedure. The lParam contains bit-coded information about the keystroke, as shown in Table 18-1.

Table 18-1

Bit and Bit Fields in the lParam of a Keystroke Message

BITS	MEANING
0-15	Repeat count field. The value is the number of times the keystroke is repeated as a result of the user holding down the key (typematic action).
16-23	OEM scan code. The value depends on the original equipment manufacturer.
24	Extended key. Bit is set when the key pressed is one duplicated in the IBM Enhanced 101- and 102-key keyboards, such as the right-hand ALT and CTRL keys, the / and Enter keys on the numeric keypad, or the Insert, Delete, Home, PageUp, PageDown, and End keys.
25-28	Reserved.
29	Context code. Bit is set if the Alt key is down while the key is pressed. Bit is clear if the WM_SYSKEYDOWN message is posted to the active window because no window has the keyboard focus.
30	Previous key state. Key is set if the key is down before the message is sent. Bit is clear if the key is up. This key allows code to determine if the keystroke resulted from a make or break action.
31	Transition state. Always 0 for a WM_SYSKEYDOWN Message.

The wParam contains the virtual-key code, which is a hardware-independent value that identifies each key. Windows uses the virtual-key codes instead of the device-dependent scan code. Typically, the virtual-key codes are processed when the application needs to recognize keys that have no associated ASCII value, such as the control keys. Table 18-2, on the following page, lists some of the most used virtual-key codes.

Notice that originally, the "w" in wParam stood for "word," since in 16-bit Windows the wParam was a word-size value. The Win32 API expanded the wParam from 16 to 32 bits. However, in the case of the virtual-key character codes, the wParam is defined as an int type. Code can typecast the wParam as follows:

Table 18-2

Virtual-Key Codes

SYMBOLIC NAME	HEX VALUE	KEY
VK_CANCEL	0x01	Ctrl + Break
VK_BACK	0x08	Backspace
VK_TAB	0x09	Tab
VK_RETURN	0x0D	Enter
VK_SHIFT	0x10	Shift
VK_CONTROL	0x11	Ctrl
VK_MENU	0x12	Alt
VK_PAUSE	0x13	Pause
VK_CAPITAL	0x14	Caps Lock
VK_ESCAPE	0x1B	Esc
VK_SPACE	0x20	Spacebar
VK_PRIOR	0x21	Page Up
VK_NEXT	0x22	Page Down
VK_END	0x23	End
VK_HOME	0x24	Home
VK_LEFT	0x25	Left arrow
VK_UP	0x26	Up arrow
VK_RIGHT	0x27	Right arrow
VK_DOWN	0x28	Down arrow
VK_SNAPSHOT	0x2C	Print Screen
VK_INSERT	0x2D	Insert
VK_DELETE	0x2E	Delete
VK_MULTIPLY	0x6A	Numeric keypad *
VK_ADD	0x6B	Numeric keypad +
VK_SUBTRACT	0x6D	Numeric keypad -
VK_DIVIDE	0x6F	Numeric keypad /
VK_F1..VK_F12	0x70..0x7B	F1 .. F12

```
int        aKeystroke;
char       aCharacter;
.
.
aKeystroke = (int) wParam;
aCharacter = (char) wParam;
```

Simple keystroke processing can be implemented by intercepting WM_KEYDOWN. Occasionally, an application needs to know when a system-level message is generated. In this case, code can intercept WM_SYSKEYDOWN. The first operation performed in a typical WM_KEYDOWN or WM_SYSKEYDOWN handler is to store in local variables the lParam, the wParam, or both. In the case of the wParam code can cast the 32-bit value into an int or a char type as necessary (see the preceding Tech Note).

Processing the keystroke usually consists of performing bitwise operations in order to isolate the required bits or bit fields. For example, to determine if the extended key flag is set, code can logically AND with a mask in which bit 24 is set and then test for a non-zero result, as in the following code fragment:

```
unsigned long     keycode;
.
.
WM_KEYDOWN:
    keycode = lParam;           // store lParam
    if(keycode & 0x01000000) {  // test bit 24
    // ASSERT:
    //   key pressed is extended key
```

Processing the virtual-key code, which is passed to your intercept routine in the lParam, consists of comparing its value with the key or keys that you wish to detect. For example, to know if the key pressed was the Backspace, you can proceed as in the following code fragment:

```
int     virtkey;
.
.
WM_KEYDOWN:
    virtkey = (int) lParam;      // cast and store lParam
    if(virtkey == VK_BACK) {     // test for Backspace
    // ASSERT:
    //   Backspace key pressed
```

18.1.3 Determining the Key State

An application can determine the state of any virtual-key by means of the GetKeyState() service. The function's general form is as follows:

```
SHORT GetKeyState(nVirtKey);
```

GetKeyState() returns a SHORT integer with the high-order bit set if the key is down and the low-order bit set if it is toggled. Toggle keys are those which have a keyboard LED to indicate their state: Num Lock, Caps Lock, and Scroll Lock. The LED for the corresponding key is lit when it is toggled and unlit otherwise. Some virtual-key constants can be used as the nVirtKey parameter of GetKeyState(). Table 18-3, on the following page, lists the virtual-keys.

Take note that in testing for the high-bit set condition returned by GetKeyState() you may be tempted to bitwise AND with a binary mask, as follows:

```
     if(0x8000 & (GetKeyState(VK_SHIFT))) {
```

Table 18-3

Virtual-Keys Used in GetKeyState()

PREDEFINED SYMBOL	KEY	RETURNS
VK_SHIFT	Shift	State of left or right Shift keys
VK_CONTROL	Ctrl	State of left or right Ctrl keys
VK_MENU	Alt	State of left or right Alt keys
VK_LSHIFT	Shift	State of left Shift key
VK_RSHIFT	Shift	State of right Shift key
VK_LCONTROL	Ctrl	State of left Ctrl key
VK_RCONTROL	Ctrl	State of right Ctrl key
VK_LMENU	Alt	State of left Alt key
VK_RMENU	Alt	State of right Alt key

The following statement is a test for the left Shift key pressed.

```
if(GetKeyState(VK_LSHIFT) < 0) {
// ASSERT:
//       Left shift key is pressed
```

Although, in many cases, such operations produce the expected results, its success depends on the size of a data type, which compromises portability. In other words, if GetKeyState() returns a 16-bit integer, then the mask 0x8000 effectively tests the high-order bit. If the value returned is stored in 32 bits, however, then the mask must be the value 0x80000000. Since any signed integer with the high-bit set represents a negative number, it is possible to test the bit condition as follows:

```
if(GetKeyState(VK_SHIFT) < 0) {
```

This test does not depend on the operand's bit size.

18.1.4 Character Code Processing

Applications often deal with keyboard input as character codes. It is possible to obtain the character code from the virtual-key code since it is encoded in the wParam of the WM_KEYDOWN, WM_SYSKEYDOWN, WM_KEYUP, and WM_SYSKEYUP messages. The codes for the alphanumeric keys are not listed in Table 18-1; however, there is also a virtual-key code for each one. The virtual-key codes for the numeric keys 0 to 9 are VK_0 to VK_9, and the ones for the alphabetic characters A through Z are VK_A through VK_Z.

This type of processing is not without complications. For example, the virtual-key code for the alphabetic characters does not specify if the character is in upper- or lower-case. Therefore, the application would have to call GetKeyState() in order to determine if the <Shift> key was down or the Caps Lock key toggled when the character key was pressed. Furthermore, the virtual-key codes for some of the character keys, such as ;, =, +, <, are not defined in the windows header files. Applications must use the numeric values assigned to these keys or define their own symbolic constants.

Fortunately, character code processing in Windows is much easier. The TranslateMessage() function converts the virtual-key code for each character into its ANSI (or Unicode) equivalent and posts it in the thread's message queue. TranslateMessage() is usually included in the program's message loop. After TranslateMessage(), the message is retrieved from the queue, typically by GetMessage() or PeekMessage(). The final result is that an application can intercept WM_CHAR, WM_DEADCHAR, WM_SYSCHAR, and WM_SYSDEADCHAR in order to obtain the ANSI character codes that correspond to the virtual-key of a WM_KEYDOWN message.

Dead-type character messages refer to the diacritical characters used in some foreign language keyboards. These are marks added to characters to distinguish them from other ones, such as the acute accent (á) or the circumflex (â). In English language processing, WM_DEADCHAR and WM_SYSDEADCHAR are usually ignored.

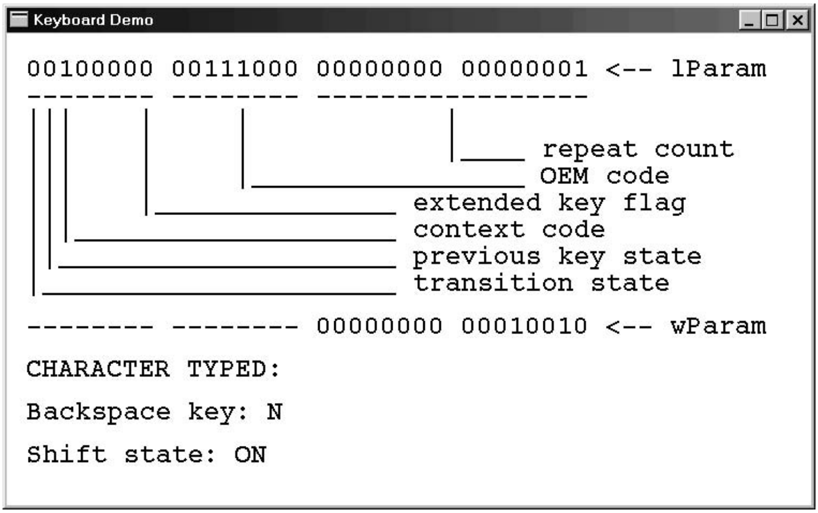
The WM_SYSCHAR message corresponds to the virtual-key that results from WM_SYSKEYDOWN. WM_SYSCHAR is posted when a character key is pressed while the Alt key is held down. Since Windows also sends the message that corresponds to a mouse click on the system item, applications often ignore WM_SYSCHAR.

This leaves us with WM_CHAR for general purpose character processing. When the WM_CHAR message is sent to your Windows procedure, the lParam is the same as for WM_KEYDOWN. However, the wParam contains the ANSI code for the character, instead of the virtual-key code. This ANSI code, which is approximately equivalent to the ASCII code, can be directly handled and displayed without additional manipulations. Processing is as follows:

```
  char      aChar;           // storage for character
  .
  .
  case WM_CHAR:
     aChar = (char) wParam;
// ASSERT:
//     aChar holds ANSI character code
```

18.1.4 Keyboard Demonstration Program

The program KBR_DEMO.CCP, located in the Keyboard Demo folder on the book's software package, is a demonstration of the keyboard processing routines described previously. The program uses a private device context; therefore, the font is selected once, during WM_CREATE processing. KBR_DEMO uses a typewriter-like, TrueType font, named Courier. Courier is a monospaced font (all characters are the same width). This makes possible the use of standard character symbols to produce a graph of the bitmaps. Figure 18-1, on the following page, is a screen snapshot of the KBD_DEMO program.



```
00100000 00111000 00000000 00000001 <-- lParam
-------- -------- -----------------
|||     |      |              |
|||     |      |              |____ repeat count
|||     |      |__________________ OEM code
|||     |__________________ extended key flag
||______________________ context code
||______________________ previous key state
|_______________________ transition state
-------- -------- 00000000 00010010 <-- wParam

CHARACTER TYPED:

Backspace key: N

Shift state: ON
```

Figure 18-1 *KBR_DEMO Program Screen*

Figure 18-1 shows the case in which the user has typed the Alt key. Note that the wParam value 00010010 is equivalent to 0x12, which is the virtual-key code for the Alt key (see Table 18-1). The critical processing in the KBD_DEMO program is as follows:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                          LPARAM lParam) {

    static int  cxChar, cyChar ;     // Character dimensions
    static int  cxClient, cyClient;  // Client area parameters
    static HDC       hdc ;           // handle to private DC
    unsigned long    keycode;        // storage for keystroke
    unsigned long    keymask;        // bit mask
    unsigned int     virtkey;        // virtual-key
    int              i, j;           // counters
    char             aChar;          // character code

    // Structures
    PAINTSTRUCT ps;
    TEXTMETRIC  tm;
    RECT        textRect;            // RECT-type
    HFONT       hFont;
    .
    .
    .
    case WM_PAINT :
        // Processing consists of displaying the text messages
        BeginPaint (hwnd, &ps) ;
         // Initialize rectangle structure
         SetRect (&textRect,          // address of structure
            2 * cxChar,               // x for start
            cyChar,                   // y for start
            cxClient -(2 * cxChar),   // x for end
            cyClient);                // y for end
         // Display multi-line text string
```

```
     DrawText( hdc, TextStr0, -1, &textRect,
        DT_LEFT | DT_WORDBREAK);

     // Display second text string
     SetRect (&textRect,          // address of structure
        2 * cxChar,               // x for start
        13 * cyChar,              // y for start
        cxClient -(2 * cxChar),   // x for end
        cyClient);                // y for end
     // Display text string
        DrawText( hdc, TextStr1, -1, &textRect,
        DT_LEFT | DT_WORDBREAK);
        .
        .
        .
        EndPaint (hwnd, &ps);
        return 0 ;

// Character code processing
case WM_CHAR:
     aChar = (char) wParam;
     // Test for control codes and replace with space
     if (aChar < 0x30)
        aChar = 0x20;

     // Test for shift key pressed
     if(GetKeyState (VK_SHIFT) < 0) {
        i = 0;        // counter
        j = 13;       // string offset
        for(i = 0; i < 3; i++){
           TextStr4[j] = StrON[i];
           j++;
        }
      }
      else {
        i = 0;        // counter
        j = 13;       // string offset
        for(i = 0; i < 3; i++){
           TextStr4[j] = StrOFF[i];
           j++;
        }
      }

    TextStr2[17] = aChar;
    return 0;

// Scan code and keystroke data processing
// Display space if a system key
case WM_SYSKEYDOWN:
     TextStr2[17] = 0x20;

case WM_KEYDOWN:
// Store bits for lParam in TextStr0[]
   keycode = lParam;    // get 32-bit keycode value
   i = 0;               // counter for keystroke bits
   j = 0;               // offset into string
   keymask = 0x80000000;// bitmask

   for (i = 0; i < 32; i++) {
     // Test for separators and skip
```

```
          if(i == 8 || i == 16 || i == 24) {
             TextStr0[j] = 0x20;
             j++;
          }
         // Test for 1 and 0 bits and display digits
         if(keycode & keymask)
            TextStr0[j] = '1';
         else
            TextStr0[j] = '0';
         keymask = keymask >> 1;
         j++;
         }

     // Store bits for wParam in TextStr1[]
        keycode = wParam;     // get 32-bit keycode value
        i = 0;                // counter for keystroke bits
        j = 18;               // initial offset into string
        keymask = 0x8000;     // bitmask

        // 16-bit loop
          for (i = 0; i < 16; i++) {
            // Test for separators and skip
            if(i == 8) {
               TextStr1[j] = 0x20;
               j++;
            }
          // Test for 1 and 0 bits and display digits
            if(keycode & keymask)
               TextStr1[j] = '1';
            else
               TextStr1[j] = '0';
               keymask = keymask >> 1;
               j++;
            }

     // Test for Backspace key pressed
     virtkey = (unsigned int) wParam;
           if (virtkey == VK_BACK)
              TextStr3[15] = 'Y';
            else
              TextStr3[15] = 'N';

     // Force WM_PAINT message
     InvalidateRect(NULL, NULL, TRUE);
     return 0;
     .
     .
     .
```

18.2 The Caret

In the MS DOS environment, the graphic character used to mark the screen position at which typed characters are displayed is called the cursor. The standard DOS cursor is a small, horizontal bar that flashes on the screen to call the user's attention to the point of text insertion. In Windows, the word cursor is used for an icon that marks the screen position associated with mouse-like pointing. Windows applications signal the location where keyboard input is to take place by means of a flashing, vertical bar called the caret.

In order to avoid confusion and ambiguity, Windows displays a single caret. The system caret, which is a shared resource, is a bitmap that can be customized by the application. The window with the input focus can request the caret to be displayed in its client area, or in a child window.

18.2.1 Caret Processing

Code can intercept the WM_SETFOCUS message to display the caret. WM_KILLFOCUS notifies the application that it has lost focus and that it should therefore destroy the caret. Caret display and processing in WM_SETFOCUS usually starts by calling CreateCaret(). The function's general form is as follows:

```
BOOL CreateCaret(hwnd, hBitmap, nWidth, nHeight);
```

The first parameter is the handle to the window that owns the caret. The second one is an optional handle to a bitmap. If this parameter is NULL then a solid caret is displayed. If it is (HBITMAP) 1, then the caret is gray. If it is a handle to a bitmap, the other parameters are ignored and the caret takes the form of the bitmap. The last two parameters define the caret's width and height, in logical units. Applications often determine the width and height of the caret in terms of character dimensions.

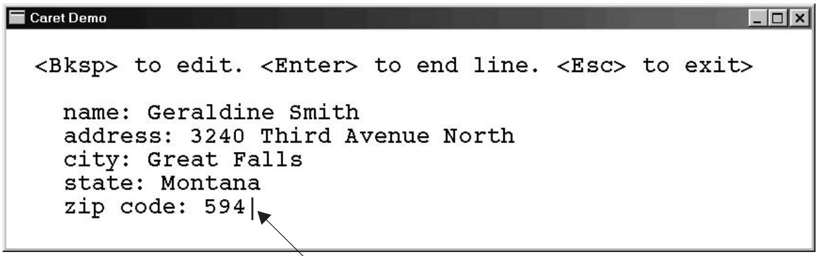
CreateCaret() defines the caret shape and size but does not set its screen position, nor does it display it. To set the caret's screen position you use the SetCaretPos() function, which takes two parameters, the first one for the caret's x-coordinate and the second one for the y-coordinate. The caret is displayed on the screen using ShowCaret(), whose only argument is the handle to the window.

Applications that use the caret usually intercept WM_KILLFOCUS. This ensures that they are notified when the window looses the keyboard focus, at which time the caret must be hidden and destroyed. The HideCaret() function takes care of the first action. Its only parameter is the handle to the window that owns the caret. DestroyCaret(), which takes no parameters, destroys the caret, erases it from the screen, and breaks the association between the caret and the window.

Applications that use the caret to signal the point of input often display the characters typed by the user. But since the caret is a graphics object, it must be erased from the screen before the character is displayed. Otherwise, the caret symbol itself, or parts of it, may pollute the screen. A program that processes the WM_CHAR message to handle user input usually starts by hiding the caret, then the code processes the input character, and finally, resets the caret position and redisplays it.

18.2.2 Caret Demonstration Program

The CAR_DEMO program, located in the Caret Demo folder on the book's software package, is a demonstration of caret processing during text input. The program displays an entry form and uses the caret to signal the current input position. When the code detects the Enter key, it moves to the next line in the entry form. The Backspace key can be used to edit the input. When Backspace is pressed, the previous character is erased and the caret position is updated. Program logic keeps track of the start location of each input line so that the user cannot backspace past this point. The Esc key erases the caret and ends input. Note that since user input is not stored by the program, the text is lost if the screen is resized or if the application looses the input focus. Figure 18-2 is a screen snapshot of the CAR_DEMO program.



```
<Bksp> to edit. <Enter> to end line. <Esc> to exit>

  name: Geraldine Smith
  address: 3240 Third Avenue North
  city: Great Falls
  state: Montana
  zip code: 594|
```

caret signals input location

Figure 18-2 *CAR_DEMO Program Screen*

Figure 18-2 shows execution of the CAR_DEMO program. The following are excerpts of the program's processing:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                          LPARAM lParam) {
 static int  cxChar, cyChar ;     // character dimensions
 static int  cxClient, cyClient; // client area parameters
 static int  xCaret, yCaret;     // caret position
 static int  xLimit;             // left limit of line
 static int  formEnd = 0;        // 1 if Esc key pressed
 static int  lineNum = 1;        // input line
 static HDC  hdc ;               // handle to private DC
 char        aChar;              // storage for character code

 // Structures
 PAINTSTRUCT ps;
 TEXTMETRIC  tm;
 RECT        textRect;
 HFONT       hFont;

 switch (iMsg) {

     case WM_CREATE :
          .
          .
          .
         // Calculate and store character dimensions
         cxChar = tm.tmAveCharWidth ;
         cyChar = tm.tmHeight + tm.tmExternalLeading ;
         // Store size of client area
         cxClient = LOWORD(lParam);
         cyClient = HIWORD(lParam);
         // Store initial caret position
         xCaret = xLimit = 10;
         yCaret = 3;
         return 0 ;
         .
         .
         .
      case WM_PAINT :
```

```
  BeginPaint (hwnd, &ps) ;
  // Initialize rectangle structure
  SetRect (&textRect,         // address of structure
         2 * cxChar,                // x for start
         cyChar,                    // y for start
         cxClient -(2 * cxChar),    // x for end
         cyClient);                 // y for end
  // Display multi-line text string
  DrawText( hdc, TextStr1, -1, &textRect,
            DT_LEFT | DT_WORDBREAK);

  EndPaint (hwnd, &ps);
  return 0 ;
  // Character input processing
 case WM_CHAR:

 HideCaret(hwnd);
 aChar = (char) wParam;

 switch (wParam) {   // wParam holds virtual-key code
 case '\r':          // Enter key pressed
      yCaret++;
      aChar = 0x20;
 // cascaded tests set x caret location in new line
      if(yCaret == 4)    // in address: line
         xCaret = xLimit = 13;

      if(yCaret == 5)    // in city: line
         xCaret = xLimit = 10;

      if(yCaret == 6)    // in state: line
         xCaret = xLimit = 11;

      if(yCaret == 7)    // in zip code: line
         xCaret = xLimit = 14;

      if(yCaret > 7) {    // Enter key ignored on
                          // last line
         yCaret--;
 }
 break;

case '\b':        // Backspace key pressed
  if (xCaret > xLimit) {
      aChar = 0x20;       // Replace with space
      xCaret--;
      // Display the blank character
      TextOut (hdc, xCaret * cxChar, yCaret * cyChar,
               &aChar, 1);
        }
     break;

case 0x1b:      // Esc key processing
     formEnd = 1;
     // Destroy the caret
     HideCaret(hwnd);
     DestroyCaret();
     break;

default:
```

```
           // Display the character if Esc not pressed
           if(formEnd == 0) {
              TextOut (hdc, xCaret * cxChar, yCaret * cyChar,
              &aChar, 1);
                xCaret++;
             }
          break;
        }
         if(formEnd == 0) {
            SetCaretPos(xCaret * cxChar, yCaret * cyChar);
            ShowCaret(hwnd);
        }
        return 0;

      case WM_SETFOCUS:
           if(formEnd == 0) {
              CreateCaret (hwnd, NULL, cxChar / 4, cyChar);
              SetCaretPos(xCaret * cxChar, yCaret * cyChar);
              ShowCaret(hwnd);
           }
           return 0;

      case WM_KILLFOCUS:
           // Destroy the caret
           HideCaret(hwnd);
           DestroyCaret();
           return 0;

      . . .
```

18.3 Mouse Programming

The use of a mouse as an input device dates back to the work at Xerox PARC, which pioneered the ideas of a graphical user interface. Since mouse and GUI have been interrelated since their original conception, one would assume that a graphical operating system, such as Windows, would require the presence of a mouse device. This is not the case. Windows documentation still considers the mouse an option and recommends that applications provide alternate keyboard controls for all mouse-driven operations.

During program development, you can make sure that a mouse is available and operational by means of the GetSystemMetrics() function, as follows:

```
assert (GetSystemMetrics(SM_MOUSEPRESENT));
```

In this case, the assert macro displays a message box if a mouse is not present or not operational. The developer can then choose to ignore the message, debug the code, or abort execution. In the release version of a program that requires a mouse you can use the abort macro to break execution. For example:

```
if (!GetSystemMetrics(SM_MOUSEPRESENT))
   abort();
```

Alternatively, an application can call PostQuitMessage(). This indicates to Windows that a thread has made a termination request and it posts a WM_QUIT message. PostQuitMessage() has an exit code parameter that is returned to Windows, but current versions of the operating system make no use of this value. The objection to using PostQuitMessage() for abnormal terminations is that execution ends abruptly, without notification of cause or reason. In this case the program should display a message box informing the user of the cause of program termination.

Windows supports other devices such as pens, touch screens, joysticks, and drawing tablets, which are all considered mouse input. The mouse itself can have up to three buttons, labeled left, middle, and right buttons. A one-button mouse is an anachronism and the three-button version is usually associated with specialized systems. The most common one is the two-button mouse, where the left button is used for clicking, double-clicking, and dragging operations and the right button activates context-sensitive program options.

An application can tell how many buttons are installed in the mouse by testing the SM_CMOUSEBUTTONS with the GetSystemMetrics() function. If the application requires a certain number of buttons, then the assert or abort macros can be used, as previously shown. For example, a program that requires a three-button mouse could test for this condition as follows:

```
 assert (GetSystemMetrics(SM_CMOUSEBOUTTONS) == 3);
```

If the three-button mouse is required in the release version of the program, then the code could be as follows:

```
if(GetSystemMetrics(SM_CMOUSEBUTTONS) != 3))
   abort();
```

Notice that the assert macro is intended to be used in debugging. If the condition is false, the macro shows information about the error and displays a message box with three options: abort, debug, and ignore. Assert has no effect on the release version of the program; it is as if the statement containing assert had been commented out of the code. For this reason conditions that must be evaluated during execution of the release version of a program should not be part of an assert statement.

The abort macro can be used to stop execution in either version. Abort provides no information about the cause of program termination.

Programs that use the assert macro must include the file assert.h. VERIFY and other debugging macros are available when coding with the Foundation Class Library, but they are not implemented in ANSI C.

18.3.1 Mouse Messages

There are 22 mouse messages currently implemented in the Windows API. Ten of these messages refer to mouse action on the client area, and ten to mouse action in the nonclient area. Of the remaining two messages WM_NCHITTEST takes place when the mouse is moved either over the client or the nonclient area. It is this message that generates all the other ones. WM_MOUSEACTIVATE takes place when a mouse button is pressed over an inactive window, an event that is usually ignored by applications.

The abundance of Windows messages should not lead you to think that mouse processing is difficult. Most applications do all their mouse processing by intercepting two or three of these messages. Table 18-4 lists the mouse messages most frequency handled by applications.

Table 18-4

Frequently Used Client Area Mouse Messages

MOUSE MESSAGE	DESCRIPTION
WM_LBUTTONDOWN	Left button pressed
WM_LBUTTONUP	Left button released
WM_RBUTTONDOWN	Right button pressed
WM_RBUTTONUP	Right button released
WM_RBUTTONDBLCLK	Right button double-clicked
WM_LBUTTONDBLCLK	Left button double-clicked
WM_MOUSEMOVE	Mouse moved into client area

In Table 18-4 lists only client area mouse messages; nonclient area messages are usually handled by the default windows procedure.

Mouse processing is similar to keyboard processing, although mouse messages do not require that the window have the input focus. Once your application gains control in a mouse message handler, it can proceed to implement whatever action is required. However, there are some differences between keyboard messages and mouse messages. To Windows, keyboard input is always given maximum attention. The operating system tries to assure that keyboard input is always preserved. Mouse messages, on the other hand, are expendable. For example, the WM_MOUSEMOVE message, which signals that the mouse cursor is over the application's client area, is not sent while the mouse is over every single pixel of the client area. The actual rate depends on the mouse hardware and on the processing speed. Therefore, it is possible, given a small enough client area and a slow enough message rate, that code may not be notified of a mouse movement action over its domain. Mouse programming must take this possibility into account.

In client area mouse messages, the wParam indicates which, if any, keyboard or mouse key was held down while the mouse action took place. Windows defines five symbolic constants to represent the three mouse keys and the keyboard Ctrl and Shift keys. These constants are listed in Table 18-5.

Table 18-5

Virtual Key Constants for Client Area Mouse Messages

CONSTANT	ORIGINATING CONDITION
MK_CONTROL	Ctrl key is down.
MK_LBUTTON	Left mouse button is down.
MK_MBUTTON	Middle mouse button is down.
MK_RBUTTON	Right mouse button is down.
MK_SHIFT	Shift key is down.

Code can determine if one of the keys was held down by ANDing with the corresponding constant. For example, the following fragment can be used to determine if the Ctrl key was held down at the time that the left mouse button was clicked in the client area:

```
case WM_LBUTTONDOWN:
  if(wParam & MK_CONTROL) {
  // ASSERT:
  //   Left mouse button clicked and <Ctrl> key down
```

The predefined constants represent individual bits in the operand; therefore, you must be careful not attempt to equate the wParam with any one of the constants. For example, the MK_LBUTTON constant is always true in the WM_LBUTTONDOWN intercept, for this reason the following test always fails:

```
case WM_LBUTTONDOWN:
  if(wParam == MK_CONTROL) {
```

On the other hand, you can determine if two or more keys were held down by performing a bitwise OR of the predefined constants before ANDing with the wParam. For example, the following expression can be used to tell if either the Ctrl keys or the Shift keys were held down while the left mouse button was clicked:

```
if(wParam & (MK_CONTROL | MK_SHIFT)) {
   // ASSERT:
   //    Either the <Ctrl> or the <Shift> key was held down
   //    when the mouse action occurred
```

To test if both the <Ctrl> and the <Shift> keys were down when the mouse action occurred, you can code as follows:

```
if((wParam & MK_CONTROL) && (wParam & MKSHIFT)) {
   // ASSERT:
   //    The <Ctrl> and <Shift> key were both down when the
   //    mouse action occurred
```

18.3.2 Cursor Location

Applications often need to know the screen position of the mouse. In the case of the client area messages, the lParam encodes the horizontal and vertical position of the mouse cursor when the action takes place. The high-order word of the lParam contains the vertical mouse position and the low-order word the horizontal position. Code can use the LOWORD and HIWORD macros to obtain the value in logical units. For example:

```
int     cursorX, cursorY;     // Storage for coordinates
  .
  .
  .
case WM_MOUSEMOVE:
     cursorX = LOWORD(lParam)
     cursorY = HIWORD(lParam);
     // ASSERT:
     //   Variables now hold x and y cursor coordinates
```

18.3.3 Double-Click Processing

Handling mouse double-clicks requires additional processing as well as some forethought. In the first place, mouse double-click messages are sent only to windows that were created with the CS_DBLCLKS style. The CS_DBLCLKS style is described in Table 16-2. The structure of type WNDCLASSES for a windows that it to receive mouse double-clicks can be defined as follows:

```
// Defining a structure of type WNDCLASSEX
   WNDCLASSEX  wndclass ;
   wndclass.cbSize    = sizeof (WNDCLASSEX) ;
   wndclass.style     = CS_HREDRAW | CS_VREDRAW |
                        CS_DBLCLKS;
   .
   .
   .
```

Three client area mouse messages are related to the double-click action, one for each mouse button. If the window class includes the CS-DBLCLKS type, then client area double-click messages take place. WM_LBUTTONDBLCLK intercepts double-clicks for the left mouse button, WM_RBUTTONDBLCLK for the right mouse button, and WM_MBUTTONDBLCLK for the center button.

The double-click notification occurs when a mouse button is clicked twice within a predefined time interval. The double-click speed is set by selecting the Mouse Properties option in the Windows Control Panel. The SetDoubleClickTime() function can also be used to change the double-click interval from within an application, although it is not a good idea to do this without user participation. The default double-click time is 500 msec (one-half second). In addition, the two actions of a double-click must occur within a rectangular area defined by Windows, according to the display resolution. If the mouse has moved outside of this rectangle between the first and the second clicks, then the action is not reported as a double-click. The parameters for the double-click rectangle can be retrieved with the GetSystemMetrics() function, using the predefined constant SM_CXDOUBLECLK for the x-coordinate, and SM_CYDOUBLECLK for the y coordinate.

A double-click intercept receives control on the second click, because at the time of the first click it is impossible to know if a second one is to follow. Therefore, if the code intercepts normal mouse clicks, it also receives notification on the first click of a double-click action. For this reason, programs are usually designed so that the action taken as a result of a double-click is a continuation of the one taken on a single click. For example, selecting an application file in Windows Explorer by means of a single mouse click has the effect of highlighting the filename. If the user double-clicks, the file is executed. In this case the double-click action complements the single-click one. Although it is possible to implement double-click processing without this constraint, the programming is more complicated and the user interface becomes sluggish.

18.3.4 Capturing the Mouse

The mouse programming logic so far discussed covers most of the conventional programming required for handling mouse action inside the active window. By intercepting the client area messages, not the nonclient area ones, we avoid being notified of actions that usually do not concern our code. However, there are common mouse operations that cannot be implemented by processing client area messages only. For example, a Windows user installs a program icon on the desktop by right-clicking on the icon and then dragging it outside of the program group window. When the right mouse button is released, Windows displays a menu box that includes options to move or copy the program item, to create a shortcut, or to cancel the operation. In this case, the action requires crossing the boundary of the active window. Therefore, client area messages cease as soon as this boundary is reached.

Another case is a drawing program that uses a mouse dragging operation to display a rectangular outline. The rectangle starts at the point where the button is clicked, and ends at the point where the button is released. But what happens if the user crosses over the client area boundary before releasing the mouse button? In this case the application is not notified of the button release action since it occurs outside the client area. Furthermore, if the drawing action is performed during the WM_MOUSEMOVE intercept, the messages also stop being sent to the applications windows procedure as soon as the client area boundary is crossed. It would be a dangerous assumption to implement this function assuming that the user never crosses the boundary of the program's client area.

Problems such as these are solved by capturing the mouse, which is done by the SetCapture() function. The only parameter to SetCapture() is the handle of the capturing window. Once the mouse is captured, all mouse actions are assumed to take place in the client area, and the corresponding message intercepts in the application code are notified. The most obvious result of a mouse capture is that the client area message handlers are active for mouse actions that take place outside the client area. Only one window can capture the mouse, and it must be the active one, also called the foreground window. While the mouse is captured all system keyboard functions are disabled. The mouse capture ends with the call to ReleaseCapture(). GetCapture() returns the handle to the window that has captured the mouse, or NULL if the mouse capture fails.

Applications should capture the mouse whenever there is a possibility, even a remote one, of the user crossing the boundary of the client area during mouse processing. Implementing a simple drag-and-drop operation usually requires capturing the mouse. Mouse operations that take place between windows, whether they be child windows or not, also require capturing the mouse. Multitasking operations are limited during mouse capture. Therefore, it is important that the capture is released as soon as it is no longer necessary.

18.3.5 The Cursor

The screen image that corresponds to the mouse device is called the cursor. Windows provides 13 built-in cursors from which an application can select. In addition, you can create your own customized cursor and use it instead of a standard one. There are over 20 Windows functions that relate to cursor operations; however, even programs that manipulate cursor images hardly ever use more than a couple of them. Figure 18-3 shows the Windows built-in cursors and their corresponding symbolic names.

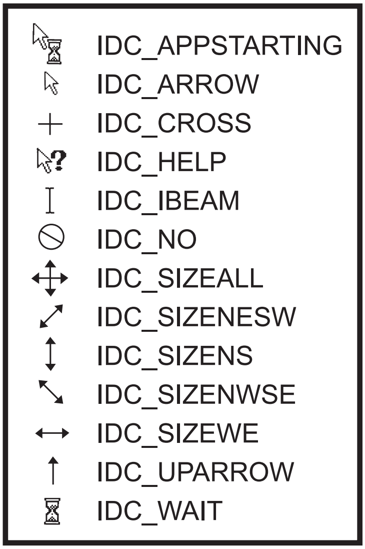


Figure 18-3 *Windows Built-In Cursors*

Code that manipulates cursor images must be aware of Windows cursor-handling operations. A mouse-related message not yet discussed is WM_SETCURSOR. This message is sent to your window procedure, and to the default window procedure, whenever a noncaptured mouse moves over the client area, or when its buttons are pressed or released. In the WM_SETCURSOR message, the wParam holds the handle to the window receiving the message. The low-order word of lParam is a code that allows determining where the action takes place, usually called the hit code. The high-order word of the lParam holds the identifier of the mouse message that triggered WM_SETCURSOR.

One of the reasons for WM_SETCURSOR is to give applications a chance to change the cursor; also for a parent window to manipulate the cursor of a child window. The problem is that Windows has a mind of its own regarding the cursor. If your application ignores the WM_SETCURSOR message, the default window procedure receives the message anyway. If Windows determines (from the hit code) that the cursor has moved over the client area of a window, then the default window procedure sets the cursor to the class cursor defined in the hCursor member of the WNDCLASSEX structure in WinMain(). If the cursor is in a nonclient area, then Windows sets it to the standard arrow shape.

What all of this means to your application code is that if you ignore the WM_SETCURSOR message, and don't take other special provisions, Windows continuously changes the cursor according to its own purposes, probably interfering with your own manipulations. The simplest solution is to intercept WM_SETCURSOR and return a nonzero value. In this case the window procedure halts all further cursor processing. You could also use the WM_SETCURSOR intercept to install your own cursor or cursors; however, the disadvantage of this approach is that WM_SETCURSOR does not provide information about the cursor's screen location.

An alternate method is to perform cursor manipulations at one of the mouse message intercepts, or any other message handler for that matter. For example, code can implement cursor changes at WM_MOUSEMOVE. In this case the lParam contains the cursor's horizontal and vertical position. Child windows can use this intercept to display their own cursors. In this case the hCursor field of the WNDCLASSEX structure is usually set to NULL, and the application takes on full responsibility for handling the cursor.

Applications that manipulate the cursor often start by setting a new program cursor during WM_CREATE processing. In cursor processing there are several ways of achieving the same purpose. The methods described are those that the authors have found more reliable. To create and display one of the built-in cursors you need a variable to store the handle to the cursor. The LoadCursor() and SetCursor() functions can then be used to load and display the cursor. To load and display the IDC_APPSTARTING cursor code can be as follows:

```
HCURSOR    aCursor;
.
.
.
aCursor = LoadCursor(NULL, IDC_APPSTARTING);
SetCursor (aCursor);
```

The first parameter of the LoadCursor() function is the handle to the program instance. This parameter is set to NULL to load one of the built-in cursors. Any of the symbolic names in Figure 18-3 can be used. The cursor is not displayed until the SetCursor() function is called, using the cursor handle as a parameter.

Graphics applications sometimes need one or more special cursors to suit their own needs. In the Visual C++ development environment, creating a custom cursor is made easy by the image editor. The process described for creating a program icon in Chapter 3, in the section, "Creating a Program Resource," is almost identical to the one for creating a custom cursor. Briefly reviewing:

1. In the Insert menu select the Resource command and then the Cursor resource type.
2. Use the editor to create a cursor. Note that all cursors are defined in terms of a 32 by 32 bit monochrome bitmap.
3. A button on the top bar of the editor allows positioning the cursor's hot spot. The default position for the hot spot is the upper left corner.
4. In the process of creating a cursor, Developer Studio also creates a new script file, or adds the information to an existing one. You must manually insert the script file into the project by selecting the Add to Project command from the Project menu and then selecting the Files option. In the "Insert Files into Project" dialog box select the script file and then click the OK button. The script file now appears on the Source Files list in the Files View window of the Project Workspace.

5. In addition to the script file, Developer Studio also creates a header file for resources. The default name of this file is resource.h. In order for resources to be available to the code you must enter an #include statement for the resource.h file in your source.

In order to use the custom cursor in your code you must know the symbolic name assigned to this resource, or its numeric value. The information can be obtained by selecting the Resource Symbols command from the View menu, or clicking the corresponding button on the toolbar.

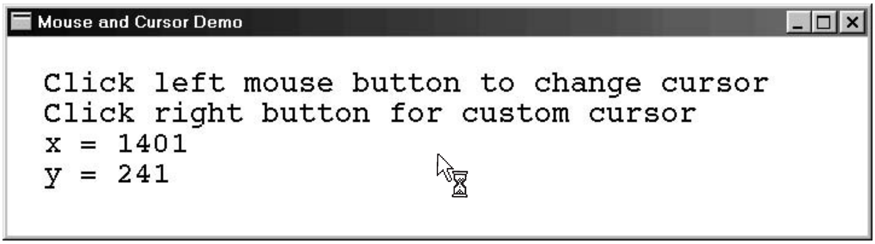
The LoadCursor() function parameters are different for a custom cursor than for a built-in one. In the case of a custom cursor, you must enter the handle to the instance as the first parameter, and use the MAKEINTRESOURCE macro to convert the numeric or symbolic value into a compatible resource type. For example, if the symbolic name of the custom cursor is IDC_CURSOR1, and the handle to the instance is stored in the variable pInstance (as is the case in the template files furnished in this book) you can proceed as follows:

```
HCURSOR        aCursor;      // handle to a cursor
.
.
.
aCursor = LoadCursor(pInstance,
                     MAKEINTRESOURCE(IDC_CURSOR1));
SetCursor(aCursor);
```

18.4 Mouse and Cursor Demonstration Program

The program named MOU_DEMO, located in the Mouse Demo project folder of the book's software package, is a demonstration of some of the mouse handling operations previously described. At this point in the book we have not yet covered the graphics services, or the implementation of user interface functions. For these reasons, it is difficult to find a meaningful demonstration for mouse operations.

MOU_DEMO monitors the left and the right mouse buttons. Clicking the left button changes to one of the built-in cursors. The cursors are displayed are the same ones as in Figure 18-3. Clicking the right mouse button displays a customized cursor in the form of the letter "A." The hot spot of the custom cursor is the vertex of the "A." When the mouse is moved in the client area, its position is displayed on the screen. Figure 18-4 is a screen snapshot of the MOU_DEMO program.



```
Click left mouse button to change cursor
Click right button for custom cursor
x = 1401
y = 241
```

Figure 18-4 *MOU_DEMO Program Screen*

The program's first interesting feature is that no class cursor is defined in the WNDCLASSEX structure. Instead, the hCursor variable is initialized as follows:

```
wndclass.hCursor    = NULL;
```

Since the program has no class cursor, one is defined during WM_CREATE processing, with the following statements:

```
// Select and display a cursor
aCursor = LoadCursor(NULL, IDC_UPARROW);
SetCursor(aCursor);
```

In this code, the variable aCursor, of type HCURSOR, is declared in the windows procedure. Toggling the built-in cursors is performed in the WM_LBUTTONDOWN message intercept. The coding is as follows:

```
case WM_LBUTTONDOWN:
curNum++;      // bump to next cursor

switch (curNum) {
case 1:
   aCursor = LoadCursor(NULL, IDC_WAIT);
   SetCursor(aCursor);
   break;
case 2:
   aCursor = LoadCursor(NULL, IDC_APPSTARTING);
   SetCursor(aCursor);
   break;
case 3:
   aCursor = LoadCursor(NULL, IDC_CROSS);
   SetCursor(aCursor);
   break;
.
.
.
case 12:
   aCursor = LoadCursor(NULL, IDC_UPARROW);
   SetCursor(aCursor);
   curNum = 0;
   break;
}
```

Note that the static variable curNum, defined in the window procedure, is used to keep track of the cursor being displayed and to index through all 13 cursor images. The custom cursor is created using the cursor editor that is part of Visual Studio. The display of the custom cursor is implemented during WM_RBUTTONDOWN processing:

```
case WM_RBUTTONDOWN:
   aCursor = LoadCursor(pInstance,
             MAKEINTRESOURCE(IDC_CURSOR1));
   SetCursor(aCursor);
   return 0;
```

The movement of the mouse in the client area is detected by intercepting the WM_MOUSEMOVE message. The processing consists of obtaining the cursor coordinates from the low-order and high-order words of lParam, and converting the numeric values into ASCII strings for display. The code uses _itoa() for this purpose.

The ASCII values are placed on the corresponding string arrays. The processing is as follows:

```
case WM_MOUSEMOVE:
   cursorX = LOWORD(lParam);
   cursorY = HIWORD(lParam);
   // Convert integer to ASCII string
   _itoa(cursorX, CurXStr + 4, 10);
   _itoa(cursorY, CurYStr + 4, 10);
   // Display x coordinate of mouse cursor
   // First initialize rectangle structure
   SetRect (&textRect,        // address of structure
      2 * cxChar,       // x for start
      3 * cyChar,        // y for start
      cxClient -(2 * cxChar),  // x for end
      cyClient);        // y for end
   // Erase the old string
   DrawText( hdc, CurXBlk, -1, &textRect,
       DT_LEFT | DT_WORDBREAK);
   // Display new string
   DrawText( hdc, CurXStr, -1, &textRect,
       DT_LEFT | DT_WORDBREAK);
 // Display y coordinate of mouse cursor
 .
 .
 .
 return 0;
```

In order to avoid having Windows change the cursor as it moves into the client area, the code intercepts the WM_SETCURSOR message, as follows:

```
case WM_SETCURSOR:
   return 1;
```

When running the MOU_DEMO program notice that if the cursor is moved at a rather fast rate out of the client area, toward the left side or the top of the screen, the last value displayed for the diminishing coordinate may not be zero. This is due to the fact, mentioned earlier in this section, that WM_MOUSEMOVE messages are not sent to the window for every pixel of screen travel. Mouse programming must also take this into account and use greater-than and smaller-than comparisons to determine screen areas of cursor travel.

Chapter 19

Child Windows and Controls

Topics:

- Windows styles
- Child windows
- Menus
- Creating a menu
- Dialog boxes
- Common controls

This chapter is about programming the Windows graphical user interface (GUI). The Windows GUI consists of child windows and built-in controls, such as status bars, toolbars, ToolTips, trackbars, up-down controls, and many others. The discussion also includes general purpose controls such as message boxes, text boxes, combo boxes, as well as the most used of the common controls. All of these components are required to build a modern Windows program; it is difficult to imagine a graphics application that does not contain most of these elements.

19.1 Window Styles

One of the members of the WNDCLASSEX structure is the windows style. In Chapter 16 we briefly discussed windows styles, and Table 16-2 is a summary of the constants that can be used to define this member. Since the eleven style constants can be ORed with each other, many more windows styles can result. Furthermore, when you create a window using the CreateWindow() function, there are 27 window style identifiers (see Table 16-5). In addition, the CreateWindowEx() function provides 21 style extensions (see Table 16-4). Although the number of possible combinations of all these elements is very large, in practice, about 20 window styles, with unique properties, are clearly identified, all of which are occasionally used. This lists can be further simplify into three general classes (overlapped, pop-up, and child windows) and three variations (owned, unowned, and child), which gives rise to five major styles.

In the sections that follow we discuss four specific window styles:

- Unclassed child windows. These are windows that are related to a parent window but that do not belong to one of the predefined classes.
- Basic controls. These are child windows that belong to one of the standard control classes: BUTTON, Combo box, EDIT, LISTBOX, MDICLIENT, SCROLLBAR, and STATIC.
- Dialog boxes. A special type of pop-up window, that usually includes several child window controls, typically used to obtain and process user input.
- Common controls. A type of controls introduced in Windows 3.1, which include status bars, toolbars, progress bars, animation controls, list and tree view controls, tabs, property sheets, wizards, rich edit controls, and a new set of dialog boxes.

Several important topics related to child windows and window types are not discussed; among them are OLE control extensions, ActiveX controls, and multiple document interface (MDI). OCX controls relate to OLE automation and ActiveX controls are used mostly in the context of Web programming.

19.1.1 Child Windows

The simplest of all child windows is one that has a parent but does not belong to any of the predefined classes. Sometimes these are called "unclassed" child windows. However, if we refer to the "classed" child windows as controls, then the "unclassed" windows can be simply called "child windows." These are the designations used in the rest of the book: we refer to unclassed child windows simply as child windows and the classed variety as controls.

A child window must have a parent, but it cannot be an owned or an unowned window. The child window can have the appearance of a main window, that is, it can have a sizing border, a title bar, a caption, one or more control buttons, an icon, a system menu, a status bar, and scroll bars. The one element it cannot have is a menu, since an application can have a single menu and it must be on the main window. On the other hand, a child window can be defined just as an area of the parent window. Moreover, a child window can be transparent; therefore, invisible on the screen. The conclusion is that it is often impossible to identify a child window by its appearance.

A child window with a caption bar can be moved inside its parent client area; however, it will be automatically clipped if moved outside of the parent. The child window overlays a portion of its parent client area. When the cursor is over the child, Windows sends messages to the child, not to the parent. By the same token, mouse action on the child window's controls, or its system menu, is sent to the child. A child window can have its own window procedure and perform input processing operations independently of the parent. When the child window is created or destroyed, or when there is a mouse-button-down action on the child, a WM_PARENTNOTIFY message is sent to the parent window. One exception to parent notification is if the child is created with the WS_EX_NOPARENTNOTIFY style.

A child window is created in a manner similar to the parent window, although there are some important variations. Creating a child window involves the same steps as creating the main window. You must first initialize the members of the WNDCLASSEX structure. Then the window class must be registered. Finally, the window is actually created and displayed when a call is made to CreateWindow() or CreateWindowEx() function.

There are not many rules regarding when and where an application creates a child window. The child window can be defined and registered in WinMain() and displayed at the same time as the main window. Or the child window can be created as the result of user input or program action. We have already mentioned the great number of windows[check] styles and style combinations that can be used to define a child window. Some of these styles are incompatible, and others are ineffective when combined. The styles used in creating the child window determine how it must be handled by the code. For example, if a child window is created with the WS_VISIBLE style, then it is displayed as it is created. If the WS_VISIBLE style is not used, then to display the child window you have to call ShowWindow() with the handle to the child window as the first parameter, and SW_SHOW, SW_SHOWNORMAL, or one of the other predefined constants, as the second parameter.

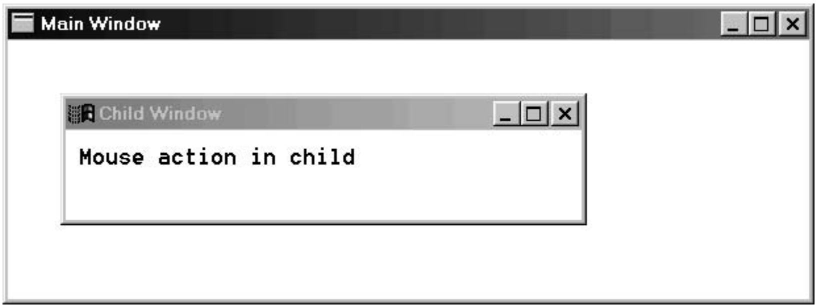
In operation, the child window provides many features that facilitate program design. For instance, a child window has its own window procedure, that can do its own message processing. This procedure receives the same parameters as the main window procedure and is notified of all the windows[check] messages that refer to the child. The child window can have its own attributes, such as icons, cursors, and background brush. If the main window is defined with an arrow cursor and the child window with a cross cursor, the cursor changes automatically to a cross as it travels over the child, and back to an arrow as it leaves the child's client area. The fact that each windows does is own message processing considerably simplifies the coding. Screen environments with multiple areas, such as the ones in Visual Studio, Windows Explorer, and many other applications, are implemented by means of child windows.

Parent and child windows can share the same display context or have different ones. In fact, each window can have any of the display contexts described in Chapter 4. If the child window is declared with the class style CS_PARENTDC, then it uses the parent's display context. This means that output performed by the child takes place in the parent's client area, and the child has no addressable client area of its own. On the other hand, parent and child can have separate device contexts. If both windows are declared with the class style CS_OWNDC, discussed in Chapter 4, then each has its own display context with a unique set of attributes. If there is more than one child window, they can be declared with the class style CS_CLASSDC, and the children share a single device context, which can be different from the one of the parent window.

Each child window is given its own integer identifier at the time it is created. Since child windows can have no menus, the HMENU parameter passed to CreateWindows() or CreateWindowsEx() is used for this purpose. The child window uses this identifier in messages sent to its parent, which enables the parent to tell to which child window the message belongs, if more than one is enabled. If multiple child windows are given the same numeric identification then it may be impossible for the parent to tell them apart.

19.1.2 Child Windows Demonstration Program

The program named CHI_DEMO, located in the Child Window Demo project folder on the book's software package, is a demonstration of a program with a child window. The program displays an overlapped child window inside the parent window. When the left mouse button is clicked inside the child window, a text message is displayed in its client area. The same happens when the left mouse button is clicked in the parent's client area. At the same time, the old messages in the parent or the child windows are erased. Figure 19-1 is a screen snapshot of the CHI_DEMO program.



Mouse action in child

Figure 19-1 *CHI_DEMO Program Screen*

The program uses a child window, which is defined using the WS_OVERLAPPEDWINDOW style. This style, which is the same one used in the parent window, gives both the parent and the child a title bar with caption, a system menu, a border, and a set of control buttons to close, minimize and restore. The child window is created during WM_CREATE message processing of the parent window, as follows:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam,
                         LPARAM lParam) {
  PAINTSTRUCT ps ;
  WNDCLASSEX  chiclass ;
  switch (iMsg) {
    case WM_CREATE:
      hdc = GetDC (hwnd) ;
      // The system monospaced font is selected
      SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
      // Create a child window
      chiclass.cbSize        = sizeof (chiclass) ;
      chiclass.style         = CS_HREDRAW | CS_VREDRAW
                               | CS_OWNDC;
      chiclass.lpfnWndProc   = ChildWndProc ;
      chiclass.cbClsExtra    = 0 ;
      chiclass.cbWndExtra    = 0 ;
      chiclass.hInstance     = pInstance ;
      chiclass.hIcon         = NULL;
      chiclass.hCursor       = LoadCursor (NULL, IDC_CROSS) ;
      chiclass.hbrBackground = (HBRUSH) GetStockObject
```

```
                              (WHITE_BRUSH);
      chiclass.lpszMenuName  = NULL;
      chiclass.lpszClassName = "ChildWindow" ;
      chiclass.hIconSm       = NULL;
      RegisterClassEx (&chiclass) ;
        hChild = CreateWindow ("ChildWindow",
                "A Child Window",  // caption
                 WS_CHILD | WS_VISIBLE |
                 WS_OVERLAPPEDWINDOW ,
                 40, 40,       // x and y of window location
                 400, 100,     // x and y of window size
                 hwnd,         // handle to the parent window
                 (HMENU) 1001, // child window designation
                 pInstance,    // program instance
                 NULL) ;
      // Make sure child window is valid
      assert(hChild != NULL);
      return 0 ;
      .
      .
      .
```

Note that the child is defined with the styles WS_CHILD, WS_VISIBLE, and WS_OVERLAPPEDWINDOW. The WS_VISIBLE class style ensures that the child becomes visible as soon as CreateWindows() is executed. The child window is assigned the arbitrary value 1001 in the HMENU parameter to CreateWindow(). The child has a private DC, the same as the parent, but the DCs are different. The assert statement ensures, during program development, that the child window is a valid one.

During the parent's WM_PAINT message processing a call is made to UpdateWindow() with the handle of the child window as a parameter. The result of this call is that the child's window procedure receives a WM_PAINT message.

The window procedure for the child, named ChildWndProc() in the demo program, is prototyped in the conventional manner and its name defined in the lpfnWndProc member of the child's WNDCLASSEX structure. The child's window procedure is coded as follows:

```
LRESULT CALLBACK ChildWndProc (HWND hChild, UINT iMsg, WPARAM wParam,
                               LPARAM lParam) {
 switch (iMsg) {
  case WM_CREATE:
      childDc = GetDC(hChild);
      SelectObject (childDc, GetStockObject
                   (SYSTEM_FIXED_FONT)) ;
      return 0;
    case   WM_LBUTTONDOWN:
     // Display message in child and erase text in parent
     TextOut(childDc, 10, 10, "Mouse action in child ", 22);
     TextOut(hdc, 10, 10, "                      ", 22);
return 0;
  case WM_DESTROY:
       return 0;
  }
 return DefWindowProc (hChild, iMsg, wParam, lParam) ;
}
```

During the WM_CREATE processing of the child's windows[check] procedure, the code obtains a handle to the child's DC. Also, the system fixed font is selected into the DC at this time.

In the CHI_DEMO program we have declared several public variables: the handles to the windows of the parent and the child and the handles to their display context. This stretches one of the fundamental rules of Windows programming: to keep public data at a minimum. In this case, however, we achieve a substantial simplification in the coding, since now the parent can have access to the child's device context, and vice versa. Therefore, when the user clicks the left mouse button in the child's client area, a text message is displayed in the child window and the one in the parent window is simultaneously erased. Similar processing takes place when the left mouse button is clicked in the parent's client area.

19.1.3 Basic Controls

These are the traditional controls that have been around since the Win16 APIs. They are predefined child windows that belong to one of the standard window classes. Table 19-1 lists the predefined classes used for controls.

Table 19-1

Predefined Control Classes

CLASS NAME	MEANING
BUTTON	A small rectangular child window representing a button. The user clicks a button to turn it on or off. Button controls can be used alone or in groups, and they can be labeled or not. Button controls typically change appearance when clicked.
COMBOBOX	Consists of a list box and a selection field similar to an edit control (see description). Depending on its style, you can or cannot edit the contents of the selection field. If the list box is visible, typing characters into the selection field highlights the first list box entry that matches the characters typed. By the same token, selecting an item in the list box displays the selected text in the selection field.
EDIT	A rectangular child window into which you type text. You select the edit box and give it the keyboard focus by clicking it or moving to it by pressing the Tab key. You can enter text into an Edit control if it displays a flashing caret. You use the mouse to move the cursor inside the box, to select characters to be replaced, or to position the cursor for inserting new characters. The Backspace key deletes characters. Edit controls use a variable-pitch system font and display characters from the ANSI character set. The WM_SETFONT message can be used to change the Default font. During input, tab characters are expanded into As many spaces as are required to move the caret to the Next tab stop. Tab stops are preset eight spaces apart.

(continues)

Table 19-1

Predefined Control Classes (continued)

CLASS NAME	MEANING
LISTBOX	A list of character strings. It is used to present a list of names, such as filenames, from which you can select. Selection is made by clicking an item in the list box. The selected string is highlighted, and a notification message is sent to the parent window. When the item list is too long for the window, you can use a vertical or horizontal scroll bar. If the scroll bar is not needed, it is automatically hidden.
SCROLLBAR	A rectangular control with a scroll box and direction arrows at both ends. The scroll bar sends a notification message to its parent window whenever the user clicks it. The parent window is responsible for updating the position of the scroll box when necessary. Scroll bar controls have the same appearance and function as scroll bars used in ordinary windows. Unlike scroll bars, however, scroll bar controls can be positioned anywhere in a window and for any purpose. The scroll bar class also includes size box controls, which is a small rectangle that you can expand to change the size of the window.
STATIC	A simple text field, box, or rectangle, used to label, group, or separate other controls. Static controls take no input and provide no output.

Figure 19-2 shows buttons of several types, a list box, a combo box, and a scroll bar control.

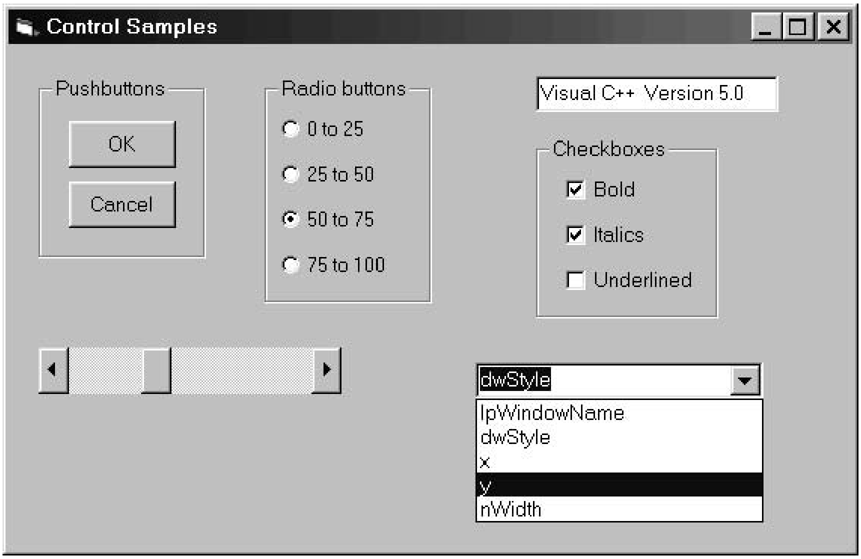


Figure 19-2 *Buttons, List Box, Combo Box, and Scroll Bar Controls*

In conventional Windows programming basic controls are not frequently used in the client area of the main window. Most often you see them in message boxes or input boxes, described later in this chapter. For this reason, Developer Studio does not provide a resource editor for inserting controls in the client area, although it does contain a powerful editor for dialog boxes. In spite of this, the use of basic controls in child windows adds considerable power to a programmer's toolkit. The result is a completely customizable message, dialog box, toolbar, or other child window, in which you are free from all the restrictions of the built-in versions of these components. The price for this power and control is that you must implement all the functionality in your own code.

The CreateWindow() or CreateWindowEx() functions are used to build any one of the controls in Table 19-1. If the control is created using the WS_VISIBLE window style, then it is displayed immediately on the window whose handle is passed as a parameter to the call. If not, then the ShowWindow() function has to be called in order to display it. The call returns a handle to the created control, or NULL if the operation fails. The following code fragment shows creating a button control.

```
static HWND    hwndRadio1;     // Handle to control
 . . .
     hwndRadio1 = CreateWindow (
      "BUTTON",        // Control class name
      "Radio 1",                // Button name text
       WS_CHILD | WS_VISIBLE | BS_RADIOBUTTON /| WS_SIZEBOX,
       20,                 // x coordinate of location
       60,                 // y coordinate
       100, 30,            // button size
       hChild,             // Handle to parent window
       (HMENU) 201,        // control id number
       pInstance,          // Instance handle
       NULL) ;             // Pointer to additional data
```

Because controls belong to predefined classes, they need not be registered as a window class. Therefore, the WNDCLASSEX structure and the call to RegisterClass() or RegisterClassEx() are not required in this case. In the case of a main window, the eighth parameter of CreateWindow() is the handle to its menu. Since controls cannot have a menu, this parameter is for the control's numeric designation, the same as with a child window. Thereafter, this numeric value, which can be also a predefined constant, identifies the control. If the control is to be addressable, this identification number should be unique.

In addition to the general window style, each of the predefined control classes has its own set of attributes. The prefixes are shown in Table 19-2.

The class-specific styles are ORed with the window style constants passed in the third parameter to CreateWindow(). Note that in the previous code fragment the BS_RADIOBUTTON constant is included in the field. There are several variations of the button class. The buttons in the first group of Figure 19-2, labeled Pushbuttons, are plain pushbuttons. They appear raised when not pushed and sunken after being pushed by the user. Pushbuttons operate independently. These buttons are usually created with the BS_PUSHBUTTON and BS_DEFPUSHBUTTON styles.

Table 19-2

Prefix for Predefined Window Classes

PREFIX	CONTROL TYPE
BS	button
CBS	combo box
ES	edit box
LBS	list box
SBS	scroll bar
SS	static

Radio buttons are reminiscent of the buttons in the radios of old-style automobiles: pushing one button automatically pops out all the others. The styles BS_RADIOBUTTON and BS_AUTORADIOBUTTONS are used for creating this type of button. Radio buttons contain a circular area with a central dot that indicates the button's state.

Another variation is the checkbox. A checkbox can have two or three states. A two-state checkbox can be checked or unchecked, while the three-state style can also be grayed. Checkboxes, like regular buttons, operate independently of each other. Two-state checkboxes are created with the BS_CHECKBOX style. The three-state version requires ORing the BS_3STATE constant with BS_CHECKBOX.

A unique style of button is the groupbox, which is enabled with the button style BS_GROUPBOX. A groupbox is used to enclose several buttons or controls in a labeled frame. It is unique in the sense that it is defined as a button, but a groupbox does not respond to user input, nor does it send messages to the parent window. Figure 19-2 shows three group boxes, one for each type of button.

Three types of controls are designed for manipulating text: the edit box, the combo box, and the list box. You select an edit box control for input by clicking it or tabbing until it has the input focus. When a caret is displayed, you can enter text until the rectangle is filled. If the edit box control is created with the ES_AUTOSCROLL style, then you can enter more characters than fit in the box since the text automatically scrolls to the left, although this practice is not recommended since part of the input disappears from the screen. If the edit box is defined with the ES_MULTILINE style then you can enter more than one text line. However, this style can create conflicts if the active window contains a default pushbutton that also responds to the Enter key. The built-in solution to this problem is that the default style of edit box requires the Ctrl+Enter key combination to end an input line. However, if the edit box is created with the style ES_WANTRETURN, then the Enter key alone serves as a line terminator.

The list box control displays a list of text items from which the user can select one or more. Code can add or remove strings from the list box. Scroll bars can be requested for a list box. If the list box is created with the LBS_NOTIFY style then the parent window receives a message whenever the user clicks or double-clicks an item. The LBS_SORT style makes the list box sort items alphabetically.

The combo box is a combination of a textbox and a list box. The user can enter text on the top portion of the combo box, or drop down the list box and select an item from it. Alternatively, the edit function of the combo box can be disabled. Figure 19-2 shows a combo box.

Scroll bar controls can be vertical or horizontal and be aligned at the bottom, top, left, or right of a rectangle defined at call time. It is important to distinguish between window and control scroll bars. Any window can have scroll bars if it is defined with the WS_VSCROLL or WS_HSCROLL styles. Scroll bar controls are individual scroll bars which can be positioned anywhere on the parent's client area. Both windows and control scroll bars send messages to the parent window whenever a user action takes place. Scroll bar controls are of little use by themselves but provide a powerful and convenient way of obtaining user input, for example, a scroll bar control that allows the user to move up or down a numeric range without typing values. In this case the scroll bar is usually combined with another control that displays the selected value. The CON_DEMO program, in this chapter, has an example of this use of a scroll bar control.

Static controls do not interact with the user since they cannot receive mouse or keyboard input. The principal use of static controls is to display rectangular frames of several colors and borders, and to provide feedback from another control. The CON_DEMO program, described later in this chapter, which is found in the Controls Demo project folder in the book's software pckage, has a child window with a static control that displays the position of a scroll bar.

19.1.4 Communicating with Controls

Controls are child windows and child windows can communicate with their parents. As is the case in all Windows functions, controls communicate with their parent window by means of a message passing mechanism. The messages passed to the parent window depend on the type of control. This communication works both ways: a control sends a message to its parent window informing it that a certain user action has taken place, or the parent window sends a message to a control requesting that it take a certain action or report some item of information stored internally. For example, when the user clicks on a pushbutton control, a WM_COMMAND message is sent to the parent window. When a parent window needs to know if a radio button is checked or unchecked it sends a BM_GETCHECK message to the radio button control.

WM_COMMAND is used to inform the parent window of action on a menu, on a control, or of an accelerator keystroke. The high-order word of the wParam is zero if the message originates in a menu, and one if it originates in an accelerator keystroke. If the message originated in a control, then the high-word of the wParam is a control-specific notification code. Table 19-3 lists the notification codes for the button controls.

Table 19-3

Notification Codes for Buttons

NOTIFICATION CODE	ACTION
BN_CLICKED	Button was clicked
BN_DBLCLK	Button was double-clicked
BN_SETFOCUS	Button has gained keyboard focus
BN_KILLFOCUS	Button has lost keyboard focus

In the case of a control, the low-order word of the wParam contains the control identifier. This identifier is the number assigned to the control in the hMenu parameter of CreateWindows() or CreateWindowsEx(). Usually, an application defines a symbolic constant for each control, since this is a mnemonic aid and helps to make sure that no two controls are assigned the same value. One or more #define statements can be used as follows:

```
#define   WARMBUTTON      101
#define   HOTBUTTON       102
#define   COLDBUTTON      103
```

A switch statement on the low word of wParam can later be used to tell which button been pressed by the user, for example:

```
int buttonID, buttonNotify;
.
.
case WM_COMMAND:
    buttonID =      LOWORD(wParam);
    buttonNotify =  HIWORD(wParam);
    //eliminate non-control actions
    if(buttonNotify <= 1)
      return 0;
    switch (buttonID):
      case WARMBUTTON:
        if(buttonNotify == BN_CLICKED)
     // ASSERT:
     //    Tested button was clicked
    .
    .
    .
```

Some controls store information about their state or other data. For example, a three-state checkbox can be in a checked, unchecked, or indeterminate state. Table 19-4 lists the checkbox constants that define the three settings. These are used with three-state checkboxes and radio buttons.

Table 19-4

Notification Codes for Three-State Controls

NOTIFICATION CODE	ACTION
BST_CHECKED	Control is checked
BST_INDETERMINATE	Control is checked and grayed
BST_UNCHECKED	Control is unchecked

If you send a BM_GETCHECK message to a three-state checkbox or radio button it responds with one of these values. Suppose a three-state checkbox, with identification code CHKBOX1, and handle hwndChkBox1, which you wished to change from the checked to indeterminate state; it can be coded as follows:

```
LRESULT butMsg;
int     buttonID, buttonNotify;
.
.
.
case WM_COMMAND:
    buttonID =      LOWORD(wParam);
    buttonNotify =  HIWORD(wParam);

    //eliminate non-control actions
    if(buttonNotify <= 1)
      return 0;
    switch (buttonID):
      case CHKBOX1:
        butMsg = SendMessage(hwndChkBox1,   // handle
                             BM_GETCHECK,   // message
                             0, 0L);        // must be zero
        if(butMsg == BST_CHECKED)
        // ASSERT:
        //     checkbox is in checked state
        SendMessage(hwndChkBox1,
                BM_SETCHECK,        // order to set new state
                BST_INDETERMINATE,  // change to this state
                0, 0l);
       .
       .
       .
```

Note, in the previous code fragment, that we used the SendMessage() function to communicate with the control. SendMessage() is used to send a message to a window or windows bypassing the message queue. In contrast, the PostMessage() function places the message in the thread's message queue. In communicating with a control, the first parameter to SendMessage() is the control's handle and the second one is the message to be sent. The third parameter is zero when we wish to obtain information from a control, and it contains a value or state when we wish to change the data stored. The BM_GETCHECK message returns a value, of type LRESULT, which is one of the notification codes in Table 19-4. The BM_SETCHECK message is used to change the button's state.

Scroll bar controls have a unique way of communicating with the parent window. Like main windows scroll bars, scroll bar controls send the WM_VSCROLL and WM_HSCROLL messages, the first one in the case of a vertical scroll bar action and the second one in the case of a horizontal scroll bar. The lParam is set to zero in windows scroll bars and to the scroll bar handle in the case of a scroll bar control. The high-order word of the wParam contains the position of the scroll box and the low-order word the scroll box value, which is one of the SB prefix constants listed in Table 19-5.

Table 19-5

User Scroll Request Constants

VALUE	MEANING
SB_BOTTOM	Scroll to the lower right
SB_ENDSCROLL	End scrolling
SB_LINELEFT	Scroll left by one unit
SB_LINERIGHT	Scroll right by one unit
SB_PAGELEFT	Scroll left by the width of the window
SB_PAGERIGHT	Scroll right by the width of the window
SB_THUMBPOSITION	Scrolls to the absolute position. The current position is specified by the nPos parameter
SB_THUMBTRACK	Drags scroll box to the specified position. The current position is specified by the NPos parameter
SB_TOP	Scroll to the upper left

In processing scroll bar controls the first step is to make sure that the message originates in the control being monitored. When the scroll action does not originate in windows scroll bars, or on those of another control, the processing usually consists in determining the new position for the scroll box. Two functions in the Windows API, SetScrollInfo() and GetScrollInfo(), provide all necessary functionality for scroll bar operation. SetScrollInfo() is used to set the minimum and maximum positions for the scroll box, to define the page size, and to set the scroll box to a specific location. GetScrollInfo() retrieves the information regarding these parameters. Four other functions, SetScrollPos(), SetScrollRange(), GetScrollPos(), and GetScrollRange() are furnished. In theory, these last four functions are furnished for backward compatibility, although they are often easier to implement in code that the new versions.

A program that implements a horizontal scroll bar usually starts by creating a scroll bar control. You can use the SBS_HORZ scroll bar style and determine its vertical and horizontal size in the sixth and seventh parameters to CreateWindows(), as follows:

```
#define  SCROLLBAR   401          // scroll bar id code
static   HWND        hwndSB;      // handle for the scroll bar
     .
   .
// create a scroll bar class child window
   hwndSB = CreateWindow ("SCROLLBAR",  // Control class name
            "",                        // Button name text
            WS_CHILD | WS_VISIBLE | SBS_HORZ ,
            20,                // x coordinate of location
            140,               // y coordinate
            150, 25,           // dimensions
            hChild,            // handle to parent window
            (HMENU) SCROLLBAR, // child window id.
            pInstance,         // instance handle
            NULL) ;
```

Once the scroll bar is created, you must determine its range, set the initial position of the scroll box, and define its page size, if page operations are implemented. This last value determines how much the scroll box moves when the bar itself is clicked. All of this can be done with a single call to SetScrollInfo(), in which case the parameters are stored in a SCROLLINFO-type structure, as follows:

```
// Store parameters in SCROLLINFO structure members
   scinfo.cbSize = sizeof(SCROLLINFO);   // structure size
   scinfo.fMask  = SIF_POS | SIF_RANGE | SIF_PAGE; mask
   scinfo.nMin   = 0;                 // minimum value
   scinfo.nMax   = 99;                // maximum value
   scinfo.nPage  = 0;                 // page size
   scinfo.nPos   = 50;                // initial position
// Store scroll bar information
   SetScrollInfo(hwndSB, SB_CTL, &scinfo, TRUE);
   //              |       |       |        |___ redraw
   //              |       |       |____ address of SCROLLINFO
   //              |       |_____ refers to a scroll bar
   //                             control
   //              |_____________ handle to the scroll bar control
```

Manipulating the scroll bar requires intercepting the corresponding scroll bar messages. The current position of the scroll box is usually stored in a local variable, in this case the variable is named sbPos. Since this is a horizontal scroll bar, you can intercept the WM_HSCROLL message and then make sure that it refers to the scroll bar you are monitoring.

```
static  int       sbPos;       // position of scroll box
  .
  .
  .
case WM_HSCROLL:
     // Make sure action refers to local scroll bar
     // not the Windows scroll bars
     if(hwndSB == (HWND) lParam) {
           switch (LOWORD (wParam))  // Scroll code
           {
        case SB_LINELEFT:     // Scroll left one unit
           if(sbPos > 0)
           sbPos--;
           break;
        case SB_LINERIGHT:    // Scroll right one unit
           if(sbPos < 99)
           sbPos++;
           break;
        // Processing for user dragging the scroll box
         case SB_THUMBTRACK:
         case SB_THUMBPOSITION:
            sbPos = HIWORD (wParam);
            break;
         }
        // Display scroll box at new position
           SetScrollPos(hwndSB,
                        SB_CTL,
                        sbPos,
                        TRUE);
        }
        return 0;
```

Finally, there is the static class of controls that are often used for text fields, for labeling boxes, and for drawing frames and rectangles. Although static controls are frequently limited to labeling and simple drawing operations, they can be made to receive mouse input by means of the SS_NOTIFY style. Furthermore, the text in a static control can be changed at run time. The CON_DEMO program, described in the following section, located in the Controls Demo project folder on the book's software package, has two static controls. One is used to display the position of the scroll bar, and the other one is a black frame that surrounds the scroll bar buttons.

19.1.5 Controls Demonstration Program

The program named CON_DEMO, in the book’s software package, is a demonstration of some of the basic controls described in previous sections and of the programming required to operate them. The controls are contained in a child window, much like the one created in the CHI_DEMO program already described. Figure 19-3 is a labeled screen snapshot of the CON_DEMO program.

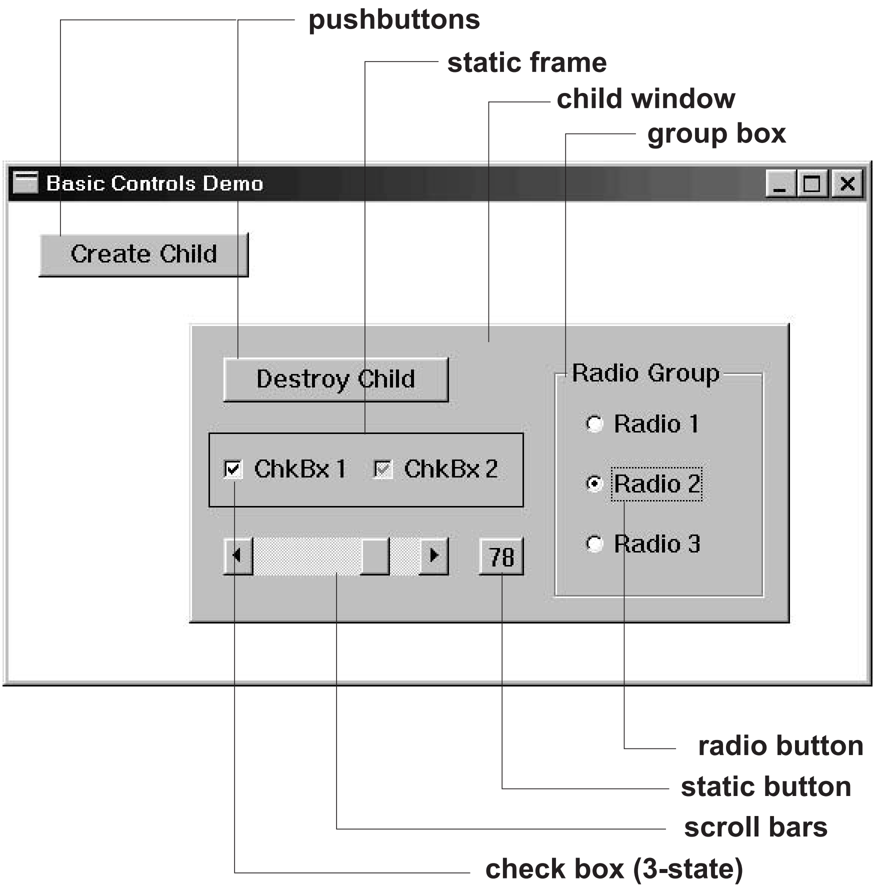


Figure 19-3 *CON_DEMO Program Screen*

The program's main screen contains a pushbutton that displays the child window. In the remainder of this section we have selected some excerpts from the program code to demonstrate the processing.

At the start of the code, the child windows and controls are defined as symbolic names. This is a useful simplification in applications that manipulate several resources or program elements that are identified by numeric values. The advantage is that the information is centralized for easy access and that it ensures that there are no repeated values.

```
// Constants for child windows and controls
#define  CHILD1      1001
#define  CREATEWIN   102
#define  DESTROYWIN  103
#define  RADGROUP    104
#define  RADIO1      201
#define  RADIO2      202
#define  RADIO3      203
#define  CHKBOX1     301
#define  CHKBOX2     302
#define  SCROLLBAR   401
#define  SCRBARWIN   501
#define  FRAME       502
```

The numeric values assigned to individual controls are arbitrary; it is a good idea, however, to follow a pattern for numbering resources and controls, since this avoid chaos in large programs. For example, child windows can be assigned a four-digit number, controls a three-digit number, and so forth. It is also recommended practice to use a dense set of integers for representing related controls, since there are Windows functions that operate on this assumption. Following this rule, the radio buttons in the CON_DEMO program are numbered 201, 202, and 203, and the checkboxes have numbers 301 and 302.

The creation of the child window in the CON_DEMO program is almost identical to the one in CHI_DEMO, previously described. The individual controls are created in the child window using the CreateWindow() function with the parameter set required in each case. The handles for the individual controls are defined as static variables in the child windows[check] procedure, as follows:

```
LRESULT CALLBACK ChildWndProc (HWND hChild, UINT iMsg, WPARAM
                               wParam, LPARAM lParam) {
  static HWND  hwndChildBut1;     // Handle to child's button
  static HWND  hwndRadio1, hwndRadio2, hwndRadio3;
  static HWND  hwndChkBx1, hwndChkBx2;
  static HWND  hwndSB, hwndVal;
  static HWND  hwndGrpBox1;
  static HWND  hwndFrame;
  .
  .
  .
```

The code in the child window intercepts the WM_CREATE message. During message processing it installs the system's fixed font in the display context and then proceeds to create the individual controls. A bool-type variable, named childStatus, is used to store the state of the child window. This variable is TRUE if the child window is displayed. This avoids creating more than one copy of the child. The first control created in the child window is the pushbutton that destroys it and returns execution to the parent. Before that, the system's fixed font is selected into the display context. Coding is as follows:

```
switch (iMsg) {
   case WM_CREATE:
    // Test that child window is not already displayed
    if(childStatus)
      return 0;
    // ASSERT:
    //      child window is not displayed
    childStatus = TRUE;    // child window is displayed
    childDc = GetDC(hChild);   // handle to private DC
        SelectObject (childDc,
                      GetStockObject (SYSTEM_FIXED_FONT));
    // Place destroy button on child window
    hwndChildBut1 = CreateWindow (
      "BUTTON",                // Control class name
      "Destroy Child",          // Button name text
      WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
      20, 20,                  // x and y location
      150,                     // Window width
      30,                      // Window height
      hChild,                  // Handle to parent window
      (HMENU) DESTROYWIN,      // Child window id.
      pInstance,               // Instance handle
      NULL) ;
      .
      .
      .
```

The user interaction with the controls is monitored and processed in the WM_COMMAND message intercept of the child window. First, the notification code and the button identifier are stored in local variables. A switch statement on the button identification code allows directing the processing to the routines for each of the buttons. The code examines the notification code to make sure that the intercept is due to action on a button control, and not on an accelerator key or a menu item.

```
case WM_COMMAND:
  buttonID = LOWORD (wParam);
  buttonNotCode = HIWORD (wParam);
  switch (buttonID) {
    if(buttonNotCode <= 1)
      return 0;

   case DESTROYWIN:
     if(buttonNotCode == BN_CLICKED) {
      childStatus = FALSE;
      DestroyWindow(hChild);
      UpdateWindow(hwnd);
      }
      break;
   // Radio button # 1 action
   case RADIO1:
      // Set radio button ON
         SendMessage(hwndRadio1, BM_SETCHECK, 1, 0L);
```

```
            SendMessage(hwndRadio2, BM_SETCHECK, 0, 0L);
            SendMessage(hwndRadio3, BM_SETCHECK, 0, 0L);
            break;
     .
     .
     .
```

19.2 Menus

The menu is one of the most important elements of the Windows user interface. It occupies the line below the title bar. Often, only the program's main window has a top-level menu. There has been considerable uncertainty regarding the names of the various elements in a menu. The following designations are based on Microsoft's The Windows Interface Guidelines for Software Design, listed in the Bibliography.

- The menu bar is a screen line directly below the title bar, which contains entries called the menu titles, or just the menus.
- Each menu title (menu) activates a drop-down box, which contains one or more menu items. Menu items are usually arranged in a single column, although Windows supports multiple column menus.
- Menu items can be of three types: menu commands, child menus, and separators. A menu command is a menu item that executes a program function directly. A child menu, also called a cascading or hierarchical menu, is a submenu, which can in turn contain menu commands, child menus, and separators. Items that activate a child menu are usually marked by a triangular arrow to the right of its name. A separator is a screen line that is used to group related menu items.
- Pop-up menus are activated by clicking the second mouse button. They are usually unrelated to the program's menu bar.
- Access keys are keystrokes that can be used instead of mouse button action to access menu items. Access keys are underscored in the menu title and in menu items. To activate a menu title by means of the access key you must hold down the Alt key. Once a drop down menu is displayed, access to the contained items is by pressing the corresponding access key. The Alt key is not required in this case.
- Shortcut keys are keystroke combinations that allow accessing a menu item directly. Shortcut keys are usually a Ctrl + key combination or a function key. Windows documentation sometimes calls these shortcut keys accelerators, but The Windows Interface Guidelines for Software Design prefers the former name.

There are also some style considerations regarding the design and implementation of menus. Although the design of the user interface is a topic outside the scope of this book, there are several general principles worth mentioning.

- A menu title should be a single word that represents the items that it contains. Each menu title should have an access key, which activates the menu when used in conjunction with the Alt key. Access keys are underlined in the menu bar. No two menu titles should have the same access key.
- Cascading menus should be used sparingly since they add complexity to the interface. Their purpose is to reduce the number of entries in the main menu and to logically orga-

nize hierarchical entries. The user should never have to navigate through more than two levels of cascading menus to reach a command.

- Menu items that are not active or are currently unavailable should be disabled and displayed in gray characters. Alternatively, a permanently inactive item can be removed from a menu.
- If a menu command requires additional data to execute, it should be followed by an ellipsis (...). The ellipsis serves as a visual key that information for executing a command is incomplete. Typically, commands with ellipses display a dialog box where the additional data is supplied. However, commands that obviously generate other informational actions should not be followed by ellipses; for example, a Properties command is expected to display information, therefore it should not have ellipsis.
- Check boxes are used in menus to indicate the status of a menu item. A checked item signals that it is functional. Code should check and uncheck items during processing to update their status.
- All menu items should have access keys, but items on the same drop down menu cannot have the same access key. The first choice for an access keys is the first character in the menu title or entry. If the first character is already used as an access key, then the next one in the item name that is not used as an access key should be selected.
- Shortcut keys that activate menu commands are best implemented with the Crtl key followed by a mnemonic letter associated with the entry. Function keys can also be used. For example, Ctrl + S can be used for a Save command and Ctrl + P for a Print command. The most used commands should be assigned a shortcut.

Figure 19-4 shows some of the most common elements in a menu.

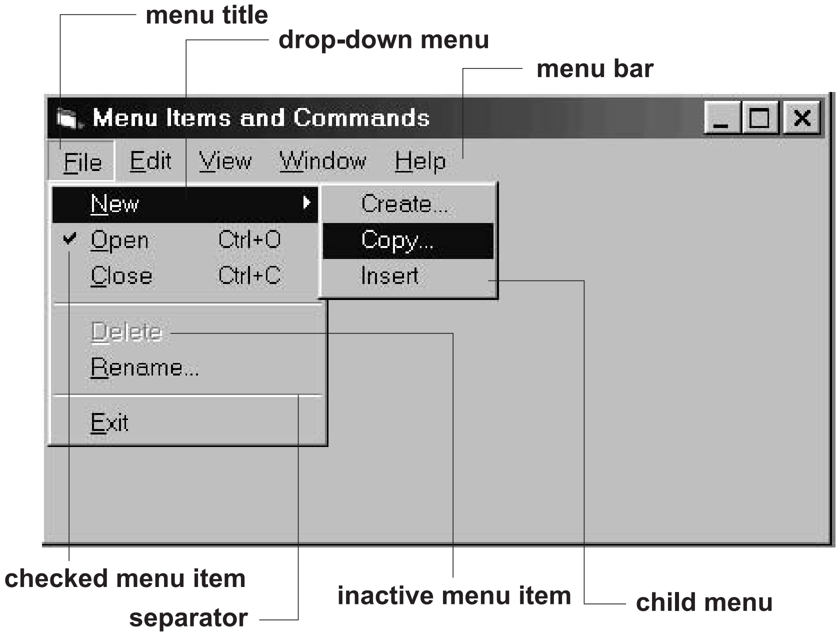


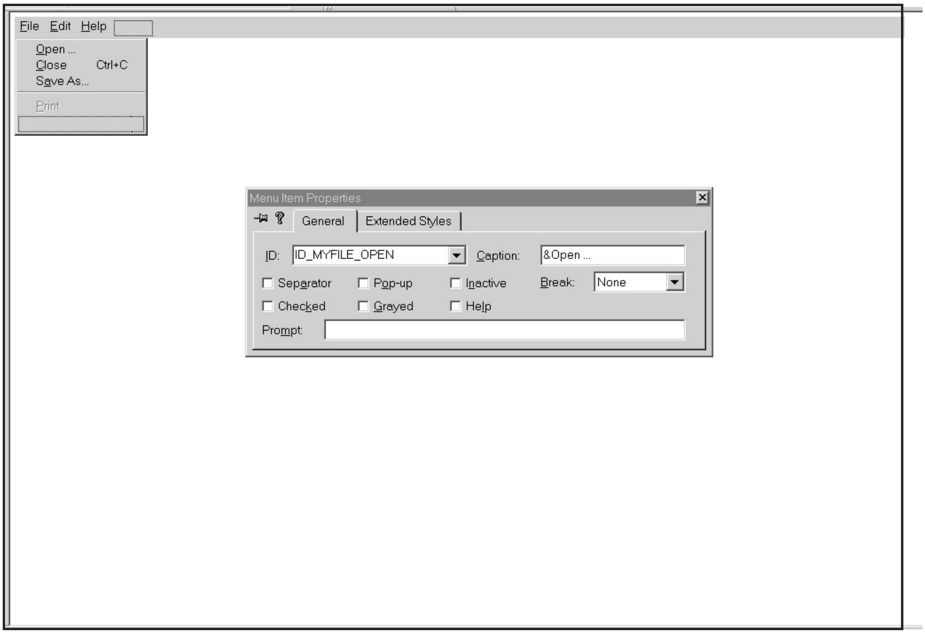
Figure 19-4 *Common Menu Elements*

19.2.1 Creating a Menu

There are several ways to create a menu. Before the Visual Studio and other development environments came into existence, menus were created using API functions. CreateMenu() creates an empty menu and returns its handle. InsertMenuItem() can be used to populate the menu with components. AppendMenu() adds a component to an existing menu. Other functions, such as DeleteMenu(), DestroyMenu(), DrawMenuBar(), ModifyMenu() and RemoveMenu() are also available. Finally, the LoadMenuIndirect() function can be used to load a menu from a memory resident menu template.

1. From the Developer Studio Insert menu, select the Resource command. Select the Menu resource type in the dialog box, and click New.

2. Create the main menu entries in you program (the menu titles) as well as the menu items in each of the drop-down menus. At this time you can assign an identification code to each menu item, define child menus (called pop-up in the input form), determine if the item is initially grayed, checked, or inactive, assign shortcut keys, and other menu attributes. Details on how to use the menu editor are available in Developer Studio online Help. Figure 19-5 shows the Developer Studio menu editor screen.



Menu Item Properties

ID: ID_MYFILE_OPEN
Caption: &Open ...

□ Separator □ Pop-up □ Inactive Break: None

□ Checked □ Grayed □ Help

Prompt:

Figure 19-5 *Developer Studio Menu Editor*

3. Once you have finished creating the menu, click on the close button of the menu editor window. If the application already has a script file, the new menu is added to it. If not, Developer Studio prompts you to save the new script file.

4. Skip this step if a script file has already been inserted into the project. If not, open the Project menu, select Add to Project, and then Files. In the Insert Files into Project dialog

box, select the script file and then click OK. The script file now appears in the project workspace window.

5. Select the Resource View button in the project workspace pane and click + on Script Resources. Click + on Menu. Note the identifier name for the menu resource, which is IDR_MENU1 if this is the first menu created.

6. Enter the menu identifier in the wndclass structure defined in WinMain(), as follows:

```
wndclass.lpszMenuName  = MAKEINTRESOURCE(IDR_MENU1);
```

7. Developer Studio creates a header file named resource.h which assigns numeric values to the program resources. The file is saved under the name "resource.h" and stored in the project's main directory. The main source file must reference this header file in an include statement, such as:

```
#include "resource.h"
```

8. To recompile the program with the new menu, select Rebuild All from the Build menu.

9. To edit the menu, double-click on the corresponding IDR_MENU1 icon.

If you receive a redefinition of symbol error at build time there are two possible solutions: one is to comment-out the redefined symbol in the file named afxres.h located in Msdev\Mfc\Include directory. The other one is to edit the resource, in this case the menu, and change the name in the ID: field. Changing the afxres.h file is a permanent way of avoiding this error, but the development system cannot be used for MFC applications if afxres.h has been altered.

19.2.2 Menu Item Processing

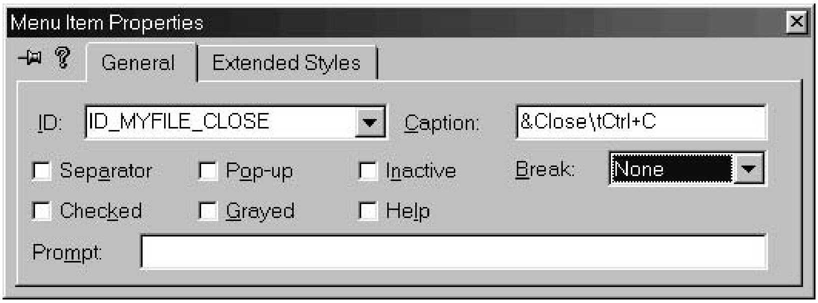
There are several intercept messages related to application menu processing. WM_MENUSELECT is sent when the mouse cursor moves among the menu items, and WM_INITITEM when the user selects an item from a menu. However, most applications do all their menu processing in the WM_COMMAND message intercept. In the case of a menu, the lParam is 0 and the wParam contains the menu ID code, which is the identification number and its corresponding string constant found in the resource.h file. System menus notify the application through the WM_SYSCOMMAND message. The following code fragment shows the intercept routine for the item named Open in the File menu:

```
case WM_COMMAND:
   switch (LOWORD (wParam)) {
   case ID_MYFILE_OPEN:
       // ASSERT:
       //   Menu item resource named ID_MY-FILE_OPEN
       //   was activated by user
   .
   .
   .
```

An important fringe benefit from using the menu editor in Developer Studio is that access keys are automatically detected and vectored to the corresponding handler. Suppose that in the preceding code fragment the Open command was defined so that the letter O is preceded by the & symbol in the editor screen. In this case, when the user presses the "O" key while the File menu is open, a WM_COMMAND message with the key code ID_MYFILE_OPEN is sent to the handler.

19.2.3 Shortcut Keys

Shortcut keys require a special treatment so that the keystrokes are vectored to the desired handler. It is recommended that shortcut keys be listed in the same line as the menu item. In order to do this you must insert the text for the control keystroke, preceded by \t in the caption window of the Menu Item Properties editor screen. In this case \t indicates a Tab code which displays the following text on the next tab field. Figure 19-6 shows the insertion of a shortcut key designation in Developer Studio menu editor.



ID: ID_MYFILE_CLOSE
Caption: &Close\tCtrl+C

□ Separator □ Pop-up □ Inactive Break: None

□ Checked □ Grayed □ Help

Prompt:

Figure 19-6 *Developer Studio Insertion of a Shortcut Key Code*

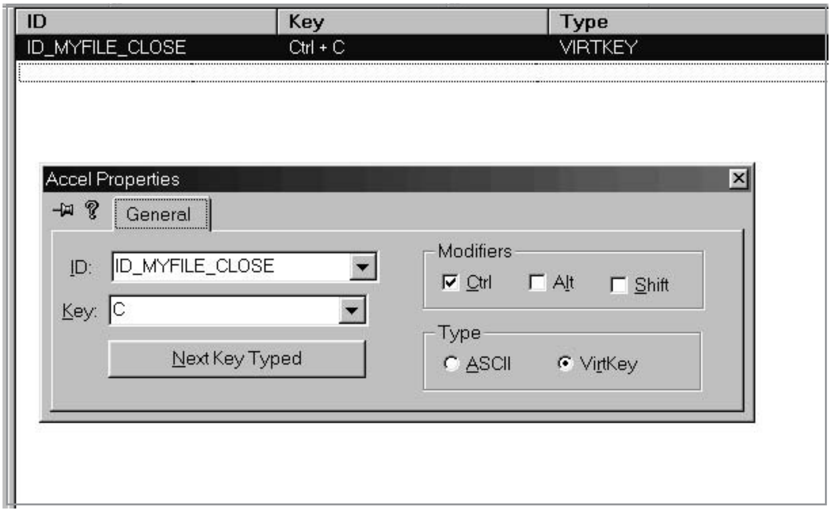
But the shortcut key label is only a caption and has no effect on the processing. In order to associate a shortcut key with a menu item you must create an accelerator table. The following steps can be followed:

1. Select Resource from the Developer Studio Insert menu. Select the Accelerator resource type in the dialog box and click New.

2. Create an accelerator table. The table includes an identification field that contains the resource ID, a key field for the keystroke that activates the shortcut, and a type field that specifies the properties of the key. Figure 19-7 shows the Accel Properties dialog box in the accelerator editor.

3. Once created, the accelerator table becomes a program resource whose name can be found in the Resource tab of Developer Studio project workspace pane, or by clicking the Resource Symbols command in the View menu or its corresponding toolbar button. Developer Studio assigns the name IDR_ACCELERATOR1 to the first accelerator table; normally, there is one per application.

4. The accelerator table must now be loaded into the application and processed so that the corresponding messages are sent to the windows[check] procedure. This requires using the LoadAccelerator() function. Its parameters are the handle to the program's instance and an identifier of the accelerator table. LoadAccelerator() returns a handle to the accelerator, of type HACCEL. Processing of accelerator keys is by means of the TranslateAccelerator() function, which takes as parameters the handle to the window whose messages are to be translated, the handle to the accelerator table returned by LoadAccelerator(), and a pointer to a message structure. Both functions are usually included in WinMain(), as in the following code fragment:



ID	Key	Type
ID_MYFILE_CLOSE	Ctrl + C	VIRTKEY

Accel Properties

General

ID: ID_MYFILE_CLOSE

Key: C

Modifiers: ☑ Ctrl □ Alt □ Shift

Type: ○ ASCII ● VirtKey

Figure 19-7 *Developer Studio Accelerator Editor*

```
LRESULT CALLBACK WinMain (HINSTANCE hInstance, HINSTANCE
                          hPrevInstance, PSTR szCmdLine,
                          int iCmdShow) {
     static char szAppName[] = "Demo" ;
     HWND         hwnd ;
     MSG          msg ;
     HACCEL       hAccel;          // Handle to accelerator
    .
    .
    .
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;
    // Load accelerators
       hAccel = LoadAccelerators (hInstance,
                   MAKEINTRESOURCE (IDR_ACCELERATOR1));
     // Message loop
     while (GetMessage (&msg, NULL, 0, 0)) {
          if (!TranslateAccelerator (hwnd, hAccel, &msg)) {
             TranslateMessage (&msg) ;
             DispatchMessage (&msg) ;
          }
     }
     return msg.wParam ;
}
```

19.2.4 Pop-Up Menus

A pop-up menu is a context-sensitive submenu that is activated by clicking the right mouse button. The pop-up menu is unrelated to the application's main menu and implemented differently. The items in a pop-up menu should be related to the context in which the right mouse button is pressed. Therefore, in a full-featured application, the processing usually requires calculating the screen coordinates where the mouse action takes place, or the object currently selected, in order to determine which, among several pop-up menus, is to be activated.

As with the program's main menu, there are several methods for creating a pop-up menu. You can use the menu editor to create a pop-up menu; however, a little trickery is required since pop-up menus have no title and the menu editor does not allow creating menu items without first entering the title. The following steps can be used to create and install a simple pop-up menu:

1. Use the menu editor to create the pop-up menu. In order to create a drop down menu you have to enter a temporary menu title. Since this title is used by Developer Studio name mangler to create the item id, it may be a good idea to used the menu title "popup1."
2. Under the temporary menu title (popup1 is the suggested one), enter the menu items as you would for a program menu. You can use all the attributes available and there can be child menus in the pop-up. Once you have finished creating the menu, double-click on the temporary menu title (popup1) and erase all the characters in the caption field. This creates a drop down menu with no menu title. To see the drop down menu you have to click on the left corner of the menu editor's title bar. This can be a little deceptive, since at times it may seem that the drop down menu has disappeared.
3. When you close the menu editor, a new menu resource appears in the Resource tab of the Program window. If this is your second menu it is named IDR_MENU2. The new menu is now included in your script resource file.
4. You need to load the pop-up menu and obtain its handle. This can be done in the WM_CREATE message intercept of the window that contains it. It requires the use of the LoadMenu() function, which returns a handle to the menu resource. The GetSubMenu() function converts this handle into a submenu handle, which can then be used by the code. Processing is usually as follows:

```
static  HMENU   pMenu;    // Handle to pop-up menu
     .

      .
      .
case WM_CREATE:
    hdc = GetDC(hwnd);
    // Get handle to pop-up menu
    pMenu = LoadMenu(pInstance,
                    (MAKEINTRESOURCE(IDR_MENU2)));
    pMenu = GetSubMenu(pMenu, 0);
    return 0;
```

5. Once you have its handle, the pop-up menu can be displayed. The TrackPopupMenu() function is used to define the screen location where the pop-up menu is shown, its position relative to the mouse cursor, and to define which mouse button actions, if any, are tracked when an item is selected. If the pop-up menu is activated by the right mouse button, as is usually the case, then the menu display code can be placed at the corresponding message intercept, as in the following code fragment.

```
case WM_RBUTTONDOWN:
    // Get mouse coordinates
    aPoint.x = LOWORD(lParam);
    aPoint.y = HIWORD(lParam);
    ClientToScreen(hwnd, &aPoint);
    TrackPopupMenu(pMenu,
                   TPM_LEFTALIGN | TPM_TOPALIGN |\
                   TPM_LEFTBUTTON,
                   aPoint.x, aPoint.y,
                   0,
                   hwnd,
                   NULL);
    return 0;
```

In the preceding code sample we start by obtaining the mouse coordinates from the lParam. One problem is that TrackPopupMenu() requires the horizontal and vertical coordinates in screen units, and the WM_RBUTTONDOWN message intercept reports the mouse position in client area units. For this reason, the ClientToScreen() function is necessary to convert client area into screen coordinates.

The TrackPopupMenu() function displays the pop-up menu. Its first parameter is the handle to the menu obtained during WM_CREATE processing. The second parameter is one or more bitwise constants. In this case we have established that the display position is relative to the upper left corner of the menu box, and that the left mouse button is the one tracked for menu selections. The display points are entered as the third and fourth parameters to the call. The fifth one is reserved (must be zero), the sixth one is the handle to the window that owns the pop-up menu, and the last one defines a RECT-type structure in which the user can click without erasing the pop-up menu. If this value is NULL then the shortcut menu disappears if the user clicks outside of its area.

6. Intercepting action on the pop-up menu is at WM_COMMAND message processing. For example, if the id of the first item in the pop-up menu is ID_POPUP1_UNDO, then the case statement at the intercept point has this label, as follows:

```
case WM_COMMAND:
    switch (LOWORD (wParam)) {
    .
    .
    .
    case ID_POPUP1_UNDO:
    // Assert:
    //   User clicked "undo" item on pop-up menu
```

19.2.5 The Menu Demonstration Program

The program named MEN_DEMO, contained in the Menu Demo project folder on the book's software package, is a trivial demonstration of an application with a main menu, a shortcut key (accelerator) to access one of the menu items, and a pop-up menu that is displayed when the user right-clicks on the client area. Processing consists of a message box that lists the menu item selected by the user.

19.3 Dialog Boxes

Dialog boxes are a programming aid; they provide no new functionality. Everything that can be done in a dialog box can also be done in a child window, as described earlier in this chapter.

What dialog boxes do for the programmer is to prepackage a series of functions that are frequently needed. Also, dialog boxes perform much of the processing and housekeeping operations for you. They handle the keyboard focus, passing keyboard input from one control to another one, they monitor mouse movements, and they provide a special procedure for tracking action on the controls contained in the dialog box. When used in conjunction with the dialog box editor in Developer Studio, dialog boxes are easy to create and implement in code.

Windows 3.1 introduced an extension to the concept of dialog boxes, usually called the common dialog boxes. The common dialog boxes are a set of prepackaged services for operations that are usually required in many applications. These include opening and saving files, selecting a font, selecting or changing color attributes, searching and replacing text strings, and controlling the printer. The common dialog boxes are discussed later in this section.

19.3.1 Modal and Modeless

There are two general types of dialog boxes: those that suspend the application until the user interacts with the dialog box, and those that do not. The first type, which are the most common ones, are called modal dialog boxes. The second type, which are often seen in floating toolbars, are called modeless dialog boxes. Modal dialog boxes do not prevent the user from switching to another application, although, upon return to the original thread, it is the modal dialog box that retains the foreground. The Windows Interface Guidelines for Software Design (see Bibliography) recommends that modal dialog boxes should have an OK button, to accept and process input, and a Cancel button to abort execution and discard the users action with the dialog box.

19.3.2 The Message Box

The simplest of all dialog boxes is used to display a message on the screen, which the user acknowledges having read by pressing a button. A special function in the Windows API allows creating message boxes directly, without having to use the dialog box editor or manipulate a program resource. The message box contains a title, a message, any one of several predefined icons, and one or more pushbuttons. The general form of the function call is as follows:

```
int MessageBox(hwnd, lpText, lpCaption, uType);
```

where hwnd is the handle to the window that owns the message box, lpText is a pointer to the text message to be displayed (or the message string itself), lpCaption is a pointer to the caption (or the caption string itself), and uType is one of several bit flags that control the behavior of the message box. Table 19-6 lists the most useful bit flags used in the MessageBox() function.

Table 19-6

Often Used Message Box Bit Flags

SYMBOLIC CONSTANT	MEANING
MB_ABORTRETRYIGNORE	Contains three push buttons: Abort, Retry, and Ignore.
MB_OK	Contains one push button: OK. This is the default.
MB_OKCANCEL	Contains two push buttons: OK and Cancel.
MB_RETRYCANCEL	The message box has two push buttons: Retry and Cancel.
MB_YESNO	Contains two push buttons: Yes and No.
MB_YESNOCANCEL	Contains three push buttons: Yes, No, and Cancel.

(continues)

Table 19-6

Often Used Message Box Bit Flags (continued)

SYMBOLIC CONSTANT	MEANING
Icon Flags:	
MB_ICONEXCLAMATION	Exclamation-point icon.
MB_ICONWARNING	Exclamation-point icon.
MB_ICONINFORMATION	Question mark icon.
MB_ICONASTERISK	Lowercase letter i icon in a circle.
MB_ICONQUESTION	Question-mark icon.
MB_ICONSTOP	Stop-sign icon.
MB_ICONERROR	Hand icon.
MB_ICONHAND	Hand icon.
Default Button Flags:	
MB_DEFBUTTON1	The first button is the default button.
MB_DEFBUTTON2	The second button is the default button.
MB_DEFBUTTON3	The third button is the default button.
MB_DEFBUTTON4	The fourth button is the default button.
Modality Flags:	
MB_APPLMODAL	User must respond to the message box before continuing work in the window. However, the user can move to the window of another application and work in those windows.
MB_SYSTEMMODAL	Same as MB_APPLMODAL except that the message box has the WS_EX_TOPMOST style. Use system-modal message boxes to notify the user of serious errors that require immediate attention.
MB_TASKMODAL	Same as MB_APPLMODAL except that all the top-level windows belonging to the current task are disabled if the hwnd parameter is NULL.
Other Flags:	
MB_HELP	Adds a Help button to the message box. Choosing the Help button or pressing F1 generates a Help event.
MB_RIGHT	The text is right-justified.
MB_SETFOREGROUND	The message box becomes the foreground window. Internally, Windows calls the SetForegroundWindow function for the message box.
MB_TOPMOST	Message box is created with the WS_EX_TOPMOST window style.

For example, the following statement creates a message box labeled "Menu Action," with the text string "File Close Requested," which contains an exclamation sign icon, and a button labeled OK:

```
MessageBox (hwnd,
           "File Close Requested",
           "Menu Action",
            MB_ICONEXCLAMATION | MB_OK);
```

Figure 19-8 shows the resulting message box.

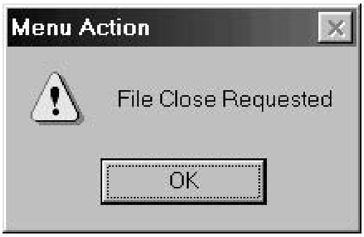


Figure 19-8 *Simple Message Box*

19.3.3 Creating a Modal Dialog Box

Developer Studio provides a dialog box editor, which is a tool for creating dialog boxes. Once the dialog box has been created, it becomes another program resource that can be referenced in the code. The dialog box editor can be used to create simple message boxes; however, in this case it is easier to use the MessageBox() function described in the previous section. Dialog boxes are useful when they are used to obtain user input.

A unique feature of dialog boxes is that they contain their own processing. In a sense, the dialog box procedure is like your window procedure.

You create a modal dialog box by means of the DialogBox() function, with the following standard form:

```
int DialogBox (hInstance, lpTemplate, hwndParent, lpDiaProc);
```

where hInstance is the handle to the program instance that contains the dialog box, lpTemplate identifies the dialog box template or resource, hwndParent is the handle to the owner window, and lpDiaProc is the name of the dialog box procedure. It is this procedure that receives control when the dialog box is created. The following code fragment shows the creation of a dialog box at the time that a menu command with the id ID_DIALOG_ABOUT is intercepted:

```
ID_DIALOG_ABOUT:
  DialogBox (pInstance,
             MAKEINTRESOURCE (IDD_DIALOG1),
             hwnd,
             (DLGPROC) AboutDlgProc);
```

In this case the dialog box resource is named IDD_DIALOG1, and the dialog box procedure that receives control is AboutDlgProc(). The dialog box procedure's general form is as follow:

```
BOOL DialogProc (hwndDlg, uMsg, wParam, lParam);
```

where DialogProc is the name of the procedure defined in the lpDiaProc field of the DialogBox() function. The first parameter passed to the dialog procedure (hwndDlg) is the handle to the dialog box. The second one is the Windows message. The wParam and lParam values contain message-specific information, as is the case in the window procedure.

As soon as the dialog box is created, and before it is displayed, Windows sends the WM_INITDIALOG message to the dialog box procedure. Typically, the dialog box procedure intercepts the message to initialize controls and perform other housekeeping functions. In WM_INITDIALOG the wParam contains the handle to the control that has focus, which is the first visible and not disabled control in the box. The application returns TRUE to accept this default focus. Alternatively, the application can set the focus to another control, in which case it returns FALSE.

The dialog box procedure receives messages for the controls in the dialog box. These messages can be intercepted in the same manner as those sent to the window procedure. The following code fragment is a dialog box procedure for a dialog box that contains a single button:

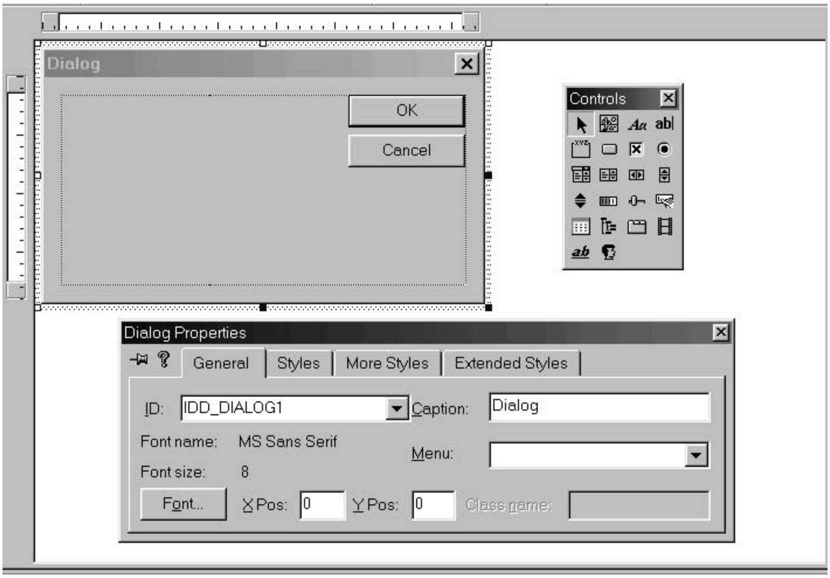
```
BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT iMsg, WPARAM wParam,
                            LPARAM lParam) {
 switch (iMsg) {
    case WM_INITDIALOG :
      return TRUE ;
    // Dialog box controls message intercepts
    case WM_COMMAND :
      switch (LOWORD (wParam)) { // Get control id
         case IDOK :
           EndDialog (hDlg, 0) ;
           return TRUE ;
         }
    break ;
    }
  return FALSE ;
  }
```

Notice that, unlike a window function, AboutDlgProc() does not return control via the default window procedure. In general, a dialog box procedure returns FALSE to indicate that default processing is to be provided by Windows and TRUE when no further processing is required. The exception is the WM_INITDIALOG message in which the return value refers to the acceptance or rejection of the default focus, as discussed previously.

Notice that dialog procedures, like all window procedures, have to be of type CALLBACK. Failing to declare a window procedure, or a callback procedure, with this type, can be the source of unpredictable errors, such as the General Protection Fault.

You can create a dialog box by means of the following steps:

1. Select Resource from the Developer Studio Insert menu. Select the Dialog resource type in the dialog box and click New.
2. The dialog box editor executes by displaying a blank form and a floating toolbox containing controls that can be inserted in the dialog box. If the toolbar is not visible, you can show it on the editor screen by opening the Tools menu, then selecting Customize, and checking the Controls box in the Toolbars tab. The controls include all those already mentioned and some others. To add a control to the dialog box you drag it onto the form and then use the handles to size it. Double-clicking on the form, or on one of the controls, displays a Dialog Properties window which allows defining the attributes of that particular element. Figure 19-9, on the following page, shows the dialog box editor with the Dialog Properties windows for the form and the Controls toolbox.
3. Once you have finished creating the dialog box, click the Close button of the menu editor window. If the application already has a script file, the dialog box is added to it. If not, Developer Studio prompts you to save the new script file.
4. Skip this step if a script file has already been inserted into the project. If not, open the Project menu, select Add to Project, and then select Files. In the Insert Files into Project dialog box, select the script file and then click OK. The script file now appears in Developer Studio project workspace pane.



ID: IDD_DIALOG1
Caption: Dialog
Font name: MS Sans Serif
Font size: 8
X Pos: 0
Y Pos: 0

Figure 19-9 *Developer Studio Dialog Editor*

5. Select Resource View button in project workspace pane and click + on Script Resources. Click + on Dialog. Note the identifier name for the dialog box resource, (usually IDD_DIALOG1) if this is the first dialog box created.
6. Your code must now create the dialog box, usually by intercepting the corresponding menu command and calling DialogBox(). Also, the dialog box procedure has to intercept the WM_INITDIALOG message and provide handlers for the controls contained in the box, as previously described.

19.3.4 Common Dialog Boxes

Windows 3.1 introduced the common dialog boxes as a set of prepackaged services for performing routine operations required in many applications. The idea behind them is to standardize frequent input functions so that they appear the same in different program functions and even in different applications. For example, the common dialog box used to select a filename and to browse through the disk storage system is the same if you are opening or saving a file. Furthermore, two applications that manipulate files can use the same common dialog box, giving the user a familiar interface. The following operations can be performed by means of common dialog boxes: opening and saving files, selecting fonts, selecting or changing color attributes, searching and replacing text strings, and controlling the printer. Common dialog boxes have a modal behavior, that is, the program is suspended until the user closes the dialog box.

Common dialog boxes are processed internally by Windows; therefore, they do not have a dialog box procedure.

The identification for the menu item or other resource that activates the common dialog box usually serves as the message intercept. The processing is done directly in the intercept routine. Each common dialog box is associated with a structure that is used to pass information to it and to receive the results of the user's action. Programs that use common dialog boxes should include the commdlg.h header file.

For example, the menu item ID_DIALOG_COLORSELECTOR can intercept user action in WM_COMMAND message processing and then proceed to fill a variable of the structure type CHOOSECOLOR (see Appendix A) as in the following code fragment:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                          LPARAM lParam) {
  HDC          hdc ;
  TEXTMETRIC   tm ;
  HBRUSH       hBrush;      // Handle to brush

  // Variables for color and font common dialog
  static CHOOSECOLOR cc ;   // Structure
  static COLORREF    custColors[16] ; // Array for custom
                                      // colors
  int    i;            // counter for custom color display
  switch (iMsg) {
     ...
    case WM_COMMAND:
       switch (LOWORD (wParam)) {
        // Color Selector common dialog
          case ID_DIALOG_COLORSELECTOR:
          cc.lStructSize    = sizeof (CHOOSECOLOR) ;
          cc.hwndOwner      = hwnd ;
          cc.hInstance      = NULL ;
          cc.rgbResult      = RGB (0x80, 0x80, 0x80) ;
          cc.lpCustColors   = custColors ;
          cc.Flags          = CC_RGBINIT | CC_FULLOPEN ;
          cc.lCustData      = 0L ;
          cc.lpfnHook       = NULL ;
          cc.lpTemplateName = NULL ;
          ...
```

Once the structure variable is filled with the necessary data, the application can call the ChooseColor() function to display the common dialog box. ChooseColor() requires a single parameter: the address of the previously mentioned structure. Most of the structure members are obvious. The lpCustColors member is an array of 16 COLORREF-type values that holds the RGB values for the custom colors in the dialog box. The Flags members are bit flags that determine the operation of the dialog box. In the previous example we set CC_RGBINIT bit so that the rgbResult member holds the initial color selection. The values 0x80 for each of the red, green, and blue components produce a middle gray color. The last three members of the structure are used for customizing the dialog box. The constant CC_FULLOPEN causes the dialog box to open in the full display mode, that is, with the controls necessary for the user to create custom colors.

ChooseColor() returns TRUE if the user clicks the OK button on the dialog box. Therefore, the coding continues as follows:

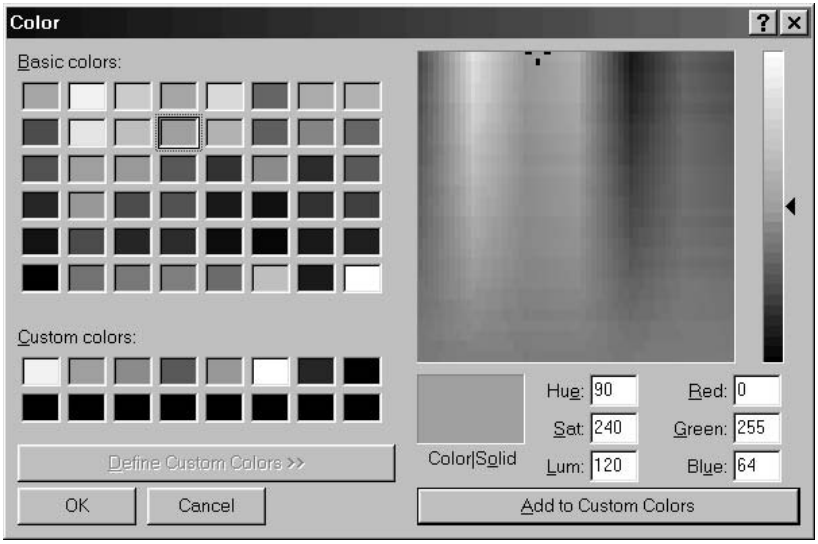
```
if (ChooseColor (&cc) == TRUE) {
// ASSERT:
```

```
//   structure members have color data selected by user
// Clear the client window
hdc = GetDC(hwnd);
InvalidateRect (hwnd, NULL, TRUE) ;
UpdateWindow (hwnd) ;
```

The colors selected by the user are stored in two members of the CHOOSECOLOR structure: rbgResult holds the solid color box, and the array variable custColor holds the 16 custom colors. The code now creates a solid brush, using the color stored in the rgbResult member, and displays a rectangle filled with this color. Then a loop displays the first eight of the 16 custom colors:

```
hBrush = CreateSolidBrush(cc.rgbResult);
// Select the brush in the DC
SelectObject (hdc, hBrush) ;
// Draw a rectangle using the brush
Rectangle (hdc, 20, 20, 100, 100) ;
// Display first eight custom colors using the
// color triplets stored in the custColors array
    for (i = 0; i < 8; i++) {
        hBrush = CreateSolidBrush(custColors[i]);
                 SelectObject (hdc, hBrush) ;
                 Rectangle (hdc, 20+(20 * i), 120,
                 40+(20 * i), 140) ;
         }
     // Clear and exit
     DeleteObject (SelectObject (hdc, hBrush)) ;
     ReleaseDC (hwnd, hdc);
}
return 0 ;
  ...
```

Figure 19-10 shows the color dialog box as displayed by this code.



Hue: 90
Sat: 240
Lum: 120
Red: 0
Green: 255
Blue: 64

Figure 19-10 *Color Selection Common Dialog Box*

19.3.5 The Dialog Box Demonstration Program

The program named DIA_DEMO, contained in the Dialog Box Demo project folder on the book's software package, is a trivial demonstration of several dialog boxes. The Dialog menu contains commands for creating a modeless dialog box, three different modal dialog boxes (one of them with a bitmap), and for the color and font common dialog boxes. The code demonstrates how information obtained by modal and common dialog boxes is passed to the application.

19.4 Common Controls

Windows 95 introduced a new set of controls that supplements the ones that existed previously. They are also available in Windows NT version 3.51 and later. These controls, sometimes referred to as the new common controls, allow the implementation of status bars, toolbars, trackbars, progress bars, animation controls, image lists, list view controls, tree view controls, property sheets, tabs, wizards, and rich edit controls. It is evident from this list that one could devote an entire volume to their discussion. Table 19-7 is a list of the Windows common controls first implemented in Windows 95.

Table 19-7

Original Set of Common Controls

CONTROL	DESCRIPTION
Frame Window Controls:	
toolbar	Displays a window with command-generating buttons.
ToolTip	Small pop-up window that describes purpose of a toolbar button or other tool.
status bar	Displays status information at the bottom screen line.
Explorer-type Controls:	
list view	Displays a list of text with icons.
tree view	Displays a hierarchical list of items.
Miscellaneous Controls:	
animation	Displays successive frames of an AVI video clip.
header	Appears above a column of text. Controls width of text displayed.
hotkey	Enables user to perform an action quickly.
image list	A collection of images used to manage large sets of icons or bitmaps. It isn't really a control, but supports lists used by other controls.
progress bar	Indicates progress of a long operation.
rich edit	Allows the user to edit with character and paragraph formatting.
slider	Displays a slider control with optional tick marks.
spin button	Displays a pair of arrow buttons user can click to increment or decrement a value.
tab	Displays divider-like elements used in tabbed dialog boxes or property sheets.

Before we can implement the new common controls, some preliminary steps are required. The reason is that the common controls library is not automatically referenced at link time, nor is it initialized for operation. The following operations are necessary:

1. The common controls library, named Comctl32.lib, must be included in the list of libraries referenced by the linker program. This is accomplished by opening the Project menu and selecting the Settings command. In the Project Settings dialog box, open the

Link tab. The "Object/library modules" edit box contains a list of all the referenced libraries, separated from one another by a space. Position the caret between two library entries and type "Comctl32.lib." Click the OK button.

2. The program code must include the common controls header file. This is accomplished with the statement:

   ```
   #include <commctrl.h>
   ```

3. The InitCommonControls() function must be called before the common controls are used. This function takes no parameters and returns nothing. The initialization can be placed in WinMain(), as follows:

   ```
   InitCommonControls();
   ```

4. Rich edit controls reside in their own library, named Riched32.dll, and have their own header file, named richedit.h. To use library controls your program must include the statement:

   ```
   LoadLibrary ("RICHED32.DLL");
   ```

At this point the application can implement common controls. In this section we sample some of the common controls that are more frequently found in graphics applications, namely toolbars and ToolTip controls. These, together with the status bar controls, are sometimes called the frame window controls. Some of the common controls are available in the toolbar of Developer Studio dialog box editor. The resource editor contains a specific toolbar editor for creating this type of common control. Most common controls can also be created by means of the CreateWindow() or CreateWindowEx() functions. Others have a dedicated function, such as CreateToolbarEx().

19.4.1 Common Controls Message Processing

Most common controls send WM_NOTIFY messages. One notable exception is the toolbar controls, which send WM_COMMAND. In processing common controls messages we follow similar methods as in processing menu selections.

The WM_NOTIFY message contains the ID of the control in wParam and a pointer to a structure in lParam. The structure is either an NMHDR structure or, more frequently, a larger structure that has an NMHDR structure as its first member. The common notifications (whose names start with NM_) and the ToolTip control's TTN_SHOW and TTN_POP notifications are the only cases in which the NMHDR structure is actually used by itself. The format of the NMHDR structure is as follows:

```
typedef struct tagNMHDR {
    HWND hwndFrom;
    UINT idFrom;
    UINT code;
} NMHDR;
```

where hwndFrom is the handle to the controls sending the message, idFrom is the control identifier, and code is one of values in Table 19-8.

Table 19-8

Common Control Notification Codes

CODE	ACTION IN CONTROL OR RESULTS
NM_CLICK	User clicked left mouse button.
NM_DBLCLK	User double-clicked left mouse button.
NM_RCLICK	User clicked right mouse button.
NM_RDBLCLK	User double-clicked right mouse button.
NM_RETURN	User pressed the Enter key.
NM_SETFOCUS	Control has been given input focus.
NM_KILLFOCUS	Control has lost input focus.
NM_OUTOFMEMORY	Control could not complete an operation because there was not enough memory available.

Most often notifications pass a pointer to a larger structure that contains an NMHDR structure as its first member. For example, the list view control uses the LVN_KEYDOWN notification message, which is sent when a key is pressed. In this case the pointer is to an LV_KEYDOWN structure, defined as follows:

```
typedef struct tagLV_KEYDOWN {
    NMHDR hdr;
    WORD wVKey;
    UINT flags;
} LV_KEYDOWN;
```

Since the NMHDR member is the first one in this structure, the pointer in the notification message can be cast to either a pointer to an NMHDR or a pointer to an LV_KEYDOWN.

19.4.2 Toolbars and ToolTips

A toolbar is a window containing graphics buttons or other controls. It is usually located between the client area and the menu bar. Although Windows applications have been using toolbars for a long time, there was no system support for toolbars until the release of the WIN-32 API. The most common use of toolbars is to provide fast access to menu commands. Toolbars often include separators, which are spaces in the toolbar that allow grouping associated buttons. A ToolTip is a small pop-up window that is displayed when the mouse is left on a toolbar button for more than one-half second. ToolTips usually consist of a short text message that explains the function of the toolbar button or control. Figure 19-11 shows a program containing a toolbar with nine buttons.

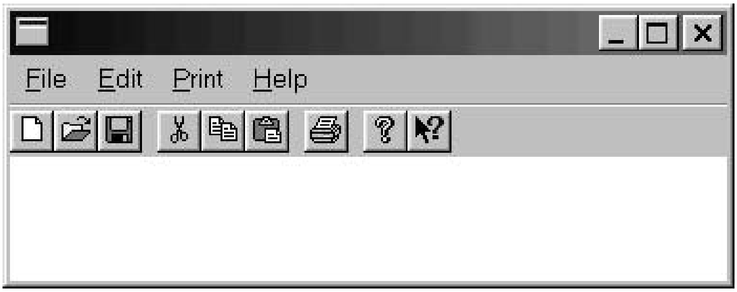


Figure 19-11 *Toolbar*

Note in Figure 19-11 that separators are used to group the toolbar buttons. In this case, the first group of buttons correspond with functions in the File menu, the second group with functions in the Edit menu, and so forth. Normally, not all menu commands have a toolbar button, but only the ones most often used.

19.4.3 Creating a Toolbar

There are several ways to create a toolbar. You can define the toolbar in code, using standard buttons furnished in Visual C++. You can use a pre-made bitmap of toolbar buttons, which can be converted into a toolbar resource and then edited. You can create custom buttons using the toolbar editor. Or you can use a combination of these methods. In this section we follow the simplest method, but even then, you must be careful to perform the steps in the same order in which we list them. The toolbar creation tools in Developer Studio were designed to be used in MFC programming; therefore, the system makes assumptions regarding the order in which the steps are performed. If you are careless in this respect, you may end up having to do some manual editing of the resource files.

We must accept that there are complications in creating a toolbar outside of the MFC, however, much suffering can be avoided if the toolbar is not created until the program menu has been defined. The idea is to use the same identification codes for the toolbar as for the corresponding menu items, such that message processing takes place at the same intercept routine. For example, if the first toolbar button in Figure 19-11 corresponds to the New command in the File menu, then both the button and the menu item could be named ID_FILE_NEW. The same applies to the other buttons in the toolbar. In the following description about the creation of a toolbar we assume that the identification strings have been defined for the corresponding menu entries:

```
File menu:
         ID_FILE_NEW
         ID_FILE_OPEN'
         ID_FILE_SAVE
Edit menu:
         ID_EDIT_CUT
         ID_EDIT_COPY
         ID_EDIT_PASTE
Print menu:
         ID_PRT_PRINT
Help menu:
         ID_HLP_ABOUT
         ID_HLP_HELP
```

All of the toolbar buttons in Figure 19-11 correspond to standard buttons contained in Developer Studio. These buttons can be loaded into the toolbar by referencing their system names, or by loading a bitmap that contains them. In the current example we use the bitmap approach.

Toolbars require that each of the buttons be defined in a structure of type TBBUTTON (see Appendix A). Your program, usually in the window procedure, creates an array of structures, with one entry for each button in the toolbar. The button separators must be included. In the case of the screen in Figure 19-11, the array of TBBUTTON structure is as follows:

```
// Array for attributes for toolbar buttons
 TBBUTTON tbb[] = {
0, ID_FILE_NEW,   TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
1, ID_FILE_OPEN,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
2, ID_FILE_SAVE,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
0, 0,             TBSTATE_ENABLED, TBSTYLE_SEP,    0, 0, 0, 0,
3, ID_EDIT_CUT,   TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
4, ID_EDIT_COPY,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
5, ID_EDIT_PASTE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
0, 0,             TBSTATE_ENABLED, TBSTYLE_SEP,    0, 0, 0, 0,
6, ID_PRT_PRINT,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
0, 0,             TBSTATE_ENABLED, TBSTYLE_SEP,    0, 0, 0, 0,
7, ID_HLP_ABOUT,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
8, ID_HLP_HELP,   TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0, }; /*
 |------------|  |-------------|  |------------|| see note |
 |        |               |               |       below
 |        |               |               |--- One or more
 |        |               |                    button styles
 |        |               |----------- One or more state flags
 |        |-------------------------- Command ID mapped to
 |                                     button
 |------------------- Zero-based index to button image in
 |                    bitmap (excluding separators)
Note:
      0, 0, 0, 0
      |  |  |  |------------ index of button string
      |  |--|--------------- application defined value
      |--------------------- Reserved
*/
```

Table 19-9 lists the style flags used with toolbars.

Table 19-9

Toolbar and Toolbar Button Style Flags

STYLE	DESCRIPTION
Toolbar Styles:	
BSTYLE_ALTDRAG	Allows the user to change the position of a toolbar button by dragging it while holding down the Alt key. If this style is not specified, the user must hold down the Shift key while dragging a Button. Note that the CCS_ADJUSTABLE style must be specified to enable toolbar buttons to be dragged.
TBSTYLE_TOOLTIPS	Creates a ToolTip control that an application can use to display descriptive text for the buttons in the toolbar.
TBSTYLE_WRAPABLE	Creates a toolbar that can have multiple lines of buttons. Toolbar buttons can "wrap" to the next line when the toolbar becomes too narrow to include all buttons on the same line. Wrapping occurs on separation and non-group boundaries.
Toolbutton Styles:	
TBSTYLE_BUTTON	Creates a standard push button.
TBSTYLE_CHECK	Button toggles between the pressed and not pressed states each time the user clicks it. The button has a different background color when it is in the pressed state.
TBSTYLE_CHECKGROUP	Creates a check button that stays pressed until another button in the group is pressed.

(continued)

Table 19-9

Toolbar and Toolbar Button Style Flags (continues)

STYLE	DESCRIPTION
Toolbutton Styles:	
TBSTYLE_GROUP	Creates a button that stays pressed until another button in the group is pressed.
TBSTYLE_SEP	Creates a separator. A button that has this style does not receive user input and is not assigned a button number.

Table 19-10 lists the toolbar states

Table 19-10

Toolbar States

TOOLBAR STATE	DESCRIPTION
TBSTATE_CHECKED	The button has the TBSTYLE_CHECKED style and is being pressed.
TBSTATE_ENABLED	The button accepts user input. A button not having this state does not accept user input and is grayed.
TBSTATE_HIDDEN	The button is not visible and cannot receive user input.
TBSTATE_INDETERMINATE	The button is grayed.
TBSTATE_PRESSED	The button is being pressed.
TBSTATE_WRAP	A line break follows the button. The button must also have the TBSTATE_ENABLED state.

The bitmap for the toolbar in Figure 19-11 is furnished with Developer Studio. We have made a copy of this bitmap and you can find it in the Resource directory on the book's software package. The name of the bitmap is toolbar.bmp. The process of creating a toolbar from a toolbar bitmap requires that you follow a certain sequence. The price to pay for changing the order of operations is that you may end up with incorrect resource files that must be manually edited. The following operations result in the toolbar resource:

1. Select Resource from the Insert menu. Select the Bitmap resource type and click the Import button.
2. In the Import Resource dialog editor, edit the filename field for that of a bitmap file. This is accomplished by entering "*.bmp". Now you can search though the file system until you find the toolbar bitmap. In this case the desired bitmap has the name "toolbar.bmp." Select the bitmap and click on the button labeled Import.
3. The toolbar bitmap is now loaded into the bitmap editor. The toolbar bitmap is shown in Figure 19-12. The buttons are labeled according to the identifications assigned in the TBBUTTON structure members listed previously.
4. Now you must convert the bitmap into a toolbar resource. This is accomplished by opening the Image menu and clicking on the Toolbar editor command. The New Toolbar Resource dialog box with the pixel size of normal toolbar buttons is displayed, which is 16 pixels wide and 15 pixels high. Click OK and the toolbar editor appears with the bitmap converted into a toolbar.

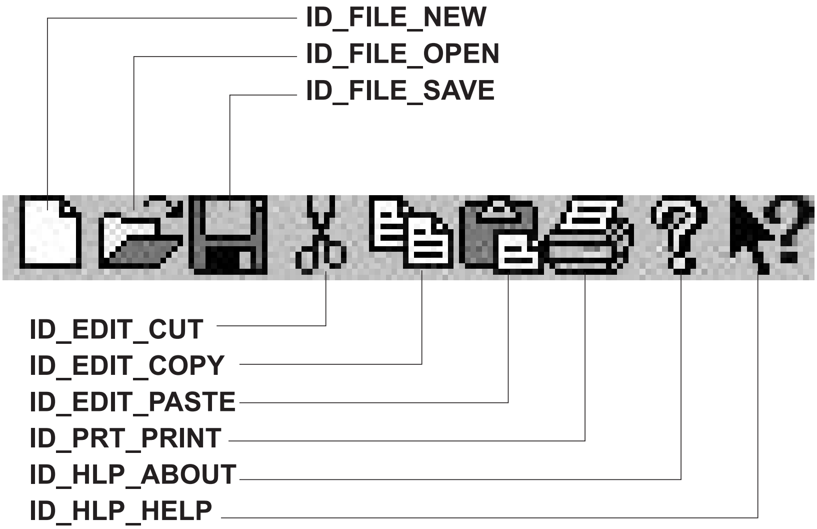
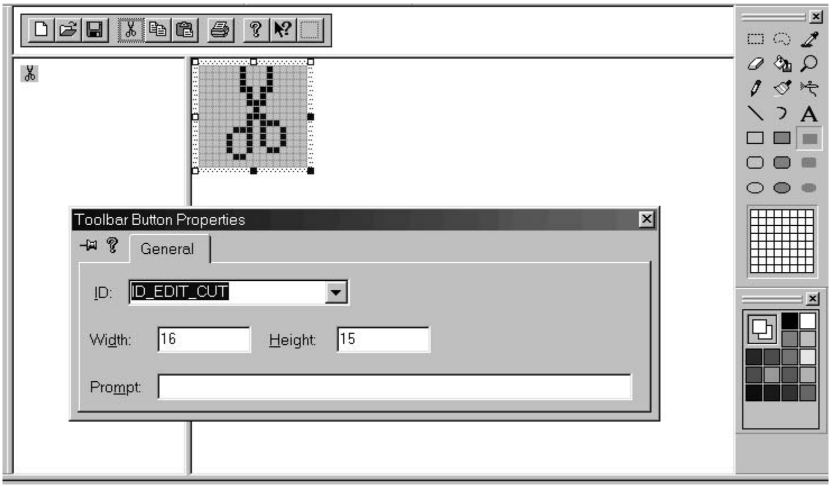


Figure 19-12 *"Toolbar.bmp" Button Identification Codes*

5. You can now proceed to edit the toolbar and assign identification codes to each of the buttons. Note that there is a blank button at the end of the toolbar, which is used for creating custom buttons. You can click on the blank button and use the editor to create a new button image. To delete a button, click on it and drag it off the toolbar. To reposition a button, click on it and drag it to it new location. To create a space in the toolbar drag the button so that it overlaps half the width of its neighbor button. To assign an identification code to a toolbar button, double-click on the button and enter the new identification in the ID: edit box of the Toolbar Button Properties dialog box. At this time you may enter the corresponding identification codes for all the buttons in the toolbar. Figure 19-13 shows the toolbar editor once the separators have been inserted.


Toolbar Button Properties

General

ID: ID_EDIT_CUT

Width: 16 Height: 15

Prompt:

Figure 19-13 *Developer Studio Toolbar Editor*

Figure 19-13 also shows the Toolbar Button Properties dialog box open and the new identification code in the ID: edit box.

6. Once the identification codes have been assigned to all the buttons, click the button labeled X to close the editor. Also close the next screen and save the resource file under the default name, or assign it a new one. Some programmers like to give the resource file the same base name as the application's main module. The extension for the resource file must be .RC.
7. The next step is one that you have already done for other resources: open the Project menu, click the Add To Project command, select Files, and add the resource file to the project. The toolbar is now in the project. You can use the Resource Symbols command in the View menu, or the corresponding toolbar button, to make sure that the identification codes are correct and coincide with those in the BBUTTON structure, and the menu items.
8. Displaying the toolbar requires a call to the CreateToolbarEx() function. The call returns a handle to the toolbar, which is of type HWND since the toolbar is a window. In this example, the call is as follows:

```
#define     ID_TOOLBAR    400   // Toolbar id number
     .
     .
     .
HWND        tbHandle;      // Handle to the toolbar
     .
     .
     .
case WM_CREATE:
  // Create toolbar
  tbHandle = CreateToolbarEx (hwnd,  // Handle to window
             WS_CHILD | WS_VISIBLE |
             WS_CLIPSIBLINGS |
             CCS_TOP | TBSTYLE_ToolTipS, // Window styles
             ID_TOOLBAR,             // Toolbar identifier
             9,                      // Number of button images
                                     // in toolbar bitmap
             hInst,                  // Module instance
             IDB_BITMAP1,            // Bitmap ID
             tbb,                    // TBBUTON structure
             12,                     // Number of buttons
                                     // plus separators
             0, 0, 0, 0,
             sizeof (TBBUTTON));
```

The second parameter in the call refers to controls bits that define the style, position, and type of toolbar. The window style WS_CHILD is always required and most toolbars use WS_VISIBLE and WS_CHILDREN. The bits with the CCS_ prefix are common control styles. Table 19-11 lists the common control styles that refer to toolbars.

In the current call to CreateToolbarEx() we used CCS_TOP and the TBSTYLE_TOOLTIPS in order to create a toolbar displayed above the application's client area, and to provide ToolTip support.

Table 19-11

Toolbar Common Control Styles

STYLE	DESCRIPTION
CCS_ADJUSTABLE	Allows toolbars to be customized by the user. If this style is used, the toolbar's owner window must handle the customization notification messages sent by the toolbar.
CCS_BOTTOM	Causes the toolbar to position itself at the bottom of the parent window's client area and sets the width to be the same as the parent window's width.
CCS_NODIVIDER	Prevents a two-pixel highlight from being drawn at the top of the control.
CCS_NOHILITE	Prevents a one-pixel highlight from being drawn at the top of the control.
CCS_NOMOVEY	Causes the toolbar to resize and move itself horizontally, but not vertically, in response to a WM_SIZE message. If the CCS_NORESIZE style is used, this style does not apply.
CCS_NOPARENTALIGN	Prevents the toolbar from automatically moving to the top or bottom of the parent window. Instead, the it keeps its position within the parent window despite changes to the size of the parent window.
CCS_NORESIZE	Prevents the toolbar from using the default width and height when setting its initial size or a new size. Instead, the control uses the width and height specified in the request for creation or sizing.
CCS_TOP	Causes the control to position itself at the top of the parent window's client. The width is set to the size of the parent window. This is the default style.

19.4.4 Standard Toolbar Buttons

The common controls library contains bitmaps for standard toolbar buttons that can be referenced by name and used by application code. In this case no toolbar bitmap is required; therefore, the button images cannot be edited in Developer Studio. There are a total of 15 button images in two sizes: 24 by 24 pixels and 16 by 16 pixels. When using the standard toolbar buttons, the TBBUTTON structure must be filled differently than when using a toolbar bitmap resource. The parameters of CreateToolbarEx() are also different. The following code fragment shows the TBBUTTON structure for loading all 15 standard toolbar buttons:

```
TBBUTTON tbb[] = {
 // File group
 STD_FILENEW,  ID_FILE_NEW,   TBSTATE_ENABLED, TBSTYLE_BUTTON,
                              0, 0, 0, 0,
 STD_FILEOPEN, ID_FILE_OPEN,  TBSTATE_ENABLED, TBSTYLE_BUTTON,
                              0, 0, 0, 0,
 STD_FILESAVE, ID_FILE_SAVE,  TBSTATE_ENABLED, TBSTYLE_BUTTON,
                              0, 0, 0, 0,
 0,            0,             TBSTATE_ENABLED, TBSTYLE_SEP,
                              0, 0, 0, 0,
 // Edit group
 STD_COPY,     ID_EDIT_COPY,  TBSTATE_ENABLED, TBSTYLE_BUTTON,
                              0, 0, 0, 0,
 STD_CUT,      ID_EDIT_CUT,   TBSTATE_ENABLED, TBSTYLE_BUTTON,
                              0, 0, 0, 0,
 STD_PASTE,    ID_EDIT_PASTE, TBSTATE_ENABLED, TBSTYLE_BUTTON,
                              0, 0, 0, 0,
STD_FIND,      ID_EDIT_FIND,  TBSTATE_ENABLED, TBSTYLE_BUTTON,
                              0, 0, 0, 0,
STD_REPLACE,   ID_EDIT_REPLACE,  TBSTATE_ENABLED,
                              TBSTYLE_BUTTON,
                              0, 0, 0, 0,
STD_UNDO,      ID_EDIT_UNDO,  TBSTATE_ENABLED, TBSTYLE_BUTTON,
                              0, 0, 0, 0,
STD_REDOW,     ID_EDIT_REDO,  TBSTATE_ENABLED, TBSTYLE_BUTTON,
                              0, 0, 0, 0,
STD_DELETE,    ID_EDIT_DELETE, TBSTATE_ENABLED,
                              TBSTYLE_BUTTON,
                              0, 0, 0, 0,
0,             0,             TBSTATE_ENABLED, TBSTYLE_SEP,
                              0, 0, 0, 0,
// Print group
STD_PRINTPRE, ID_PRINT_PREVIEW, TBSTATE_ENABLED,
                              TBSTYLE_BUTTON,
                              0, 0, 0, 0,
STD_PRINT,     ID_PRINT_PRINT, TBSTATE_ENABLED,
                              TBSTYLE_BUTTON,
                              0, 0, 0, 0,
0,             0,             TBSTATE_ENABLED, TBSTYLE_SEP,
                              0, 0, 0, 0,
// Help and properties group
STD_PROPERTIES,ID_PROPS,      TBSTATE_ENABLED, TBSTYLE_BUTTON,
                              0, 0, 0, 0,
STD_HELP,       ID_HELP,      TBSTATE_ENABLED, TBSTYLE_BUTTON,
                              0, 0, 0, 0, } ;
```

The call to CreateToolbarEx() is also different. The fourth parameter, which indicates the number of button images in the toolbar bitmap is set to zero in the case of standard buttons. The fifth parameter, which in the case of a toolbar bitmap is set to the application instance, is now the constant HINST_COMMCTRL defined in the common controls library. The sixth parameter is the constant IDB_STD_SMALL_COLOR. The resulting call to CreateToolbarEx() is as follows:

```
tbHandle = CreateToolbarEx (hwnd,
           WS_CHILD | WS_VISIBLE | CCS_TOP |
           TBSTYLE_WRAPABLE,
           ID_TOOLBAR,          // Toolbar ID number
           0,                   // Number of bitmaps (none)
           (HINSTANCE)HINST_COMMCTRL, // Special resource
                                      // instance for
                                      // standard buttons
           IDB_STD_SMALL_COLOR, // Bitmap resource ID
           tbb,                 // TBBUTTON variable
           18,                  // Count of buttons plus
                              // separators
           0, 0, 0, 0,          // Not required for standard
                                // buttons
           sizeof (TBBUTTON));
```

The program named TB1_DEMO, located in the Toolbar Demo No 1 project folder in the book's software package, is a demonstration of using the standard toolbar buttons. When you click on any of the toolbar buttons, a message box is displayed that contains the button's name. Figure 19-14 is a screen snapshot of the TN1_DEMO program.

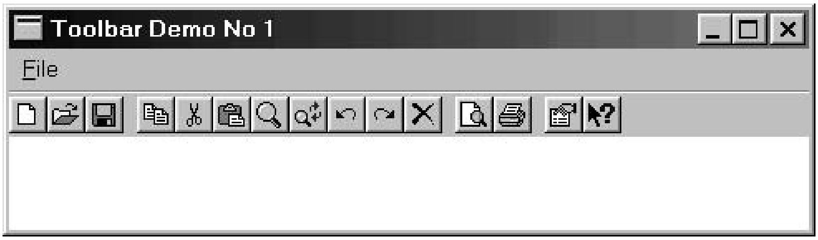


Figure 19-14 *TB1_DEMO Program Screen*

19.4.5 Combo Box in a Toolbar

Windows programs, including Developer Studio, often contain a combo box as part of the toolbar. This application of the combo box is a powerful one. For example, the combo box that is part of Developer Studio standard menu bar is used to remember search strings that have been entered by the user. At any time, you can inspect the combo box and select one of the stored strings for a new search operation. Not only does it save you the effort of retyping the string, it is also a record of past searches.

The position of the combo box in the toolbar is an important consideration. If the combo box is to the right of the last button in the toolbar, then it is a matter of calculating the length of the toolbar in order to position the combo box. However, if, as is often the case, the combo box is located between buttons in the toolbar, or at its start, then code must make space in the toolbar. The method suggested by Nancy Cluts in her book *Programming the Windows 95 User Interface* (see Bibliography) is based on adding separators to make space for the combo box. Since each separator is 8 pixels wide, we can calculate that for a 130-pixels-wide combo box we would need at least 17 separators. In many cases a little experimentation may be necessary to find the number of separators.

The creation of the combo box requires calling CreateWindow() with "COMBOBOX" as the fist parameter. If the combo box is to have a series of string items, as is usually the case, then it is created with the style CBS_HASSTRINGS. If the combo box is to have an edit box feature, then the CBS_DROPDOWN style is used. If it is to have a list of selectable items but no editing possibilities, then the CBS_DROPDOWNLIST style is used. The following code fragment shows the creation of a combo box in a toolbar:

```
static HWND  cbHandle;      // Handle to combo box
static HWND  tbHandle;      // Handle to toolbar
  .
  .
  .
cbHandle = CreateWindow ("COMBOBOX",
           NULL,                        // No class name
           WS_CHILD | WS_VISIBLE | WS_BORDER |
           CBS_HASSTRINGS |CBS_DROPDOWNLIST,
           0,                   // x origin
           0,                   // y origin
           130,                 // width
           144,                 // height
```

```
            tbHandle,            // Parent window handle
            (HMENU) IDR_MENU1,   // Menu resource ID
            pInstance,           // Application instance
            NULL);
```

Once the combo box is created, we need to add the text strings with which it is originally furnished. This is accomplished by a series of calls to SendMessage() with the message code CB_INSERTSTRING. Typical coding is as follows:

```
  char *szStrings[] = { "Visual C++",
                        "Borland C",
                        "Pascal",
                        "Fortran 80",
                        "Visual Basic" };
   .
   .
   .
  //Add strings to combo box
    for (i=0; i < 5; i++)
        SendMessage(cbHandle,
                    CB_INSERTSTRING,
                    (WPARAM)-1,
                    (LPARAM)szStrings[i]);
```

The program TB2_DEMO, located in the Toolbar Demo No 2 project folder on the book's software package, demonstrates the creation of a toolbar that includes a combo box.

19.4.6 ToolTip Support

A ToolTip is a small window that contains a brief descriptive message. Although ToolTips can be activated in relation to any screen object, we are presently concerned with ToolTips associated with a toolbar. For a toolbar to support ToolTips, it must have been created with the TBSTYLE_TOOLTIPS, listed in Table 19-9.

Providing ToolTip support for toolbar buttons is straightforward and simple. However, when you need to furnish ToolTips for other elements in the toolbar, such as the combo box previously mentioned, then ToolTip processing may get more complicated. The first step in creating ToolTips is retrieving a handle for the ToolTip window. This is usually performed in the WM_CREATE message intercept. It consists of calling SendMessage() with the first parameter set to the toolbar handle and the second parameter set to the TB_GETTOOLTIPS message identifier. The following code fragment shows the creation of a three-button toolbar and its corresponding ToolTip window:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                          LPARAM lParam) {
  static HWND     tbHandle;    // Handle to toolbar
  static HWND     hWndTT;      // Handle to ToolTip
     .
     .
     .
   switch (iMsg)
          {
   case WM_CREATE:
      // Create a toolbar
      tbHandle = CreateToolbarEx (hwnd,  // Handle to window
                 WS_CHILD | WS_VISIBLE |
                 WS_CLIPSIBLINGS |
                 CCS_TOP | TBSTYLE_TOOLTIPSS, // Window styles
                 0,                 // Toolbar identifier
                 3,                 // Number of button images
                                    // in toolbar bitmap
                 pInstance,         // Module instance
                 IDB_BITMAP1,       // Bitmap ID
                 tbb,               // TBBUTON structure
                 3,                 // Number of buttons
                                    // plus separators
                 0, 0, 0, 0,        // Not required
                 sizeof (TBBUTTON));
      // Get the handle to the ToolTip window.
          hWndTT = (HWND)SendMessage(tbHandle,
                           TB_GETToolTipS, 0, 0);
            .
            .
            .
```

Once you create the ToolTip window and obtain its handle, the next step is to create and initialize a structure of type TOOLINFO. The coding proceeds as follows:

```
    if (hWndTT) {
      // Fill in the TOOLINFO structure.
      lpToolInfo.cbSize = sizeof(lpToolInfo);
      lpToolInfo.uFlags = TTF_IDISHWND | TTF_CENTERTIP;
      lpToolInfo.hwnd = hwnd;
      lpToolInfo.uId = (UINT)tbHandle;
      lpToolInfo.hinst = pInstance;
      lpToolInfo.lpszText = LPSTR_TEXTCALLBACK;
    }
```

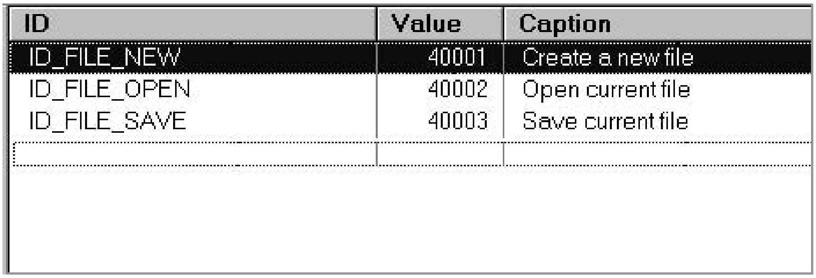
The first flag, TTF_IDISHWND, indicates that the fourth structure member (uId) is a handle to a window, in this case, the toolbar. The flag TTF_CENTERTIP determines that the ToolTip is displayed below the window specified in the uId member, here again, the toolbar. Finally, the lpszText member is set to the constant LPSTR_TEXTCALLBACK, which makes the control send the TTN_NEEDTEXT notification message to the owner window. The values entered in the other structure members are self-explanatory.

Processing of ToolTip messages, as is the case with most controls, takes place at the WM_NOTIFY message intercept. At the time the message handler receives control, the lParam is a pointer to a structure of type HMHDR (see Appendix A), or to a larger structure that has NMHDR as its first member. The third member of the HMHDR structure contains the control-specific notification code. This parameter is TTN_NEEDTEXT when text is required for a ToolTip. Therefore, code can switch on this structure member and provide processing in a case statement, as shown in the following code fragment:

```
   LPNMHDR        pnmh;        // Pointer to HMHDR structure
   TOOLTIPINFO    lpToolTipInfo;
   LPTOOLTIPTEXT  lpToolTipText;
   static char    szBuf[128]; // Buffer for ToolTip text
   .
   .
   .
    case WM_NOTIFY:
       pnmh = (LPNMHDR) lParam;
       switch  (pnmh->code) {
           case TTN_NEEDTEXT:
           // Display ToolTip text.
              lpToolTipText = (LPTOOLTIPTEXT)lParam;
              LoadString (pInstance,
                         lpToolTipText->hdr.idFrom,
                         szBuf,
                         sizeof(szBuf));
                         lpToolTipText->lpszText = szBuf;
              break;
            default:
               return TRUE;
               break;
         }
      return 0;
    break;
```

Note that the TTN_NEEDTEXT message intercept contains a pointer to a structure of type TOOLTIPTEXT in the lParam (see Appendix A). The first member of TOOLlTIPTEXT (hdr) is a structure of type NMHDR, and the idFrom member of HMHDR is the identifier of the control sending the message. Code uses this information and the LoadString() function to move the text into the buffer named szBuf. The text moved into szBuf comes from the lpszText member of a structure variable of type TOOLTIPTEXT. This second member is a pointer to a text string defined as a string resource in the application's executable.

The string resource that contains the messages that are displayed with each ToolTip is the last missing element of ToolTip implementation. You create the string resource by opening the Insert menu and selecting the Resource command. In the Insert Resource dialog box select String Table and then click on the New button. An example of a resource table is seen in Figure 19-15.



ID	Value	Caption
ID_FILE_NEW	40001	Create a new file
ID_FILE_OPEN	40002	Open current file
ID_FILE_SAVE	40003	Save current file

Figure 19-15 *Developer Studio Resource Table Editor*

The resource table consists of three entries: the id, the value, and the caption fields. You fill the id field so that it contains the same identification code as the button for which you are providing a ToolTip. In the caption field, you enter the text that is to be displayed at the ToolTip. Developer Studio automatically fills the value field for the one assigned to the corresponding toolbar button. Double-clicking on the entry displays a dialog box where these values can be input.

The program named TT_DEMO, located in the ToolTip Demo project folder on the book's software package, is a demonstration of the processing required for the implementation of ToolTip controls.

Chapter 20

Pixels, Lines, and Curves

Topics:

- Basic architecture of a Windows graphics application
- Graphics device interface attributes
- The device context
- Graphic objects: pens, brushes, mixes, pen position, and arc direction
- Drawing pixels, lines, and curves using GDI functions

This chapter is on graphics programming using the services in the Windows Graphics Device Interface. It discusses the simpler of these services, which are used for reading and setting individual pixels and for drawing lines and curves in a two-dimensional space. The described graphics functions are among the most often used in conventional Windows graphics.

The chapter starts with the architecture of a Windows graphics application, the GDI itself, and a more extensive look at the Windows Device Context. It is in the Device Context where system-level graphics information is stored. Applications must often read these attributes. It also covers Windows graphics objects and their attributes, that is, pens, brushes, bitmaps, palettes, fonts, paths, and regions, as well as some of the attributes of the Device Context: color, mix mode, background mode, pen position, and arc direction. These attributes determine how graphics output takes place.

20.1 Drawing in a Window

Windows programs are event driven; applications share resources with all other running programs and with the operating system. This determines that a graphics program cannot make exclusive use of the display, or of other system resources, since these are part of a pool that is accessible to all code in a multitasking environment. The following implications result from this architecture:

- A typical Windows application must obtain information about the system and the display device before performing output operations. The application must know the structure and dimensions of the output surface, as well as its capabilities, in order to manage the display function.

- In Windows, output to devices is performed by means of a logical link between the application, the device driver, and the hardware components. This link is called a device context. A display context is a special device context for the display device. Applications that draw to a window using conventional Windows functions must first obtain the display context. The handle to this display context is passed as a parameter to all API drawing functions.
- Unlike a DOS program, a Windows applications cannot draw to the screen and assume that the resulting image remains undisturbed for unlimited time. On the contrary, a Windows program must take into account that the video display is a shared resource. Windows notifies the application that its client area needs to be painted or repainted by posting a WM_PAINT message to the program's message queue. A well-designed Windows programs must be able to redraw its client area upon receiving this message.

The first two of these topics, that is, obtaining the device context handle and the display device attributes, are discussed in a separate section later in this chapter. Here we are concerned with the mechanisms used by Windows applications for accessing the display device in a way that is consistent with the multitasking nature of the environment.

20.1.1 The Redraw Responsibility

Windows applications are burdened with the responsibility of redrawing their client area at any time. This is an obligation to be taken seriously since it implies that code must have ways for reconstructing the display on demand. What data structures and other controls are necessary to redraw the screen and how code handles this responsibility depends on the application itself. In some programs the screen redraw burden is met simply by keeping tabs of which on several possible displays is active. In other applications the screen redraw obligation can entail such elaborate processing that it becomes a major consideration in program design.

The operating system, or your own code, sends the WM_PAINT message whenever the client area, or a portion thereof, needs to be redrawn. Application code responds to its screen redraw responsibility during the WM_PAINT message intercept. The following events cause the operating system to send WM_PAINT:

- The user has brought into view a previously hidden area of the application window. This happens when the window has been moved or uncovered.
- The user has resized the window.
- The user has scrolled the window contents.

The WM_PAINT message is not produced when a window is merely moved to another position on the desktop, since in this case, the client area has not been changed by the translation. Therefore, the operating system is able to maintain the screen contents because no new graphics elements were introduced or removed, and the screen size remains the same. However, the operating system cannot anticipate how an application handles a screen resizing operation. There are several possible processing options: are the screen contents scaled to the new dimension of the client area, or is their original size maintained? Are the positions of the graphics elements changed as a consequence of the resize operation, or do they remain in the same place? Not knowing how these alternatives are to be handled, the operating system responds by sending WM_PAINT to the application and letting it take whatever redraw action considers appropriate. The same logic applies when the client area is scrolled or when portions of the window are uncovered.

There are other times during which Windows attempts to restore the application's screen, but may occasionally post the WM_PAINT message if it fails in this effort. These occasions are when a message or dialog box is displayed, when a menu is pulled down, or when a tooltip is enabled. Finally, there are cases in which Windows always saves and restores the screen automatically, for example, when the mouse cursor or a program icon is dragged or moved across the client area.

20.1.2 The Invalid Rectangle

In an effort to minimize the processing, Windows keeps tabs on which portion of the application's client area needs to be redrawn. This notion is based on the following logic: it is wasteful for application code to repaint the entire screen when only a small portion of the program's client area needs to be redrawn. In practice, for simpler programs, it is often easier to assume that the entire client area needs redrawing than to get into the complications of repainting parts of the screen. However, in more complex applications, particularly those that use multiple child windows, it may save considerable time and effort if code can determine which of these elements need redrawing and which can be left unchanged.

The screen area that needs to be redrawn is called the update region. The smallest rectangle that binds the update region is called the invalid rectangle. When the WM_PAINT message is placed in the message queue, Windows attaches to it a structure of type RECT that contains the dimensions and location of the invalid rectangle. If another screen area becomes invalid before WM_PAINT is posted, Windows then makes the necessary correction in the invalid rectangle. This scheme saves posting more than one WM_PAINT message on the queue. Applications can call GetUpdateRect() to obtain the coordinates of the top-left and bottom-right corner of the update region.

An application can force Windows to send a WM_PAINT message to its own window procedure. This is accomplished by means of the InvalidateRect() or InvalidateRgn() functions. InvalidateRect() has the effect of adding a rectangle to a window's update region. The function has the following general form:

```
BOOL InvalidateRect(
HWND hWnd,          // 1
CONST RECT* lpRect, // 2
BOOL bErase         // 3
             );
```

The first parameter identifies the window whose update region has changed. The second parameter is a pointer to a structure variable of type RECT that contains the coordinates of the rectangle to be added to the update region. If this parameter is NULL, then the entire client area is added to the update region. The third parameter is a flag that indicates if Windows should erase or not erase the background. If this parameter is TRUE, then the background is erased when the BeginPaint() function is called by the application. If it is FALSE, the background remains unchanged.

20.1.3 Screen Updates On-Demand

The standard reply of an application that has received a WM_PAINT message is to redraw its own client area. This implies that the application has been designed so that a screen update takes place every time a WM_PAINT message is received. In this case the application design has to take into account the message-driven characteristic of a Windows program.

Consider a program that contains three menu commands: one to display a circle, another one to display a rectangle, and a third one to display a triangle. When the user clicks on any one of the three menu items, a WM_COMMAND message is posted to the application's message queue. The low word of the WPARAM encodes the menu item selected. Application code usually switches on this value in order to field all possible commands. However, the screen should not be updated during WM_COMMAND processing. What code can do at this point is set a switch that indicates the selected command. In this example a static variable of type int, named drawMode, could be set to 1 to indicate a circle drawing request, to 2 to indicate a rectangle, and to 3 to indicate a triangle. After this switch is set according to the menu command entered by the user, code calls InvalidateRect() so that Windows posts WM_PAINT to the application's own message queue. The application then processes WM_PAINT inspecting the value of the drawMode variable. If the value is 1 it draws a circle, if its is 2 it draws a rectangle, and if it is 3 it draws a triangle.

To a non-Windows programmer this may appear to be quite a round-about way of doing things. Why not draw the geometrical figures at the time that the menu commands are received? The problem with drawing as the commands are received is that if the window is resized or covered there is no mechanism in place to restore its screen image. The result would be either a partially or a totally blank client area. However, if the screen updates take place during WM_PAINT message processing, then when Windows sends WM_PAINT to the application because of a screen contents change, the application redraws itself and the client area is correctly restored.

On the other hand, not all screen drawing operations can take place during WM_PAINT message processing. Applications sometimes have to perform display functions that are directly linked to a user action, for example, a rubber-band image that is drawn in direct and immediate response to a mouse movement. In this case code cannot postpone the drawing until WM_PAINT is received.

20.1.4 Intercepting the WM_PAINT Message

The WM_PAINT message is generated only for windows that were created with the styles CS_HREDRAW or CS_VREDRAW. Receiving WM_PAINT indicates to application code that all or part of the client area must be repainted. The message can originate in Windows, typically because the user has minimized, overlapped, or resized the client area. Or also because the application itself has produced the message by calling InvalidateRect() or InvalidateRgn(), as previously discussed.

Typically, WM_PAINT processing begins with the BeginPaint() function. BeginPaint() prepares the window for a paint operation. In the first place it fills a variable of type PAINTSTRUCT, which is defined as follows:

```
typedef struct tagPAINTSTRUCT {
    HDC    hdc;             // Identifies display device
    BOOL   fErase;          // TRUE if background must be
                            // erased
    RECT   rcPaint;         // Rectangle structure specifying
                            // the update region
    BOOL   fRestore;        // RESERVED
    BOOL   fIncUpdate;      // RESERVED
    BYTE   rgbReserved[32]; // RESERVED
} PAINTSTRUCT;
```

If the screen erasing flag is set, BeginPaint() uses the window's background brush to erase the background. In this case, when execution returns from BeginPaint() code can assume that the update region has been erased. At this point the application can call GetClientRect() to obtain the coordinates of the update region, or proceed on the assumption that the entire client area must be redrawn.

Processing ends with EndPaint(). EndPaint() notifies Windows that the paint operation has concluded. The parameters passed to EndPaint() are the same ones passed to BeginPaint(): the handle to the window and the address of the structure variable of type PAINTSTRUCT. One important consequence of the EndPaint() function is that the invalid region is validated. Drawing operations by themselves have no validating effect. Placing the drawing operations between BeginPaint() and EndPaint() functions automatically validates the invalid region so that other WM_PAINT messages are not produced. In fact, placing the BeginPaint() EndPaint() functions in the WM_PAINT intercept, with no other processing operation, has the effect of validating the update region. The DefWindowProc() function operates in this manner.

The project Pixel and Line Demo in the book's software package demonstrates image display and update in response to WM_PAINT messages. The processing uses a static variable to store the state of the display. A switch construct in the WM_PAINT routine performs the screen updates, as in the following code fragment:

```
// Drawing command selector
static int   drawMode = 0;
//                      0 = no menu command active
//                          Active menu command:
//                      1 = Set Pixel
//                      2 = LineTo
//                      3 = Polyline
//                      4 = PolylineTo
//                      5 = PolyPolyline
//                      6 = Arc
//                      7 = AngleArc
//                      8 = PolyBezier
//                      9 = PolyDraw
  . . .
//*******************************
//     menu command processing
//*******************************
```

```
case WM_COMMAND:
switch (LOWORD (wParam)) {
         //****************************
//    SetPixel command
         //****************************
case ID_DRAWOP_PIXELDRAW:
drawMode = 1;   // Command to draw line
InvalidateRect(hwnd, NULL, TRUE);
break;
//****************************
//    LineTo command
//****************************
case ID_DRAWOP_LINE_LINETO:
drawMode = 2;   // Command to draw line
InvalidateRect(hwnd, NULL, TRUE);
break;
   . . .
//********************************
//     WM_PAINT processing
//********************************
case WM_PAINT :
BeginPaint (hwnd, &ps) ;
switch(drawMode)
         {
// 1 = SetPixel command
case 1:
pixColor = RGB(0xff, 0x0, 0x0); // Red
for (i = 0; i < 1000; i++) {
x = i * cxClient / 1000;
y = (int) (cyClient / 2 *
(1- sin (pix2 * i / 1000)));
SetPixelV (hdc, x, y, pixColor);
               }
break;
// 2 = LineTo command
case 2:
// Create a solid blue pen, 4 pixels wide
SelectObject(hdc, bluePen4);
MoveToEx (hdc, 140, 140, NULL);
LineTo (hdc, 300, 140);
LineTo (hdc, 300, 300);
LineTo (hdc, 140, 300);
LineTo (hdc, 140, 140);
break;
   . . .
```

20.2 Graphics Device Interface

The Graphics Device Interface (GDI) consists of a series of functions and related data structures that applications can use to generate graphics output. The GDI can output to any compatible device, but most frequently the device is either the video display, a graphics hard copy device (such as a printer or plotter), or a metafile in memory. By means of GDI functions you draw lines, curves, closed figures, paths, bitmapped images, and text. The objects are drawn according to the style selected for drawing objects, such as pens, brushes, and fonts. The pen object determines how lines and curves are drawn; the brush object determines how the interior of closed figures is filled. Fonts determine the attributes of text.

Output can be directed to physical devices, such as the video display or a printer, or to a logical device, such as a metafile. A metafile is a memory object that stores output instructions so that they can later be used to produce graphics on a physical device. It works much like a tape recording that can be played back at any time, any number of times.

The GDI is a layer between the application and the graphics hardware. It ensures device-independence and frees the programmer from having to deal with hardware details of individual devices. The device context, mentioned in Chapter 4, is one of the fundamental mechanisms used by the GDI to implement device-independent graphics. The GDI is a two-dimensional interface, which contains no 3D graphics primitives or transformations. It is also a static system, with very little support for animation. Therefore, the GDI is not capable of doing everything that a graphics programmer may desire, but within these limitations, it provides an easy and convenient toolkit of fundamental functions.

GDI functions can be classified into three very general categories:

- Functions that relate to the device context. These are used to create and release the DC, to get information about it, and to get and set its attributes.
- Drawing primitives. These are used to draw lines and curves, fill areas, and display bitmaps and text.
- Functions that operate on GDI objects. These perform manipulation of graphics objects such as pens, brushes, and bitmaps, which are not part of the device context.

20.2.1 Device Context Attributes

The GDI can output to any compatible device, including hard copy graphics devices and memory. For this reason, when referring to the GDI functions, we always use the term device context, instead of the more restrictive display context. In Chapter 4 we discussed the fundamentals of the device context and developed a template file TEMPL02.CPP, found in the Templates directory on the book's software package; it creates a program that uses a private device context. A private device context has the advantage that it need be retrieved only once and that attributes assigned to it are retained until they are explicitly changed. In the following examples and demonstration programs, we continue to use a private device context to take advantage of these simplifications.

The mapping modes are among the most important attributes of the device context. Two scalable mapping modes, named MM_ANISOTROPIC and MM_ISOTROPIC, are use in shrinking and expanding graphics by manipulating the coordinate system. They provide a powerful image manipulation mechanism and are discussed in Chapter 21. For now, we continue to use the default mapping mode, MM_TEXT, in the demonstrations and examples.

Device context operations belong to two types: those that obtain information and those that set attributes. For example, the GDI function GetTextColor() retrieves the current text color from the device context, while the function SetTextColor() is used to change the text color attribute. Although these functions are sometimes referred to as get- and set-types, the function names do not always start with these words. For example, the SelectObject() function is used to both get and set the attributes of pens, brushes, fonts, and bitmaps.

Graphics applications often need to obtain information regarding the device context. For example, a program may need to know the screen resolution or the number of display colors. One of the most useful functions for obtaining information regarding the capabilities of a device context is GetDeviceCaps(). The call to GetDeviceCaps() requires two parameters: the first one is the handle to the device context, and the second one is an index value that identifies the capability being queried. Table 20.1 lists some of the most useful information returned by this function.

Table 20-1

Information Returned by GetDeviceCaps()

INDEX	MEANING	
DRIVERVERSION	Version number of device driver.	
TECHNOLOGY	Any one of the following:	
	Value	Meaning
	DT_PLOTTER	Vector plotter
	DT_RASDISPLAY	Raster display
	DT_RASPRINTER	Raster printer
	DT_RASCAMERA	Raster camera
	DT_CHARSTREAM	Character stream
	DT_METAFILE	Metafile
	DT_DISPFILE	Display file
HORZSIZE	Width of the physical screen (millimeters).	
VERTSIZE	Height of the physical screen (millimeters).	
HORZRES	Width of the screen (pixels).	
VERTRES	Height of the screen (raster lines).	
LOGPIXELSX	Number of pixels per logical inch along the screen width.	
LOGPIXELSY	Number of pixels per logical inch along the screen height.	
BITSPIXEL	Number of color bits per pixel.	
PLANES	Number of color planes.	
NUMBRUSHES	Number of device-specific brushes.	
NUMPENS	Number of device-specific pens.	
NUMFONTS	Number of device-specific fonts.	
NUMCOLORS	Number of entries in the color table, if the device has a color depth of no more than 8 bits per pixel. Otherwise, –1 is returned.	
ASPECTX	Relative width of a device pixel used for line drawing.	
ASPECTY	Relative height of a device pixel used for line drawing.	
ASPECTXY	Diagonal width of the device pixel.	

(continues)

Table 20-1

Information Returned by GetDeviceCaps() (continued)

INDEX	MEANING	
CLIPCAPS	Flag indicating clipping capabilities of the device. Value is 1 if the device can clip to a rectangle. Otherwise, it is 0.	
SIZEPALETTE	Number of entries in the system palette.	
NUMRESERVED	Number of reserved entries in the system palette.	
COLORRES	Actual color resolution of the device, in bits per pixel.	
PHYSICALWIDTH	For printing devices: the width of the physical page, in device units.	
PHYSICALHEIGHT	For printing devices: the height of the physical page, in device units.	
PHYSICALOFFSETX	For printing devices: the distance from the left edge of the physical page to the left edge of the printable area, in device units.	
PHYSICALOFFSETY	For printing devices: the distance from the top edge of the physical page to the top edge of the printable area, in device units.	
RASTERCAPS	Value that indicates the raster capabilities of the device, as follows:	
	Capability	Meaning
	RC_BANDING	Requires banding support.
	RC_BITBLT	Capable of transferring bitmaps.
	RC_BITMAP64	Supports bitmaps larger than 64K.
	RC_DI_BITMAP	Supports SetDIBits() and GetDIBits functions.
	RC_DIBTODEV	Capable of supporting the SetDIBitsToDevice function.
	RC_FLOODFILL	Capable of performing flood fills.
	RC_PALETTE	Palette-based device.
	RC_SCALING	Capable of scaling.
	RC_STRETCHBLT	Capable of performing the StretchBlt function.
	RC_STRETCHDIB	Capable of performing the StretchDIBits function.
CURVECAPS	Indicates the curve capabilities of the device, as follows:	
	Value	Meaning
	CC_NONE	Does not support curves.
	CC_CIRCLES	Device can draw circles.
	CC_PIE	Device can draw pie wedges.
	CC_CHORD	Device can draw chord arcs.
	CC_ELLIPSES	Device can draw ellipses.
	CC_WIDE	Device can draw wide borders.
	CC_STYLED	Device can draw styled borders.
	CC_WIDESTYLED	Device can draw wide and styled borders.
	CC_INTERIORS	Device can draw interiors.
	CC_ROUNDRECT	Device can draw rounded Rectangles.

(continues)

Table 20-1

Information Returned by GetDeviceCaps() (continued)

INDEX	MEANING	
LINECAPS	Indicates the line capabilities of the device, as follows:	
	Value	Meaning
	LC_NONE	Does not support lines.
	LC_POLYLINE	Device can draw a polyline.
	LC_MARKER	Device can draw a marker.
	LC_POLYMARKER	Device can draw multiple markers.
	LC_WIDE	Device can draw wide lines.
	LC_STYLED	Device can draw styled lines.
	LC_WIDESTYLED	Device can draw lines that are wide and styled.
	LC_INTERIORS	Device can draw interiors.
POLYGONALCAPS	Indicates the polygon capabilities of the device, as follows:	
	Value	Meaning
	PC_NONE	Does not support polygons.
	PC_POLYGON	Device can draw alternate-fill polygons.
	PC_RECTANGLE	Device can draw rectangles.
	PC_WINDPOLYGON	Device can draw winding-fill polygons.
	PC_SCANLINE	Device can draw a single scanline.
	PC_WIDE	Device can draw wide borders.
	PC_STYLED	Device can draw styled borders.
	PC_WIDESTYLED	Device can draw borders that Are wide and styled.
	PC_INTERIORS	Device can draw interiors.
TEXTCAPS	Indicates the text capabilities of the device, as follows:	
	Value	Meaning
	TC_OP_CHARACTER	Device is capable of character output precision.
	TC_OP_STROKE	Device is capable of stroke output precision.
	TC_CP_STROKE	Device is capable of stroke clip precision.
	TC_CR_90	Device is capable of 90-degree character rotation.
	TC_CR_ANY	Device is capable of any character rotation.
	TC_SF_X_YINDEP	Device can scale independently in the x- and y-directions.
	TC_SA_DOUBLE	Device is capable of doubled character for scaling.
	TC_SA_INTEGER	Device uses integer multiples only for character scaling.
	TC_SA_CONTIN	Device uses any multiples for exact character scaling.

(continues)

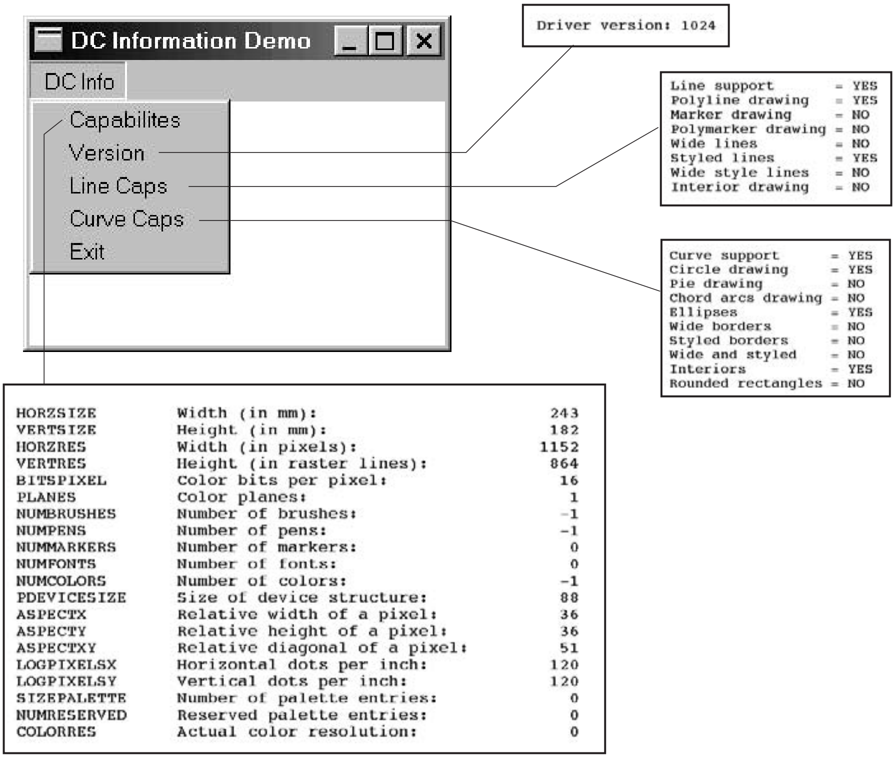
Table 20-1

Information Returned by GetDeviceCaps() (continued)

	VALUE	MEANING
	TC_EA_DOUBLE	Device can draw double-weight characters.
	TC_IA_ABLE	Device can italicize.
	TC_UA_ABLE	Device can underline.
	TC_SO_ABLE	Device can draw strikeouts.
	TC_RA_ABLE	Device can draw raster fonts.
	TC_VA_ABLE	Device can draw vector fonts.
	TC_SCROLLBLT	Device cannot scroll using a bit-block transfer.

20.2.2 DC Info Demonstration Program

The program named DCI_DEMO, located in the DC Info Demo project folder on the book's software package, shows how to obtain device context information. The menu labeled "DC Info" contains commands for displaying the most used general device context capabilities, the device driver version, as well as the specific line and curve drawing capabilities. Figure 20-1 shows the various menu commands in the DCI_DEMO program.



```
Driver version: 1024
```

```
Line support        = YES
Polyline drawing    = YES
Marker drawing      = NO
Polymarker drawing  = NO
Wide lines          = NO
Styled lines        = YES
Wide style lines    = NO
Interior drawing    = NO
```

```
Curve support       = YES
Circle drawing      = YES
Pie drawing         = NO
Chord arcs drawing  = NO
Ellipses            = YES
Wide borders        = NO
Styled borders      = NO
Wide and styled     = NO
Interiors           = YES
Rounded rectangles  = NO
```

```
HORZSIZE       Width (in mm):                      243
VERTSIZE       Height (in mm):                     182
HORZRES        Width (in pixels):                 1152
VERTRES        Height (in raster lines):           864
BITSPIXEL      Color bits per pixel:                16
PLANES         Color planes:                         1
NUMBRUSHES     Number of brushes:                   -1
NUMPENS        Number of pens:                      -1
NUMMARKERS     Number of markers:                    0
NUMFONTS       Number of fonts:                      0
NUMCOLORS      Number of colors:                    -1
PDEVICESIZE    Size of device structure:            88
ASPECTX        Relative width of a pixel:           36
ASPECTY        Relative height of a pixel:          36
ASPECTXY       Relative diagonal of a pixel:        51
LOGPIXELSX     Horizontal dots per inch:           120
LOGPIXELSY     Vertical dots per inch:             120
SIZEPALETTE    Number of palette entries:            0
NUMRESERVED    Reserved palette entries:             0
COLORRES       Actual color resolution:              0
```

Figure 20-1 *Screen Snapshots of the DC Info Program*

The Capabilities command in the DC Info menu displays the device context values for some of the most used elements returned by the GetDeviceCaps() function. To simplify the programming, the data required during processing is stored in a header file named DC_Caps.h, which can be found in the project directory. The header file is formatted as follows:

```
// Header file for DC Info Demo project
// Contains array of structures

#define LINES ((int) (sizeof DCcaps / sizeof DCcaps [0]))
struct
     {
int  iIndex ;
char *szLabel ;
char *szDesc ;
     }
DCcaps [] =
          {
          HORZSIZE,      "HORZSIZE",     "Width (in mm):",
          VERTSIZE,      "VERTSIZE",     "Height (in mm):",
          HORZRES,       "HORZRES",      "Width (in pixels):",
          .
          .
          .
          NUMRESERVED,   "NUMRESERVED",  "Reserved palette entries:",
          COLORRES,      "COLORRES",     "Actual color resolution:"
          } ;
```

Each entry in the array of structures contains three elements. The first one (int iIndex) is the index name required in the GetDeviceCaps() call. The two other elements are strings used at display time. Processing takes place in a loop in which the number of iterations is determined by the constant LINES, which is calculated by dividing the number of entries in the structure by the number of elements in each entry. This coding allows us to change the number of entries in the array without having to change the loop.

```
// Obtain and display DC capabilities
for (i = 0 ; i < LINES ; i++) {
TextOut (hdc, cxChar, cyChar * (1 + i),
DCcaps[i].szLabel,
strlen (DCcaps[i].szLabel)) ;
TextOut (hdc, cxChar + 16 * cxCaps, cyChar * (1 + i),
DCcaps[i].szDesc,
strlen (DCcaps[i].szDesc)) ;
SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
TextOut (hdc, cxChar + 16 * cxCaps + 40 * cxChar,
cyChar * (1 + i), szBuffer,
wsprintf (szBuffer, "%5d",
GetDeviceCaps (hdc, DCcaps[i].iIndex))) ;
SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
       }
break;
```

In the previous code fragment, the first TextOut() call displays the szLabel variable in the DCcaps structure. The second call to TextOut() displays the szDesc string. The value in the device context is obtained with the GetDeviceCaps() function that is part of the third call to TextOut(). In this case the iIndex element in the array is used as the second parameter to the call. The wsprintf() function takes care of converting and formatting the integer value returned by GetDeviceCaps() into a displayable string.

Obtaining and displaying the driver version is much simpler. The coding is as follows:

```
// Get driver version
_itoa(GetDeviceCaps(hdc, DRIVERVERSION),
szVersion + 16, 10);
// Initialize rectangle structure
    SetRect (&textRect,                // address of structure
             2 * cxChar,               // x for start
             cyChar,                   // y for start
             cxClient,                 // x for end
             cyClient);                // y for end
DrawText( hdc, szVersion, -1, &textRect,
DT_LEFT | DT_WORDBREAK);
break;
```

In this case we use the _itoa() function to convert the value returned by GetDeviceCaps() into a string. SetRect() and DrawText() are then used to format and display the string.

Obtaining and displaying the curve drawing and line drawing capabilities of the device context requires different processing. These values (see Table 20-1) are returned as bit flags associated with an index variable. For example, we make the call to GetDeviceCaps() using the index constant CURVECAPS as the second parameter. The integer returned by the call contains all the bit flags that start with the prefix CC (CurveCaps) in Figure 20-1. Code can then use a bitwise AND to test for one or more of curve drawing capabilities. The following code fragment shows one possible approach for obtaining curve-drawing capabilities:

```
// Get curve drawing capabilities
curvecaps = GetDeviceCaps (hdc, CURVECAPS);
// Test individual bit flags and change default
// string if necessary
if (curvecaps & CC_NONE)
strncpy(szCurvCaps + 21, strNo, 3);
if (curvecaps & CC_CIRCLES)
strncpy(szCurvCaps + (26 + 21), strYes, 3);
    .
    .
    .
if (curvecaps & CC_ROUNDRECT)
strncpy(szCurvCaps + (9 * 26 + 21), strYes, 3);
// Initialize rectangle structure
        SetRect (&textRect,                // address of
                                           // structure
                 2 * cxChar,               // x for start
                 cyChar,                   // y for start
                 cxClient,                 // x for end
                 cyClient);                // y for end

DrawText( hdc, szCurvCaps, -1, &textRect,
DT_LEFT | DT_WORDBREAK);
break;
```

Each of the if statements in the processing routine tests one of the bit flags returned by GetDeviceCaps(). If the bit is set, then a text string containing the words YES or NO is moved into the display string. When all the bits have been examined, the message string named szCurvCaps is displayed in the conventional manner.

20.2.3 Color in the Device Context

Monochrome displays are a thing of the past. Virtually all Windows machines have a color display and most of them can go up to 16.7 million displayable colors. In graphics programming you will often have to investigate the color capabilities of a device as well as select and manipulate colors.

In Chapter 1 we discussed the primary and the complementary color components of white light. In Windows programming, colors are defined by the relative intensity of the red, green, and blue primary components. Each color value is encoded in 8 bits, therefore, all three primary components require 24 bits. Since no C++ data type is exactly 24 bits, however, the color value in Windows is stored in a type called COLORREF, which contains 32 bits. The resulting encoding is said to be in RGB format, where the letters stand for the red, green, and blue components, respectively. Figure 20-2 shows the bit structure of the COLORREF type.

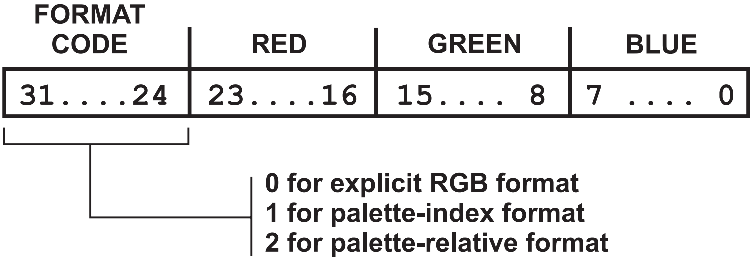

FORMAT CODE
RED
GREEN
BLUE
31....24
23....16
15.... 8
7 0
0 for explicit RGB format
1 for palette-index format
2 for palette-relative format

Figure 20-2 *COLORREF Bitmap*

Windows provides a macro named RGB, defined in the windows.h header file; it simplifies entering the color values into a data variable of type COLORREF. The macro takes care of inserting the zeros in bits 24 to 31, and in positioning each color in its corresponding field. As the name RGB indicates, the first value corresponds to the red primary, the second one to the green, and the third one to the blue. For example, to enter a middle-gray value, in which each of the primary colors is set to 128, proceed as follows:

```
COLORREF     midGray;  // Variable of type COLORREF
midGray = RGB(128, 128, 128);
```

The COLORREF data type is also used to encode palette colors. Windows uses the high-order 8 bits to determine if a color value is in explicit RGB, palette-index, or palette-relative format. If the high-order byte is zero, then the color is an explicit RGB value; if it is 1 then it is a palette-index value; if it is 2 then the color is a palette-relative value. Using the RGB macro when creating explicit-RGB values ensures that the high-order byte is set correctly.

Obtaining color information from the device context requires careful consideration. Note in Table 20-1that the index constant NUMCOLORS is valid only if the color depth is no more than 8 bits per pixel. The device queried in Figure 20-1 has 16 bits per pixel; therefore, the NUMCOLORS value is set to –1. By the same token, the COLORRES index constant is valid only if the device sets the RC_PALETTE bit. In Figure 20-1 the value of this field is 0. The two most useful constants for obtaining general color depth information are PLANES and BITPIXEL. PLANES returns the number of color planes and BITPIXEL returns the number of bits used in encoding each plane.

20.3 Graphic Objects and GDI Attributes

We should first mention that Windows graphics objects are not objects in the object-oriented sense. Windows graphics objects are pens, brushes, bitmaps, palettes, fonts, paths, and regions. Of these, pens and brushes are the objects most directly related to pixel and line drawing operations.

20.3.1 Pens

The pen graphics object determines a line's color, width, and style. Windows uses the pen currently selected in the device context with any of the pen-based drawing functions. Three stock pens are defined: BLACK_PEN, WHITE_PEN, and NULL_PEN. The default pen is BLACK_PEN, which draws solid black lines. Applications refer to a pen by means of its handle, which is stored in a variable of type HPEN. The GetStockObject() function is used to obtain a handle to one of the stock pens. The pen must be selected into the device context before it is used, as follows:

```
HPEN     aPen;       // handle to pen
   .
   .
   .
aPen = GetStockObject (WHITE_PEN);
SelectObject (hdc, aPen);
```

The two functions can be combined in a single statement, as follows:

```
SelectObject (hdc, GetStockObject (WHITE_PEN));
```

In this case, no pen handle variable is required. SelectObject() returns the handle to the pen previously installed in the device context. This can be used to save the original pen so that it can be restored later.

Drawing applications sometimes require one or more custom pens, which have a particular style, width, and color. Custom pens can be created with the functions CreatePen(), CreatePenIndirect(), and ExtCreatePen(). In the CreatePen() function the pen's style, width, and color are passed as parameters. CreatePenIndirect() uses a structure of type LOGPEN to hold the pen's style, width, and color. ExtCreatePen(), introduced in Windows 95, is the more powerful of the three. The iStyle parameter is a combination of pen type, styles, end cap style, and line join attributes. The constants used in defining this parameter are listed in Table 20-2, on the following page.

Table 20-2

Values Defined for the ExtCreatePen() iStyle Parameter

PEN TYPE	DESCRIPTION
PS_GEOMETRIC	Pen is geometric.
PS_COSMETIC	Pen is cosmetic. Same as those created with CreatePen() and CreatePenIndirect(). Width must be 1 pixel.
Pen Style	
PS_ALTERNATE	Windows NT: Pen sets every other pixel. (Cosmetic pens only.) Windows 95: Not supported.
PS_SOLID	Pen is solid.
PS_DASH	Pen is dashed.
PS_DOT	Pen is dotted.
PS_DASHDOT	Pen has alternating dashes and dots.
PS_DASHDOTDOT	Pen has alternating dashes and double dots.
PS_NULL	Pen is invisible.
PS_USERSTYLE	Windows NT: Pen uses a styling array supplied by the user. Windows 95: Not supported.
PS_INSIDEFRAME	Pen is solid. Any drawing function that takes a bounding rectangle, the dimensions of the figure are shrunk so that it fits entirely in the bounding rectangle. Geometric pens only.
End Cap Style (only in stroked paths)	
PS_ENDCAP_ROUND	End caps are round.
PS_ENDCAP_SQUARE	End caps are square.
PS_ENDCAP_FLAT	End caps are flat.
Join Style (only in stroked paths)	
PS_JOIN_BEVEL	Joins are beveled.
PS_JOIN_MITER	Joins are mitered when they are within the current limit set by the SetMiterLimit() function. If it exceeds this limit, the join is beveled. SetMiterLimit() is discussed in Chapter 21.
PS_JOIN_ROUND	Joins are round.

The standard form of the ExtCreatePen() function is as follows:

```
HPEN ExtCreatePen (iStyle,    // pen style
                   iWidth,    // pen width
                   &aBrush,   // pointer to a LOGBRUSH
                              // structure (next section)
dwStyleCount,// length of next parameter
lpStyle);    // dot-dash pattern array
```

The second parameter to ExtCreatePen() defines the pen's width. If the pen is a geometric pen, then its width is specified in logical units. If it is a cosmetic pen then the width must be set to 1.

A geometric pen created with ExtCreatePen() has brush-like attributes. The third parameter is a pointer to LOGBRUSH. The LOGBRUSH structure, described in the following section, is defined as follows:

```
struct tagLOGBRUSH {
UINT      lbStyle;
COLORREF  lbColor;
LONG      lbHatch;
} LOGBRUSH
```

If the pen is a cosmetic pen, then the lbStyle member must be BS_SOLID and the lbColor member defines the pen's color. In this case the lbHatch member, which sets a brush's hatch pattern, is ignored. If the pen is geometric, then all three structure members are meaningful and must be used to specify the corresponding attributes.

The fourth parameter, dwStyleCount, determines the length of the fifth parameter. The fifth parameter, lpStyle, is a pointer to an array of doubleword values. The first value in the array is the length of the first dash of a user-defined pen style, the second one is the length of the first space, and so on. If the pen style does not contain the PS_USERSTYLE constant, then the fourth parameter must be zero, and the fifth parameter must be NULL. Note that PS_USERSTYLE is supported in Windows NT but not in Windows 95 or 98.

The end cap styles determine the appearance of the line ends. Three constants are defined for round, square, and flat line ends. The end join style determines the appearance of the connecting point of two lines. Both styles are available only for geometric pens. Figure 20-3 shows the pen styles and the effects of the different end caps and joins.

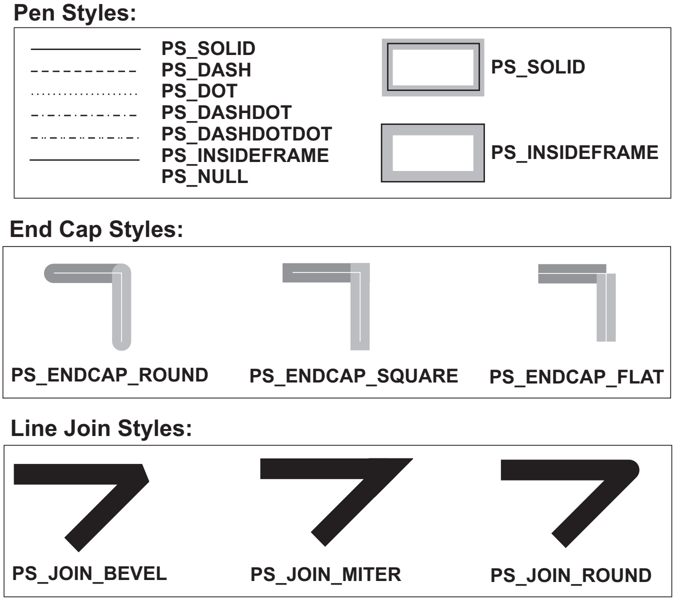


Figure 20-3 *Pen Syles, End Caps, and Joins*

Note in Figure 20-3 that the difference between square and flat caps is that the square style extends the line by one-half its width. The white lines in the end cap style insert are drawn with the white stock pen, to better show the style's effect. The NULL_PEN style creates a pen that draws with transparent ink, therefore it leaves no mark as it moves on the drawing surface. This style is occassionaly used in creating figures that are filled with a particular brush style but have no border.

20.3.2 Brushes

The brush object determines the attributes used in filling a solid figure. The outline of these figures is determined by the brush selected in the device context. A brush has a style, color, and hatch pattern. There are several stock brushes: WHITE_BRUSH, LTGRAY_BRUSH, GRAY_BRUSH, DKGRAY_BRUSH, BLACK_BRUSH, and NULL_BRUSH. All stock brushes are solid, that is, they fill the entire enclosed area of the figure. The NULL_BRUSH is used to draw figures without filling the interior. If a solid figure is drawn with the NULL_PEN, then it is filled but has no outline.

Applications refer to a brush by its handle, which is stored in a variable of type HBRUSH. The GetStockObject() function is used to obtain a handle to one of the stock brushes. The brush must be selected into the device context before use, as follows:

```
HBRUSH     aBrush;       // handle to brush
   .
   .
   .
aBrush = GetStockObject (WHITE_BRUSH);
SelectObject (hdc, aBrush);
```

As in the case of a pen, the two functions can be combined in a single statement, as follows:

```
SelectObject (hdc, GetStockObject (WHITE_BRUSH));
```

In this case, no brush handle variable is required. SelectObject() returns the handle to the brush previously installed in the device context. This can be used to save the original brush so that it can later be restored.

A custom brush is created by means of the CreateBrushIndirect() function. The call returns a handle to the brush, of type HBRUSH. The only parameter is a pointer to a structure of type LOGBRUSH which holds the brush style, color, and hatch pattern. The LOGBRUSH structure is also used by the ExtCreatePen() previously described. Table 20-3 lists the predefined constants used for members of the LOGBRUSH structure.

The foreground mix mode attribute of the device context, also called the drawing mode, determines how Windows combines the pen or brush color with the display surface when performing drawing operations. The mixing is a raster operation based on a boolean function of two variables: the pen and the background. For this reason it is described as a binary raster operation, or ROP2. All four boolean primitives are used in setting the mix mode: AND, OR, NOT, and XOR. The function for setting the foreground mix mode is SetROP2(). GetROP2() returns the current mix mode in the device context. The general form of the SetROP2() function is as follows:

Table 20-3

Constants in the LOGBRUSH Structure Members

BRUSH STYLE	DESCRIPTION
BS_DIBPATTERN	A pattern brush defined by a device-independent bitmap. If lbStyle is BS_DIBPATTERN, the lbHatch member contains a handle to a packed DIB. Note: DIB stands for Device Independent Bitmap. DIBs are discussed in Chapter 8.
BS_DIBPATTERNPT	Same as BS_DIBPATTERN but the lbHatch member contains a pointer to a packed DIB.
BS_HATCHED	Hatched brush.
BS_HOLLOW	Hollow brush.
BS_NULL	Same as BS_HOLLOW.
BS_PATTERN	Pattern brush defined by a memory bitmap.
BS_SOLID	Solid brush.
Brush Color	Description
DIB_PAL_COLORS	The color table consists of an array of 16-bit indices into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.
Hatch Style	
HS_BDIAGONAL	A 45-degree upward, left-to-right hatch.
HS_CROSS	Horizontal and vertical cross-hatch.
HS_DIAGCROSS	45-degree crosshatch.
HS_FDIAGONAL	A 45-degree downward, left-to-right hatch.
HS_HORIZONTAL	Horizontal hatch.
HS_VERTICAL	Vertical hatch.

```
int SetROP2(
        HDC hdc,         // 1
        int fnDrawMode   // 2
            );
```

Figure 20-4 shows the brush hatch patterns.

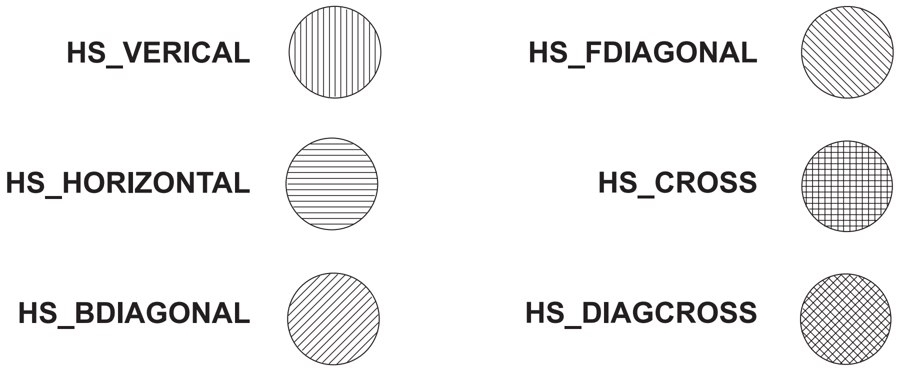


Figure 20-4 *Brush Hatch Patterns*

20.3.3 Foreground Mix Mode

The first parameter is the handle to the device context and the second parameter is one of 16 mix modes defined by Windows. The function returns the previous mix mode, which can be used to restore the original condition. Table 20-4 lists the ROP2 mix modes. The center column shows how the pen (P) and the screen (S) pixels are logically combined at draw time. The boolean operators correspond to the symbols used in C.

Table 20-4

Mix Modes in SetROP2()

CONSTANT	BOOLEAN OPERATION	DESCRIPTION
R2_BLACK	0	Pixel is always 0.
R2_COPYPEN	P	Pixel is the pen color. This is the default mix mode.
R2_MASKNOTPEN	~P&S	Pixel is a combination of the colors common to both the screen and the inverse of the pen.
R2_MASKPEN	P&S	Pixel is a combination of the colors common to both the pen and the screen.
R2_MASKPENNOT	P&~S	Pixel is a combination of the colors common to both the pen and the inverse of the screen.
R2_MERGENOTPEN	~P|S	Pixel is a combination of the screen color and the inverse of the pen color.
R2_MERGEPEN	P|S	Pixel is a combination of the pen color and the screen color.
R2_MERGEPENNOT	P|~S	Pixel is a combination of the pen color and the inverse of the screen color.
R2_NOP	S	Pixel remains unchanged.
R2_NOT	~S	Pixel is the inverse of the screen color.
R2_NOTCOPYPEN	~P	Pixel is the inverse of the pen color.
R2_NOTMASKPEN	~(P&S)	Pixel is the inverse of the R2_MASKPEN color.
R2_NOTMERGEPEN	~(P|S)	Pixel is the inverse of the R2_MERGEPEN color.
R2_NOTXORPEN	~(P^S)	Pixel is the inverse of the R2_XORPEN color.
R2_WHITE	1	Pixel is always 1.
R2_XORPEN	P^S	Pixel is a combination of the colors in the pen and in the screen, but not in both.

Legend:

~ = boolean NOT
| = boolean OR
& = boolean AND
^ = boolean XOR

20.3.4 Background Modes

Windows recognizes two background modes that determine how the gaps between dots and dashes are filled when drawing discontinuous lines, as well as with text and hatched brushes. The background modes, named OPAQUE and TRANSPARENT, are set in the device context by means of the SetBkMode() function. The function's general form is as follows:

```
int SetBkMode(
         HDC hdc,      // 1
         int iBkMode   // 2
              );
```

The first parameter is the handle to the device context, and the second one the constants OPAQUE or TRANSPARENT. If the opaque mode is selected, the background is filled with the current screen background color. If the mode is TRANSPARENT, then the background is left unchanged.

The background mode affects lines that result from a pen created with CreatePen() or CreatePenIndirect(), but not by those created with ExtCreatePen().

20.3.5 Current Pen Position

Many GDI drawing functions start at a screen location known as the current pen position, or the current position. The pen position is an attribute of the device context. The initial position of the pen is at logical coordinates (0, 0). Two functions relate directly to the current pen position: MoveToEx() and GetCurrent Position(). Some drawing functions change the pen position as they execute. The MoveToEx() function is used to set the current pen position. Its general form is as follows:

```
BOOL MoveToEx(
         HDC hdc,          // 1
         int X,            // 2
         int Y,            // 3
         LPPOINT lpPoint   // 4
              );
```

The first parameter is the handle to the device context. The second and third parameters are the x- and y-coordinates of the new pen position, in logical units. The fourth parameter is a pointer to a structure of type POINT that holds the x- and y-coordinates of the previous current pen position. If this parameter is set to NULL the old pen position is not returned. The function returns a boolean that is TRUE if the function succeeds and FALSE if it fails.

The GetCurrentPositionEx() function can be used to obtain the current pen position. Its general form is as follows:

```
BOOL MoveToEx(
         HDC hdc,          // 1
         int X,            // 2
         int Y,            // 3
         LPPOINT lpPoint   // 4
              );
```

The second parameter is a pointer to a structure variable of type POINT that receives the coordinates of the current pen position. The function returns TRUE if it succeeds and FALSE if it fails.

Drawing functions whose names contain the word "To" use and change the current pen position; these are LineTo(), PolylineTo(), and PolyBezierTo(). Windows is not always consistent in this use of the word "To", since the functions AngleArc() and PolyDraw() also use and update the current pen position.

20.3.6 Arc Direction

One start-point and one end-point on the circumference of a circle define two different arcs: one drawn clockwise and one drawn counterclockwise. The exception is when the start and end points coincide. Figure 20-5 shows this possible ambiguity.

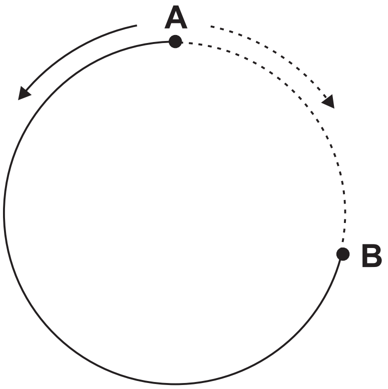


Figure 20-5 *The Arc Drawing Direction*

In Figure 20-5 the solid line arc is drawn counterclockwise from point A to point B, while the dotted line arc is drawn clockwise between these same points. The SetArcDirection() function is used to resolve this problem. The function's general form is as follows:

```
int SetArcDirection(
              HDC hdc,            // 1
              int ArcDirection    // 2
                  );
```

The second parameter is either the constant AD_CLOCKWISE, or the constant AD_COUNTERCLOCKWISE. The function returns the previous arc drawing direction.

20.4 Pixels, Lines, and Curves

The lowest-level graphics primitives are to set a screen pixel to a particular attribute and to read the attributes of a screen pixel. In theory, with functions to set and read a pixel, all the other graphics operations can be developed in software. For example, a line can be drawn by setting a series of adjacent pixels, a closed figure can be filled by setting all the pixels within its boundaries, and so on. However, in actual programming practice these simple primitives are not sufficient. In the first place, high-level language code requires considerable overhead in performing the pixel set and read operations. To draw lines and figures by successively calling these functions would be prohibitively time consuming. On the other hand, there are cases in which the programmer must resort to pixel-by-pixel drawing since other higher-level functions are not available.

There are 11 functions in the Windows API that can be used to draw lines. For one of them, StrokePath(), we postpone the discussion until Chapter 7, since we must first discuss paths in greater detail. Table 20-5 lists the remaining ten line-drawing functions.

Table 20-5

Line-Drawing Functions

FUNCTION	DRAWING OPERATION
LineTo()	A straight line from current position up to a point. Pen position is updated to line's end point.
PolylineTo()	One or more straight lines between the current position and points in an array. Pen position is used for the first line and updated to end point of last line.
Polyline()	A series of straight line segments between points defined in an array.
PolyPolyLine()	Multiple polylines.
ArcTo()	An elliptical arc updating current pen position.
Arc()	An elliptical arc without updating current pen position.
AngleArc()	A segment of arc starting at current pen position.
PolyBezier()	One or more Bezier curves without updating the current pen position.
PolyBezierTo()	One or more Bezier curves updating the current pen position.
PolyDraw()	A set of lines and Bezier curves.
StrokePath()	See Chapter 21.

20.4.1 Pixel Operations

Two Windows functions operate on single pixels: SetPixel() and GetPixel(). SetPixel() is used to set a pixel at any screen location to a particular color attribute. GetPixel() reads the color attribute of a pixel at a given screen location. The general form of SetPixel() is as follows:

```
COLORREF SetPixel(
         HDC hdc,           // 1
         int X,             // 2
         int Y,             // 3
         COLORREF crColor   // 4
              );
```

The first parameter is the handle to the device context. The second and third parameters are the x- and y-coordinates of the pixel to set, in logical units. The fourth parameter contains the pixel color in a COLORREF type structure. The function returns the RGB color to which the pixel was set, which may not coincide with the one requested in the call because of limitations of the video hardware. A faster version of this function is SetPixelV(). It takes the same parameters but returns a boolean value that is TRUE if the operation succeeded and FALSE if it failed. In most cases SetPixelV() is preferred over SetPixel() because of its better performance. The following code fragment shows how to draw a box of 100-by-100 pixels using the SetPixelV() function:

```
      int x, y, i, j;       // control variables
      COLORREF   pixColor;
      .
      .
      .
      x = 120;        // start x
      y = 120;        // start y
pixColor = RGB(0xff, 0x0, 0x0); // Red
// Draw a 100-by-100 pixel box
for (i = 0; i < 100; i++) {
for (j = 0; j < 100; j++) {
SetPixelV (hdc, x, y, pixColor);
x++;
           }
x = 120;
y++;
      }
```

20.4.2 Drawing with LineTo()

The simplest of all line-drawing functions is LineTo(). The function requires three parameters: the handle to the device context, and the coordinates of the end points of the line. The line is drawn with the currently selected pen. The start point is the current pen position; for this reason LineTo() is often preceded by MoveToEx() or another drawing function that sets the current pen position. LineTo() returns TRUE if the function succeeds and FALSE if it fails, but most often the return value is not used by code. If the LineTo() function succeeds, the current pen position is reset to the line's end point; therefore, the function can be used to draw a series of connected line segments.

The following code fragment draws a rectangle using four lines:

```
  HPEN          bluePen4;     // handle for a pen
  int           x, y, i, j;   // local variables
  .
  .
  .
// Create and select pen
bluePen4 = CreatePen (PS_SOLID, 4, RGB (0x00, 0x00, 0xff);
SelectObject (hdc, bluePen4);

// Set current pen position for start point
MoveToEx (hdc, 140, 140, NULL);
  LineTo (hdc, 300, 140);      // draw first segment
  LineTo (hdc, 300, 200);      // second segment
  LineTo (hdc, 140, 300);      // third segment
  LineTo (hdc, 140, 140);      // last segment
```

20.4.3 Drawing with PolylineTo()

The PolylineTo() function draws one or more straight lines between points contained in an array of type POINT. The current pen position is used as a start point and is reset to the location of the last point in the array. PolylineTo() provides an easier way of drawing several connected line segments, or an unfilled closed figure. The function uses the current pen. Its general form is as follows:

```
BOOL PolylineTo(
         HDC hdc,            // 1
         CONST POINT *lppt,  // 2
         DWORD cCount        // 3
              );
```

The second parameter is the address of an array of points that contains the x- and y-coordinate pairs. The third parameter is the count of the number of points in the array. The function returns TRUE if it succeeds and FALSE otherwise. The following code fragment shows the drawing of a rectangle using the PolylineTo() function:

```
  HPEN       redPen2;
  POINT      pointsArray[4];  // array of four points
  .
  .
  .
// Create a solid red pen, 2 pixels wide
redPen2 = CreatePen (PS_SOLID, 2, RGB(0xff, 0x00, 0x00));
SelectObject (hdc, redPen2);
// Fill array of points
pointsArray[0].x = 300; pointsArray[0].y = 160;
pointsArray[1].x = 300; pointsArray[1].y = 300;
pointsArray[2].x = 160; pointsArray[2].y = 300;
pointsArray[3].x = 160; pointsArray[3].y = 160;
// Set start point for first segment
MoveToEx (hdc, 160, 160, NULL);
// Draw polyline
PolylineTo (hdc, pointsArray, 4);
```

20.4.4 Drawing with Polyline()

The Polyline() function is similar to PolylineTo() except that it does not use or change the current pen position. Therefore, you need one more entry in the array of points to draw a figure with Polyline() since the initial position of the drawing pen cannot be used as the starting point for the first line segment. The following code fragment shows drawing a rectangle using the Polyline() function.

```
  HPEN       blackPen;
  POINT      pointsArray[4];  // array of four points
  .
  .
  .
// Create a solid red pen, 2 pixels wide
blackPen = CreatePen (PS_DASH, 1, 0);
SelectObject (hdc, blackPen);
// Fill array of points
pointsArray[0].x = 160; pointsArray[0].y = 160;
pointsArray[1].x = 300; pointsArray[1].y = 160;
pointsArray[2].x = 300; pointsArray[2].y = 300;
pointsArray[3].x = 160; pointsArray[3].y = 300;
pointsArray[4].x = 160; pointsArray[4].y = 160;
// Draw polyline
Polyline (hdc, pointsArray, 5);
```

20.4.5 Drawing with PolyPolyline()

As the function name implies, PolyPolyline() is used to draw several groups of lines or "polylines." Since the points array contains sets of points for more than one polyline, the function requires an array of values that holds the number of points for each polyline. PolyPolyline(), like Polyline(), does not use or change the current pen position. The function's general form is as follows:

```
BOOL PolyPolyline(
         HDC hdc,                     // 1
         CONST POINT *lppt,           // 2
         CONST DWORD *lpdwPolyPoints, // 3
         DWORD cCount                 // 4
             );
```

The second parameter is an array containing vertices of the various polylines. The third parameter is an array that contains the number of vertices in each of the polylines. The fourth parameter is the count of the number of elements in the third parameter, which is the number of polylines to be drawn. The function returns TRUE if it succeeds and FALSE otherwise. The following code fragment shows the drawing of two polylines, each with five vertices, using the PolyPolyline() function.

```
 POINT      pointsArray[10];     // array of points
 DWORD      vertexArray[2];      // vertices per polyline

// Fill array of points for first polyline
pointsArray[0].x = 160; pointsArray[0].y = 160;
pointsArray[1].x = 300; pointsArray[1].y = 160;
pointsArray[2].x = 300; pointsArray[2].y = 300;
pointsArray[3].x = 160; pointsArray[3].y = 300;
pointsArray[4].x = 160; pointsArray[4].y = 160;

// Fill array of points for second polyline
pointsArray[5].x = 160; pointsArray[5].y = 230;
pointsArray[6].x = 230; pointsArray[6].y = 160;
pointsArray[7].x = 300; pointsArray[7].y = 230;
pointsArray[8].x = 230; pointsArray[8].y = 300;
pointsArray[9].x = 160; pointsArray[9].y = 230;

// Fill number of vertices in array
vertexArray[0] = 5;
vertexArray[1] = 5;
// Draw two polylines
PolyPolyline (hdc, pointsArray, vertexArray, 2);
```

Figure 20-6 shows the figures that result from executing the previous code sample. The second polyline is shown in dashed lines to visually distinguish it from the first one. However, in an actual drawing there is no way of changing pens inside a call to PolyPolyline().

20.4.6 Drawing with Arc()

The Arc() function draws an elliptical arc. It is also used to draw circles, since the circle is a special case of the ellipse. The function's general form is as follows:

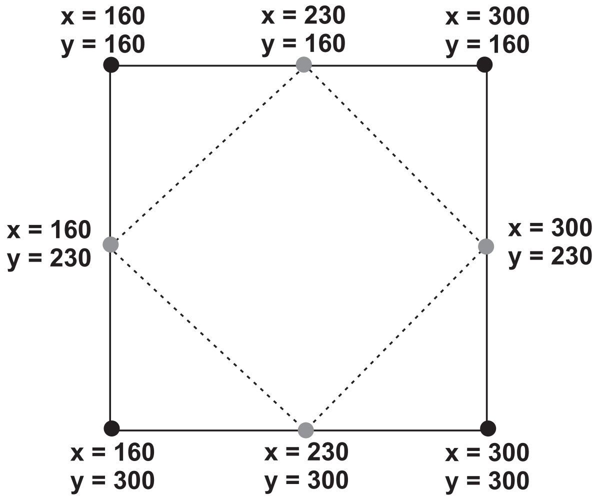


Figure 20-6 *Coordinates of Two Polylines in the Sample Code*

```
BOOL Arc(
          HDC hdc,          // 1
          int nLeftRect,    // 2
          int nTopRect,     // 3
          int nRightRect,   // 4
          int nBottomRect,  // 5
          int nXStartArc,   // 6
          int nYStartArc,   // 7
          int nXEndArc,     // 8
          int nYEndArc      // 9
               );
```

The second and third parameters are the x- and y-coordinates of the upper-left corner of a rectangle that contains the ellipse, while the fourth and fifth parameters are the coordinates of its lower-right corner. By using a bounding rectangle to define the ellipse, the Windows API avoids dealing with elliptical semi-axes. However, whenever necessary, the bounding rectangle can be calculated from the semi-axes. The sixth and seventh parameters define the coordinates of a point that sets the start point of the elliptical arc. The last two parameters set the end points of the elliptical arc. The elliptical arc is always drawn in the counterclockwise direction. The SetArcDirection() function has no effect in this case.

The coordinates of the start and end points of the elliptical arc need not coincide with the arc itself. Windows draws an imaginary line from the center of the ellipse to the start and end points. The point at which this line (or its prolongation) intersects the elliptical arc is used as the start or end point. If the start and end points are the same, then a complete ellipse is drawn. The following code fragment draws an elliptical arc:

```
Arc (hdc,
       150, 150,           // upper-left of rectangle
       350, 250,           // lower-right
       250, 260,           // start point
       200, 140;           // end point
```

Figure 20-7 shows the location of each of the points in the preceding call to the Arc() function and the resulting ellipse.

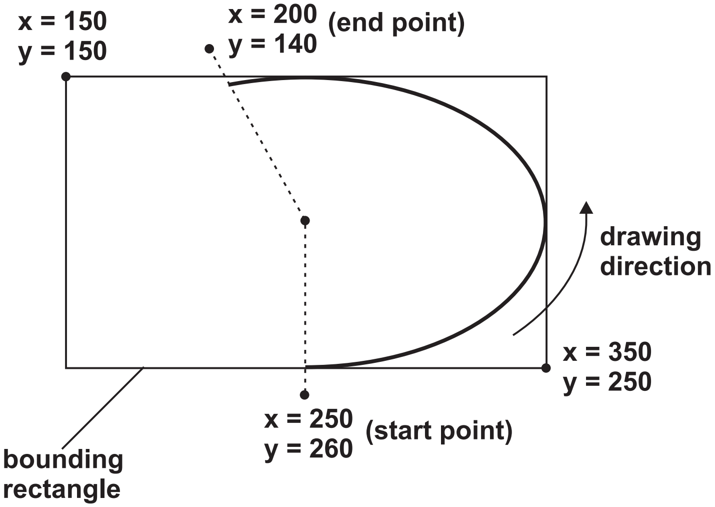


Figure 20-7 *Coordinates of an Elliptical Arc in Sample Code*

20.4.7 Drawing with ArcTo()

ArcTo() is a version of the Arc() function that updates the current pen position to the end point of the elliptical arc. This function requires Windows NT Version 3.1 or later. It is not available in Windows 95 or 98. The function parameters are identical to those of the Arc() function.

20.4.8 Drawing with AngleArc()

The AngleArc() function draws a straight line segment and an arc of a circle. The straight line segment is from the current pen position to the arc's starting point. The arc is defined by the circle's radius and two angles: the starting position, in degrees, relative to the x-axis, and the angle sweep, also in degrees, relative to the starting position. The arc is drawn in a counterclockwise direction. The function's general form is as follows:

```
BOOL AngleArc(
         HDC hdc,            // 1
         int X,              // 2
         int Y,              // 3
         DWORD dwRadius,     // 4
         FLOAT eStartAngle,  // 5
         FLOAT eSweepAngle   // 6
             );
```

The second and third parameters are the coordinates of the center of the circle that defines the arc, in logical units. The fourth parameter is the radius of the circle, also in logical units. The fifth parameter is the start angle in degrees, relative to the x-axis. The last parameter is the sweep angle, also in degrees, relative to the angle's starting position. Figure 20-8 shows the various elements in the AngleArc() function.

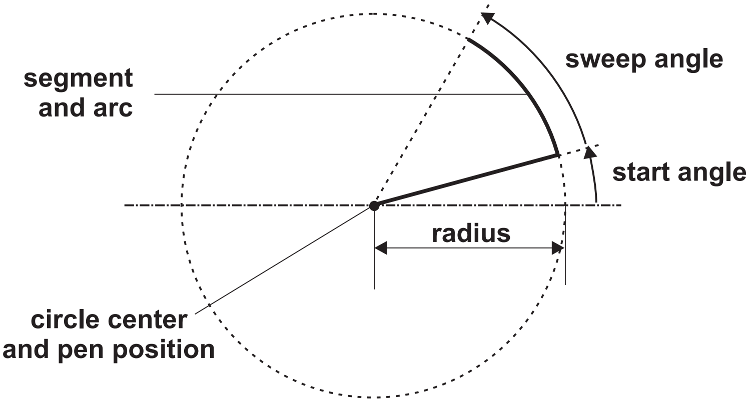


Figure 20-8 *AngleArc() Function Elements*

The AngleArc() function is not available in Windows 95 or 98; however, it can be emulated in code. Microsoft Developers Network contains the following listing which allows implementing the AngleArc() function in software:

```
BOOL AngleArc2(HDC hdc, int X, int Y, DWORD dwRadius,
float fStartDegrees, float fSweepDegrees) {
int iXStart, iYStart;  // End point of starting radial line
  int iXEnd, iYEnd;      // End point of ending radial line
  float fStartRadians;   // Start angle in radians
  float fEndRadians;     // End angle in radians
  BOOL bResult;          // Function result
float fTwoPi = 2.0f * 3.141592f;
/* Get the starting and ending angle in radians */
if (fSweepDegrees > 0.0f) {
fStartRadians = ((fStartDegrees / 360.0f) * fTwoPi);
fEndRadians = (((fStartDegrees + fSweepDegrees) / 360.0f) *
fTwoPi);
} else {
fStartRadians = (((fStartDegrees + fSweepDegrees)  / 360.0f) *
fTwoPi);
fEndRadians =  ((fStartDegrees / 360.0f) * fTwoPi);
  }

/* Calculate a point on the starting radial line via */
/* polar -> cartesian conversion */
iXStart = X + (int)((float)dwRadius * (float)cos(fStartRadians));
iYStart = Y - (int)((float)dwRadius * (float)sin(fStartRadians));

/* Calculate a point on the ending radial line via */
/* polar -> cartesian conversion */
iXEnd = X + (int)((float)dwRadius * (float)cos(fEndRadians));
iYEnd = Y - (int)((float)dwRadius * (float)sin(fEndRadians));

/* Draw a line to the starting point */
LineTo(hdc, iXStart, iYStart);

/* Draw the arc */
bResult = Arc(hdc, X - dwRadius, Y - dwRadius,
X + dwRadius, Y + dwRadius,
iXStart, iYStart,
iXEnd, iYEnd);

// Move to the ending point - Arc() wont do this and ArcTo()
// wont work on Win32s or Win16 */
MoveToEx(hdc, iXEnd, iYEnd, NULL);

return bResult;
}
```

Notice that the one documented difference between the preceding listing of AngleArc2() and the GDI AngleArc() function is that if the value entered in the sixth parameter exceeds 360 degrees, the software version will not sweep the angle multiple times. In most cases this is not a problem.

The program named PXL_DEMO, in the Pixel and Line Demo project folder on the book's software package, uses the AngleArc2() function to display a curve similar to the one in Figure 20-8.

20.4.9 Drawing with PolyBezier()

In mechanical drafting, a spline is a flexible edge that is used to connect several points on an irregular curve. Two French engineers, Pierre Bezier and Paul de Casteljau, almost simultaneously discovered a mathematical expression for a spline curve that can be easily adapted to computer representations. This curve is known as the Bezier spline or curve, since it was Bezier who first published his findings. The Bezier curve is defined by its end points, called the nodes, and by one or more control points. The control points serve as magnets or attractors that "pull" the curve in their direction, but never enough for the curve to intersect the control point. Figure 20-9 shows the elements of a simple Bezier curve.

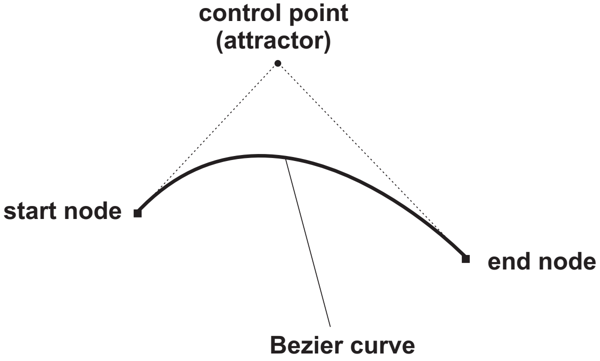


Figure 20-9 *The Bezier Spline*

The Bezier curve in Figure 20-9 can be generated by a geometrical method that consists of creating a series of progressively smaller line segments. The process, sometimes called the divide and conquer method, starts by joining the half-way points between the nodes and the attractor, thus creating a new set of nodes and a new attractor. The process continues until a sufficiently accurate approximation of the spline is reached. Figure 20-10 shows the progressive steps in creating a Bezier spline by this method.

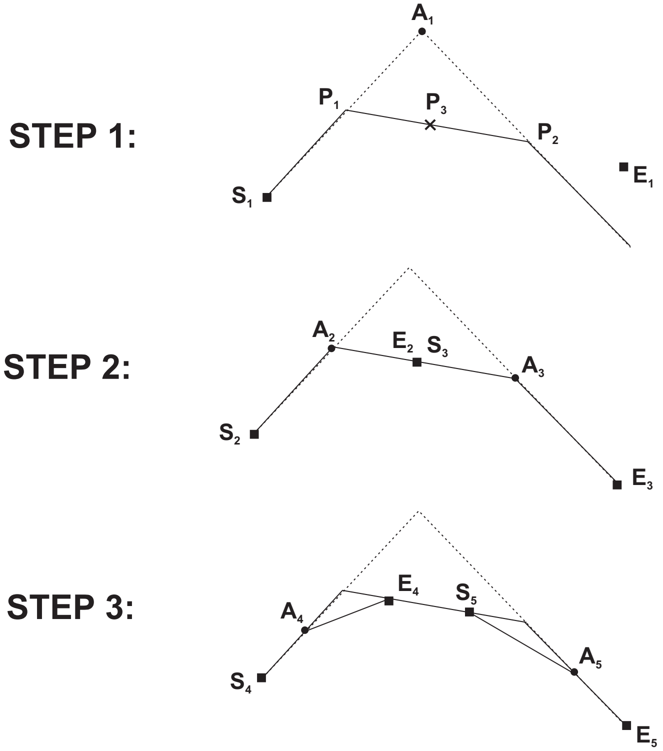


Figure 20-10 *Divide-and-Conquer Method of Creating a Bezier Curve*

In Step 1of Figure 20-10, we see the start node S1, the end node E1, and the attractor A1. We first find a point midway between S1 and A1 and label it P1. Another point midway between A1 and E1 is labeled P2. Points P1 and P2 are joined by a line segmen, whose midpoint is labeled P3. In Step 2 we can see two new figures. The first one has nodes at S2 and E2, and the attractor at A2. The second figure has nodes at S3 and E3, and the attractor at A3. In Step 3 we have joined the midpoints between the nodes and the attractors with a line segment, thus continuing the process. The two new figures have their new respective nodes and attractors, so the process can be again repeated. In Step 3 we can see how the resulting line segments begin to approximate the Bezier curve in Figure 20-9.

The divide and conquer process makes evident the fundamental assumption of the Bezier spline: the curve is in the same direction and tangent to straight lines from the nodes to the attractors. A second assumption is that the curve never intersects the attractors. The Bezier formulas are based on these assumptions.

The Bezier curve generated by the divide and conquer method is known as a quadratic Bezier. In computer graphics the most useful Bezier is the cubic form. In the cubic form the Bezier curve is defined by two nodes and two attractors. The development of the cubic Bezier is almost identical to that of the quadratic. Figure 20-11 shows the elements of a cubic Bezier curve.

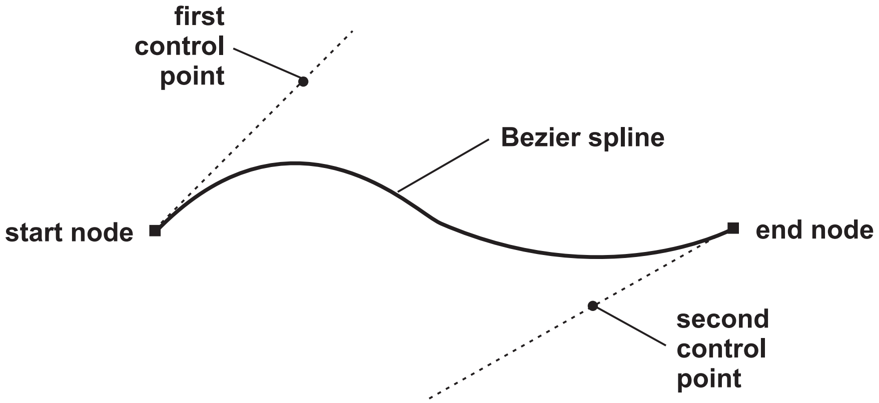


Figure 20-11 *Elements of the Cubic Bezier*

The PolyBezier() function, introduced in Windows 95, draws one or more cubic Bezier curves, each one defined by its nodes and two attractors. The function can be called to draw multiple Bezier curves. In this case the first curve requires four parameters, and all the other curves require three parameters. This is because the end node of the preceding Bezier curve serves as the start node for the next one. PolyBezier() does not change the current pen position. The Bezier curve is drawn using the pen selected in the device context. The function's general form is as follows:

```
BOOL PolyBezier(
         HDC hdc,             // 1
         CONST POINT *lppt,  // 2
         DWORD cPoints       // 3
             );
```

The first parameter is the handle to the device context. The second parameter is the address of an array of points that contains the x- and y-coordinate pairs for the nodes and control points. The third parameter is the count of the number of points in the array. This value must be one more than three times the number of curves to be drawn. For example, if the PolyBezier() function is called to draw four curves, there must be 13 coordinate pairs in the array (1 + (3 * 4)). The function returns TRUE if it succeeds and FALSE otherwise.

The Bezier data is stored in the array of points in a specific order. In the first Bezier curve, the first and fourth entries are the nodes and the second and third are attractors. Note that in the array the first and fourth entries are at offset 0 and 3; respectively, and the second and third entries are at offset 1 and 2. If there are other Bezier curves in the array, the first node is not explicit in the data, since it coincides with the end node of the preceding curve. Therefore, after the first curve, the following two entries are attractors, and the third entry is the end node. Table 20-6 shows the sequence of nodes and control points for an array with multiple Bezier curves.

Table 20-6

Nodes and Control Points for the PolyBezier() Function

NUMBER	OFFSET	TYPE
1	0	Start node of curve 1
2	1	First attractor of curve 1
3	2	Second attractor of curve 1
4	3	End node of curve 1
5	4	First attractor of curve 2
6	5	Second attractor of curve 2
7	6	End node of curve 2
8	7	First attractor of curve 3
9	8	Second attractor of curve 3
10	9	End node of curve 3

The following code fragment shows the drawing of a Bezier curve using the PolyBezier() function:

```
POINTS      pointsArray[4];  // Array of x/y coordinates
   .
   .
   .
// Fill array of points for Bezier spline
// Entries 0 and 3 are nodes
// Entries 1 and 2 are attractors
pointsArray[0].x = 150; pointsArray[0].y = 150;
pointsArray[1].x = 200; pointsArray[1].y = 75;
pointsArray[2].x = 280; pointsArray[2].y = 190;
pointsArray[3].x = 350; pointsArray[3].y = 150;

// Draw a Bezier spline
PolyBezier (hdc, pointsArray, 4);
```

The resulting Bezier curve is similar to the one in Figure 20-9.

20.4.10 Drawing with PolyBezierTo()

The PolyBezierTo() function is very similar to PolyBezier() except that the start node for the first curve is the current pen position, and the current pen position is updated to the end node of the last curve. The return value and parameters are the same for both functions. In the case of PolyBezierTo() each curve is defined by three points: two control points and the end node. Table 20-7, on the following page, shows the sequence of points stored in the points array for the PolyBezierTo() function.

20.4.11 Drawing with PolyDraw()

PolyDraw() is the most complex of the Windows line-drawing functions. It creates the possibility of drawing a series of line segments and Bezier curves, which can be joint or disjoint. PolyDraw() can be used in place of several calls to MoveTo(), LineTo(), and PolyBezierTo() functions. All the figures are drawn with the pen currently selected in the device context. The function's general form is as follows:

```
BOOL PolyDraw(
         HDC hdc,               // 1
         CONST POINT *lppt,     // 2
         CONST BYTE *lpbTypes,  // 3
         int cCount             // 4
              );
```

Table 20-7

Nodes and Control Points for the PolyBezierTo() Function

NUMBER	OFFSET	TYPE
1	0	First attractor of curve 1
2	1	Second attractor of curve 1
3	2	End node of curve 1
4	3	First attractor of curve 2
5	4	Second attractor of curve 2
6	5	End node of curve 2
7	6	First attractor of curve 3
8	7	Second attractor of curve 3
9	7	End node of curve 3

The second parameter is the address of an array of points that contains x- and y-coordinate pairs. The third parameter is an array of type BYTE that contains identifiers that define the purpose of each of the points in the array. The fourth parameter is the count of the number of points in the array of points. The function returns TRUE if it succeeds and FALSE otherwise. Table 20-8 lists the constants used to represent the identifiers entered in the function's third parameter.

Table 20-8

Constants for PolyDraw() Point Specifiers

TYPE	MEANING
PT_MOVETO	This point starts a disjoint figure. The point becomes the new current pen position.
PT_LINETO	A line is to be drawn from the current position to this point, which then becomes the new current pen position.
PT_BEZIERTO	This is a control point or end node for a Bezier curve. This constant always occurs in sets of three. The current position defines the start node for the Bezier curve. The other two coordinates are control points. The third entry is the end node.
PT_CLOSEFIGURE	The figure is automatically closed after the PT_LINETO or PT_BEZIERTO type for this point is executed. A line is drawn from the end point to the most recent PT_MOVETO or MoveTo() point. The PT_CLOSEFIGURE constant is combined by means of a bitwise OR operator with a PT_LINETO or PT_BEZIERTO constant. This indicates that the corresponding point is the last one in a figure and that the figure is to be closed.

The PolyDraw() function is not available in Windows 95 or 98. Microsoft has published the following code for implementing the function in software:

```
//*****************************
// Win95 version of PolyDraw()
// as published by Microsoft)
//*****************************
BOOL PolyDraw95(HDC  hdc,                 // handle of a device context
                CONST LPPOINT lppt,       // array of points
                CONST LPBYTE lpbTypes,    // line and curve identifiers
                int  cCount)              // count of points
{
int i;
for (i=0; i<cCount; i++)
switch (lpbTypes[i]) {
case PT_MOVETO :
MoveToEx(hdc, lppt[i].x, lppt[i].y, NULL);
break;

case PT_LINETO | PT_CLOSEFIGURE:
case PT_LINETO :
LineTo(hdc, lppt[i].x, lppt[i].y);
break;

case PT_BEZIERTO | PT_CLOSEFIGURE:
case PT_BEZIERTO :
PolyBezierTo(hdc, &lppt[i], 3);
i+=2;
break;
    }

return TRUE;
}
```

Notice that in the function PolyDraw95() the processing for closed and open figures takes place in the same intercepts. Therefore, there is no closing action implemented. When using this software implementation, including the PT_CLOSEFIGURE constant has no effect on the drawing. We have coded the following modification, named PolyDraw95A(), which closes open figures:

```
//*****************************
// Win95 version of PolyDraw()
//       improved!
//*****************************
BOOL PolyDraw95A (HDC  hdc,        // handle to device context
CONST LPPOINT lppt,  // array of points
CONST LPBYTE lpbTypes, // array of identifiers
int  cCount)           // count of points
{
int i;
  static long lastPenx, lastPeny;   // Storage for last pen position
  POINT       currentPoints[1];

// Store initial position of drawing pen
GetCurrentPositionEx (hdc, currentPoints);
lastPenx = currentPoints[0].x;
lastPeny = currentPoints[0].y;

for (i=0; i<cCount; i++)
switch (lpbTypes[i]) {
case PT_MOVETO :
MoveToEx(hdc, lppt[i].x, lppt[i].y, NULL);
// Store position for closed figures
lastPenx = lppt[i].x;
lastPeny = lppt[i].y;
break;

case PT_LINETO | PT_CLOSEFIGURE:
LineTo(hdc, lppt[i].x, lppt[i].y);
LineTo(hdc, lastPenx, lastPeny);
break;

case PT_LINETO :
LineTo(hdc, lppt[i].x, lppt[i].y);
break;

case PT_BEZIERTO | PT_CLOSEFIGURE:
// Store start points of Bezier for closing
GetCurrentPositionEx (hdc, currentPoints);
lastPenx = currentPoints[0].x;
lastPeny = currentPoints[0].y;
// Draw curve
PolyBezierTo(hdc, &lppt[i], 3);
i+=2;
// Close with line
LineTo(hdc, lastPenx, lastPeny);
break;

case PT_BEZIERTO :
// Draw Bezier
PolyBezierTo(hdc, &lppt[i], 3);
i+=2;
break;
    }

return TRUE;
}
```

The following code fragment displays several open and close figures using the PolyDraw() function or its software version Polydraw95A():

```
 POINT      pointsArray[16];     // array of points
 BYTE       controlArray[16];
 .
 .
 .
// In this example, pen is moved to start position externally
MoveToEx (hdc, 150, 50, NULL);

// Filling array of points for three lines
  // offset:       purpose:
  //   0           end point of line 1
  //   1           start of line 2
  //   2           end of line 2
  //   3           start of line 3
  //   4           end of line 3
pointsArray[0].x = 250; pointsArray[0].y = 50;
pointsArray[1].x = 150; pointsArray[1].y = 70;
pointsArray[2].x = 250; pointsArray[2].y = 70;
pointsArray[3].x = 150; pointsArray[3].y = 90;
pointsArray[4].x = 250; pointsArray[4].y = 90;
```

```
// Move to start node of Bezier curve
pointsArray[5].x = 150; pointsArray[5].y = 150;

// Filling array of points for first Bezier spline
pointsArray[6].x = 200; pointsArray[6].y = 75;
pointsArray[7].x = 280; pointsArray[7].y = 190;
pointsArray[8].x = 350; pointsArray[8].y = 150;

// Filling array for closed figure
pointsArray[9].x = 200; pointsArray[9].y = 200;
pointsArray[10].x = 300; pointsArray[10].y = 200;
pointsArray[11].x = 300; pointsArray[11].y = 300;
pointsArray[12].x = 200; pointsArray[12].y = 300;

// Filling array for second Bezier spline
pointsArray[13].x = 300; pointsArray[13].y = 90;
pointsArray[14].x = 350; pointsArray[14].y = 40;
pointsArray[15].x = 350; pointsArray[15].y = 40;
pointsArray[16].x = 400; pointsArray[16].y = 90;

// Filling control array
controlArray[0] = PT_LINETO;
controlArray[1] = PT_MOVETO;
controlArray[2] = PT_LINETO;
controlArray[3] = PT_MOVETO;
controlArray[4] = PT_LINETO;
controlArray[5] = PT_MOVETO;
controlArray[6] = PT_BEZIERTO;
controlArray[7] = PT_BEZIERTO;
controlArray[8] = PT_BEZIERTO;
controlArray[9] = PT_MOVETO;
controlArray[10] = PT_LINETO;
controlArray[11] = PT_LINETO;
controlArray[12] = PT_LINETO | PT_CLOSEFIGURE;
controlArray[13] = PT_MOVETO;
controlArray[14] = PT_BEZIERTO | PT_CLOSEFIGURE;
controlArray[15] = PT_BEZIERTO;
controlArray[16] = PT_BEZIERTO;
// Drawing lines and Bezier curves
PolyDraw95A (hdc, pointsArray, controlArray, 17);
```

Figure 20-12 is an approximation of the figures that result from the previous code sample.

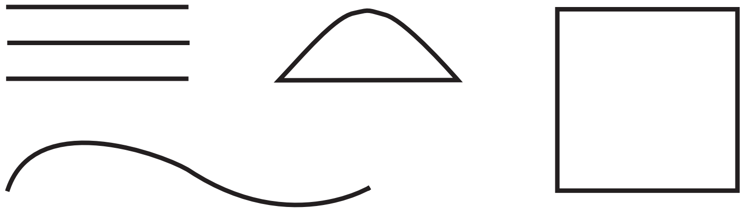


Figure 20-12 *Approximate Result of the PolyDraw() Code Sample*

20.4.12 Pixel and Line Demonstration Program

The program named PXL_DEMO, located in the Pixel and Line Demo project folder of the book's software package, is a demonstration of the drawing functions discussed in this chapter. Pixel-level functions are used to display the point plot of a sine curve. Also, the program contains a function named DrawDot(), which uses the SetPixelV() function to draw a black screen dot by setting five adjacent pixels. The demo program displays a pop-up menu, named Line Functions, which has menu commands for exercising LineTo(), PolyLineTo(), PolyLine(), PolyPolyLine(), Arc(), AngleArc(), PolyBezier(), and PolyDraw(). Code for implementing PolyDraw() and AngleArc() in software is also included in the demo program.

Chapter 21

Drawing Figures, Regions, and Paths

Topics:

- Setting the drawing attributes
- Drawing closed figures such as rectangles, ellipses, chords, pie sections, and polygons
- Drawing operations on rectangles
- Creating, combining, filling, and painting regions
- Clipping operations
- Creating , deleting, and converting paths
- Path information and rendering
- Filled Figures Demo program

In this chapter we continue exploring the graphics functions in the Windows GDI, concentrating on geometrical figures that contain an interior region, in addition to a perimeter or outline. These are called solid or closed figures. The interior area allows them to be filled with a given color, hatch pattern, or bitmap image. At the same time, the perimeter of a closed figure can be rendered differently than the filled area. For example, the circumference of a circle can be outlined with a 2-pixel-wide black pen, and the circle's interior filled with 1-pixel-wide red lines, slanted at 45 degrees, and separated from each other by 10 pixels.

21.1 Closed Figures

Closed figures allow several graphics manipulations. For instance, a solid figure can be used to define the output area to which Windows can perform drawing operations. This area, called the clipping region, allows you to produce unique and interesting graphics effects, such as filling geometrical figures with text or pictures.

Some closed figures are geometrically simple: a rectangle, an ellipse, or a symmetrical polygon. More complex figures are created by combining simpler ones. A region is an area composed of one or more rectangles, polygons, or ellipses. Regions are used to define irregular areas that can be filled, to clip output, or to establish an area for mouse input.

Paths are relatively new graphics objects, since they were introduced with Windows NT, and are also supported in Windows 95/98. A path is the route the drawing instrument follows in creating a figure or set of figures. It is used to define the outline of a graphics object. After a path is created, you can draw its outline (called stroking the path), fill its interior, or both. A path can also be used in clipping, or converted into a region. Paths and regions add a powerful dimension to Windows graphics.

21.1.1 Area of a Closed Figure

A closed figure has both a perimeter and an interior. The perimeter of a closed figure is drawn using the current pen and the GDI line-related attributes discussed in Chapter 20. The interior is filled using the current brush, also partly discussed in Chapter 20. There are several closed figures that can be drawn with the Windows GDI; among them are ellipses, polygons, chords, pies, and rectangles. Later in this chapter we see that the Windows names for some of these figures are not geometrically correct. Areas bound by complex lines, such as irregular polygons, Bezier curves, and text characters, can also be filled.

Like lines and curves, closed figures have attributes that determine their characteristics. Most of the attributes that relate to closed figures are described in Chapter 20. These include the mix mode, the background mode, the arc direction, the brush pattern, the pen styles, as well as the brush, pen, and background colors. Two attributes that are specific to closed figures are the brush origin and the polygon filling mode.

21.1.2 Brush Origin

Figure 20-4, in the preceding chapter, shows the various hatch patterns that can be used with a brush. Windows locates the hatch pattern in reference to coordinates (0,0). It is important to know that this origin is in device units, not in logical units. The hatch pattern is a bitmap. In Windows 95/98, the bitmap is 8-by-8 pixels. In Windows NT, it can have any size. The painting process consists of repeating the bitmap horizontally and vertically until the area is filled.

In some cases the default origin of the bitmap produces undesirable results. This usually happens when the alignment of a filled figure does not coincide with that of the brush hatch pattern. Figure 21-1 shows two rectangles, one filled with an unaligned hatch pattern and the other one filled with an aligned hatch pattern.

The SetBrushOrgEx() function can be used to reposition the hatch bitmap in relation to the origin of the client area. The function's general form is as follows:

```
BOOL SetBrushOrgEx(
         HDC hdc,        // 1
         int nXOrg,      // 2
         int nYOrg,      // 3
         LPPOINT lppt    // 4
              );
```

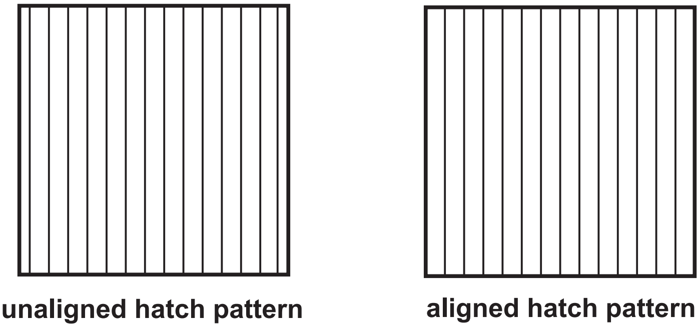


Figure 21-1 *Brush Hatch Patterns*

The second parameter specifies the x-coordinate of the new brush origin. In Windows 95/98, the range is 0 to 7. In Windows NT, the range cannot be greater than the width of the bitmap. In either case, if the entered value exceeds the width of the bitmap, it is adjusted by performing the modulus operation:

```
xOrg = xOrg % bitmap width
```

The third parameter is the y-coordinate of the new brush origin. Its range and adjustments are the same as for the second parameter. The fourth parameter is a pointer to a POINT structure that stores the origin of the brush previously selected in the device context. If this information is not required, NULL can be entered in this parameter. The function returns TRUE if the operation succeeds and FALSE otherwise.

A call to SetBrushOrgEx() sets the origin of the next brush that an application selects into the device context. Note that the first parameter of the SetBrushOrgEx() function is the handle to the device context, and that the brush variable is nowhere to be found in the parameter list. Therefore, the brush origin is associated with the device context, not with a particular brush. The origin in the device context is assigned to the next brush created.

The following code fragment shows the display of two rectangles. The brush origin is changed for the second one. The Rectangle() function is described later in this chapter.

```
static HBRUSH     vertBrush1, vertBrush2;
LOGBRUSH          brush1;
.
.
.
// Create a brush
brush1.lbStyle = BS_HATCHED;
brush1.lbColor = RGB(0x0, 0xff, 0x0);
brush1.lbHatch = HS_VERTICAL;
vertBrush1 = CreateBrushIndirect (&brush1);
SelectObject (hdc, (HGDIOBJ)(HBRUSH) vertBrush1);
// Draw a rectangle with this brush
Rectangle (hdc, 150, 150, 302, 300);
```

```
  // Create a new hatched brush with offset origin
  brush1.lbStyle = BS_HATCHED;
  brush1.lbColor = RGB(0x0, 0x0, 0x0);
  brush1.lbHatch = HS_VERTICAL;
  // Offset the new brush 6 pixels
  SetBrushOrgEx (hdc, 5, 0, NULL);
  vertBrush2 = CreateBrushIndirect (&brush1);
  SelectObject (hdc, (HGDIOBJ)(HBRUSH) vertBrush2);
  // Draw a rectangle with the new brush
  Rectangle (hdc, 350, 150, 502, 300);
```

The results of executing this code are similar to the rectangles in Figure 21-1. The GetBrushOrg() function can be used to retrieve the origin of the current brush.

Notice that Windows documentation recommends that to avoid brush misalignment an application should call SetStretchBltMode() with the stretching mode set to HALFTONE before calling SetBrushOrgEx().

21.1.3 Object Selection Macros

The Windows header file windowsx.h contains four macros that can be used in selecting a pen, brush, font, or bitmap. The advantage of using these macros is that the objects are automatically typecast correctly. The macros are named SelectPen(), SelectBrush(), SelectFont(), and SelectBitmap(). They are all defined similarly. The SelectBrush() macro is as follows:

```
#define SelectBrush (hdc, hbr) \
        ((HBRUSH) SelectObject ((hdc, (HGDIOBJ)(HBRUSH)(hbr)))
```

You can use these macros to easily produce correct code that is correct and more portable. Programs that use the object selection macros must contain the statement:

```
#include <windowsx.h>
```

21.1.4 Polygon Fill Mode

The polygon fill mode attribute determines how overlapping areas of complex polygons and regions are filled. The polygon fill mode is set with the SetPolyFillMode() function. The function's general form is as follows:

```
int SetPolyFillMode(
          HDC hdc,              // 1
          int iPolyFillMode     // 2
               );
```

The second parameter is one of two constants: ALTERNATE and WINDING. ALTERNATE defines a mode that fills between odd-numbered and even-numbered polygon sides, or in other words, those areas that can be reached from the outside of the polygon by crossing an odd number of lines, excluding the vertices. This fill algorithm is based on what is called the parity rule.

The WINDING mode is based on the nonzero winding rule. In the WINDING mode, the direction in which the figure is drawn determines whether an area is to be filled. A polygon line segment is drawn either in a clockwise or a counterclockwise direction. The term winding relates to the clockwise and counterclockwise drawing of the polygon segments. An imaginary line, called a ray, is extended from an enclosed area in the figure, to a point distant from the figure and outside of it. The ray must be on a positive x-direction. Every time the ray crosses a clockwise winding, a counter is incremented. The same counter is decremented whenever the line crosses a counterclockwise winding. The winding counter is examined when the ray reaches the outside of the figure. If the winding counter is nonzero, the area is filled.

In figures that have a single interior region the fill mode is not important. This is not the case in figures that have enclosed areas. A typical case is a polygon in the shape of a five-pointed star with an enclosed pentagon. In this case, the ALTERNATE mode does not fill the interior pentagon, while the WINDING fill mode does. In more complex figures the same rules apply, although they may not be immediately evident. Figure 21-2 shows the results of the polygon fill modes in two different figures. Recall that in the WINDING fill mode, the direction in which the line segments are drawn is significant.

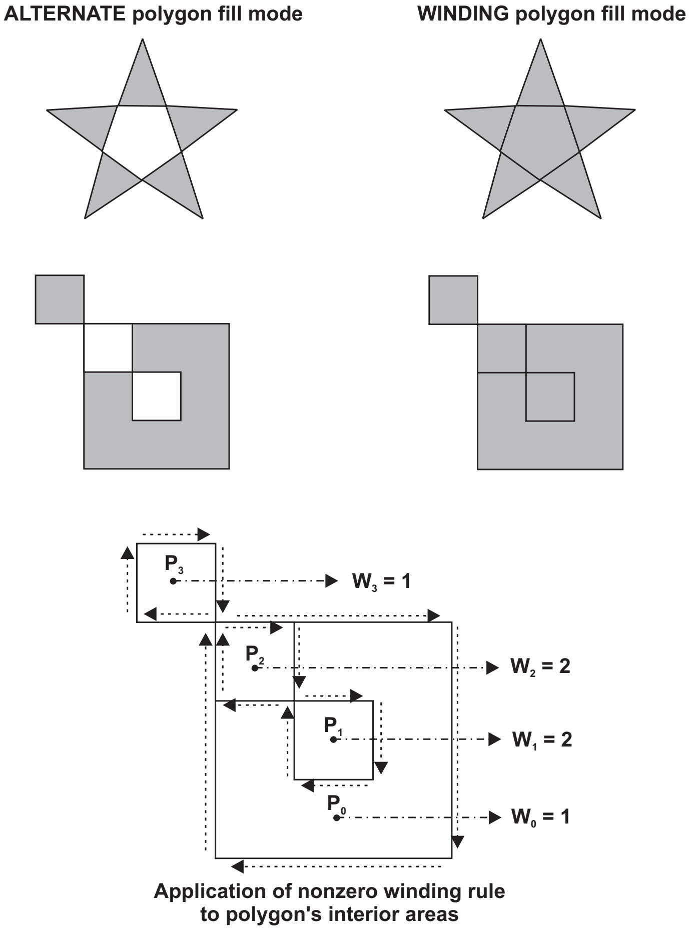

ALTERNATE polygon fill mode
WINDING polygon fill mode
P_3
$W_3 = 1$
P_2
$W_2 = 2$
P_1
$W_1 = 2$
P_0
$W_0 = 1$
Application of nonzero winding rule
to polygon's interior areas

Figure 21-2 *Effects of the Polygon Fill Modes*

In Figure 21-2 you can see the application of the nonzero winding rule to the interior areas of a complex polygon. For example, the ray from point P1 to the exterior of the figure crosses two clockwise segments (windings). Therefore, it has a winding value of 2. Since the winding is nonzero, the area is filled. The same rule can be applied to other points in the figure's interior, as shown in Figure 21-2.

Notice that some Windows documentation states that in the WINDING mode all interior areas of a figure are filled. This oversimplification is not correct. If the interior segments of the polygon in Figure 21-2 were drawn in the opposite direction, some areas would have zero winding and would not be filled. The program named FIL_DEMO, located in the Filled Figure Demo project folder, in the book's software package, contains the menu command Polygon (2), on the Draw Figures pop-up menu, which displays a complex polygon that has an unfilled interior in the WINDING mode.

You can retrieve the current polygon fill mode with the GetPolyFillMode() function. The only parameter to the call is the handle to the device context. The value returned is either ALTERNATE or WINDING.

21.1.5 Creating Custom Brushes

In Chapter 20 we mentioned that a logical brush can be created with the CreateBrushIndirect() function. CreateBrushIndirect() has the following general form:

```
HBRUSH CreateBrushIndirect (
CONST LOGBRUSH* lplb    // 1
);
```

The only parameter of the function is the address of a structure of type LOGBRUSH. The structure members are divided into three groups: brush style, brush color, and hatch style. The function returns the handle to the created brush. Table 21-1 lists the members of the LOGBRUSH structure.

Once the brush is created, it can be selected into the device context by either calling the SelectObject() function or the SelectBrush() macro discussed previously in this chapter. In addition, you can create specific types of brushes more easily by using the functions CreateSolidBrush(), CreateHatchBrush(), or CreatePatternBrush(). CreateSolidBrush() is used to create a brush of a specific color and no hatch pattern. The function's general form is as follows:

```
HBRUSH CreateSolidBrush (COLORREF colorref);
```

The only parameter is a color value in the form of a COLORREF type structure. The color value can be entered directly using the RGB macro. For example, the following code fragment creates a solid blue brush:

```
  static HBRUSH        solidBlueBrush;
  .
  .
  .
  solidBlueBrush = CreateSolidBrush ( RGB (0x0, 0x0, 0xff));
```

Table 21-1

LOGBRUSH Structure Members

BRUSH STYLE	DESCRIPTION
BS_DIBPATTERN	A pattern brush defined by a device-independent bitmap. If lbStyle is BS_DIBPATTERN, the lbHatch member contains a handle to a packed DIB.
BS_DIBPATTERNPT	Same a BS_DIBPATTERN but the lbHatch member contains a pointer to a packed DIB.
BS_HATCHED	Hatched brush.
BS_HOLLOW	Hollow brush.
BS_NULL	Same as BS_HOLLOW.
BS_PATTERN	Pattern brush defined by a memory bitmap.
BS_SOLID	Solid brush.
BRUSH COLOR	**DESCRIPTION**
DIB_PAL_COLORS	The color table consists of an array of 16-bit indices into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.
HATCH STYLE	**DESCRIPTION**
HS_BDIAGONAL	A 45-degree upward, left-to-right hatch.
HS_CROSS	Horizontal and vertical cross-hatch.
HS_DIAGCROSS	45-degree crosshatch.
HS_FDIAGONAL	A 45-degree downward, left-to-right hatch.
HS_HORIZONTAL	Horizontal hatch.
HS_VERTICAL	Vertical hatch.

CreatehatchBrush() creates a logical brush with a hatch pattern and color. The function's general form is as follows:

```
HBRUSH CreateHatchBrush(
          int fnStyle,       // 1
          COLORREF clrref    // 2
               );
```

The first parameter is one of the hatch style identifiers listed in Figure 21-2. The second parameter is a color value of COLORREF type. The function returns the handle to the logical brush.

If an application requires a brush with a hatch pattern different from the ones predefined in Windows, it can create a custom brush with its own bitmap. In Windows 95/98, the size of the bitmap cannot exceed 8-by-8 pixels, but there is no size restriction in Windows NT. The function's general form is as follows:

```
HBRUSH CreatePatternBrush (
HBITMAP hbmp       // 1
);
```

The function's only parameter is a handle to the bitmap that defines the brush. The bitmap can be created with CreateBitmap(), CreateBitmapIndirect() or CreateCompatibleBitmap() functions. These functions are described in Chapter 8.

21.2 Drawing Closed Figures

There are seven Windows functions that draw closed figures, shown in Table 21-2.

Table 21-2

Windows Functions for Drawing Closed Figures

FUNCTION	FIGURE
Rectangle()	Rectangle with sharp corners
RoundRect()	Rectangle with rounded corners
Ellipse()	Ellipse or circle
Chord()	Solid figure created by an arc on the circumference of an ellipse connected by a chord
Pie()	Pie-shaped wedge created by joining the end points of an arc on the perimeter of an ellipse with the center of the arc
Polygon()	Closed polygon
PolyPolygon()	Series of closed polygons, possibly overlapping

All functions that draw closed figures use the pen currently selected in the device context for the figure outline, and the current brush for filling the interior. All of the line attributes discussed in Chapter 20 apply to the perimeter of solid figures. The programmer has control of the width of the perimeter, its line style, and its color. By selecting NULL_PEN you can draw a figure with no perimeter. The fill is determined by the current brush. Windows approximates the color of the brush according to the device capabilities.

This often requires manipulating dot sizes by a process called dithering. Dithering is a technique that creates the illusion of colors or shades of gray by treating the targeted areas as a dot pattern. The process takes advantage of the fact that the human eye tends to blur small spots of different color by averaging them into a single color or shade. For example, a pink color effect can be produced by mixing red and white dots.

The brush can be any one of the stock brushes: WHITE_BRUSH, LTGRAY_BRUSH, GRAY_BRUSH, DKGRAY_BRUSH, BLACK_BRUSH, and NULL_BRUSH. All stock brushes are solid. NULL_BRUSH is used to draw figures without filling the interior. The GetStockObject() function is used to obtain a handle to one of the stock brushes. Since stock brushes need not be stored locally, the most common case is that the stock brush is retrieved and installed in the device context at the time it is needed. SelectBrush() and GetStockObject() can be combined as follows:

```
SelectBrush (hdc, GetStockObject (WHITE_BRUSH));
```

The creation and installation of custom brushes was discussed previously in this chapter.

21.2.1 Drawing with Rectangle()

The simplest solid-figure drawing function is Rectangle(). This function draws a rectangle using the current pen for the outline and fills it with the current brush. The function's general form is as follows:

```
BOOL Rectangle(
         HDC hdc,          // 1
         int nLeftRect,    // 2
         int nTopRect,     // 3
         int nRightRect,   // 4
         int nBottomRect   // 5
         );
```

The second and third parameters are the coordinates of the upper-left corner of the rectangle. The fourth and fifth parameters are the coordinates of the lower-right corner. The function returns TRUE if it succeeds and FALSE if it fails. Figure 21-3 shows a rectangle drawn using this function.

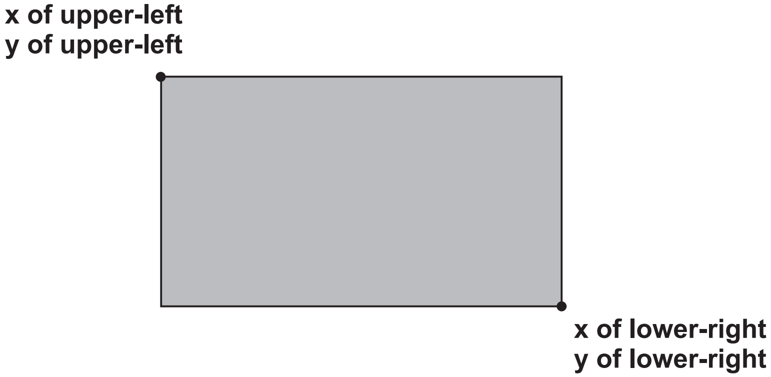


Figure 21-3 *Figure Definition in the Rectangle() Function*

21.2.2 Drawing with RoundRect()

The RoundRect() function draws a rectangle with rounded corners. Like all the solid figure drawing functions, it uses the current pen for the outline and fills the figure with the current brush. The function's general form is as follows:

```
BOOL RoundRect(
         HDC hdc,          // 1
         int nLeftRect,    // 2
         int nTopRect,     // 3
         int nRightRect,   // 4
         int nBottomRect,  // 5
         int nWidth,       // 6
         int nHeight       // 7
         );
```

The second and third parameters are the coordinates of the upper-left corner of the bounding rectangle. The fourth and fifth parameters are the coordinates of the lower-right corner. The sixth parameter is the width of the ellipse that is used for drawing the rounded corner arc. The seventh parameter is the height of this ellipse. The function returns TRUE if it succeeds and FALSE if it fails. Figure 21-4 shows the values that define a rounded-corner rectangle drawn using this function.

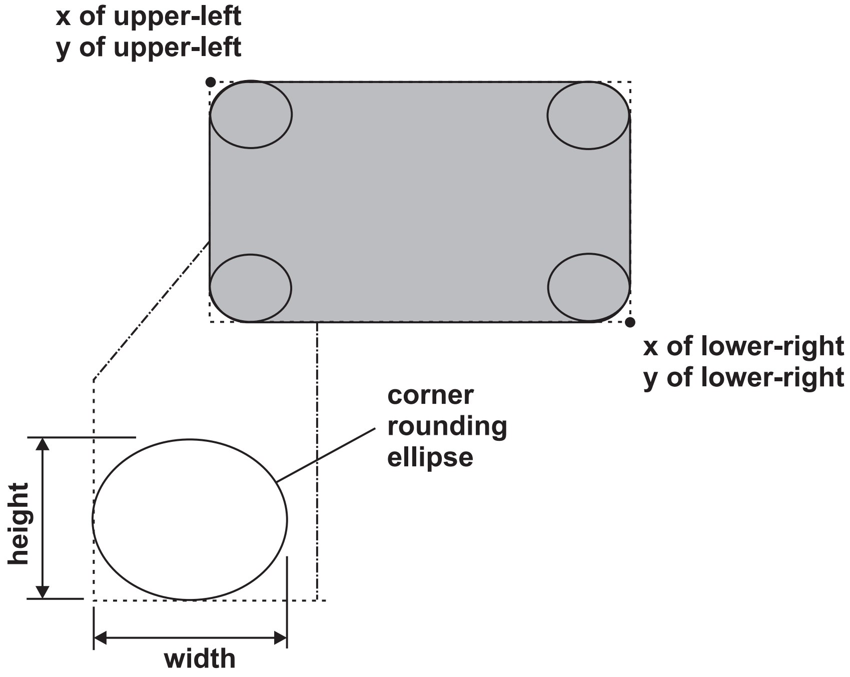


Figure 21-4 *Definition Parameters for the RoundRect() Function*

21.2.3 Drawing with Ellipse()

The Ellipse() function draws a solid ellipse. Ellipse() uses the current pen for the outline and fills the figure with the current brush. The function's general form is as follows:

```
BOOL Ellipse(
         HDC hdc,         // 1
         int nLeftRect,   // 2
         int nTopRect,    // 3
         int nRightRect,  // 4
         int nBottomRect  // 5
         );
```

The second and third parameters are the coordinates of the upper-left corner of a rectangle that binds the ellipse. The fourth and fifth parameters are the coordinates of the lower-right corner of this rectangle. The function returns TRUE if it succeeds and FALSE if it fails. Figure 21-5 shows an ellipse drawn using this function.

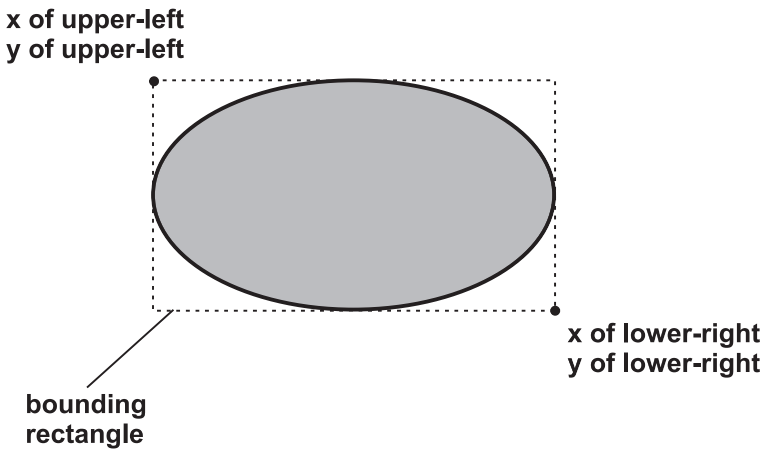


Figure 21-5 *Figure Definition in the Ellipse() Function*

21.2.4 Drawing with Chord()

Chord() draws a solid figure composed of an arc of an ellipse whose ends are connected to each other by a straight line, called a secant. The Chord() function is related to the Arc() function described in Chapter 20. The parameters that define the elliptical arc are the same for the Arc() as for the Chord() function. The function's general form is as follows:

```
BOOL Chord(
         HDC hdc,          // 1
         int nLeftRect,    // 2
         int nTopRect,     // 3
         int nRightRect,   // 4
         int nBottomRect,  // 5
         int nXRadial1,    // 6
         int nYRadial1,    // 7
         int nXRadial2,    // 8
         int nYRadial2     // 9
         );
```

The second and third parameters are the x- and y-coordinates of the upper-left corner of a rectangle that contains the ellipse, while the fourth and fifth parameters are the coordinates of its lower-right corner. The sixth and seventh parameters define the coordinates of a point that sets the start point of the secant. The last two parameters set the end points of the secant. The elliptical arc is always drawn in the counterclockwise direction. The SetArcDirection() function has no effect in this case.

The coordinates of the start and end points of the secant need not coincide with the elliptical arc, since Windows prolongs the secant until it intersects the elliptical arc. Figure 21-6, on the following page, shows the elements that define the figure.

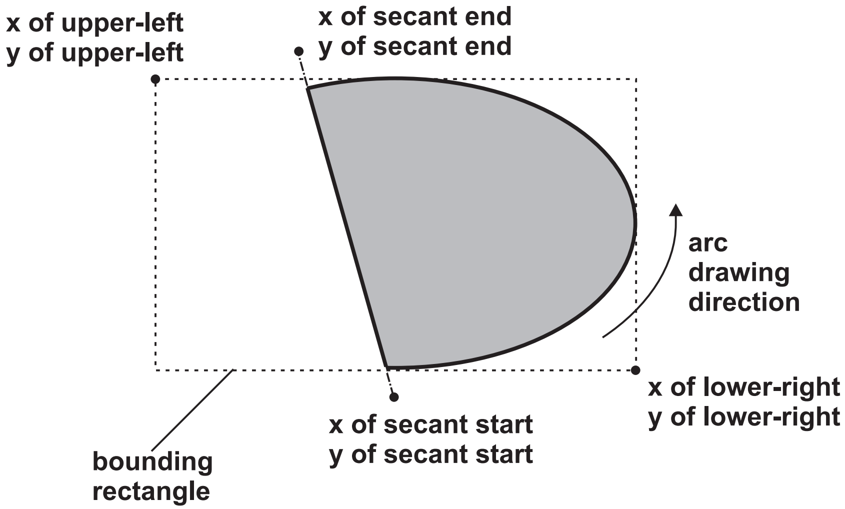


Figure 21-6 *Figure Definition in the Chord() Function*

Notice that the name of the Chord() function does not coincide with its mathematical connotation. Geometrically, a chord is the portion of a secant line that joins two points on a curve, not a solid figure.

21.2.5 Drawing with Pie()

Pie() draws a solid figure composed of the arc of an ellipse whose ends are connected to the center by straight lines. In Windows terminology the two straight lines are called radials. The Pie() function is related to the Arc() function described in Chapter 20. The parameters that define the elliptical arc are the same for the Arc() as for the Pie() functions. It is also similar to the Chord() function previously described. The difference between Chord() and Pie() is that in Chord() the line points are connected to each other and in Pie() they are connected to the center of the ellipse. The function's general form is as follows:

```
BOOL Pie(
          HDC hdc,          // 1
          int nLeftRect,    // 2
          int nTopRect,     // 3
          int nRightRect,   // 4
          int nBottomRect,  // 5
          int nXRadial1,    // 6
          int nYRadial1,    // 7
          int nXRadial2,    // 8
          int nYRadial2     // 9
          );
```

The second and third parameters are the x- and y-coordinates of the upper-left corner of a rectangle that contains the ellipse, while the fourth and fifth parameters are the coordinates of its lower-right corner. The sixth and seventh parameters define the coordinates of the end point of the start radial line. The last two parameters set the coordinates of the end points of the end radial line. The elliptical arc is always drawn in the counterclockwise direction. The SetArcDirection() function has no effect in this case.

The coordinates of the start and end points of the radials need not coincide with the elliptical arc, since Windows prolongs these lines until they intersect the elliptical arc. Figure 21-7 shows the elements that define the figure.

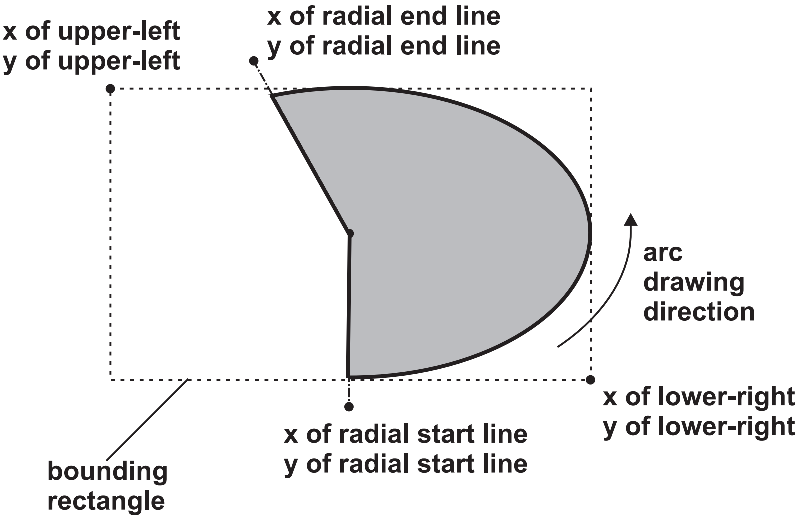


Figure 21-7 *Figure Definition in the Arc() Function*

21.2.6 Drawing with Polygon()

The Polygon() function is similar to the Polyline() function described in Chapter 20. The main difference between a polygon and a polyline is that the polygon is closed automatically by drawing a straight line from the last vertex to the first one. The polygon is drawn with the current pen and filled with the current brush. The inside of the polygon is filled according to the current polygon fill mode, which can be ALTERNATE or WINDING. Polygon fill modes were discussed in detail earlier in this chapter. The function's general form is as follows:

```
BOOL Polygon(
         HDC hdc,                 // 1
         CONST POINT *lpPoints,   // 2
         int nCount               // 3
         );
```

The second parameter is the address of an array of points that contains the x- and y-coordinate pairs of the polygon vertices. The third parameter is the count of the number of vertices in the array. The function returns TRUE if it succeeds and FALSE otherwise.

When drawing the lines that define a polygon you can repeat the same segment. It is not necessary to avoid going over an existing line. When the WINDING fill mode is selected, however, the direction of each edge determines the fill action. The following code fragment shows the drawing of a complex polygon that is defined in an array of structures of type POINT.

```
// Arrays of POINT structures for polygon vertices
POINT  polyPoints1[] = {
      { 100, 100 }, // 1
      { 150, 100 }, // 2
      { 150, 150 }, // 3
      { 300, 150 }, // 4
      { 300, 300 }, // 5
      { 150, 300 }, // 6
      { 150, 150 }, // 7
      { 200, 150 }, // 8
      { 200, 200 }, // 9
      { 250, 200 }, // 10
      { 250, 250 }, // 11
      { 200, 250 }, // 12
      { 200, 200 }, // 13
      { 150, 200 }, // 14
      { 150, 150 }, // 15
      { 100, 150 }  // 16
} ;
.
.
.
// Draw the polygon using array data
SetPolyFillMode (hdc, ALTERNATE);
Polygon (hdc, polyPoints1, 17);
```

Figure 21-8 shows the figure that results from this code when the ALTERNATE fill mode is active. The polygon vertices are numbered and labeled.

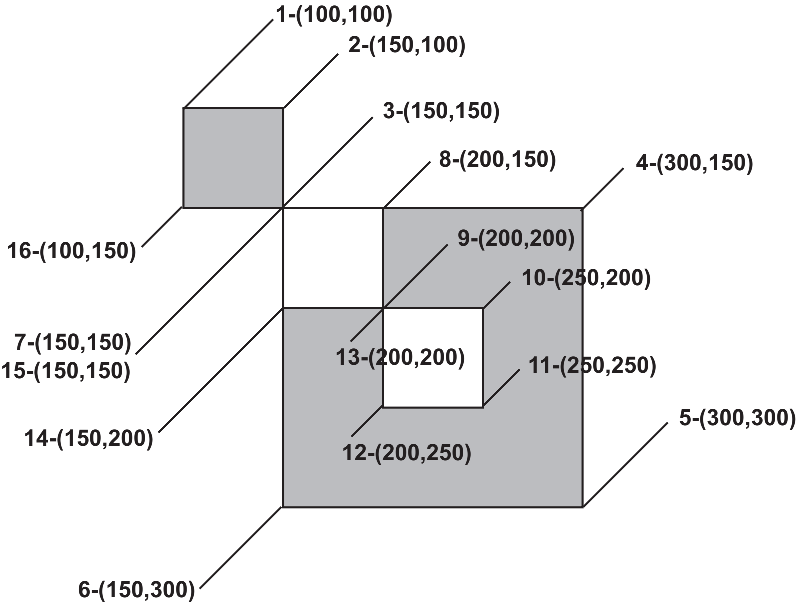


Figure 21-8 *Figure Produced by the Polygon Program*

21.2.7 Drawing with PolyPolygon()

As the function name implies, PolyPolygon() is used to draw several closed polygons. The outlines of all the polygons are drawn with the current pen and the interiors are filled with the current brush and according to the selected fill mode. The polygons can overlap. Unlike the Polygon() function, the figures drawn with PolyPolygon() are not automatically closed. The PolyPolygon() function is similar to the PolyPolyline() function described in Chapter 20. Like the PolyPolyline() function, PolyPolygon() requires an array of values that holds the number of points for each polygon. The function's general form is as follows:

```
BOOL PolyPolygon(
         HDC hdc,                    // 1
         CONST POINT *lpPoints,      // 2
         CONST INT *lpPolyCounts,    // 3
         int nCount                  // 4
         );
```

The second parameter is an array containing the vertices of the various polygons. The third parameter is an array that contains the number of vertices in each of the polygons. The fourth parameter is the count of the number of elements in the third parameter, which is also the number of polygons to be drawn. The function returns TRUE if it succeeds and FALSE otherwise. The following code fragment shows the drawing of four polygons, each one with four vertices, using the PolyPolygon() function.

```
  // Arrays of POINT structures for holding the vertices
  // of all four polygons
  POINT  polyPoly1[] = {
     { 150, 150 }, // 1  |
     { 300, 150 }, // 2  |
     { 300, 300 }, // 3  |-- first polygon
     { 150, 300 }, // 4  |
     { 150, 150 }, // 5  |

     { 200, 200 }, // 6  |
     { 250, 200 }, // 7  |
     { 250, 250 }, // 8  |-- second polygon
     { 200, 250 }, // 9  |
     { 200, 200 }, // 10 |

     { 150, 150 }, // 11 |
     { 200, 150 }, // 12 |
     { 200, 200 }, // 13 |-- third polygon
     { 150, 200 }, // 14 |
     { 150, 150 }, // 15 |

     { 100, 100 }, // 11 |
     { 150, 100 }, // 12 |
     { 150, 150 }, // 13 |-- fourth polygon
     { 100, 150 }, // 14 |
     { 100, 100 }, // 15 |
   } ;

  // Array holding the number of segments in each
  // polygon
  int  vertexArray[] = {
     { 5 },
     { 5 },
     { 5 },
     { 5 }
   };
   .
   .
   .
   // Draw the polygon using array data
   SetPolyFillMode (hdc, ALTERNATE);
   PolyPolygon (hdc, polyPoly1, vertexArray, 4);
```

The resulting polygon is identical to the one in Figure 21-8.

21.3 Operations on Rectangles

Rectangular areas are often used in Windows programming. Child windows are usually in the form of a rectangle, as are message and input boxes as well as many other graphics components. For this reason, the Windows API includes several functions that operate on rectangles. These are listed in Table 21-3.

Table 21-3

Windows Functions Related to Rectangular Areas

FUNCTION	FIGURE
FillRect()	Fills the interior of a rectangle using a brush defined by its handle
FrameRect()	Draws a frame around a rectangle
InvertRect()	Inverts the pixels in a rectangular area
DrawFocusRect()	Draws rectangle with special dotted pen to indicate that the object has focus

One common characteristic of all the rectangular functions is that the rectangle coordinates are stored in a structure of type RECT. The use of a RECT structure is a more convenient way of defining a rectangular area than by passing coordinates as function parameters. It allows the application to easily change the location of a rectangle,and to define a rectangular area without hard-coding the values, thus making the code more flexible. The RECT structure is as follows:

```
typdef struct _RECT {
  LONG   left;     // x coordinate of upper-left corner
  LONG   top;      // y of upper-left corner
  LONG   right;    // x coordinate of bottom-right corner
  LONG   bottom;   // y of bottom-right
} RECT;
```

21.3.1 Drawing with FillRect()

The FillRect() function fills the interior of a rectangular area, whose coordinates are defined in a RECT structure. The function uses a brush specified by its handle. The filled area includes the upper-left corner of the rectangle but excludes the bottom-right corner. The function's general form is as follows:

```
int FillRect(
         HDC hDC,            // 1
         CONST RECT *lprc,   // 2
```

```
        HBRUSH hbr         // 3
        );
```

The second parameter is a pointer to a structure of type RECT that contains the rectangle's coordinates. The third parameter is the handle to a brush or a system color. If a handle to a brush, it must have been obtained with CreateSolidBrush(), CreatePatternBrush(), or CreateHatchBrush() functions described previously. Additionally, you may use a stock brush and obtain its handle by means of GetStockObject(). The function returns TRUE if it succeeds and FALSE if it fails. Table 21-4 lists the constants that are used to identify the system colors in Windows.

Table 21-4

Windows System Colors

VALUE	MEANING
COLOR_3DDKSHADOW	Dark shadow display elements
COLOR_3DFACE, COLOR_BTNFACE	Face color for display elements
COLOR_3DHILIGHT, COLOR_3DHIGHLIGHT, COLOR_BTNHILIGHT, COLOR_BTNHIGHLIGHT	Highlight color for edges facing the light source
COLOR_3DLIGHT	Light color for edges facing the light source
COLOR_3DSHADOW, COLOR_BTNSHADOW	Shadow color for edges facing away from the light source
COLOR_ACTIVEBORDER	Active window border
COLOR_ACTIVECAPTION	Active window caption
COLOR_APPWORKSPACE	Background color of multiple document Interface. (MDI) applications
COLOR_BACKGROUND, COLOR_DESKTOP	Desktop color
COLOR_BTNTEXT	Text on push buttons
COLOR_CAPTIONTEXT	Text in caption, size box, and scroll bar Arrow box
COLOR_GRAYTEXT	Grayed (disabled) text Set to 0 if the Current display driver does not support a solid gray color
COLOR_HIGHLIGHT	Item(s) selected in a control
COLOR_HIGHLIGHTTEXT	Text of item(s) selected in a control
COLOR_INACTIVEBORDER	Inactive window border
COLOR_INACTIVECAPTION	Inactive window caption
COLOR_INACTIVECAPTIONTEXT	Color of text in an inactive caption
COLOR_INFOBK	Background color for ToolTip controls
COLOR_INFOTEXT	Text color for ToolTip controls
COLOR_MENU	Menu background
COLOR_MENUTEXT	Text in menus
COLOR_SCROLLBAR	Scroll bar gray area
COLOR_WINDOW	Window background
COLOR_WINDOWFRAME	Window frame
COLOR_WINDOWTEXT	Text in windows

21.3.2 Drawing with FrameRect()

The FrameRect() function draws border around a rectangular area, whose coordinates are defined in a RECT structure. The width and height of this border are one logical unit. The border is drawn with a brush, not with a pen. The brush is specified by its handle. The function's general form is as follows:

```
int FrameRect(
          HDC hDC,            // 1
          CONST RECT *lprc,   // 2
          HBRUSH hbr          // 3
          );
```

The second parameter is a pointer to a structure of type RECT that contains the coordinates. The third parameter is the handle to a brush, which must have been obtained with CreateSolidBrush(), CreatePatternBrush(), or CreateHatchBrush() functions described previously. Additionally, you may use a stock brush and obtain its handle by means of GetStockObject(). The function returns TRUE if it succeeds and FALSE if it fails.

Because the borders of the rectangle are drawn with a brush, rather than with a pen, the function is used to produce figures that can not be obtained by other means. For example, if you select a brush with the vertical hatch pattern HS_VERTICAL, the resulting rectangle has dotted lines for the upper and lower segments since this is the brush pattern. The vertical segments of the rectangle are displayed as solid lines only when the rectangle's side coincides with the brush's bitmap pattern. Another characteristic of the FrameRect() function is that dithered colors can be used to draw the rectangle's border.

21.3.3 Drawing with DrawFocusRect()

The DrawFocusRect() function draws a rectangle of dotted lines. The rectangle's interior is not filled. The function's name relates to its intention, not to its operation, since the drawn rectangle is not given the keyboard focus automatically. The DrawFocusRect() function uses neither a pen nor a brush to draw the perimeter. The dotted lines used for the rectangle are one pixel wide, one pixel high, and are separated by one pixel. The function's general form is as follows:

```
BOOL DrawFocusRect(
          HDC hDC,           // 1
          CONST RECT *lprc   // 2
          );
```

The second parameter is a pointer to a structure of type RECT that contains the coordinates. The function returns TRUE if it succeeds and FALSE if it fails. Figure 21-9 shows a rectangle drawn with the DrawFocusRect() function.

There are several unique features of the DrawFocusRect() function. The most important feature is that the rectangle is displayed by means of an XOR operation on the background pixels. This ensures that it is visible on most backgrounds. Also, that the rectangle can be erased by calling the function a second time with the same parameters. This is a powerful feature of this function since an application can call DrawFocusRect() to draw a rectangle around an object or background, and then erase the rectangle and restore the display without having to preserve the overdrawn area.

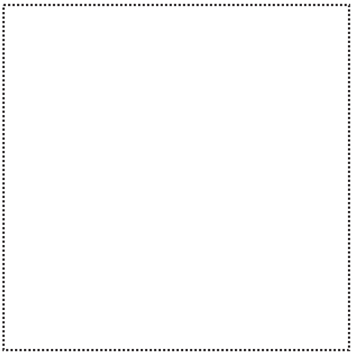


Figure 21-9 *Rectangle Drawn with DrawFocusRect()*

The area that contains a rectangle drawn with DrawFocusRect() cannot be scrolled. In order to scroll this area you can call DrawFocusRect() a second time to erase the rectangle, scroll the display, then call the function again to redraw the focus rectangle.

21.3.4 Auxiliary Operations on Rectangles

Windows provides several auxiliary functions designed to facilitate manipulating structures of type RECT. Although these functions have no unique functionality, they do simplify the coding. Table 21-5 lists these auxiliary functions.

Table 21-5

Rectangle-Related Functions

FUNCTION	FIGURE
SetRect()	Fills a RECT structure variable with coordinates
CopyRect()	Copies the data in a RECT structure variable to another one
SetEmptyRect()	Fills a RECT structure variable with zeros thus creating an empty rectangle
OffsetRect()	Translates a rectangle along the x- and y-axes
InflateRect()	Increases or decreases the width and height of a rectangle
IntersectRect()	Creates a rectangle that is the intersection of two other rectangles
UnionRect()	Creates a rectangle that is the union of two other rectangles
SubratctRect()	Creates a rectangle that is the difference between two other rectangles
IsRectEmpty()	Determines if a rectangle is empty
PtInRect()	Determines if a point is located within the perimeter of a rectangle
EqualRect()	Determines if two rectangles are equal

The function SetRect() is used to set the coordinates in a RECT structure. It is equivalent to entering these values into the structure member variables. The function's general form is as follows:

```
BOOL SetRect(
         LPRECT lprc, // 1
         int xLeft,   // 2
         int yTop,    // 3
         int xRight,  // 4
         int yBottom  // 5
         );
```

The first parameter is a pointer to the structure variable that references the rectangle to be set. The second and third parameters are the x and y-coordinates of the upper-left corner. The fourth and fifth parameters are the coordinates of the lower-right corner.

The CopyRect() function is used to copy the parameters from one rectangle structure variable to another one. The function's parameters are the addresses of the destination and source structures. Its general form is as follows:

```
BOOL CopyRect(
         LPRECT lprcDst,       // 1
         CONST RECT *lprcSrc   // 2
         );
```

The first parameter is a pointer to a structure of type RECT that receives the copied coordinates. The second parameter is a pointer to the structure that holds the source coordinates.

The function SetRectEmpty() takes as a parameter the address of a structure variable of type RECT and sets all its values to zero. The result is an empty rectangle that does not show on the screen. Its general form is as follows:

```
BOOL SetEmptyRect (LPRECT rect);
```

The function's only parameter is the address of the RECT structure that is to be cleared.

Notice that there is a difference between an empty rectangle and a NULL rectangle. An empty rectangle is one with no area, that is, one in which the coordinate of the right side is less than or equal to that of the left side, or the coordinate of the bottom side is less than or equal to that of the top side. A NULL rectangle is one in which all the coordinates are zero. The Foundation Class Library contains different member functions for detecting an empty and a NULL rectangle. The Windows API, however, has no function for detecting a NULL rectangle.

OffsetRect() translates a rectangle along both axes. The function's general form is as follows:

```
BOOL OffsetRect(
         LPRECT lprc,  // 1
         int dx,       // 2
         int dy        // 3
         );
```

The first parameter is a pointer to a structure variable of type RECT that contains the parameters of the rectangle to be moved. The second parameter is the amount to move the rectangle along the x-axis. The third parameter is the amount to move the rectangle along the y-axis. Positive values indicate movement to the right or down. Negative values indicate movement to the left or up.

In reality, the OffsetRect() function does not move the rectangle, but simply changes the values in the RECT structure variable referenced in the call. Another call to a rectangle display function is necessary in order to show the translated rectangle on the screen. Figure 21-10 shows the effect of OffsetRect().

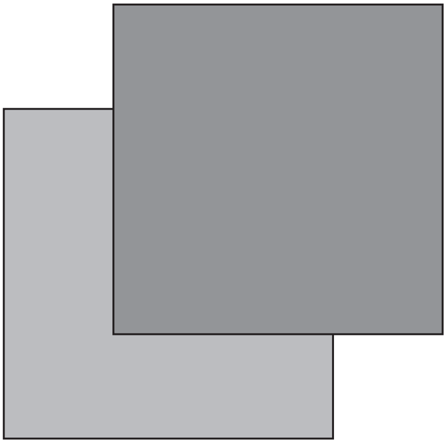


Figure 21-10 *Effect of the OffsetRect() Function*

In Figure 21-10, the light-gray rectangle shows the original figure. The OffsetRect() function was applied to the data in the figure's RECT structure variable, adding 50 pixels along the x-axis and subtracting 50 pixels along the y-axis. The resulting rectangle is shown with a dark-gray fill.

InflateRect() serves to increase or decrease the size of a rectangle. The function's general form is as follows:

```
BOOL InflateRect(
        LPRECT lprc,   // 1
        int dx,        // 2
        int dy         // 3
        );
```

The first parameter is a pointer to a structure variable of type RECT that contains the rectangle to be resized. The second parameter is the amount to add or subtract from the rectangle's width. The third parameter is the amount to add or subtract from the rectangle's height. In both cases, positive values indicate an increase of the dimension and negative values a decrease. The InflateRect() function does not change the displayed rectangle, but modifies the values in the RECT structure variable referenced in the call. Another call to a rectangle display function is necessary in order to show the modified rectangle on the screen. Figure 21-11, on the following page, shows the effect of the InflateRect() function.

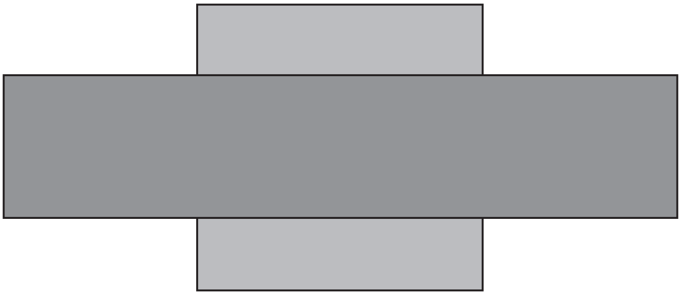


Figure 21-11 *Effect of the InflateRect() Function*

In Figure 21-11, the light-gray rectangle shows the original 150-by-150 pixels figure. The InflateRect() function was applied to increase the width by 100 pixels and decrease the height by 75 pixels. The results are shown in the dark-gray rectangle.

The IntersectRect() function applies a logical AND operation on two rectangles to create a new rectangle that represents the intersection of the two figures. If there are no common points in the source rectangles, then an empty rectangle is produced. The function's general form is as follows:

```
BOOL IntersectRect (LPRECT, CONST LPRECT, CONST LPRECT);
                    ------  ------------  ------------
                      |          |              |
                      1          2              3
```

The first parameter is the address of a RECT structure variable where the intersection coordinates are placed. The second parameter is a pointer to a RECT structure variable that holds the coordinates of the first rectangle. The third parameter is a pointer to a RECT structure variable with the coordinates of the second rectangle. Figure 21-12 shows the effect of the IntersectRect() function.

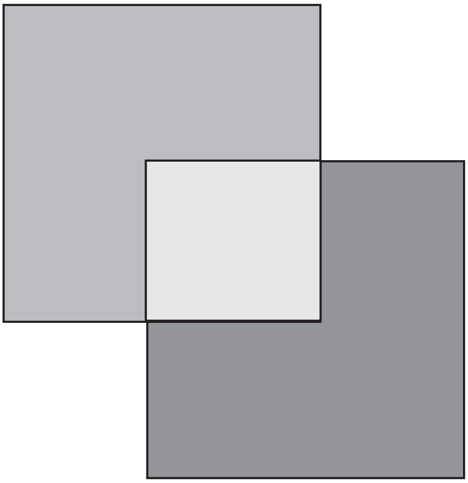


Figure 21-12 *Effect of the IntersectRect() Function*

The UnionRect() function applies a logical OR operation on two rectangles to create a new rectangle that represents the union of the two figures. If there are no common points in the source rectangles, then an empty rectangle is produced. The resulting image is the smallest rectangle that contains both sources. The function's general form is as follows:

```
BOOL UnionRect(
         LPRECT lprcDst,         // 1
         CONST RECT *lprcSrc1,  // 2
         CONST RECT *lprcSrc2   // 3
         );
```

The first parameter is the address of a RECT structure variable where the union rectangle coordinates are placed. The second parameter is a pointer to a RECT structure variable that holds the coordinates of the first rectangle, and the third parameter is a pointer to a RECT structure variable with the coordinates of the second rectangle. Figure 21-13 shows the effect of the UnionRect() function.

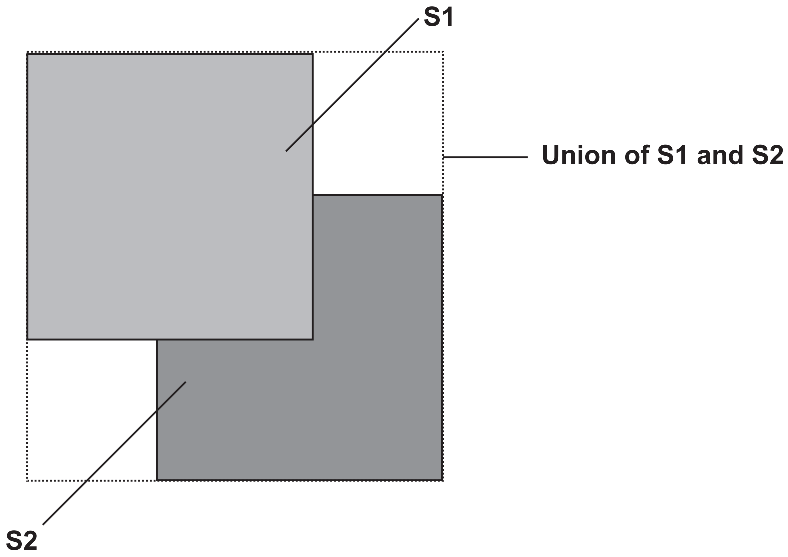


Figure 21-13 *Effect of the UnionRect() Function*

The SubtractRect() function creates a new rectangle by subtracting the coordinates of two source rectangles. The function's general form is as follows:

```
BOOL SubtractRect(
         LPRECT lprcDst,         // 1
         CONST RECT *lprcSrc1,  // 2
         CONST RECT *lprcSrc2   // 3
         );
```

The first parameter is the address of a RECT structure variable where the resulting coordinates are placed. The second parameter is a pointer to a RECT structure variable that holds the coordinates of the first source rectangle. It is from this rectangle that the coordinates of the second source rectangle are subtracted. The third parameter is a pointer to a RECT structure variable with the coordinates of the second source rectangle. The coordinates of this rectangle are subtracted from the ones of the first source rectangle.

The result of the operation must be a rectangle, not a polygon or any other non-rectangular surface. This imposes the restriction that the rectangles must completely overlap in either the vertical or the horizontal direction. If not, the coordinates of the resulting rectangle are the same as those of the first source rectangle. Figure 21-14 shows three possible cases of rectangle subtraction.

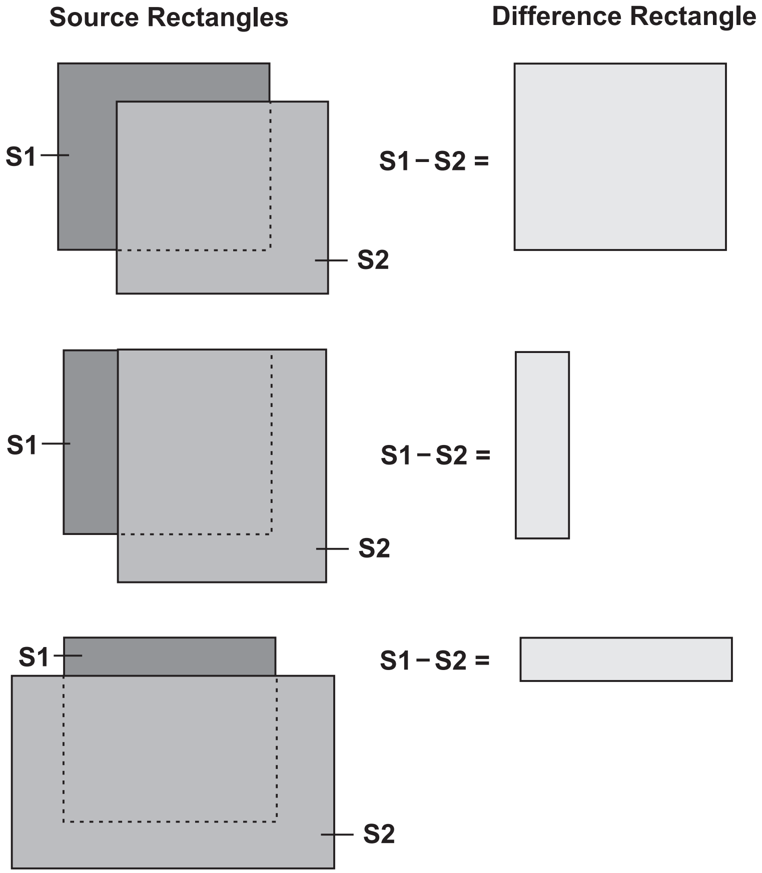


Figure 21-14 *Cases in the SubtractRect() Function*

The IsRectEmpty() function determines whether a rectangle is empty. An empty rectangle is one with no area, that is, one in which the width and/or the height are zero or negative. The function's general form is as follows:

```
BOOL IsRectEmpty(
          CONST RECT *lprc   // 1
          );
```

The only parameter is the address of the RECT structure variable that contains the rectangle's parameters. The function returns TRUE if the rectangle is empty and FALSE otherwise.

The PtInRect() function determines whether a point lies within a rectangle. A point that lies on the rectangle's top or left side is considered to be within the rectangle, but a point within the right or bottom side is not. The function's general form is as follows:

```
BOOL PtInRect(
         CONST RECT *lprc,  // 1
         POINT pt           // 2
         );
```

The first parameter is the address of a RECT structure variable that contains the rectangle's dimensions. The second parameter is a structure of type POINT which holds the coordinates of the point being tested. The function returns TRUE if the point is within the rectangle and FALSE otherwise.

The EqualRect() function determines whether two rectangles are equal. For two rectangles to be equal all their coordinates must be identical. The function's general form is as follows:

```
BOOL EqualRect(
         CONST RECT *lprc1,  // 1
         CONST RECT *lprc2   // 2
         );
```

The first parameter points to a RECT structure variable that contains the parameters of one rectangle. The second parameter points to a RECT structure variable with the parameters of the second rectangle. The function returns TRUE if both rectangles are equal and FALSE otherwise.

21.3.5 Updating the Rectangle() Function

All of the rectangle operations described in the preceding section receive the coordinates in a structure of type RECT. The basic rectangle-drawing function, Rectangle(), receives the figure coordinates as parameters to the call. This difference in data formats, which is due to the evolution of Windows, makes it difficult to transfer the results of a rectangle operation into the Rectangle() function. To solve this problem we have coded a rectangle-drawing function, called DrawRect(), which takes the figure coordinates from a RECT structure. The function is as follows:

```
BOOL DrawRect (HDC hdc, LPRECT aRect) {
  return (Rectangle (hdc, aRect->left,
         aRect->top,
                        aRect->right,
                        aRect->bottom));
}
```

Since DrawRect() uses Rectangle() to draw the figure, the return values are the same for both functions.

21.4 Regions

A region is an area composed of one or more polygons or ellipses. Since a rectangle is a polygon, a region can also be (or contain) a rectangle, or even a rounded rectangle. In Windows programming, regions are used for three main purposes:

- To fill or frame an irregular area
- To clip output to an irregular area
- To test for mouse input in an irregular area

From these uses we can conclude that the main role of a region is to serve as a boundary. Regions can be combined logically, copied, subtracted, and translated to another location. In Windows 95/98 and NT, a new region can be produced by performing a rotation, scaling, reflection, or shearing transformation on another region. Transformations are discussed in Chapter 9. Here, we deal with the simpler operations on regions. There is a rich set of functions that relate to regions and region operations. These are listed in Table 21-6.

Table 21-6

Region-Related GDI Functions

FUNCTION	ACTION
CREATING REGIONS:	
CreateRectRgn()	Creates a rectangular-shaped region, given the four coordinates of the rectangle
CreateRectRgnIndirect()	Creates a rectangular-shaped region, given a RECT structure with the coordinates of the rectangle
CreateRoundRectRgn()	Creates a region shaped like a rounded-corner rectangle, given the coordinates of the rectangle and the dimensions of the corner ellipse
CreateEllipticRgn()	Creates an elliptically shaped region from a bounding rectangle
CreateEllipticRegionIndirect()	Creates an elliptically shaped region from the parameters of a bounding rectangle in a RECT structure
CreatePolygonRgn()	Creates a polygon-shaped region from an array of points that define the polygon
CreatePolyPolygonRgn()	Creates one or more polygon-shaped regions from an array of points that define the polygons
PathToRegion()	Converts the current path into a region
ExtCreateRgn()	Creates a region based on a transformation performed on another regions.
COMBINING REGIONS:	
CombineRgn()	Combines two regions into one by performing a logical, subtraction, or copy Operation
FILLING AND PAINTING REGIONS:	
FillRgn()	Fills a region using a brush

(continues)

Table 21-6

Region-Related GDI Functions (continued)

FUNCTION	ACTION
GetPolyFillMode()	Gets fill mode used by FillRgn()
SetPolyFillMode()	Sets the fill mode for FillRgn()
FrameRgn()	Frames a region using a brush
PaintRgn()	Paints the interior of a region with the brush currently selected in the device context
InvertRgn()	Inverts the colors in a region
REGION STATUS AND CONTROL:	
SetWindowRgn()	Sets the window regions. The window region is the area where the operating system allows drawing operations to take place
GetWindowRgn()	Retrieves the window region established by SetWindowRgn()
OffsetRgn()	Moves a region along the x- or y-axis
SelectClipRgn()	Makes a region the current clipping region
ExtSelectClipRgn()	Combines a region with the current clipping region
GetClipRgn()	Gets handle of the current clipping region
ValidateRgn()	Validates the client area removing the area in the region from the current update region
InvalidateRgn()	Forces a WM_PAINT message by invalidating a screen area defined by a region
OBTAIN REGION DATA:	
PtInRegion()	Tests if a point is located within a region
RectInRegion()	Tests if a given rectangle overlaps any part of a region
EqualRgn()	Tests if two regions are equal
GetRgnBox()	Retrieves a region's bounding box
GetRegionData()	Retrieves internal structure information about a region

In the sections that follow we discuss some of the region-related functions. Other region operations are discussed, in context, later in the book.

21.4.1 Creating Regions

A region is a GDI object, hence, it must be explicitly created. The functions that create a region return a handle of type HRGN (handle to a region). With this handle you can perform many region-based operations, such as filling the region, drawing its outline, and combining it with another region. You often create two or more simple regions by calling their primitive functions, and then combine them into a more complex region, usually by means of the CombineRgn() function.

CreateRectRgn() is used to create a rectangular region. The function's general form is as follows:

```
HRGN CreateRectRgn(
         int nLeftRect,   // 1
         int nTopRect,    // 2
```

```
          int nRightRect,  // 3
          int nBottomRect  // 4
          );
```

The first and second parameters are the coordinates of the upper-left corner of the rectangle. The third and fourth parameters are the coordinates of the lower-right corner.

CreateRectRgnIndirect() creates a rectangular-shaped region, identical to the one produced by CreateRectRgn(); the only difference is that CreateRectRgnIndirect() receives the coordinates in a RECT structure variable. The function's general form is as follows:

```
HRGN CreateRectRgnIndirect(
          CONST RECT *lprc   // 1
               );
```

CreateRoundRectRgn() creates a region shaped like a rounded rectangle. Its general form is as follows:

```
HRGN CreateRoundRectRgn(
          int nLeftRect,      // 1
          int nTopRect,       // 2
          int nRightRect,     // 3
          int nBottomRect,    // 4
          int nWidthEllipse,  // 5
          int nHeightEllipse  // 6
          );
```

The first and second parameters are the coordinates of the upper-left corner of the bounding rectangle. The third and fourth parameters are the coordinates of the lower-right corner. The fifth parameter is the width of the ellipse that is used for drawing the rounded corner arc. The sixth parameter is the height of this ellipse. The shape of the resulting region is the same as that of the rectangle in Figure 21-4.

CreateEllipticRgn() creates an elliptically shaped region. The function's general form is as follows:

```
HRGN CreateEllipticRgn(
          int nLeftRect,   // 1
          int nTopRect,    // 2
          int nRightRect,  // 3
          int nBottomRect  // 4
);
```

The first and second parameters are the coordinates of the upper-left corner of a rectangle that bounds the ellipse. The third and fourth parameters are the coordinates of the lower-right corner of this bounding rectangle. The shape of the resulting region is similar to the one in Figure 21-5.

CreateEllipticRegionIndirect() creates an elliptically shaped region identical to the one produced by CreateEllipticRgn() except that in this case the parameters are read from a RECT structure variable. The function's general form is as follows:

```
HRGN CreateEllipticRgnIndirect(
         CONST RECT *lprc   // 1
);
```

CreatePolygonRgn() creates a polygon-shaped region. The call assumes that the polygon is closed; no automatic closing is provided. The function's general form is as follows:

```
HRGN CreatePolygonRgn(
         CONST POINT *lppt,  // 1
         int cPoints,        // 2
         int fnPolyFillMode  // 3
);
```

The first parameter is the address of an array of points that contains the x- and y-coordinate pairs of the polygon vertices. The second parameter is the count of the number of vertices in the array. The third parameter specifies the polygon fill mode, which can be ALTERNATE or WINDING. ALTERNATE defines a mode that fills between odd-numbered and even-numbered polygon sides, that is, those areas that can be reached from the outside of the polygon by crossing an odd number of lines. WINDING mode fills all internal regions of the polygon. These are the same constants as used in the SetPolyFillMode() function described earlier in this chapter. In the CreatePolygonRgn() function call the fill mode determines which points are included in the region.

CreatePolyPolygonRgn() creates one or more polygon-shaped regions. The call assumes that the polygons are closed figures. No automatic closing is provided. CreatePolyPolygonRgn() is similar to PolyPolygon(). The function's general form is as follows:

```
HRGN CreatePolyPolygonRgn(
         CONST POINT *lppt,        // 1
         CONST INT *lpPolyCounts,  // 2
         int nCount,               // 3
         int fnPolyFillMode        // 4
);
```

The first parameter is a pointer to an array containing vertices of the various polygons. The second parameter is a pointer to an array that contains the number of vertices in each of the polygons. The third parameter is the count of the number of elements in the second parameter, which is the same as the number of polygons to be drawn. The fourth parameter specifies the polygon fill mode, which can be ALTERNATE or WINDING. These two constants have the same effect as described in the CreatePolygonRgn() function.

All the region-creation functions discussed so far return the handle to the region if the call succeeds, and NULL if it fails.

A region can be created from a path by means of the PathToRegion() function. Paths are discussed later in this chapter. The ExtCreateRgn() function allows creating a new region by performing a transformation on another region. Transformations are discussed in Chapter 23.

21.4.2 Combining Regions

Sometimes a region consists of a simple, primitive area such as a rectangle, and ellipse, or a polygon. On other occasions a region is a complex figure, composed of two or more simple figures of the same or different types, which can overlap, be adjacent, or disjoint. The CombineRgn() function is used to create a complex region from two simpler ones. The function's general form is as follows:

```
int CombineRgn(
         HRGN hrgnDest,       // 1
         HRGN hrgnSrc1,       // 2
         HRGN hrgnSrc2,       // 3
         int fnCombineMode    // 4
);
```

The first parameter is the handle to the resulting combined region. The second parameter is the handle to the first source region to be combined. The third parameter is the handle to the second source region to be combined. The fourth parameter is one of five possible combination modes, listed in Table 21-7.

Table 21-7

Region Combination Modes

MODE	EFFECT
RGN_AND	The intersection of the two combined regions
RGN_COPY	A copy of the first source region
RGN_DIFF	Combines the parts of the first source region that are not in the second source region
RGN_OR	The union of two combined regions
RGN_XOR	The union of two combined regions except for any overlapping area

CombineRgn() returns one of four integer values, as shown in Table 21-8.

Table 21-8

Region Type Return Values

VALUE	MEANING
NULLREGION	The region is empty
SIMPLEREGION	The region is a single rectangle
COMPLEXREGION	The region is more complex than a single rectangle
ERROR	No region was created

One property of CombineRgn() is that the destination region, expressed in the first parameter, must exist as a region prior to the call. Creating a memory variable to hold the handle to this region is not sufficient. The region must have been first created by means of one of the region-creation functions, otherwise CombineRgn() returns ERROR. The following code fragment shows the required processing for creating two simple regions and then combining them into a complex region using the RGN_AND combination mode:

```
HRGN      rectRgn, ellipRgn, resultRgn;
.
.
```

```
.
// Create a rectangular region
rectRgn = CreateRectRgn (100, 100, 300, 200);
// Create an elliptical region
ellipRgn = CreateEllipticRgn (200, 100, 400, 200);
// Create a dummy region for results. Skipping this
// step results in an ERROR from the CombineRgn() call
resultRgn = CreateRectRgn (0, 0, 0, 0);
// Combine regions and fill
CombineRgn (resultRgn, rectRgn, ellipRgn, RGN_AND);
FillRgn (hdc, resultRgn, redSolBrush);
```

Figure 21-15 shows the results of applying the various region combination modes on two simple, overlapping regions.

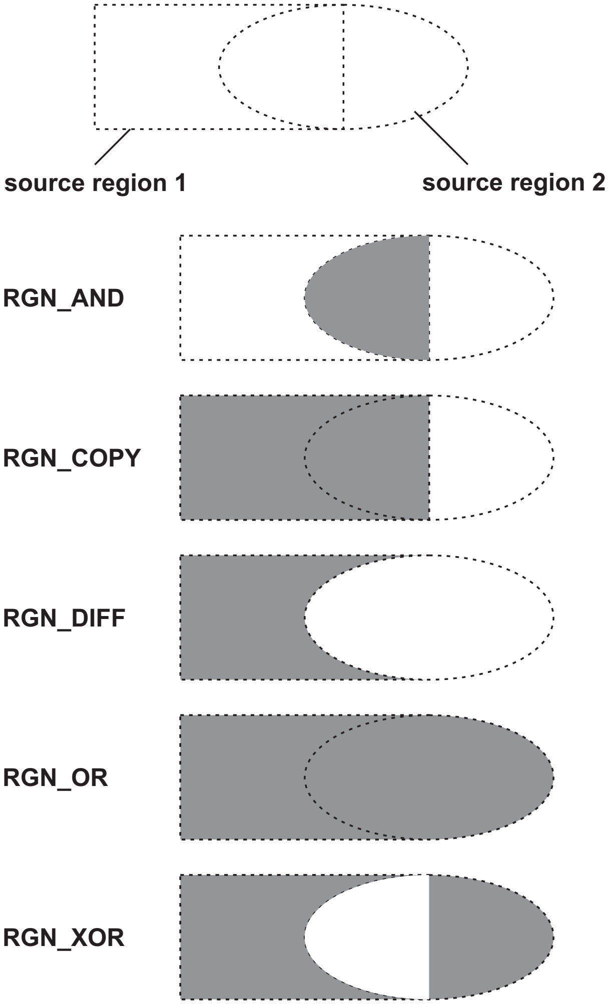


Figure 21-15 *Regions Resulting from CombineRgn() Modes*

In addition to the CombineRgn() function, the windowsx.h header files define several macros that facilitate region combinations. These macros implement the five combination modes that are entered as the last parameter of the CombineRgn() call. They are as follows:

```
CopyRgn      (hrgnDest, hrgnScr1);
IntersectRgn (hrgnDest, hrgnSrc1, hrgnSrc2);
SubtractRgn  (hrgnDest, hrgnScr1, hrgnScr2);
UnionRgn     (hrgnDest, hrgnSrc1, hrgnSrc2);
XorRgn       (hrgnDest, hrgnSrc1, hrgnScr2);
```

In all of the macros, hrgnDest is the handle to the destination region, while hrgnScr1 and hrgnSrc2 are the handles to the source regions.

21.4.3 Filling and Painting Regions

Several functions relate to filling, painting, and framing regions. The difference between filling and painting is that fill operations require a handle to a brush, while paint operations use the brush currently selected in the device context.

The FillRgn() function fills a region using a brush defined by its handle. The function's general form is as follows:

```
BOOL FillRgn(
         HDC hdc,     // 1
         HRGN hrgn,   // 2
         HBRUSH hbr   // 3
         );
```

The first parameter is the handle to the device context. The second one is the handle to the region to be filled. The third parameter is the handle to the brush used in filling the region. The function returns TRUE if it succeeds and FALSE if it fails.

PaintRgn() paints the interior of a region with the brush currently selected in the device context. The function's general form is as follows:

```
BOOL PaintRgn(
         HDC hdc,     // 1
         HRGN hrgn    // 2
         );
```

The second parameter is the handle to the region to be filled. The function returns TRUE if it succeeds and FALSE if it fails.

FrameRgn() draws the perimeter of a region using a brush defined by its handle. The function's general form is as follows:

```
BOOL FrameRgn(
         HDC hdc,      // 1
         HRGN hrgn,    // 2
         HBRUSH hbr,   // 3
         int nWidth,   // 4
         int nHeight   // 5
         );
```

The second parameter is the handle to the region to be filled. The third one is the handle to the brush used in filling the region. The fourth parameter specifies the width of the brush, in logical units. The fifth parameter specifies the height of the brush, also in logical units. The function returns TRUE if it succeeds and FALSE if it fails. If the width and height of the brush are different, then oblique portions of the image are assigned an intermediate thickness. The result is similar to using a calligraphy pen. Figure 21-16 shows a region drawn with the FrameRgn() function.

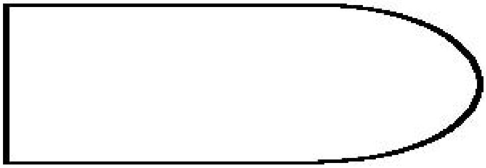


Figure 21-16 *Region Border Drawn with FrameRgn()*

The InvertRgn() function inverts the colors in a region. In a monochrome screen, inversion consists of turning white pixels to black and black pixels to white. In a color screen, inversion depends on the display technology. In general terms, inverting a color produces its complement. Therefore, inverting blue produces yellow, inverting red produces cyan, and inverting green produces magenta. The function's general form is as follows:

```
BOOL InvertRgn(
         HDC hdc,     // 1
         HRGN hrgn    // 2
         );
```

The second parameter is the handle to the region to be inverted. The function returns TRUE if it succeeds and FALSE if it fails.

21.4.4 Region Manipulations

Several functions allow the manipulation of regions. These manipulations include moving a region, using a region to define the program's output area, setting the clipping region, obtaining the clipping region handle, and validating or invalidating a screen area defined by a region. The region manipulations related to clipping are discussed in the following section.

A powerful, but rarely used function in the Windows API is SetWindowRgn(). It allows you to redefine the window area of a window, thus redefining the area where drawing operations take place. In a sense, SetWindowRgn() is a form of clipping that includes not only the client area, but the entire window. The SetWindowRgn() function allows you to create a window that includes only part of the title bar, or to eliminate one or more of the window borders, as well as many other effects. The function's general form is as follows:

```
int SetWindowRgn(
         HWND hWnd,      // 1
```

```
          HRGN hRgn,      // 2
          BOOL bRedraw    // 3
          );
```

The first parameter is the handle to the window whose region is to be changed. The second parameter is the handle to the region that is to be used in redefining the window area. If this parameter is NULL then the window has no window area, therefore becoming invisible. The third parameter is a redraw flag. If set to TRUE, the operating system automatically redraws the window to the new output area. If the window is visible the redraw flag is usually TRUE. The function returns nonzero if it succeeds and zero if it fails.

The function GetWindowRgn() is used to obtain the window area of a window, which usually has been set by SetWindowRgn(). The function's general form is as follows:

```
int GetWindowRgn(
          HWND hWnd, // 1
          HRGN hRgn  // 2
          );
```

The first parameter is the handle to the window whose region is to be obtained. The second parameter is the handle to a region that receives a copy of the window region. The return value is one of the constants listed in Table 21-8.

The OffsetRgn() function is used to move a region to another location. The function's general form is as follows:

```
int OffsetRgn(
          HRGN hrgn,      // 1
          int nXOffset,   // 2
          int nYOffset    // 3
          );
```

The first parameter is the handle to the region that is to be moved. The second parameter is the number of logical units that the region is to be moved along the x-axis. The third parameter is the number of logical units along the y-axis. The function returns one of the constants listed in Table 21-8.

Sometimes the OffsetRgn() function does not perform as expected. It appears that when a region is moved by means of this function, some of the region attributes are not preserved. For example, assume a region that has been filled red is moved to a new location that does not overlap the old position. If we now call InvertRgn() on the translated window, the result is not a cyan-colored window, but one that is the reverse of the background color. In this case the red fill attribute of the original window was lost as it was translated into a new position, and the translated window has no fill. If the translated window partially overlaps the original one, however, then the overlap area's original color is negated when the InvertRgn() function is called on the translated region. Figure 21-17 shows the result of inverting a region translated by means of OffsetRgn().

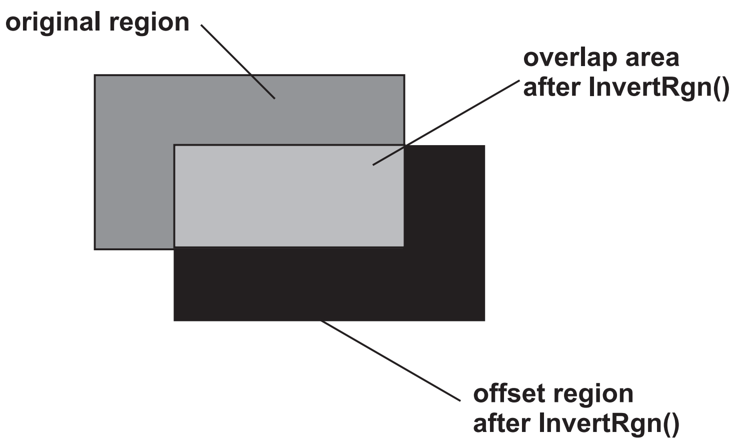


Figure 21-17 *Effect of OffsetRgn() on Region Fill*

Two functions, SelectClipRgn() and ExtSelectClipRgn(), refer to the use of regions in clipping. These functions, along with clipping operations, are discussed later in this chapter.

The InvalidateRgn() function adds the specified region to the current update region of the window. The invalidated region is marked for update when the next WM_PAINT message occurs. The function's general form is as follows:

```
BOOL InvalidateRgn(
         HWND hWnd,     // 1
         HRGN hRgn,     // 2
         BOOL bErase    // 3
         );
```

The first parameter is the handle to the window that is to be updated. The second parameter is the handle to the region to be added to the update area. If this parameter is NULL then the entire client area is added to the update area. The third parameter is an update flag for the background area. If this parameter is TRUE then the background is erased. The function always returns a nonzero value.

The ValidateRgn() function removes the region from the update area. It has the reverse effect as InvalidateRgn(). The function's general form is as follows:

```
BOOL ValidateRgn(
         HWND hWnd,  // 1
         HRGN hRgn   // 2
         );
```

The first parameter is the handle to the window. The second parameter is the handle to the region to be removed from the update area. If this parameter is NULL then the entire client area is removed from the update area. The function returns TRUE if it succeeds and FALSE if it fails.

21.4.5 Obtaining Region Data

A few region-related functions are designed to provide region data to application code. The GetRegionData() function is used mainly in relation to the ExtCreateRegion() function. Both of these functions relate to geometric transformations and are discussed in Chapter 23.

PtInRegion() tests if a point defined by its coordinates is located within a region. The function's general form is as follows:

```
BOOL PtInRegion(
         HRGN hrgn,   // 1
         int X,       // 2
         int Y        // 3
         );
```

The first parameter is the handle to the region to be examined. The second and third parameters are the x- and y-coordinates of the point. If the point is located within the region, the function returns TRUE. If not, the function returns FALSE.

The RectInRegion() function determines if any portion of a given rectangle is within a specified region. The function's general form is as follows:

```
BOOL RectInRegion (
                   HRGN,          // 1
                   CONST RECT *   // 2
                   );
```

The first parameter is the handle to the region to be examined. The second parameter is a pointer to a RECT structure that holds the coordinates of the rectangle. If any part of the specified rectangle lies within the region, the function returns TRUE. If not, the function returns FALSE.

EqualRgn() tests if two regions are identical in size and shape. The function's general form is as follows:

```
BOOL EqualRgn(
         HRGN hSrcRgn1,   // 1
         HRGN hSrcRgn2    // 2
         );
```

The first parameter identifies one of the regions and the second parameter the other one. If the two regions are identical, the function returns TRUE. If not, the function returns FALSE.

The GetRgnBox() function retrieves the bounding rectangle that encloses the specified region. The function's general form is as follows:

```
int GetRgnBox(
         HRGN hrgn,    // 1
         LPRECT lprc   // 2
         );
```

The first parameter is the handle to the region. The second parameter is a pointer to a RECT structure variable that receives the coordinates of the bounding rectangle. The function returns one of the first three constants listed in Table 21-8. If the first parameter does not identify a region then the function returns zero.

21.5 Clipping Operations

One of the fundamental graphics manipulations is clipping. In Windows programming, clipping is associated with regions, since the clip action is defined by a region. In practice, a clipping region is often of rectangular shape, which explains why some clipping operations refer specifically to rectangles. Figure 21-18 shows the results of a clipping operation.

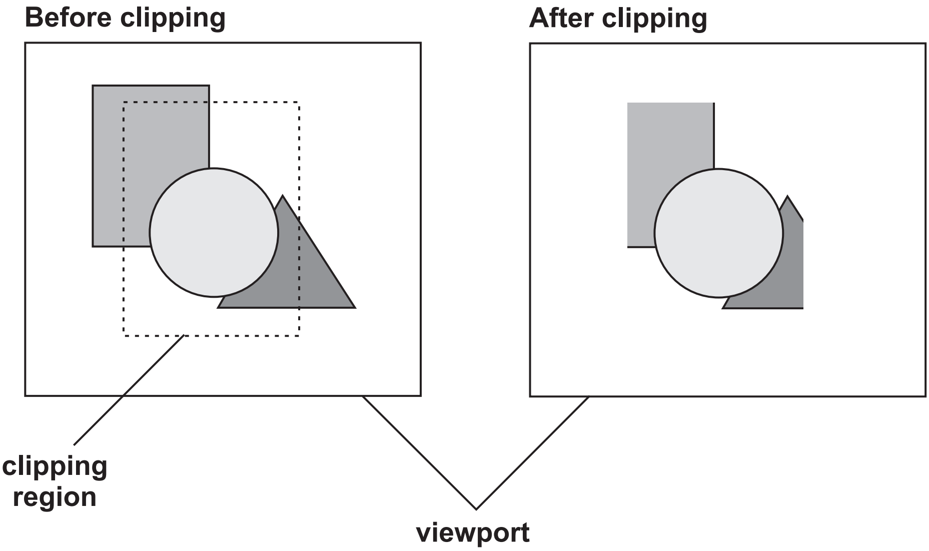


Figure 21-18 *Results of Clipping*

A clipping region is an object of the device context. The default clipping region is the client area. Not all device contexts support a predefined clipping region. If the device context is supplied by a call to BeginPaint(), then Windows creates a default clipping region and assigns to it the area of the window that needs to be repainted. If the device context was created with a call to CreateDC() or GetDC(), as is the case with a private device context, then no default clipping region exists. In this case an application can explicitly create a clipping region. Table 21-9, on the following page, lists the functions that relate to clipping.

Note that Metaregions were introduced in Windows NT and are supported in Windows 95/98. However, very little has been printed about their meaning or possible uses. Microsoft documentation for Visual C++, up to the May prerelease of version 6.0, has nothing on metaregions beyond a brief mentioning of the two related functions listed in Table 21-9. For this reason, it is impossible to determine at this time if a metaregion is a trivial alias for a conventional region, or some other concept not yet documented. Metaregions are not discussed in the text.

Table 21-9

Windows Clipping Functions

FUNCTION	ACTION
CREATING OR MODIFYING A CLIPPING REGION:	
SelectClipRgn()	Makes a region the clipping region for a specified device context
ExtSelectClipRgn()	Combines a specified region with the clipping region according to a predefined mode
IntersectClipRect()	Creates a new clipping region from the interception of a rectangle and the current clipping region in a device context
ExcludeClipRect()	Subtracts a rectangle from the clipping region
OffsetClipRgn()	Moves the clipping region horizontally or vertically
SelectClipPath()	Appends the current path to the clipping region of a device context, according to a predefined mode
OBTAIN CLIPPING REGION INFORMATION:	
GetClipBox()	Retrieves the bounding rectangle for the clipping region
GetClipRgn()	Retrieves the handle of the clipping region for a specified device context
PtVisible()	Determines if a specified point is within the clipping region of a device context
RectVisible()	Determines whether any part of a rectangle lies within the clipping region
METAREGION OPERATIONS:	
GetMetaRgn()	Retrieves the metaregion for the specified device context
SetMetaRgn()	Creates a metaregion, which is the intersection of the current metaregion and the clipping region

21.5.1 Creating or Modifying a Clipping Region

In order for a region to be used to clip output, it must be selected as such in a device context. The SelectClipRgn() function is the primary method of achieving clipping. The region must first be defined and a handle for it obtained. Then the handle to the device context and the handle to the region are used to enforce the clipping. The function's general form is as follows:

```
int SelectClipRgn(
         HDC hdc,     // 1
         HRGN hrgn    // 2
         );
```

The first parameter is the handle to the device context that is to be clipped. The second parameter is the handle to the region used in clipping. This handle is obtained by any of the region-creating calls listed in Table 21-6. The function returns one of the values in Table 21-8.

Once the call is made, all future output is clipped; however, the existing screen display is not automatically changed to reflect the clipping area. Some unexpected or undesirable effects are possible during clipping. Once a clipping region is defined for a device context, then all output is limited to the clipping region. This requires that clipping be handled carefully, usually by installing and restoring clipping regions as necessary.

When a call is made to SelectClipRgn(), Windows preserves a copy of the previous clipping region. The newly installed clipping region can be removed from the device context by means of a call to SelectClipRgn() specifying a NULL region handle.

The ExtSelectClipRgn() function allows combining the current clipping region with a new one, according to one of five predefined modes. The function's general form is as follows:

```
int ExtSelectClipRgn(
         HDC hdc,            // 1
         HRGN hrgn,          // 2
         int fnMode          // 3
);
```

The first parameter is the handle to the device context that is to be clipped. The second parameter is the handle to the region used in clipping. The third parameter is one of the constants listed in Table 21-10.

Table 21-10

Clipping Modes

VALUE	ACTION
RGN_AND	The resulting clipping region combines the overlapping areas of the current clipping region and the one identified in the call, by performing a logical AND between the two regions
RGN_COPY	The resulting clipping region is a copy of the region identified in the call. The result is identical to calling SelectClipRgn(). If the region identified in the call is NULL, the new clipping region is the default clipping region
RGN_DIFF	The resulting clipping region is the difference between the current clipping region and the one identified in the call
RGN_OR	The resulting clipping region is the result of performing a logical OR operation on the current clipping region and the region identified in the call
RGN_XOR	The resulting clipping region is the result of performing a logical XOR operation on the current clipping region and the region identified in the Call

The clipping regions that result from these selection modes are the same as those used in the CombineRgn() function, as shown in Figure 21-15. The ExtSelectClipRgn() function returns one of the values in Table 21-8.

The IntersectClipRect() function creates a new clipping region by performing a logical AND between the current clipping region and a rectangular area defined in the call. The function's general form is as follows:

```
int IntersectClipRect(
         HDC hdc,            // 1
         int nLeftRect,      // 2
         int nTopRect,       // 3
         int nRightRect,     // 4
         int nBottomRect     // 5
         );
```

The second and third parameters are the coordinates of the upper-left corner of the rectangle. The fourth and fifth parameters are the coordinates of the lower-right corner. The function returns one of the values in Table 21-8.

The function ExcludeClipRect() subtracts a rectangle specified in the call from the clipping region. The function's general form is as follows:

```
int ExcludeClipRect(
         HDC hdc,          // 1
         int nLeftRect,    // 2
         int nTopRect,     // 3
         int nRightRect,   // 4
         int nBottomRect   // 5
         );
```

The second and third parameters are the coordinates of the upper-left corner of the rectangle. The fourth and fifth parameters are the coordinates of the lower-right corner. The function returns one of the values in Table 21-8.

The function OffsetClipRgn() translates the clipping region along the horizontal or vertical axes. The function's general form is as follows:

```
int OffsetClipRgn(
         HDC hdc,        // 1
         int nXOffset,   // 2
         int nYOffset    // 3
         );
```

The second parameter is the amount to move the clipping region along the x-axis. The third parameter is the amount to move along the y-axis. Positive values indicate movement to the right or down. Negative values indicate movement to the left or up. The function returns one of the values in Table 21-8.

The SelectClipPath() function appends the current path to the clipping region of a device context, according to a predefined mode. The function's general form is as follows:

```
BOOL SelectClipPath(
         HDC hdc,     // 1
         int iMode    // 2
         );
```

The second parameter is one of the constants listed in Table 21-10. The function returns TRUE if it succeeds and FALSE if it fails. Paths are discussed later in this chapter.

21.5.2 Clipping Region Information

Code that uses clipping often needs to obtain information about the clipping region. Several functions are available for this purpose. The GetClipBox() function retrieves the bounding rectangle for the clipping region. This rectangle is the smallest one that can be drawn around the visible area of the device context. The function's general form is as follows:

```
int GetClipBox(
         HDC hdc,       // 1
         LPRECT lprc    // 2
         );
```

The second parameter is a pointer to a RECT structure that receives the coordinates of the bounding rectangle. The function returns one of the values in Table 21-8.

The GetClipRgn() function retrieves the handle of the clipping region for a specified device context. The function's general form is as follows:

```
int GetClipRgn(
         HDC hdc,            // 1
         HRGN hrgn           // 2
         );
```

The first parameter is the handle to the device context whose clipping region is desired. The second parameter is the handle to an existing clipping region that holds the results of the call. The function returns zero if there is no clipping region in the device context. The return value 1 indicates that there is a clipping region and that the function's second parameter holds its handle. A return value of –1 indicates an error. The function refers to clipping regions that result from SelectClipRgn() of ExtSelectClipRgn() functions. Clipping regions assigned by the system on calls to the BeginPaint() function are not returned by GetClipRgn().

The PtVisible() function is used to determine if a specified point is within the clipping region of a device context. The function's general form is as follows:

```
BOOL PtVisible(
         HDC hdc,  // 1
         int X,    // 2
         int Y     // 3
         );
```

The first parameter is the handle to the device context under consideration. The second and third parameters are the x- and y-coordinates of the point in question. The function returns TRUE if the point is within the clipping region, and FALSE otherwise.

The function RectVisible() is used to determine whether any part of a rectangle lies within the clipping region of a device context. The function's general form is as follows:

```
BOOL RectVisible(
         HDC hdc,            // 1
         CONST RECT *lprc    // 2
         );
```

The first parameter is the handle to the device context under consideration. The second parameter is a pointer to a structure variable of type RECT that holds the coordinates of the rectangle in question. The function returns TRUE if any portion of the rectangle is within the clipping region, and FALSE otherwise.

21.6 Paths

In previous chapters we have discussed paths rather informally. The project folder Text Demo No 3, in the book's software package, contains a program that uses paths to achieve graphics effects in text display. We now consider revisit paths in a more rigorous manner, and apply paths to other graphics operations.

Paths were introduced with Windows NT and are also supported by Windows 95/98. As its name implies, a path is the route the drawing instrument follows in creating a particular figure or set of figures. A path, which is stored internally by the GDI, can serve to define the outline of a graphics object. For example, if we start at coordinates 100, 100, and move to the point at (150, 100), then to (150, 200), from there to (100, 200), and finally to the start point, we have defined the path for a rectangular figure. We can now stroke the path to draw the rectangle's outline, fill the path to produce a solid figure, or both stroke and fill the path to produce a figure with both outline and fill. In general, there are path-related functions to perform the following operations:

- To draw the outline of the path using the current pen.
- To paint the interior of the path using the current brush.
- To draw the outline and paint the interior of a path.
- To modify a path converting curves to line segments.
- To convert the path into a clip path.
- To convert the path into a region.
- To flatten the path by converting each curve in the path into a series of line segments.
- To retrieve the coordinates of the lines and curves that compose a path.

The path is an object of the device context, such as a region, a pen, a brush, or a bitmap. One characteristic of a path is that there is no default path in the device context. Another one is that there is only one path in each device context; this determines that there is no need for a path handle. Every path is initiated by means of the BeginPath() function. This clears any old path from the device context and prepares to record the drawing primitives that create the new path, sometimes called the path bracket. Any of the functions listed in Table 21-11 can be used for defining a path in Windows NT. The subset of functions that can be used in paths in Windows 95/98 is listed in Table 21-12.

Since paths are mostly utilized in clipping operations, the CloseFigure() function is generally used to close an open figure in a path. After all the figures that form the path have been drawn into the path bracket, the application calls EndPath() to select the path into the specified device context. The path can then be made into a clipping region by means of a call to SelectClipPath().

Table 21-11

Path-Defining Functions in Windows NT

AngleArc()	LineTo()	Polyline()
Arc()	MoveToEx()	PolylineTo()
ArcTo()	Pie()	PolyPolygon()
Chord()	PolyBezier()	PolyPolyline()
CloseFigure()	PolyBezierTo()	Rectangle()
Ellipse()	PolyDraw()	RoundRect()
ExtTextOut()	Polygon()	TextOut()

Table 21-12

Path-Defining Functions in Windows 95 and Later

ExtTextOut()	PolyBezierTo()	PolyPolygon()
LineTo()	Polygon()	PolyPolyline()
MoveToEx()	Polyline()	TextOut()
PolyBezier()	PolylineTo()	

Notice that the term clip path, or clipping path, sometimes found in the Windows documentation, can be somewhat confusing. It is better to say that the SelectClipPath() function converts a path to a clipping region, thus eliminating the notion of a clip path as a separate entity.

Table 21-13 lists the paths-related functions.

Table 21-13

Path-Related Functions

FUNCTION	ACTION
PATH CREATION, DELETION, AND CONVERSION:	
BeginPath()	Opens a path bracket
EndPath()	Closes the path bracket and selects the path into the device context
AbortPath()	Closes and discards any open path bracket on the Device context
SelectClipPath()	Makes the current path into a clipping region for a specified device context. Combines the new clipping region with any existing one according to a predefined mode
PathToRegion()	Closes an open path and converts is to a region
PATH RENDERING OPERATIONS:	
StrokePath()	Renders the outline of the current path using the current pen
FillPath()	Closes and opens figure in the current path and fills the path interior with the current brush, using the current polygon fill mode
StrokeAndFillPath()	Renders the outline of the current path using the current pen and fills the interior with the current brush
CloseFigure()	Draws a line from the current pen position to the figure's start point. The closing line is connected to the figure's first line using the current line join style

(continues)

Table 21-13

Path-Related Functions (continued)

FUNCTION	ACTION
PolyDraw()	Draws lines and curves that result from GetPath() (Windows NT only)
PATH MANIPULATIONS:	
FlattenPath()	Converts curves in the current path into line segments
WidenPath()	Redefines the current path in a given device context as the area that would be painted if the path were stroked with the current pen
SetMiterLimit()	Sets the length of the miter joins for the specified device context
OBTAIN PATH INFORMATION:	
GetPath()	Retrieves the coordinates of the endpoints of lines and control points of curves in a path
GetMiterLimit()	Returns the limit for the length of the miter joins in the specified device context
GetPolyFillMode()	Returns the current polygon fill mode

21.6.1 Creating, Deleting, and Converting Paths

A path is initiated by calling the BeginPath() function. The call discards any existing path in the device context and opens a path bracket. The function's general form is as follows:

```
BOOL BeginPath (HDC hdc);
```

The only parameter is the handle to the device context. The function returns TRUE if it succeeds and FALSE if it fails. After the call to BeginPath() is made an application can call any of the functions in Table 21-11 or 21-12, according to the operating system platform.

The EndPath() function closes a path bracket and selects the path into the specified device context. The function's general form is as follows:

```
BOOL EndPath (HDC hdc);
```

The only parameter is the handle to the device context. The function returns TRUE if it succeeds and FALSE if it fails.

The AbortPath() functions closes and discards any open path bracket on the specified device context. The function's general form is as follows:

```
BOOL AbortPath (HDC hdc);
```

The only parameter is the handle to the device context.

A path bracket is created by calling BeginPath(), followed by one or more of the drawing functions listed in Table 21-11 and 21-12, and closed by a call to EndPath(). At this point applications usually proceed to stroke, fill, or stroke-and-fill the path or to install it as a clipping region. Two possible methods can be followed for converting a path into a clipping region. One method is to use PathToRegion() to create a region and then call ExtSelectClipRgn() to make the region a clipping region. Alternatively, code can call SelectClipPath() and perform both functions in a single call.

SelectClipPath() makes the current path into a clipping region for a specified device context, according to a predefined combination mode. The function's general form is as follows:

```
BOOL SelectClipPath(
          HDC hdc,     // 1
          int iMode    // 2
          );
```

The second parameter is one of the values listed in Table 21-7. The function returns TRUE if it succeeds and FALSE if it fails.

The PathToRegion() function closes an open path and converts is to a region. The function's general form is as follows:

```
HRGN PathToRegion (HDC hdc);
```

The function's only parameter is the handle to the device context. The call assumes that the path in the device context is closed. PathToRegion() returns the handle to the created region. Since there are no path handles, this function provides a way of identifying a particular path, although it must be first converted into a region. Unfortunately, there is no method for converting a region into a path.

21.6.2 Path-Rendering Operations

After a path is created it is possible to render it as an image by stroking it, filling it, or both. In Windows NT it is also possible to directly draw line segments and Bezier curves that form a path, whose end and control points are stored in an array of type POINT.

The StrokePath() function renders the outline of the current path using the current pen. The function's general form is as follows:

```
BOOL StrokePath (HDC hdc);
```

The only parameter is the handle to the device context that contains a closed path. Since a device context can only have a single path, there is no need for further specification. The path is automatically discarded from the device context after it is stroked. The function returns TRUE if it succeeds and FALSE if it fails.

Notice that Microsoft Visual C++ documentation does not mention that StrokePath() discards the path automatically. What is worse, the remarks on the StrokeAndFillPath() function suggest that it is possible to first stroke and then fill the same path by making separate calls to the StrokePath() and FillPath() function. In reality, the StrokePath() function destroys the path before exiting execution. A subsequent call to FillPath() has no effect, since there is no longer a path in the device context. This is the reason why the StrokeAndFillPath() function exists. Without this function it would be impossible to both stroke and fill a path.

The FillPath() function closes an open figure in the current path and fills the path interior with the current brush, using the current polygon fill mode. The function's general form is as follows:

```
BOOL FillPath (HDC hdc);
```

The only parameter is the handle to the device context that contains a valid path. Since a device context can only have a single path, there is no need for further specification. The path is automatically discarded from the device context after it is filled. The function returns TRUE if it succeeds and FALSE if it fails.

The StrokeAndFillPath() function closes an open figure in the current path, strokes the path outline using the current pen, and fills the path's interior with the current brush, using the current polygon fill mode. The function's general form is as follows:

```
BOOL StrokeAndFillPath (HDC hdc);
```

The only parameter is the handle to the device context that contains a valid path. The path is automatically discarded from the device context after it is stroked and filled. StrokeAndFillPath() provides the only way for both stroking and filling a path in Windows, since StrokePath() and FillPath() destroy the path after they execute. The function returns TRUE if it succeeds and FALSE if it fails.

The CloseFigure() function draws a line from the current pen position to the figure's start point. The closing line is connected to the figure's first line using the current line join style. The function's general form is as follows:

```
BOOL CloseFigure (HDC hdc);
```

The only parameter is the handle to the device context that contains a valid path. A figure in a path is open unless the CloseFigure() call has been made, even if the figure's starting point and the current point coincide. Usually, the starting point of the figure is the one in the most recent call to MoveToEx().

The effect of closing a figure using the CloseFigure() function is not the same as using a call to a drawing primitive. For example, when the figure is closed with a call to the LineTo() function, end caps are used at the last corner, instead of a join. If the figure is drawn with a thick, geometric pen, the results can be quite different. Figure 21-19 shows the difference between closing a figure by calling LineTo() or by calling CloseFigure().

The triangles in Figure 21-19 are both drawn with a pen style that has a miter join and a round end cap. One of the figures is closed using the LineTo() drawing function and the other one with CloseFigure(). The apex of the triangle closed using the LineTo() function is rounded while the one closed using the CloseFigure() function is mitered. This is due to the fact that two segments drawn with LineTo() do not have a join at a common end point. In this case, the appearance of the apex is determined by the figure's round end cap. On the other hand, when the figure is closed with the CloseFigure() function, the selected join is used in all three vertices.

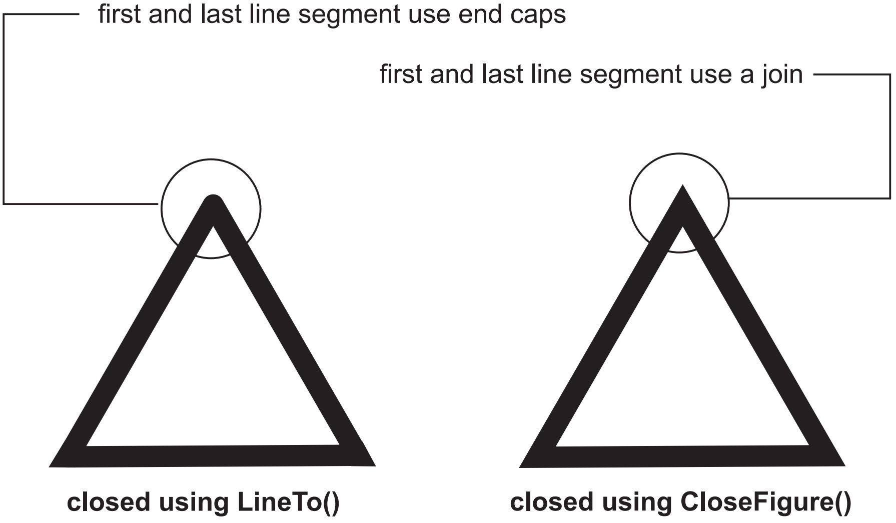

first and last line segment use end caps
first and last line segment use a join
closed using LineTo()
closed using CloseFigure()

Figure 21-19 *Figure Closing Differences*

The PolyDraw() function, available only in Windows NT, draws lines segments and Bezier curves. Because of its limited portability we do not discuss it here.

21.6.3 Path Manipulations

Several functions allow modifying existing paths or determining the path characteristics. FlattenPath() converts curves in the current path into line segments. The function's general form is as follows:

```
BOOL FlattenPath (HDC hdc);
```

The only parameter is the handle to the device context that contains a valid path. The function returns TRUE if it succeeds and FALSE if it fails. There are few documented uses for the FlattenPath() function. The screen appearance of a flattened path is virtually undetectable. The documented application of this function is to fit text on a curve. Once a curved path has been flattened, a call to GetPath() retrieves the series of line segments that replaced the curves of the original path. Code can now use this information to fit the individual characters along the line segments.

The WidenPath() function redefines the current path in a given device context as the area that would be painted if the path were stroked with the current pen. The function's general form is as follows:

```
BOOL WidenPath (HDC hdc);
```

The only parameter is the handle to the device context that contains a valid path. Any Bezier curves in the path are converted to straight lines. The function makes a difference when the current pen is a geometric pen or when it has a width or more than one device unit. WidenPath() returns TRUE if it succeeds and FALSE if it fails. This is another function with few documented uses. The fact that curves are converted into line segments suggests that it can be used in text fitting operations, such as the one described for the FlattenPath() function.

The SetMiterLimit() function sets the length of the miter joins for the specified device context. The function's general form is as follows:

```
BOOL SetMiterLimit(
          HDC hdc,           // 1
          FLOAT eNewLimit,   // 2
          PFLOAT peOldLimit  // 3
          );
```

The first parameter is the handle to the device context. The second parameter specifies the new miter limit. The third parameter is a pointer to a floating-point variable that holds the previous miter limit. If this parameter is NULL the value of the previous miter limit is not returned. The function returns TRUE if it succeeds and FALSE if it fails.

The miter length is the distance from the intersection of the line walls on the inside of the join to the intersection of the line walls on the outside of the join. The miter limit is the ratio between the miter length, to the line width. Figure 21-20 shows the miter length, the line width, and the miter limit.

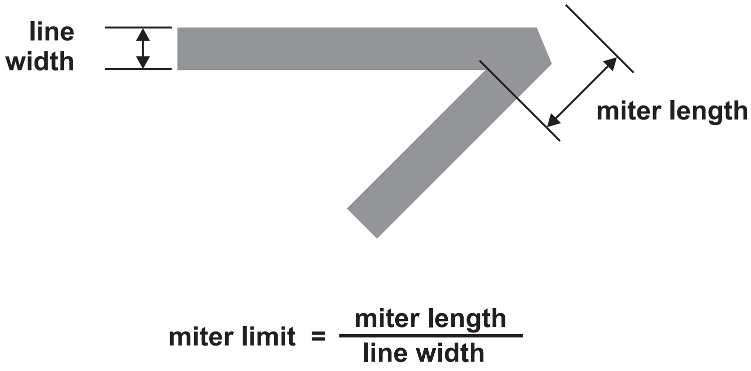


Figure 21-20 *Miter Length, Line Width, and Miter Limit*

The miter limit determines if the vertex of a join that was defined with the PS_JOIN_MITER style (see Figure 20-3 in the previous chapter) is drawn using a miter or a bevel join. If the miter limit is not exceeded, then the join is mitered. Otherwise, it is beveled. Mitered and beveled joins apply only to pens created with the ExtCreatePen() function and to stroked paths. The following code fragment shows the creation of two joins.

```
  static HPEN    fatPen;    // Handle for pen
  static FLOAT   oldMiter;  // Storage for miter limit
  .
  .
  .
  // Create a special pen
  fatPen = ExtCreatePen (PS_GEOMETRIC | PS_SOLID |
           PS_ENDCAP_ROUND | PS_JOIN_MITER,
           15,
           &fatBrush, 0, NULL);
  SelectBrush (hdc, GetStockObject (LTGRAY_BRUSH));
  SelectPen (hdc, fatPen);
  // Draw first angle
  BeginPath (hdc);
  MoveToEx (hdc, 100, 100, NULL);
  LineTo (hdc, 250, 100);
  LineTo (hdc, 150, 180);
  EndPath (hdc);
  StrokePath (hdc);
  // Draw second angle
  GetMiterLimit (hdc, &oldMiter);
  SetMiterLimit (hdc, 2, &oldMiter);
  BeginPath (hdc);
  MoveToEx (hdc, 300, 100, NULL);
  LineTo (hdc, 450, 100);
  LineTo (hdc, 350, 180);
  EndPath (hdc);
  StrokePath (hdc);
  SetMiterLimit (hdc, oldMiter, NULL);
```

Figure 21-21 is a screen snapshot of the execution of the preceding code fragment. Notice in Figure 21-21 that the image on the left, in which the default miter limit of 10 is used, is drawn with a miter join. In the right-hand figure the miter limit was changed to 1, therefore, the figure is drawn using a bevel join.

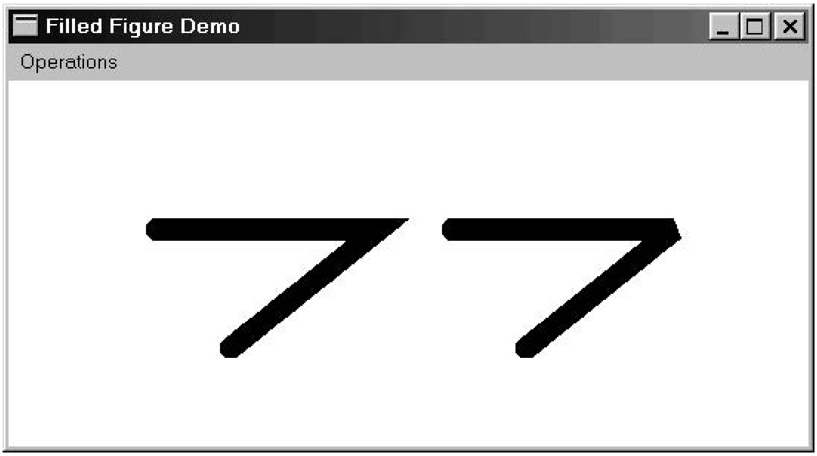


Figure 21-21 *Effect of the SetMiterLimit() Function*

21.6.4 Obtaining Path Information

Several functions provide information about the path, or about GDI parameters that affect the path. The GetPath() function retrieves the coordinates of the endpoints of lines and control points of curves in a path. The function's structure is quite similar to the PolyDraw() function discussed in Chapter 20. GetPath() is related to the PolyDraw() function mentioned earlier. The function's general form is as follows:

```
int GetPath(
         HDC hdc,            // 1
         LPPOINT lpPoints,   // 2
         LPBYTE lpTypes,     // 3
         int nSize           // 4
         );
```

The first parameter identifies the device context. The second parameter is a pointer to an array of POINT structures that contains the endpoints of the lines and the control points of the curves that form the path. The third parameter is an array of type BYTE which contains identifiers that define the purpose of each of the points in the array. The fourth parameter is the count of the number of points in the array of points. The function returns TRUE if it succeeds and FALSE otherwise. Table 21-14 lists the constants used to represent the identifiers entered in the function's third parameter.

Table 21-14

Constants for the GetPath() Vertex Types

TYPE	MEANING
PT_MOVETO	This point starts a disjoint figure. The point becomes the new current pen position.
PT_LINETO	A line is to be drawn from the current position to this point, which then becomes the new current pen position.
PT_BEZIERTO	This is a control point or end node for a Bezier curve. This constant always occurs in sets of three.The current position defines the start node for The Bezier curve. The other two coordinates are control points. The third entry (if coded) is the end node.
PT_CLOSEFIGURE	The figure is automatically closed after the PT_LINETO or PT_BEZIERTO type for this point is executed. A line is drawn from the end point to the most recent PT_MOVETO or MoveTo() point. The PT_CLOSEFIGURE constant is combined by means of a bitwise OR operator with a PT_LINETO or PT_BEZIERTO constant. This indicates that the corresponding point is the last one in a figure and that the figure is to be closed.

The GetMiterLimit() function, which was mentioned in regards to SetMiterLimit(), returns the limit for the length of the miter join in the specified device context. The function's general form is as follows:

```
BOOL GetMiterLimit(
         HDC hdc,          // 1
         PFLOAT peLimit    // 2
         );
```

The first parameter is the handle to the device context. The second parameter stores the current miter limit. The function returns TRUE if it succeeds and FALSE if it fails.

The GetPolyFillMode() returns the current polygon fill mode. The only parameter is the handle to the device context. The value returned is either ALTERNATE or WINDING. The fill more affects the operation of the FillPath() and StrokeAndFillPath() functions.

21.7 Filled Figures Demo Program

The program named FIL_DEMO, located in the Filled Figure Demo project folder of the book's software package, is a demonstration of the graphics functions and operations discussed in this chapter. The first entry in the Operations menu shows the offset of the hatch origin to visually improve a filled rectangle. The main menu contains several pop-up menus that demonstrate most of the graphics primitives discussed in the text. These include drawing solid figures, operations on rectangles, regions, clipping, and paths. Another menu entry demonstrates the use of the SetMiterLimit() function. Many of the illustrations used in this chapter were taken from the images displayed by the demonstration program.

Chapter 22

Windows Bitmapped Graphics

Topics:

- Raster and vector graphics on the PC
- Windows bitmap formats and structures
- Bitmap programming and the bitblt operation
- Manipulating and transforming bitmaps

This chapter is about bitmaps. A bitmap is a digitized image in which each dot is represented by a numeric value. Bitmap images are used in graphics programming at least as frequently as vector representation. The high resolution and extensive color range of current video display systems allow encoding bitmapped images with photo-realistic accuracy. The powerful storage and processing capabilities of the modern day PC make possible for software to rapidly and effectively manipulate and transform bitmaps. Computer simulations, virtual reality, artificial life, and electronic games are fields of application that rely heavily on bitmap operations.

22.1 Raster and Vector Graphics

The two possible ways of representing images in a computer screen, or a digital graphics device, are based on vector and raster graphics technologies. All of the graphics primitives discussed in Chapters 6 and 7 are based on vector techniques. Commercially speaking, vector graphics are associated with drawing programs, while raster graphics are associated with painting programs. The vector representation of a line consists of its start and end points and its attributes, which usually include width, color, and type. The raster representation of the same line is a mapping of adjacent screen dots. Most current computer systems are raster based, that is, the screen is a two-dimensional pixel grid and all graphics objects are composed of individual screen dots, as described in Chapter 1. Vector graphics are a way of logically defining images, but the images must be rasterized at display time.

Vector and raster representations have their advantages and drawbacks. Vector images can be transformed mathematically (as you will see in Chapter 9); they can also be scaled without loss of quality. Furthermore, vector images are usually more compact. Many images can be conveniently represented in vector form, such as an engineering drawing composed of geometrical elements that can be mathematically defined. The same applies to illustrations, and even to artwork created by combining geometrical elements.

22.1.1 The Bitmap

An image of Leonardo's Mona Lisa, or a photograph of the Crab nebulae, can hardly be vectorized. When geometrical elements are not present, or when the image is rich in minute details, vector representations cease to be practical. In these cases it is better to encode the image as a data structure containing all the individual picture elements. This pixel-by-bit encoding is called a bitmap.

A bitmap is a form of raster image. A raster image can be defined as pixel-by-pixel enumeration, usually in scan-line order. A bitmap is a formatted raster image encoded according to some predefined standard or convention. A raster image, on the other hand, can be in raw format. For example, a scanning instrument onboard a satellite or space craft acquires and transmits image data in raster form. Once received, the raster data can be processed and stored as bitmaps that can be easily displayed on a computer screen. Television images are in raster form.

A bitmap is a memory object, not a screen image. It is the memory encoding of an image at the pixel level. Although bitmaps are often represented in image form, it is important to remember that a bitmap is a data construct. Bitmaps cannot be easily transformed mathematically, as is the case with vector images, nor can they be scaled without some loss of information. However, bitmaps offer a more faithful reproduction of small details than is practical in vector representations.

In bit-mapping, one or more memory bits are used to represent the attribute of a screen pixel. The simplest and most compact scheme is that in which a memory bit represents a single screen pixel: if the bit is set, so is the pixel. This one-bit-to-one-pixel representation leaves no choice about pixel attributes, that is, a pixel is either set or not. In a monochrome video system a set bit can correspond to a bright pixel and a reset bit to a black one. Figure 22-1 shows a bit-to-pixel image and bitmap.

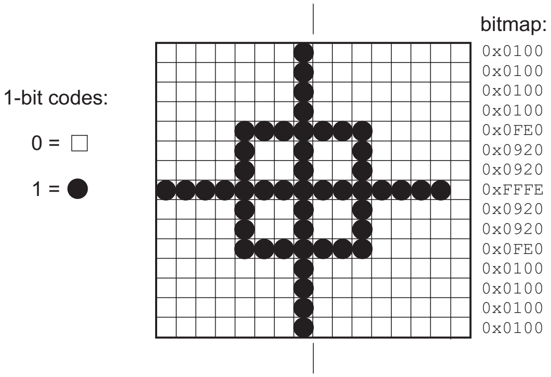
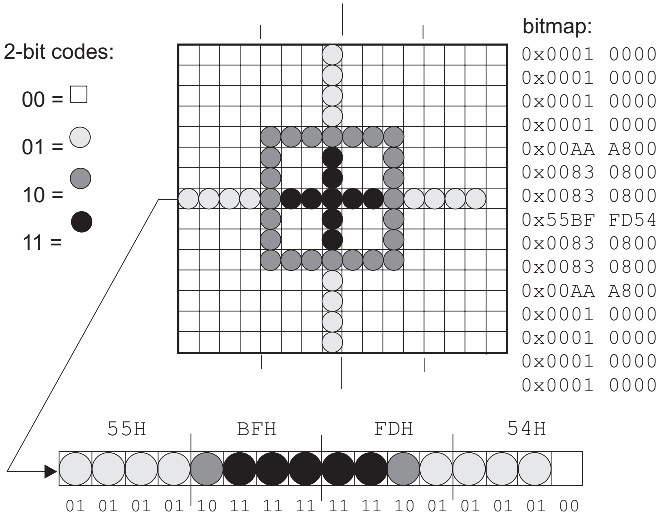


Figure 22-1 *One Bit Per Pixel Image and Bitmap*

Most current video systems support multiple attributes per screen pixel. PC color video systems are usually capable of representing 16, 256, 65,535, and 16.7 million colors. Shades of gray can also be encoded in a bitmap. The number of bits devoted to each pixel determines the attribute range. Sixteen colors or shades of gray can be represented in four bits. Eight bits can represent 256 colors. Sixteen bits encode 65,535 colors. So-called true color systems, which can display approximately 16.7 million colors, require 24 data bits per screen dot. Some color data formats use 32-bits per pixel, but this representation is actually 24 bits of pixel data plus 8 bits of padding. Figure 22-2 shows an image and bitmap in which each pixel is represented by two memory bits.



bitmap:

```
0x0001 0000
0x0001 0000
0x0001 0000
0x0001 0000
0x00AA A800
0x0083 0800
0x0083 0800
0x55BF FD54
0x0083 0800
0x0083 0800
0x00AA A800
0x0001 0000
0x0001 0000
0x0001 0000
0x0001 0000
```

Figure 22-2 *Two Bits Per Pixel Image and Bitmap*

In Figure 22-2, each screen pixel can be in one of four attributes: background, light gray, dark gray, or black. In order to represent these four states it is necessary to assign a 2-bit field for each screen pixel. The four bit combinations that correspond with the attribute options are shown on the left side of Figure 22-2. At the bottom of the illustration is a map of one of the pixel rows, with the corresponding binary codes for each pixel, as well as the hexadecimal digits of the bitmap.

22.1.2 Image Processing

The fact that raster images cannot be transformed mathematically does not mean that they cannot be manipulated and processed. Image Processing is a major field of computer graphics. It deals exclusively with the manipulation and analysis of two-dimensional pictorial data in digital form. In practice, this means processing raster data. Although digital image processing originated in the science programs of the National Aeronautics and Space Administration (NASA) it has since been applied to many other technological fields, including biological research, document processing, factory automation, forensics, medical diagnostics, photography, prepress publishing operations, space exploration, and special effects on film and video.

22.1.3 Bitblt Operations

The fundamental bitmap operation is the bit block transfer, or bitblt (pronounced bit-blit). In the bitblt a rectangular block of memory bits, representing pixel attributes, is transferred as a block. If the destination of the transfer is screen memory, then the bitmapped image is displayed. At the time of the transfer, the source and destination bit blocks can be combined logically or arithmetically, or a unary operation can be performed on the source or the destination bit blocks. Figure 22-3 shows several binary and unary operations on bit blocks.

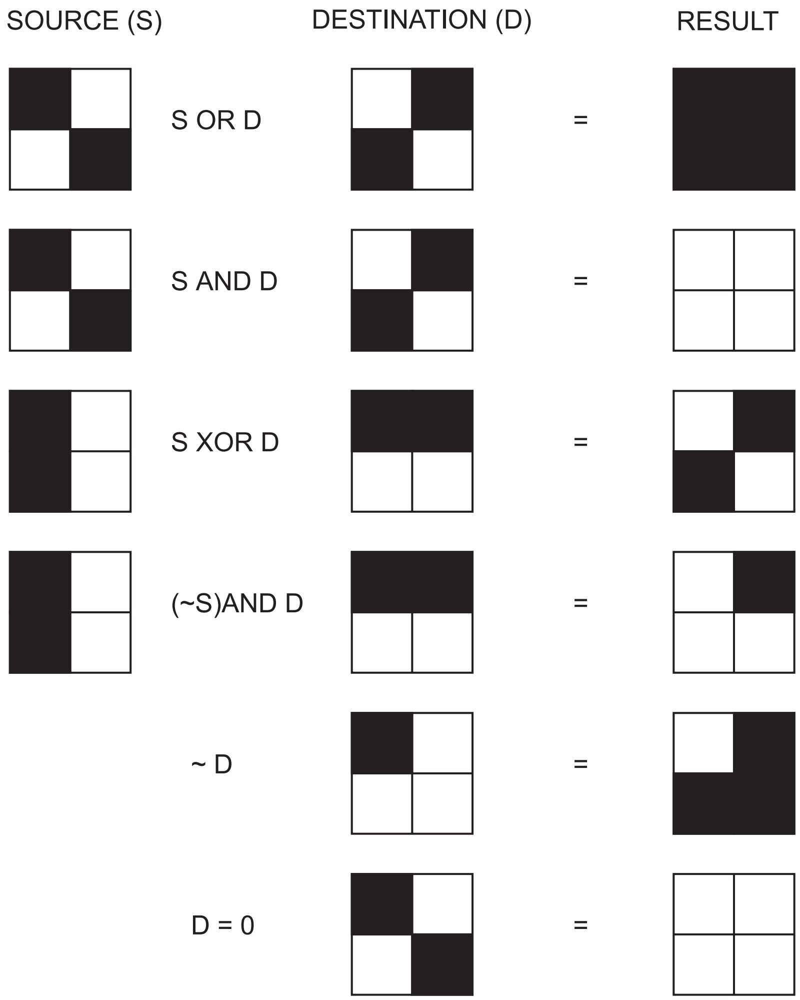


Figure 22-3 *Binary and Unary Operations on Bit Blocks*

22.2 Bitmap Constructs

Several Windows data structures and concepts relate to bitmaps:

- Bitmap formats
- Structures for bitmap operations
- Creating bitmap resources

You must understand these concepts in order to be able to manipulate and display bitmaps.

22.2.1 Windows Bitmap Formats

The computer establishment has created hundreds of bitmap image formats over the past two or three decades; among the best known are GIF, PCX, Targa, TIFF, and Jpeg. Windows does not provide support for manipulating image files in any commercial format. The only bitmap formats that can be handled are those specifically created for Windows. These are the original device-specific bitmaps (extension BMP) and the newer device-independent bitmaps (extension DIB).

The device-specific bitmap format was created in Windows 3.0. It can store images with up to 24-bit color. Files in BMP format are uncompressed, which makes them quick to read and display. The disadvantage of uncompressed files is that they take up considerable memory and storage space. Another objection to using the device-specific bitmap is that the BMP header cannot directly encode the original colors in the bitmap. For this reason, the format works well for copying parts of the screen to memory and pasting them back to other locations. But problems often occur when the device-specific bitmap must be saved in a disk file and then output to a different device, since the destination device may not support the same colors as the original one. For this reason it is usually preferable to use the device-independent format, although its data structures are slightly larger.

Note that many Windows bitmap functions operate both on device-specific and device-independent bitmaps. The Bitmap Demo project on the book's software package displays bitmaps in BMP and DIB format using the same Windows functions. In the remainder of this chapter we deal exclusively with device-independent bitmaps encoded in DIB format, although device independent bitmaps can also be stored in .BMP files.

22.2.2 Windows Bitmap Structures

There are eleven structures directly or indirectly related to Windows bitmaps, listed in Table 22-1, on the following page.

22.2.3 The Bitmap as a Resource

In Visual C++ a bitmap can be a program resource. Developer Studio includes a bitmap editor that can be used for creating and editing bitmaps that do not exceed a certain size and color range. Bitmaps in BMP or DIB format created with other applications can also be imported into a program. In this case you can select the Resource command in the Insert menu, and then select the bitmap resource type and click the Import button. The dialog that appears allows you to browse through the file system in order to locate the bitmap file. It is usually better to copy the bitmap file into the project's folder, thus ensuring that it is not deleted or modified inadvertently.

Table 22-1

Bitmap-Related Structures

STRUCTURE	CONTENTS
BITMAP	Defines the width, height, type, format, and bit values of a bitmap.
BITMAPCOREHEADER	Contains information about the dimensions and color format of a DIB.
BITMAPCOREINFO	Contains a BITMAPCOREHEADER structure and a RGBTRIPLE structure array with the bitmap's color intensities.
BITMAPFILEHEADER	Contains information about a file that holds a DIB.
BITMAPINFO	Defines the dimensions and color data of a DIB. Includes a BITMAPINFOHEADER structure and a RGBQUAD structure.
BITMAPINFOHEADER	Contains information about the dimensions and color format of a DIB.
COLORADJUSTMENT	Defines the color adjustment values used by the StrtchBlt() and StretchDIBits() functions.
DIBSECTION	Contains information about a bitmap created by means of the CreateDIBSection() function.
RGBQUAD	Describes the relative intensities of the red, green, and blue color components. The fourth byte of the structure is reserved.
RGBTRIPLE	Describes the relative intensities of the red, green, and blue color components. The BITMAPCOREINFO structure contains an array of RGBTRIPLE structures.
SIZE	Defines the width and height of a rectangle, which can be used to represent the dimensions of a bitmap.

Some of the structures in Table 22-1 are listed and described in this chapter.

Once a bitmap has been imported in the project, it is listed in the Resource tab of the Project Workspace pane. At this point it is possible to use the MAKEINTRESOURCE macro to convert it into a resource type that can be managed by the application. Since the final objective is to obtain the handle to a bitmap, code usually proceeds as follows:

```
HBITMAP       aBitmap;
.
.
.
aBitmap = LoadBitmap (pInstance,
                      MAKEINTRESOURCE (IDB_BITMAP1));
```

In this case, IDB_BITMAP1 is the resource name assigned by Developer Studio when the bitmap was imported. It is listed in the Bitmap section of the Resource tab in the Project Workspace pane and also by the Resource Symbols command in the View menu.

22.3 Bitmap Programming Fundamentals

The bitmap is an object of the device context, such as a brush, a pen, a font, or a region. Bitmap manipulations and display functions in Windows are powerful, but not without some limitations and complications. This overview presents a preliminary discussion of simple bitmap programming. Later in this chapter we get into more complicated operations.

22.3.1 Creating the Memory DC

The unique characteristic of a bitmap is that it can be selected only into a memory device context. The memory DC is defined as a device context with a display surface. It exists only in memory and is related to a particular device context. In order to use a memory device context you must first create it. The CreateCompatibleDC() function is used for this purpose. Its general form is as follows:

```
HDC CreateCompatibleDC (HDC hdc);
```

Its only parameter is the handle to the device context with which the memory device context is to be compatible. This parameter can be NULL if the memory device context is to be compatible with the video screen. If the function succeeds, it returns the handle to the memory device context. The call returns NULL if it fails. CreateCompatibleDC() assumes that the device context supports raster operations, which is true in all PC video systems, but not necessarily so for other graphics devices, such as a plotter. The GetDeviceCaps() function with the RASTERCAPS constant can be used to determine if a particular device supports raster operations (see Table 20-4).

Like the device context, a memory device context has a mapping mode attribute. Applications often use the same mapping mode for the memory device context and for the device context. In this case the SetMapMode() and GetMapMode() functions can be combined as follows:

```
HDC         hdc;        // Handle to device context
HDC         memDC;      // Handle to memory device context
.
.
.
SetMapMode (memDC, GetMapMode (hdc));
```

22.3.2 Selecting the Bitmap

When the memory device context is created, it is assigned a display surface of a single monochrome pixel. The single pixel acts as a placeholder until a real one is selected into the memory DC. The SelectObject() function, discussed in Chapter 6, can be used to select a bitmap into a memory device context, but the SelectBitmap() macro, discussed in Chapter 7, serves the same purpose. Both, SelectObject() and SelectBitmap() have the same interface. For SelectBitmap() it is as follows:

```
HBITMAP SelectBitmap (HDC, HBITMAP);
                       |    -------
                       |       |
                       1       2
```

The first parameter must be the handle of a memory device context. The second parameter is the handle to the bitmap being installed. If the call succeeds, the macro returns the handle to the device context object being replaced. If the call fails, it returns NULL. Using the SelectBitmap() macro instead of the SelectObject() function produces code that is correct and the coding is made easier. Recall that programs that use the object selection macros must include the windowsx.h file.

The handle to the bitmap used in the SelectBitmap() macro is usually obtained with the LoadBitmap() function previously discussed.

22.3.3 Obtaining Bitmap Dimensions

Bitmap functions often require information about the dimensions and other characteristics of the bitmap. For example, the function most often used to display a bitmap is BitBlt(); it requires the width and height of the bitmap. If the bitmap is loaded as a resource from a file, the application must obtain the bitmap dimensions before blitting it to the screen. The GetObject() function is used to obtain information about a bitmap. The function's general form is as follows:

```
int GetObject(
          HGDIOBJ hgdiobj,  // 1
          int cbBuffer,     // 2
          LPVOID lpvObject  // 3
);
```

The first parameter is the handle to a graphics object; in this case, the handle to a bitmap. The second parameter is the size of the buffer that holds the information returned by the call. In the case of a bitmap, this parameter can be coded as sizeof (BITMAP). The third parameter is a pointer to the buffer that holds the information returned by the call. In the case of a bitmap, the buffer is a structure variable of type BITMAP. The BITMAP structure is defined as follows:

```
typedef struct tagBITMAP {
   LONG   bmType;       // Must be zero
   LONG   bmWidth;      // bitmap width (in pixels)
   LONG   bmHeight;     // bitmap height  (in pixels)
   LONG   bmWidthBytes; // bytes per scan line
   WORD   bmPlanes;     // number of color planes
   WORD   bmBitsPixel;  // bits per pixel color
   LPVOID bmBits;       // points to bitmap values array
} BITMAP;
```

The structure member bmType specifies the bitmap type. It must be zero. The member bmWidth specifies the width, in pixels, of the bitmap. Its value must be greater than zero. The member bmHeight specifies the height, in pixels, of the bitmap. The height must be greater than zero. The bmWidthBytes member specifies the number of bytes in each scan line. Since Windows assumes that the bitmap is word-aligned, its value must be divisible by 2. The member bmPlanes specifies the number of color planes. The member bmBitsPixel specifies the number of bits required to indicate the color of a pixel. The member bmBits points to the location of the bit values for the bitmap. It is a long pointer to an array of character-size values.

When the target of the GetObject() call is a bitmap, the information returned is the structure members related to the bitmap width, height, and color format. The GetObject() function cannot be used to read the value of the pointer to the bitmap data in the bmBits structure member. GetDIBits() retrieves the bit data of a bitmap.

The mapping mode can also be a factor in regard to bitmap data. The GetObject() function returns the bitmap width and height in the BITMAP structure. The values returned are in pixels, which are device units. This works well if the mapping mode of the memory device context is MM_TEXT, but is not acceptable in any of the mapping modes that use logical units. The DPtoLP() function allows the conversion of device coordinates (pixels) into logical coordinates for a particular device context. The function's general form is as follows:

```
BOOL DPtoLP(
         HDC hdc,           // 1
         LPPOINT lpPoints,  // 2
         int nCount         // 3
);
```

The first parameter identifies the device context. In the case of a bitmap, it is a memory device context. The second parameter points to an array of POINT structures that holds the transformed values for the x- and y-coordinates. The third parameter holds the number of points in the array specified in the second parameter. The function returns TRUE if it succeeds and FALSE if it fails.

A bitmap size is defined by two values: the x-coordinate is the width and the y-coordinate the height. When a call to DPtoLP() is made to obtain the bitmap size, the third parameter is set to 1. This indicates that the coordinates to be transformed refer to a single point. By the same token, the second parameter is a pointer to a single POINT structure that holds the bitmap width in the x member and the bitmap height in the y member.

22.3.4 Blitting the Bitmap

Once a bitmap has been selected onto a memory device context, and the code has obtained the necessary information about its width and height, it is possible to display it at any screen position by blitting the memory stored bitmap onto the screen. The BitBlt() function is the simplest and most direct method of performing the bitmap display operation. The function's general form is as follows:

```
BOOL BitBlt(
         HDC hdcDest, // 1
         int nXDest,  // 2
         int nYDest,  // 3
         int nWidth,  // 4
         int nHeight, // 5
         HDC hdcSrc,  // 6
         int nXSrc,   // 7
         int nYSrc,   // 8
         DWORD dwRop  // 9
);
```

The first parameter identifies the destination device context. If the call to BitBlt() is made to display a bitmap, this will be the display context. The second and third parameters are the x- and y-coordinates of the upper-left corner of the destination rectangle. Which is also the screen location where the upper-left corner of the bitmap is displayed. The fourth and fifth parameters are the width and height of the bitmap, in logical units. The sixth parameter is the source device context. In the case of a bitmap display operation, this parameter holds the memory device context where the bitmap is stored. The seventh and eighth parameters are the x- and y-coordinates of the source bitmap. Since the blitted rectangles must be of the same dimensions, as defined by the fourth and fifth parameters, the seventh and eighth parameters are usually set to zero.

The ninth parameter defines the raster operation code. These codes are called ternary raster operations. They differ from the binary raster operation codes (ROP2) discussed in Chapter 6 in that the ternary codes take into account the source, the destination, and a pattern determined by the brush currently selected in the device context. There are 256 possible raster operations, fifteen of which have symbolic names defined in the windows.h header file. The raster operation code determines how the color data of the source and destination rectangles, together with the current brush, are to be combined. Table 22-2 lists the fifteen raster operations with symbolic names.

Table 22-2

Symbolic Names for Raster Operations

NAME	DESCRIPTION
BLACKNESS	Fills the destination rectangle using the color associated with index 0 in the physical palette. The default value is black.
DSTINVERT	Inverts the destination rectangle.
MERGECOPY	Merges the colors of the source rectangle with the specified pattern using an AND operation.
MERGEPAINT	Merges the colors of the inverted source rectangle with the colors of the destination rectangle using an OR operation.
NOTSRCCOPY	Inverts the bits in the source rectangle and copies it to The destination.
NOTSRCERASE	Combines the colors of the source and destination rectangles using an OR operation, and then inverts the result.
PATCOPY	Copies the specified pattern to the destination bitmap.
PATINVERT	Combines the colors of the specified pattern with the colors of the destination rectangle using an XOR operation.
PATPAINT	Combines the colors of the pattern with the colors of the inverted source rectangle using an OR operation. The result of this operation is combined with the colors of the destination rectangle using an OR operation.
SRCAND	Combines the colors of the source and destination rectangles using an AND operation SRCCOPY Copies the source rectangle directly to the destination rectangle. This is, by far, the most-used mode in bitblt operations.

(continues)

Table 22-2

Symbolic Names for Raster Operations (continued)

NAME	DESCRIPTION
SRCERASE	Combines the inverted colors of the destination rectangle with the colors of the source rectangle using an AND operation.
SRCINVERT	Combines the colors of the source and destination rectangles using an XOR operation.
SRCPAINT	Combines the colors of the source and destination rectangles using an OR operation.
WHITENESS	Fills the destination rectangle using the color associated with index 1 in the physical palette. The default value is White.

22.3.5 A Bitmap Display Function

Displaying a bitmap is a multistage process that includes the following operations:

1. Creating a memory device context.
2. Selecting the bitmap into the memory device context.
3. Obtaining the bitmap dimensions and converting device units to logical units.
4. Blitting the bitmap onto the screen according to a ternary raster operation code

Many graphics applications can make use of a function that performs all of the previous operations. The function named ShowBitmap() is used in the Bitmap Demo project on the book's software package. The function's prototype is as follows:

```
void ShowBitmap (HDC, HBITMAP, int, int, DWORD);
                  |   -------   |    |   -----
                  |      |      |    |     |
                  1      2      3    4     5
```

The first parameter is the handle to the device context. The ShowBitmap() function creates its own memory device context. The second parameter is the handle to the bitmap that is to be displayed. The third and fourth parameters are the screen coordinates for displaying the upper-left corner of the bitmap. The fifth parameter is the ternary ROP code used in blitting the bitmap. If this parameter is NULL then the bitmap is displayed using the SRCCOPY raster operation. The following is a listing of the ShowBitmap() function:

```
void ShowBitmap (HDC hdc, HBITMAP hBitmap, int xStart, int yStart,\
                 DWORD rop3) {

BITMAP     bm;             // BITMAP structure
HDC        memoryDc;       // Handle to memory DC
POINT      ptSize;         // POINT for DC
POINT      ptOrigin;       // POINT for memory DC
int        mapMode;        // Mapping mode

    // Test for NULL ROP3 code
    if (rop3 == NULL)
        rop3 = SRCCOPY;

    memoryDc = CreateCompatibleDC (hdc); // Memory device
```

```
                                             // handle
    mapMode = GetMapMode (hdc);              // Obtain mapping
                                             // mode
    SetMapMode (memoryDc, mapMode);          // Set memory DC
                                             // mapping mode

    // Select bitmap into memory DC
    // Note: assert statement facilitates detecting invalid
    //       bitmaps during program development
    assert (SelectBitmap (memoryDc, hBitmap));

    // Obtain bitmap dimensions
    GetObject (hBitmap, sizeof(BITMAP), (LPVOID) &bm);

    // Convert device units to logical units
    ptSize.x = bm.bmWidth;
    ptSize.y = bm.bmHeight;
    DPtoLP (hdc, &ptSize, 1);

    ptOrigin.x = 0;
    ptOrigin.y = 0;
    DPtoLP (memoryDc, &ptOrigin, 1);

    // Bitblt bitmap onto display memory
    BitBlt( hdc, xStart, yStart, ptSize.x, ptSize.y, memoryDc,
            ptOrigin.x, ptOrigin.y, rop3);

    // Delete memory DC with bitmap
    DeleteDC (memoryDc);
}
```

22.4 Bitmap Manipulations

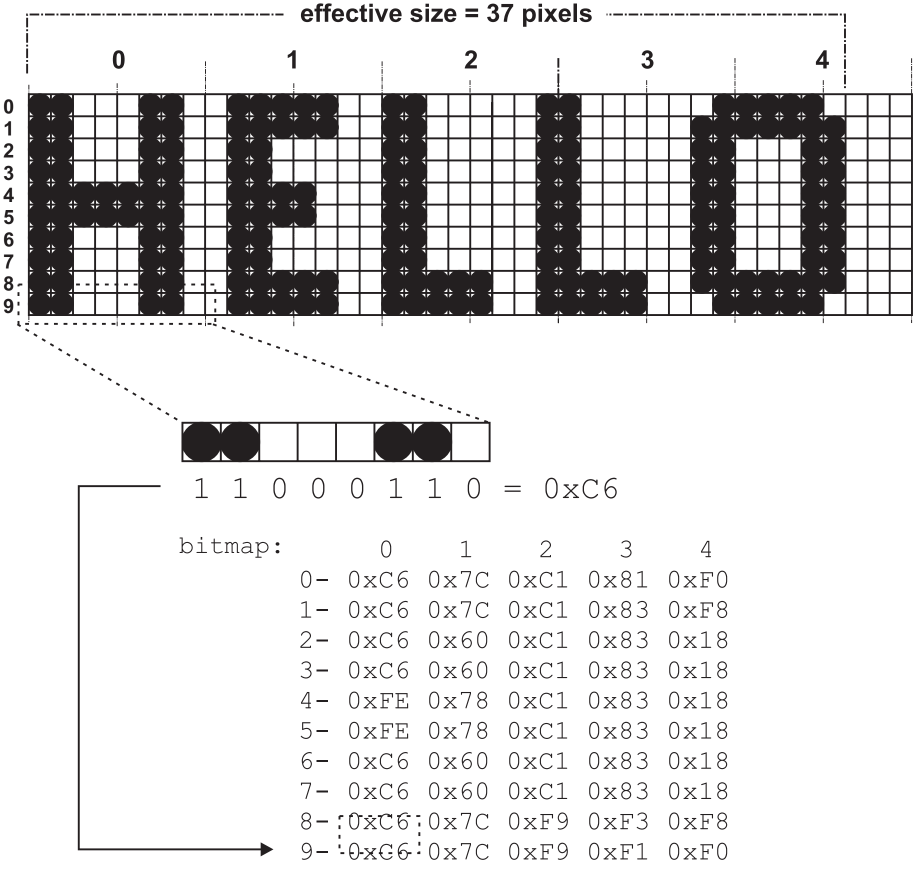
In addition to displaying a bitmap stored in a disk file, graphics applications often need to perform other bitmap-related operations. The following are among the most common operations:

- Creating and displaying a hard-coded bitmap
- Creating a bitmap in heap memory
- Creating a blank bitmap and filling it by means of GDI functions
- Creating a system-memory bitmap which applications can access directly
- Using a bitmap to create a pattern brush

22.4.1 Hard-Coding a Monochrome Bitmap

With the facilities available in Developer Studio for creating bitmaps, the programmer is seldom forced to hard-code a bitmap. Our rationale for discussing this option is that hard-coding bitmaps is a basic skill for a graphics programmer, and that it helps you to understand bitmaps in general.

A monochrome bitmap has one color plane and is encoded in a 1-bit per pixel format. In Windows, a monochrome bitmap is displayed by showing the 0-bits in the foreground color and the 1-bits in the background color. If the screen has a white foreground and a black background, 0-bits in the bitmap are displayed as white pixels, and vice versa. If the bitmap is to be blitted on the screen using the BitBlt() function, the action of the bits can be reversed by changing the raster operation code. This gives the programmer the flexibility of using either zero or one bits for background or foreground attributes. Figure 22-4 shows a hard-coded monochrome bitmap.



```
1 1 0 0 0 1 1 0 = 0xC6

bitmap:       0    1    2    3    4
         0- 0xC6 0x7C 0xC1 0x81 0xF0
         1- 0xC6 0x7C 0xC1 0x83 0xF8
         2- 0xC6 0x60 0xC1 0x83 0x18
         3- 0xC6 0x60 0xC1 0x83 0x18
         4- 0xFE 0x78 0xC1 0x83 0x18
         5- 0xFE 0x78 0xC1 0x83 0x18
         6- 0xC6 0x60 0xC1 0x83 0x18
         7- 0xC6 0x60 0xC1 0x83 0x18
         8- 0xC6 0x7C 0xF9 0xF3 0xF8
         9- 0xC6 0x7C 0xF9 0xF1 0xF0
```

Figure 22-4 *Hard-Coded, Monochrome Bitmap*

In Figure 22-4 the dark pixels in the image are represented by 0-bits in the bitmap. The resulting data structure has five bytes per scan line and a total of 10 scan lines. The illustration shows how eight pixels in the bottom row of the image are represented by 8 bits (1 byte) of the bitmap. The dark pixels are encoded as 0 bits and the light pixels as 1-bits. In order to display this bitmap so that the letters are black on a white screen background, the black and white pixels have to be reversed by changing the raster operation mode from the default value, SCRCOPY, to NOTSCRCOPY, as previously explained.

Once the bit image of a monochrome bitmap has been calculated, we can proceed to store this bit-to-pixel information in an array of type BYTE. Windows requires that bitmaps be word-aligned; therefore, each scan line in the array must have a number of bytes divisible by 2. In regards to the bitmap in Figure 22-4, you would have to add a padding byte to each scan line in order to satisfy this requirement. The resulting array could be coded as follows:

```
//                          0      1      2      3      4     5
static BYTE hexBits[] = {0xc6, 0x7c, 0xc1, 0x81, 0xf0, 0x0,
                         0xc6, 0x7c, 0xc1, 0x83, 0xf8, 0x0,
                         0xc6, 0x60, 0xc1, 0x83, 0x18, 0x0,
                         0xc6, 0x60, 0xc1, 0x83, 0x18, 0x0,
                         0xfe, 0x78, 0xc1, 0x83, 0x18, 0x0,
                         0xfe, 0x78, 0xc1, 0x83, 0x18, 0x0,
                         0xc6, 0x60, 0xc1, 0x83, 0x18, 0x0,
                         0xc6, 0x60, 0xc1, 0x83, 0x18, 0x0,
                         0xc6, 0x7c, 0xf9, 0xf3, 0xf8, 0x0,
                         0xc6, 0x7c, 0xf9, 0xf1, 0xf0, 0x0};
//                                                        |
//                               padding byte -----------|
```

The CreateBitmap() function can be used to create a bitmap given the bit-to-pixel data array. The function's general form is as follows:

```
HBITMAP CreateBitmap(
         int nWidth,          // 1
         int nHeight,         // 2
         UINT cPlanes,        // 3
         UINT cBitsPerPel,    // 4
         CONST VOID *lpvBits  // 5
);
```

The first parameter is the actual width of the bitmap, in pixels. In relation to the bitmap in Figure 22-4, this value is 37 pixels, as shown in the illustration. The second parameter is the number of scan lines in the bitmap. The third parameter is the number of color planes. In the case of a monochrome bitmap this value is 1. The fourth parameter is the number of bits per pixel. In a monochrome bitmap this value is also 1. The fifth parameter is a pointer to the location where the bitmap data is stored. If the function succeeds, the returned value is the handle to a bitmap. If the function fails, the return value is NULL. The following code fragment initializes a monochrome bitmap using the bitmap data in the hexBits[] array previously listed:

```
static HBITMAP      bmImage1;
.
.
.
// Initialize monochrome bitmap
bmImage1 = CreateBitmap (37, 10, 1, 1, hexBits);
```

Alternatively, a bitmap can be defined using a structure of type BITMAP, listed previously in this chapter and in Appendix A. Before the bitmap is created, the structure members must be initialized, as in the following code fragment:

```
static BITMAP       monoBM;
.
.
.
// Initialize data structure for a monochrome bitmap
monoBM.bmType = 0;       // must be zero
monoBM.bmWidth = 37;     // actual pixels used
```

```
    monoBM.bmHeight = 10;    // scan lines
    monoBM.bmWidthBytes = 6; // width (must be word aligned)
    monoBM.bmPlanes = 1;     // 1 for monochrome bitmaps
    monoBM.bmBitsPixel = 1;  // 1 for monochrome bitmaps
    monoBM.bmBits = (LPVOID) &hexBits; // address of bit field
```

When the bitmap data is stored in a structure of type BITMAP, the bitmap can be created by means of the CreateBitmapIndirect() function. The function's general form is as follows:

```
HBITMAP CreateBitmapIndirect(
         CONST BITMAP *lpbm    // 1
);
```

The function's only parameter is a pointer to a structure of type BITMAP. If the function succeeds, the returned value is the handle to a bitmap. If the function fails, the return value is NULL. The following code fragment uses CreateBitmapIndirect() to initialize a monochrome bitmap using the bitmap data in the hexBits[] array previously listed:

```
    static HBITMAP      bmImage1;
    .
    .
    .
    // Initialize monochrome bitmap
    bmImage1 = CreateBitmapIndirect (&monoBM);
```

Whether the bitmap was created using CreateBitmap() or CreateBitmapIndirect(), it can now be displayed by means of the ShowBitmap() function developed and listed previously in this chapter.

22.4.2 Bitmaps in Heap Memory

A bitmap can take up a considerable amount of memory or storage resources. For example, a 1200-by-1200 pixel bitmap encoded in 32-bit color takes up approximately 5.7 Mb. Applications that store large bitmaps in their own memory space can run into memory management problems. One possible solution is to store large bitmaps in dynamically allocated memory, which can be freed as soon as the bitmap is no longer needed. Note that freeing memory where the bitmap is stored does not affect the screen image.

Several programming techniques can be used to allocate and release heap memory for a bitmap. The one most suitable depends on the particular needs of each particular programming problem. In one case you may allocate heap memory for a bitmap during WM_CREATE message processing, and then use this allocated space to create or copy different bitmaps during program execution. The allocated memory can then be freed during WM_DESTROY processing. Another option is to allocate memory at the time it is required, create or copy the bitmap to this allocated space, and deallocate the memory when the bitmap is no longer needed.

```
Many fables and fantastic theories have originated in the complications and
misunderstandings of Windows memory management. Although most of these
problems were corrected in Windows 3.1, an entire programming subculture
still thrives on discussions related to these topics. A programmer
encountering memory management in Windows finds that there are three
```

separate sets of memory allocation and deallocation operators that serve apparently identical purposes: the C++ operators new and delete, the traditional C operators malloc and free, and the Windows kernel functions LocalAlloc(), GlobalAlloc(), LocalFree(), and GlobalFree().

In Win32 programming, the first simplification is a result of the fact that there is no difference between the global and the local heaps. Therefore, GlobalAlloc() and LocalAlloc(), as well as GlobalFree() and LocalFree(), actually perform virtually identical functions. Because of their greater flexibility we will use GlobalAlloc() and GlobalFree() instead of new and delete or malloc and free operators from the C++ and C libraries. Another reason for this preference is that most Windows compilers implement malloc and new in terms of GlobalAlloc(); therefore, the traditional operators offer no advantage.

Traditionally, three types of memory are documented as being available to applications: fixed memory, moveable memory, and discardable memory. The justification for preferring moveable memory disappeared with Windows 95, in which a memory block can be moved in virtual memory while retaining the same address. For the same reason, the use of fixed memory no longer needs to be avoided. Discardable memory is only indicated when the data can be easily recreated, which is not usually the case with image data structures. In conclusion, fixed memory is usually quite suitable for dynamically storing bitmaps and other image data.

Note that the terms moveable and movable are both accepted, although movable is preferred. However, the Windows API contains the constants GMEM_MOVEABLE and LMEM_MOVEABLE. For this reason we have used the former.

In this section we discuss the bare essentials of memory allocation and deallocation in Windows. The topic of Windows memory management can easily fill a good-size volume. Graphics applications are often memory-intensive and may require sophisticated memory management techniques. A graphics programmer should have a thorough knowledge of Win32 memory architecture, virtual memory, and heap management. The book Advanced Windows, by Richter (see Bibliography) has chapters devoted to each of these topics.

The GlobalAlloc() function is used to allocate memory from the default heap. The default heap is initially 1Mb, but under Windows, this heap grows as it becomes necessary. The function's general form is as follows:

```
   HGLOBAL GlobalAlloc(
          UINT uFlags,     // 1
          DWORD dwBytes    // 2
);
```

The first parameter is a flag that determines how memory is allocated to the caller. Table 22-3 lists the most commonly used allocation flags. The second parameter is the number of bytes of memory to be allocated. If it succeeds, the function returns the handle to the allocated memory object. If the allocation flag is GMEM_FIXED, the handle can be directly cast into a pointer and the memory used. If the allocated memory is not fixed, then it must be locked using GlobalLock() before it can be used by code. The function returns NULL if the allocation request fails.

Table 22-3

Win-32 Commonly Used Memory Allocation Flags

FLAG	MEANING
GMEM_FIXED	Allocates fixed memory. This flag cannot be Combined with the GMEM_MOVEABLE or GMEM_DISCARDABLE flag. The return value is a pointer to the memory block.
GMEM_MOVEABLE	Allocates moveable memory. This flag cannot be combined with the GMEM_FIXED flag. The return Value is the handle of the memory object, which is A 32-bit quantity private to the calling process.
GMEM_DISCARDABLE	Allocates discardable memory. This flag cannot be combined with the flag. Win32-based operating systems ignore this flag.
GMEM_ZEROINIT	Initializes memory to 0.
GPTR	Combines the GMEM_FIXED and GMEM_ZEROINIT flags.
GHND	Combines the GMEM_MOVEABLE and GMEM_ZEROINIT flags.

The process of creating a bitmap in heap memory and displaying it on the video screen can be accomplished by the following steps:

1. Dynamically allocate the memory required for the bitmap and the corresponding data structures.
2. Store the bitmap data and the bitmap dimensions and color information, thus creating a DIB.
3. Convert the device independent bitmap (DIB) into a device dependent bitmap using CreateDIBitmap().
4. Display the bitmap using SetDIBitsToDevice().

Suppose you wanted to create a 200-pixel wide bitmap, with 255 scan lines, in 32-bit color. Since each pixel requires four bytes, each scan line consists of 800 bytes, and the entire bitmap occupies 204,000 bytes. A BITMAPINFO structure variable is used to hold the bitmap information. Notice that because this is a true-color bitmap, the RGBQUAD structure is not necessary. In this case the memory allocation operations can be coded as follows:

```
static PBITMAPINFO  pDibInfo;    // pointer to BITMAPINFO structure
static BYTE        *pDib;        // pointer to bitmap data
.
.
.
pDibInfo = (PBITMAPINFO) LocalAlloc(LMEM_FIXED, \
                         sizeof(BITMAPINFOHEADER));
pDib = (BYTE*) LocalAlloc(LMEM_FIXED, 204000);
```

At this point the code has allocated memory for both the bitmap and the BITMAPINFOHEADER structure variable that is to hold the bitmap format information. The pointers to each of these memory areas can be used to fill them in. First the bitmap information:

```
pDibInfo->bmiHeader.biSize = (LONG) sizeof(BITMAPINFOHEADER);
pDibInfo->bmiHeader.biWidth = (LONG) 200;     // pixel width
pDibInfo->bmiHeader.biHeight = (LONG) 255;    // pixel height
pDibInfo->bmiHeader.biPlanes = 1;             // number of planes
pDibInfo->bmiHeader.biBitCount = 32;          // bits per pixel
```

Assume that the bitmap is to represent a blue rectangle with 255 decreasing intensities of blue, along the scan lines. The code to fill this bitmap can be coded as follows:

```
int    i, j, k;        // counters
BYTE  shade;           // shade of blue
.
.
.
// Fill the bitmap using 32-bit color data
// <--------- 200 pixels (4 bytes each) -------->
// |
// | ... 255 scan lines
shade = 0;
  for (k = 0; k < 255; k++){       // Counts 255 scan lines
     for (i = 0; i < 200; i++){    // Counts 200 pixels
        for(j = 0; j < 4; j++) {   // Counts 4 bytes
            pDib[((k*800)+(i*4)+0] = shade; // blue
            pDib[((k*800)+(i*4)+1] = 0;     // green
            pDib[((k*800)+(i*4)+2] = 0;     // red
            pDib[((k*800)+(i*4)+3] = 0;     // must be zero
            };
        };
     shade++;
  };
```

Now that the bitmap data structures have been initialized and the bitmap data entered into the allocated heap memory, it is time to create the device-dependent bitmap that can be displayed on the display context. The function used for this purpose is named CreateDIBitmap(); this name is somewhat confusing since it actually creates a dependent device from a device-independent bitmap. The function's general form is as follows:

```
HBITMAP CreateDIBitmap(
         HDC hdc,                          // 1
CONST BITMAPINFOHEADER *lpbmih,   // 2
         DWORD fdwInit,                    // 3
         CONST VOID *lpbInit,              // 4
         CONST BITMAPINFO *lpbmi,          // 5
         UINT fuUsage                      // 6
);
```

The first parameter is the handle to the device context for which the device dependent bitmap is to be configured. The second parameter is a pointer to a BITMAPINFOHEADER structure variable that contains the bitmap data. The third parameter is a flag that determines how the operating system initializes the bitmap bits. If this parameter is zero the bitmap data is not initialized and parameters 4 and 5 are not used. If it is set to CBM_INIT, then parameters 4 and 5 are used as pointers to the data used in initializing the bitmap bits. The fourth parameter is a pointer to the array of type BYTE that contains the bitmap data. The fifth parameter is a pointer to a BITMAPINFO structure that contains the bitmap size and color data.

The sixth parameter is a flag that determines whether the bmiColors member of the BITMAPINFO structure contains explicit color values in RGB format or palette indices. In the first case the constant DIB_RGB_COLORS is used for this parameter, and in the second case the constant is DIB_PAL_COLORS. The function returns the handle to the bitmap if it succeeds, or NULL if it fails.

In the example that we have been following, the device-dependent bitmap is created as follows:

```
  static HBITMAP      hBitmap;     // handle to a bitmap
  .
  .
  .

  hBitmap = CreateDIBitmap ( hdc,
               (LPBITMAPINFOHEADER) &pDibInfo->bmiHeader,
               CBM_INIT,
               (LPSTR) pDib,
               (LPBITMAPINFO) pDibInfo,
               DIB_RGB_COLORS );
```

Having obtained its handle, the bitmap can be displayed using the ShowBitmap() function developed earlier in this chapter. Alternatively, you can use SetDIBitsToDevice() to set the screen pixels. The function's general form is as follows:

```
int SetDIBitsToDevice(
         HDC hdc,                    // 1
         int XDest,                  // 2
         int YDest,                  // 3
         DWORD dwWidth,              // 4
         DWORD dwHeight,             // 5
         int XSrc,                   // 6
         int YSrc,                   // 7
         UINT uStartScan,            // 8
         UINT cScanLines,            // 9
         CONST VOID *lpvBits,        // 10
         CONST BITMAPINFO *lpbmi,    // 11
         UINT fuColorUse             // 12
);
```

The first parameter is the handle to the display context to which the bitmap is to be output. The second and third parameters are the x- and y-coordinates of the destination rectangle, in logical units. This is the screen location where the bitmap is displayed. The fourth and fifth parameters are the width and height of the DIB. These values are often read from the corresponding members of the BITMAPINFOHEADER structure variable that defines the bitmap. The sixth and seventh parameters are the x- and y-coordinates of the lower-left corner of the DIB. The eighth parameter is the starting scan line of the DIB. The ninth parameter is the number of scan lines. The tenth parameter is a pointer to the bitmap data and the eleventh parameter is a pointer to the BITMAPINFO structure variable that describes the bitmap. The twelfth parameter is a flag that determines whether the bmiColors member of the BITMAPINFO structure contains explicit color values in RGB format or palette indices. In the first case the constant DIB_RGB_COLORS is used, and in the second case the constant DIB_PAL_COLORS. If the function succeeds the return value is the number of scan lines displayed. The function returns NULL if it fails.

In the current example, the bitmap can be displayed with the following call to SetDIBitsToDevice():

```
 SetDIBitsToDevice (hdc, 50, 50,
          pDibInfo->bmiHeader.biWidth,
          pDibInfo->bmiHeader.biHeight,
          0, 0, 0,
          pDibInfo->bmiHeader.biHeight,
          pDib,
          (BITMAPINFO FAR*) pDibInfo,
          DIB_RGB_COLORS);
```

22.4.3 Operations on Blank Bitmaps

Sometimes an application needs to fill a blank bitmap using GDI functions. The functions can include all the drawing and text display primitives discussed in previous chapters. There are several programming approaches to creating a bitmap on which GDI operations can be performed. The simplest approach is to select a bitmap into a memory device context and then perform the draw operation on the memory device context. Note that all the drawing functions discussed previously require a handle to the device context. When the drawing takes place on a memory DC, the results are not seen on the video display until the memory DC is blitted to the screen. In this approach the following steps are required:

1. Select the bitmap into a memory device context using the SelectObject() function.
2. Clear or otherwise paint the bitmap using the PatBlt() function.
3. Perform drawing operations on the memory device context that contains the bitmap.
4. Display the bitmap by blitting it on the screen, typically with BitBlt().

The CreateCompatibleBitmap() function has the following general form:

```
HBITMAP CreateCompatibleBitmap(
            HDC hdc,          // 1
            int nWidth,       // 2
            int nHeight       // 3
);
```

The first parameter is the handle to the device context with which the created bitmap is to be compatible. If the bitmap is to be displayed this parameter is set to the display context. The second and third parameters are the width and height of the bitmap, in pixels. If the function succeeds it returns the handle to the bitmap. The function returns NULL if it fails.

In the following code sample we create a blank, 300 by 300-pixel bitmap, draw a rectangle and an ellipse on it, and then blit it to the screen. First we start by creating the blank bitmap:

```
  static HDC        aMemDC;           // memory device context
  static HBITMAP    bmBlank;          // handle to a bitmap
  static HGDIOBJ    oldObject;        // storage for current object
  .
```

```
.
.
// Preliminary operations
aMemDC = CreateCompatibleDC (NULL);  // Memory device handle
mapMode = GetMapMode (hdc);          // Obtain mapping mode
SetMapMode (aMemDC, mapMode);        // Set memory DC mapping mode
// Create the bitmap
bmBlank = CreateCompatibleBitmap (hdc, 300, 300);
oldObject = SelectObject (aMemDC, bmBlank);
```

Note that we use a generic handle (HGDIOBJ) to store the current handle in the device context.

There is no guarantee that the bitmap thus created and selected into the device is initialized. The PatBlt() function can be used to set all the bitmap bits to a particular attribute or to a predefined pattern. The function's general form is as follows:

```
BOOL PatBlt(
         HDC hdc,      // 1
         int nXLeft,   // 2
         int nYLeft,   // 3
         int nWidth,   // 4
         int nHeight,  // 5
         DWORD dwRop   // 6
);
```

The first parameter is the handle to the device context, which can be a memory device context. The second and third parameters are the x- and y-coordinates of the upper-left corner of the rectangle to be filled. The fourth and fifth parameters are the width and height of the bitmap, in logical units. The sixth parameter is one of the following constants: PATCOPY, PATINVERT, DSTINVERT, BLACKNESS, or WHITENESS. The constants are defined in Table 22-2. The call returns TRUE if it succeeds and FALSE if it fails.

Following the current example, the call clears the bitmap and sets all bits to the white attribute:

```
PatBlt (aMemDC, 0, 0, 300, 300, WHITENESS);
```

At this point in the code we can start performing drawing operations on the bitmap. The only requirement is that the drawing primitives reference the handle to the memory device context where the blank bitmap was selected, as in the following example:

```
Ellipse (aMemDC, 10, 10, 210, 110);
Polyline (aMemDC, rectangle, 5);
```

Once you have finished drawing on the blank bitmap, it can be displayed by means of a bitblt, as in the following example:

```
BitBlt(hdc, 50, 50, 300, 300, aMemDC, 0, 0, SRCCOPY);
```

In this case, the call references both the display context (hdc) and the memory device context containing the bitmap (aMemDC).

Clean-up operations consist of reselecting the original object to the memory device context, then deleting the device context and the bitmap.

22.4.4 Creating a DIB Section

The methods described in the preceding section are satisfactory when the bitmap area requires drawing operations that can be implemented by GDI functions, but code has no direct access to the bitmap itself. This is due to the fact that CreateCompatibleBitmap() does not return a pointer to the bitmap data area. The CreateDIBSection() function, first introduced in Win32 and formalized in Windows 95, allows creating a device-independent bitmap that applications can access directly.

Note that the original Windows documentation for Win32 contained incorrect information about the CreateDIBSection() function and the associated DIBSECTION structure. Some of the errors and omissions were later corrected so that current documentation is more accurate, although not very clear.

Before CreateDIBSection(), an application would access bitmap data by calling the GetDIBits() function, which copies the bitmap into a buffer supplied by the caller. At the same time, the bitmap size and color data is copied into a BITMAPINFO structure from which the application can read these values. After the bitmap is changed, the SetDIBits() function is used to redisplay the bitmap. Both functions, GetDIBits() and SetDIBits(), allow selecting the first scan line and the number of scan lines. When operating on large bitmaps, this feature makes it possible to save memory by reading and writing portions of it at a time.

There are several shortcomings to modifying bitmaps at run time using GetDIBits() and SetDIBits(). The most obvious one is that the system bitmap must be copied into the application's memory space, then back into system memory. The process is wasteful and inefficient. If the entire bitmap is read during the GetDIBits() call, there are two copies of the same data, thus wasting memory. If it is broken down into regions in order to reduce the waste, then processing speed suffers considerably. The solution offered by CreateDIBSection() is to create a bitmap that can be accessed by both the system and the application. Figure 22-5 shows both cases.

Although CreateDIBSection() provides a better alternative than GetDIBits() and SetDIBits(), it is by no means the ultimate in high-performance graphics. DirectDraw methods, discussed starting in Chapter 9, provide ways of accessing video memory directly and of taking advantage of raster graphics hardware accelerators.

In the following example, we create a DIB section, using the pointer returned by the CreateDIBSection() call to fill the bitmap, and the bitmap handle to perform GDI drawing functions on its memory space. The bitmap is 50 pixels wide and 255 scan lines long. It is encoded in 32-bit true color format. The code starts by defining the necessary data structures, initializing the variables, and allocating memory.

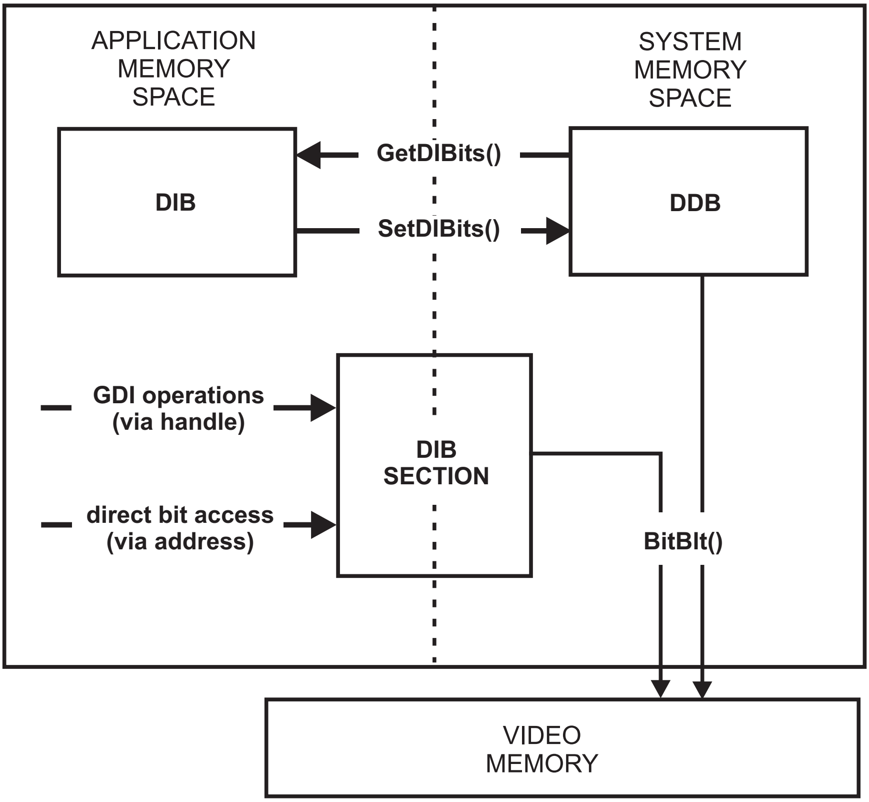


Figure 22-5 *Memory Image of Conventional and DIB Section Bitmaps*

```
HDC                 aMemDC;        // Memory DC
static HBITMAP      aBitmap;
static BYTE*        lpBits;

BITMAPINFOHEADER    bi;
BITMAPINFOHEADER*   lpbi;
HANDLE              hDIB;
int                 shade;
static int BMScanLines;     // Bitmap y-dimension
static int BMWidth;         // Bitmap x-dimension
.
.
.
// Initialize size variables
BMScanLines = 255;
BMWidth = 50;
// Initialize BITMAPINFOHEADER structure
bi.biSize = sizeof(BITMAPINFOHEADER);
bi.biWidth = BMWidth;
bi.biHeight = BMScanLines;
bi.biPlanes = 1;
bi.biBitCount = 32;
bi.biCompression = BI_RGB;
bi.biSizeImage = 0;
```

```
     bi.biXPelsPerMeter = 0;
     bi.biYPelsPerMeter = 0;
     bi.biClrUsed = 0;
     bi.biClrImportant = 0;

    // Allocate memory for DIB
    hDIB = GlobalAlloc (GMEM_FIXED, sizeof(BITMAPINFOHEADER));
    // Initialize bitmap pointers
    lpbi = (BITMAPINFOHEADER*) hDIB;
    *lpbi = bi;
```

At this point everything is ready to call CreateDIBSection(). The function's general form is as follows:

```
HBITMAP CreateDIBSection(
              HDC hdc,                     // 1
              CONST BITMAPINFO *pbmi,      // 2
              UINT iUsage,                 // 3
              VOID *ppvBits,               // 4
              HANDLE hSection,             // 5
              DWORD dwOffset               // 6
);
```

The first parameter is the handle to a device context associated with the DIB section. The second parameter is a pointer to a structure variable of type BITMAPINFOHEADER, which holds the bitmap attributes. The first five members of the BITMAPINFOHEADER structure are required; the other ones can often be omitted, although it is usually a good idea to fill in the entire structure. The third parameter is either the constant DIB_PAL_COLORS or DIB_RGB_COLORS. In the first case the bmiColors array member of the RGBQUAD structure in BITMAPINFO is a set of 16-bit palette color indices. In the second case the bmiColors member is not used and the colors are encoded directly in the bitmap. The fourth parameter is a pointer to a pointer to type VOID that contains the location of the bitmap values. This parameter is incorrectly documented in the Windows help files as a pointer to type VOID. If the parameter is not correctly typecast to (VOID**) the CreateDIBSection() call fails.

The fifth parameter is a handle to a file-mapping object. In file mapping, a physical file on disk is associated with a portion of the virtual address space of a process. The file-mapping object is the mechanism that maintains this association. Its main purpose is to share data between applications and to facilitate access to files. Although file mapping is a powerful mechanism, it is outside the scope of this book and is not discussed any further. If no file mapping is used, the fifth parameter is set to NULL, and the sixth one, which sets the offset of the file mapping object, is set to zero.

Following the current example, the call to CreateDIBSection() is coded as follows:

```
  aBitmap = CreateDIBSection (hdc,
            (LPBITMAPINFO)lpbi,  // Pointer to
                                 // BITMAPINFOHEADER
            DIB_RGB_COLORS,      // True color in RGB format
            (VOID**) &lpBits,    // Pointer to bitmap data
            NULL,                // File mapping object
```

```
           (DWORD) 0);          // File mapping object offset
  assert (aBitmap);
  assert (lpBits);
```

The two assertions that follow the call ensure that a valid bitmap and pointer are returned. If the call succeeds we now have a handle to a bitmap and its address in system memory. Using the address, we can fill the bitmap. The following code fragment uses the soft-coded bitmap parameters to fill the entire bitmap, scan line by scan line, with increasing intensities of blue. The access to the bitmap is by means of the pointer (lpBits) returned by the previous call.

```
   // Fill the bitmap using 32-bit color data
   // <--------- BMWidth * 4 -------->
   // |
   // | ... BMScanLines

   shade = 0;
   for (k = 0; k < BMScanLines; k++){    // Counts 255 lines
      for (i = 0; i < BMWidth; i++){     // Counts 50 pixels
         for(j = 0; j < 4; j++) {  //Counts 4 bytes per pixel
           lpBits[(k*(BMWidth*4))+(i*4)+0] = shade; // blue
           lpBits[(k*(BMWidth*4))+(i*4)+1] = 0x0;   // green
           lpBits[(k*(BMWidth*4))+(i*4)+2] = 0x0;   // red
           lpBits[(k*(BMWidth*4))+(i*4)+3] = 0;     // zero
           };
         };
       shade++;
   };
```

Since we have also acquired the handle to the bitmap, we can use GDI functions to perform drawing operations on its surface. As described earlier in this chapter, the GDI functions require that the bitmap be first selected into a memory device context. The following code fragment shows one possible processing method:

```
   aMemDC = CreateCompatibleDC (NULL); // Memory device handle
   mapMode = GetMapMode (hdc);         // Obtain mapping mode
   SetMapMode (aMemDC, mapMode);       // Set memory DC
                                       // mapping mode
   // Select the bitmap into the memory DC
   oldObject = SelectObject (aMemDC, aBitmap);
Drawing operations can now take place, as follows:
 // Draw on the bitmap
 blackPenSol = CreatePen (PS_SOLID, 2, 0);
 redPenSol = CreatePen (PS_SOLID, 2, (RGB (0xff, 0x0, 0x0)));
 SelectPen (aMemDC, blackPenSol);
 Polyline (aMemDC, rectsmall, 5);     // Draw a rectangle
 SelectPen (aMemDC, redPenSol);
 Ellipse (aMemDC, 4, 4, 47, 47);      // Draw a circle
```

You may be tempted to display the bitmap at this time; the display operation, however, cannot take place until the memory device context has been deleted. In the following instructions we re-select the original object in the memory device context and then delete it. We also delete the pens used in the drawing operations.

```
  // Erase bitmap and free heap memory
  SelectObject (aMemDC, oldObject);
  DeleteDC (aMemDC);
  DeleteObject (redPenSol);
```

```
DeleteObject (blackPenSol);
```

Displaying the bitmap can be performed by any of the methods already discussed. In this code fragment we use the ShowBitmap() function developed earlier in the chapter. A necessary precaution relates to the fact that some versions of Windows NT place GDI calls that return a boolean value in a batch for later execution. In this case, it is possible to attempt to display a DIB section bitmap before all the calls in the GDI batch have been executed. In order to prevent this problem, it is a good idea to flush the GDI batch buffer before displaying a DIB section bitmap, as shown in the following code:

```
GdiFlush();       // Clear the batch buffer
ShowBitmap (hdc, Abitmap, 50, 50, SRCCOPY);
```

Now that you have finished displaying the bitmap, a tricky problem arises: how to free the system memory space allocated by CreateDIBSection(). The solution is easy. Since the bitmap resides in system memory, all we have to do in application code is delete the bitmap; Windows takes care of freeing the memory. On the other hand, if the BITMAPINFOHEADER structure was defined in heap memory, your code must take care of freeing this memory space in the conventional manner. Processing is as follows:

```
// Erase bitmap and free heap memory
// Note: deleting a DIB section bitmap also frees
//       the allocated memory resources
DeleteObject (aBitmap);   // Delete the bitmap
GlobalFree (hDIB);
```

Figure 22-6 is a screen snapshot of a program that executes the listed code. The listing is found in the Bitmap Demo project folder on the book's software package

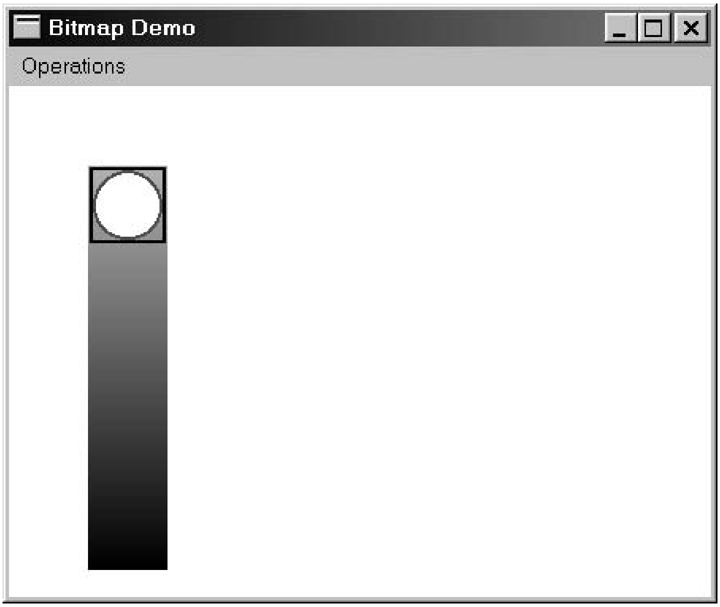


Figure 22-6 *Screen Snapshot Showing a DIB Section Bitmap Manipulation*

22.4.5 Creating a Pattern Brush

In Chapter 21 we mentioned that applications can create a brush with a hatch pattern different than the ones predefined in Windows. This is done by using a bitmap to define the brush pattern. In Windows 95/98 the size of the bitmap cannot exceed 8-by-8 pixels, but there is no size restriction in Windows NT. The function's general form is as follows:

```
HBRUSH CreatePatternBrush (HBITMAP hbitmap);
```

The function's only parameter is a handle to the bitmap that defines the brush. The bitmap can be created with CreateBitmap(), CreateBitmapIndirect(), or CreateCompatibleBitmap() functions. It can also be a bitmap drawn using Developer Studio bitmap editor, or any other similar utility, and loaded with the LoadBitmap() function. The one type of bitmap that is not allowed is one created with the CreateDIBSection() function. CreatePatternBrush() returns the handle to the brush if it succeeds, and NULL if it fails.

Once the handle to the brush has been obtained, the pattern brush is selected into the device context. Thereafter, all GDI drawing functions that use a brush use the selected pattern brush. The following code fragment shows the creation of a pattern brush from a bitmap resource named IDC_BITMAP5. The pattern brush is then used to draw a rectangle.

```
static HBITMAP      brushBM1;   // Handle to a bitmap
static HBRUSH       patBrush;   // Handle to a brush
.
.
.
brushBM1 = LoadBitmap (pInstance,
                       MAKEINTRESOURCE (IDB_BITMAP5);
patBrush = CreatePatternBrush (brushBM1);
SelectBrush (hdc, patBrush);
Rectangle (hdc, 10, 10, 110, 160);
DeleteObject (patBrush);
```

In displaying solid figures that use a pattern brush, Windows sets the origin of the brush bitmap to the origin of the client area. The SetBrushOrgEx() function is used to reposition the brush bitmap in relation to the origin of the client area. This matter was discussed in Chapter 7 in relation to brush hatch patterns.

22.5 Bitmap Transformations

In addition to manipulating bitmaps, Windows provides functions that transform the bitmaps themselves. You have already seen that the BitBlt() function allows you to define a ternary raster operation code that determines how the source bitmap and a pattern are combined to form the destination bitmap. In this section we discuss the following transformations that are useful in bitmap programming:

- Painting a bitmap using a raster operation based on the brush selected in the device context
- Stretching or compressing a bitmap according to the dimensions of a destination rectangle, a predefined stretch mode, and the selected ternary raster operation code

Windows NT provides two powerful bitmap transforming functions named MaskBlt() and PlgBlt(). Since the scope of this book includes functions that are available only in Windows 95/98, these functions are not discussed.

22.5.1 Pattern Brush Transfer

A pattern brush transfer consists of transferring the pattern in the current brush into a bitmap. The PatBlt() function is used in this case. If the PATCOPY raster operation code is selected, as is usually the case, the brush pattern is copied to the destination bitmap. If the PATINVERT raster operation code is used, then the brush and the destination bitmap are combined by performing a boolean XOR operation. The remaining raster operation codes that are documented for the PatBlt() function with symbolic names (DSTINVERT, BLACKNESS, and WHITENESS) ignore the brush and are useless in a pattern block transfer. The raster operation performed by PatBlt() is a binary one since it combines a pattern and a destination. In theory, any of the raster operations codes listed in Appendix B that do not have a source operand can be used in PatBlt(), although it may be difficult to find a useful application for most of them.

Note that there is a not-so-subtle difference between a rectangle filled with a pattern brush, and a bitmap created by means of a pattern transfer. Although the results can be made graphically identical by drawing the rectangle with a NULL pen, the possibilities of further manipulating and transforming a bitmap are not possible with a figure created by means of a GDI drawing function.

The following code fragment creates a blank bitmap in a memory device context and fills it with a pattern brush. Since the processing is based on functions already discussed, the code listing needs little comment.

```
static HBITMAP     brushBM1;  // Handle to a bitmap
static HBRUSH      patBrush;  // Handle to a brush
.
.
// Create the brush pattern bitmap from a resource
brushBM1 = LoadBitmap (pInstance,
                       MAKEINTRESOURCE (IDB_BITMAP5);
// Create a pattern brush
patBrush = CreatePatternBrush (patBM1);
// Create a memory device context
aMemDC = CreateCompatibleDC (NULL);   // Memory DC
mapMode = GetMapMode (hdc);           // Obtain mapping mode
SetMapMode (aMemDC, mapMode);         // Set memory DC
                                      // mapping mode
// Create the bitmap
bmBlank = CreateCompatibleBitmap (hdc, 300, 300);
oldObject = SelectObject (aMemDC, bmBlank);
// Select the pattern brush into the memory DC
SelectBrush (aMemDC, patBrush);
// Blit the pattern onto the memory DC
PatBlt (aMemDC, 0, 0, 300, 300, PATCOPY);
// Display the bitmap
BitBlt(hdc, 50, 50, 300, 300, aMemDC, 0, 0, SRCCOPY);
// Clean-up
SelectObject (aMemDC, oldObject);
DeleteDC (aMemDC);
DeleteObject (bmBlank);
```

The demonstration program named Bitmap Demo, in the book's software package, displays a pattern bitmap using code very similar to the one listed.

22.5.2 Bitmap Stretching and Compressing

Occasionally, an application must fit a bitmap into a destination rectangle that is of different dimensions, and even of different proportions. In order to do this, the source bitmap must be either stretched or compressed. One possible use of bitmap stretching or compressing is adapting imagery to a display device that has a different aspect ratio than the one for which it was created. The method can also be used to accommodate a bitmap to a resizable window, as well as for producing intentional distortions, such as simulating the effect of a concave or convex mirror, or other special visual effects.

The StretchBlt() function, one of the more elaborate ones in the API, allows stretching or compressing a bitmap if this is necessary to fit it into a destination rectangle. StretchBlt() is a variation of BitBlt(); therefore, it is used to stretch or compress and later display the resulting bitmap. StretchBlt() is also used to reverse (vertically) or invert (horizontally) a bitmap image. The stretching or compressing is done according to the stretching mode attribute selected in the device context. The stretch mode is selected by means of the SetStretchBltMode() function, which has the following general form:

```
int SetStretchBltMode(
         HDC hdc,            // 1
         int iStretchMode    // 2
);
```

The first parameter is the handle to the device context to which the stretch mode attribute is applied. The second parameter is a predefined constant that corresponds to one of four possible stretching modes. All stretching modes have an old and a new name. Table 22-4, on the following page, lists and describes the stretching modes. The new, preferred names are listed first.

Note that, on many systems, the entire discussion on stretch modes is purely academic, since Microsoft has reported a Windows 95 bug in which the StretchBlt() function always uses the STRETCH_DELETESCANS mode, no matter which one has been selected by means of SetStretchBltMode(). The Microsoft Knowledge Base article describing this problem is number Q138105. We have found no other Microsoft Knowledge Base update regarding this matter.

The actual stretching or compression of the bitmap is performed by means of the StretchBlt() function. The function's general form is as follows:

```
BOOL StretchBlt(
         HDC hdcDest,       // 1
         int nXOriginDest, // 2
         int nYOriginDest, // 3
         int nWidthDest,   // 4
         int nHeightDest,  // 5
         HDC hdcSrc,       // 6
         int nXOriginSrc,  // 7
         int nYOriginSrc,  // 8
         int nWidthSrc,    // 9
```

```
        int nHeightSrc,   // 10
        DWORD dwRop       // 11
);
```

Table 22-4

Windows Stretching Modes

NAME	DESCRIPTION
STRETCH_ANDSCANS BLACKONWHITE	Performs a logical AND operation using the color values for the dropped pixels and the retained ones. If the bitmap is a monochrome bitmap, this mode preserves black pixels at the expense of whiteones.
STRETCH_DELETESCANS COLORONCOLOR	Deletes the pixels. This mode deletes all dropped pixel lines without trying to preserve their information. This mode is typically used to preserve color in a color bitmap.
STRETCH_HALFTONE HALFTONE	Maps pixels from the source rectangle into blocks of pixels in the destination rectangle. The average color over the destination block of pixels approximates the color of the source pixels. Windows documentation recommends that after setting the HALFTONE stretching mode, an application must call the SetBrushOrgEx() function in order to avoid brush misalignment.
STRETCH_ANDSCANS WHITEONBLACK	Performs a logical OR operation using the color values for the dropped and preserved pixels. If the bitmap is a monochrome bitmap, this mode preserves white pixels at the expense of black ones.

The first parameter is the destination device context and the sixth parameter is the source device context. The second and third parameters are the x- and y-coordinates of the upper-left corner of the destination rectangle. The fourth and fifth parameters are the width and height of the destination rectangle. The seventh and eighth parameters are the x- and y-coordinates of the upper-left corner of the source rectangle. The ninth and tenth parameters are the width and height of the source rectangle. The eleventh parameter is one of the ternary raster operation codes listed in Table 22-2 and in Appendix B.

Although the function's parameter list is rather large, it can be easily simplified. Parameters 1 through 5 are the handle to the device context and the location and size of the destination rectangle. Parameters 6 through 10 contain the same information in regards to the source rectangle. The last parameter defines the raster operation code, which is usually set to SRCCOPY.

If the source and destination width parameters have opposite signs, the bitmap is flipped about its vertical axis. In this case the left side of the original bitmap is displayed starting at the right edge. If the source and destination height parameters have opposite signs the image is flipped about its horizontal axis. If both, the width and the height parameters have opposite signs, the original bitmap is flipped about both axes during the transfer. Figure 22-7 shows the image changes in each case.

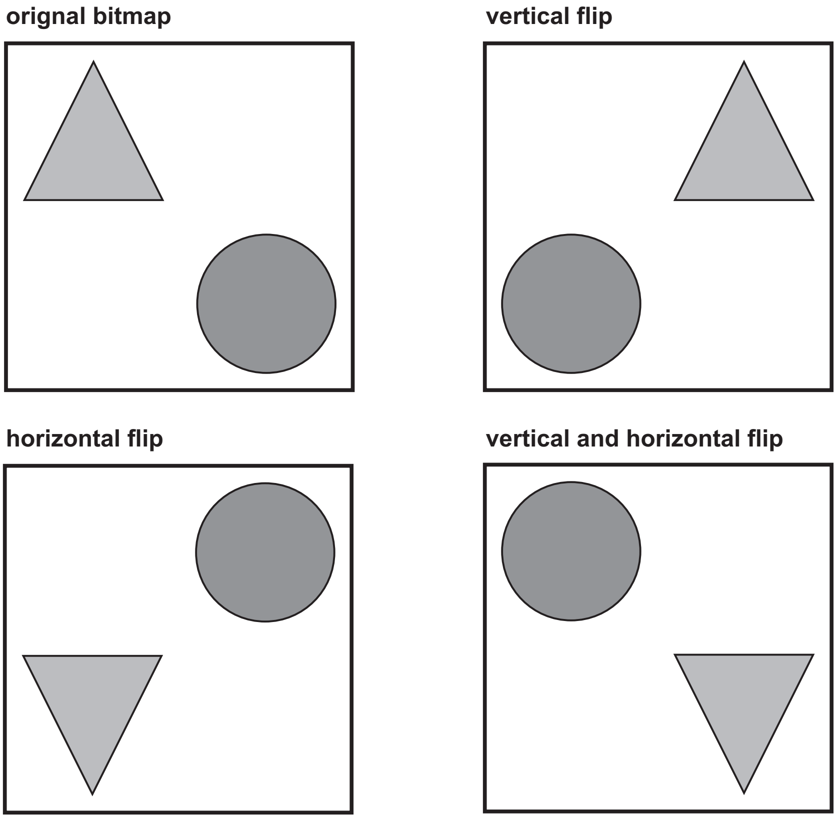


Figure 22-7 *Horizontal and Vertical Bitmap Inversion with StretchBlt()*

The following example takes an existing bitmap and stretches or compresses it to fit the size of the client area. Code assumes an existing bitmap resource named IDB_BITMAP2. The code starts by creating a bitmap from the resource and storing its dimensions in a structure variable of type BITMAP. The dimensions of the client area, which serves as a destination bitmap, are also retrieved and stored in a structure variable of type RECT.

```
BITMAP              bm;       // Storage for bitmap data
RECT                rect;     // Client area dimensions
static  HBITMAP     hstScope; // Handle for a bitmap
.  .  .
// Create bitmap from resource
hstScope = LoadBitmap (pInstance, MAKEINTRESOURCE (IDB_BITMAP2);
// Get bitmap dimensions into BITMAP structure variable
GetObject (hstScope, sizeof(BITMAP), &bm);
// Get client area dimensions
GetClientRect (hwnd, &rect);
```

The StretchBlt() function requires two device contexts: one for the source bitmap and another one for the destination. In this case we create a memory device context and select the bitmap into it. This device context is the source rectangle. The code also sets the stretch mode.

```
aMemDC = CreateCompatibleDC (hdc);
SelectObject (aMemDC, hstScope);
SetStretchBltMode (hdc, STRETCH_DELETESCANS);
```

All that is left is to display the bitmap using StretchBlt(). Once the bitmap is blitted, the destination device context can be deleted. If the bitmap is not necessary, it can also be erased at this time.

```
StretchBlt(hdc,                 // Destination DC
           0, 0, rect.right, rect.bottom,  // dest. dimensions
           aMemDC,              // Source DC
           0, 0, bm.bmWidth, bm.bmHeight,  // Source dimensions
           SRCCOPY);
DeleteDC (aMemDC);
```

22.6 Bitmap Demonstration Program

The program named BMP_DEMO, located in the Bitmap Demo project folder of the book's software package, is a demonstration of the bitmap operations and functions discussed in this chapter. The Operations menu contains commands that correspond to all the bitmap programming primitives, manipulations, and transformations discussed in the text.

Part IV

DirectX Graphics

Chapter 23

Introducing DirectX

Topics:

- Why was DirectX created
- 2D and 3D graphics
- DirectX components and features
- Obtaining and installing the DirectX SDK
- Overview of the DirectX software components
- Testing the SDK installation

In this chapter we start our discussion about graphics programming with DirectX. We begin the chapter with a short review of Microsoft's reasons for creating the DirectX package and describe its fundamental features. Then we look at obtaining, installing, and testing the DirectX software.

23.1 Why DirectX?

Computer games and other high-performance graphics programs require interactive processing, animation, and realistic object rendering, all of which rapidly consume CPU cycles and video resources. Game programmers in particular have traditionally pressed the boundaries of machine performance in order to improve the quality of their products.

In the PC world the first computer games were developed in DOS. Because DOS is a single-user single-task operating system, a DOS program can use any operation that is valid in the machine's instruction set. In other words, a DOS application is in total control of the machine hardware: it is “the god of the machine.” Because of this power, a DOS program can accidentally (or intentionally) destroy files and resources that are not its own, including the operating system itself.

As the PC evolved into a serious business platform, it became a major concern that an application could destroy code, erase data belonging to other programs, or create havoc with the operating system itself. In the business world a computer environment that is intrinsically unsafe is intolerable. Who would ask a client to trust its valuable business information and processing operations to such a machine? If the PC were to be used in business, this situation had to be resolved.

The problem first had to be addressed in hardware. An operating system capable of providing a safe and reliable environment requires hardware components that support this protection. The 286 was the first Intel microprocessor that came equipped with such hardware features. The 286 CPU allows the operating system to detect and prevent access to restricted memory areas and to disallow instructions that are considered dangerous to the integrity of other programs, or to the environment's stability. These special features made possible an operating system environment generically called “protected mode.” Protected mode functions were expanded and enhanced in the 386, the 486, and in the various versions of the Pentium.

In the mid eighties Microsoft and other companies started developing PC operating systems that would execute in protected mode. The results were several new operating systems, of which Windows has been the only major survivor. Although safer and more reliable, Microsoft Windows imposes many restrictions on applications. In the original versions of Windows, games and other high-performance graphics applications could not access the hardware resources directly. This resulted in applications with much less performance and limited functionality. The natural consequence of this situation was that game programmers continued to exercise the craft in DOS. Windows users had to switch to the DOS mode in order to run games, simulations, and other high-end graphical programs. In the PC this state-of-affairs created a major contradiction: Windows was a graphical operating system in which graphics applications would execute with marginal performance.

Microsoft attempted to remedy the situation by providing programmers with limited access to hardware and system resources. The goal was to allow applications sufficient control of video hardware and other resources so as to improve performance and control, and to do it in a way that does not compromise system stability. The first effort in this direction was a product named WinG, in reference to Windows for Games. WinG, which was first made available in 1994, required Windows 3.1 in Win32 mode. WinG's main feature was to allow game programmers to rapidly transfer bitmaps from system memory into video memory. The result was a host of new Windows games that executed with performance comparable to DOS. The immediate success of WinG prompted Microsoft to develop a more elaborate product, called the Game Software Development Kit, or Game SDK.

23.1.1 From the Game SDK to DirectX 8.1

The first version of the Game SDK made evident that the usefulness of direct access to video memory and hardware extended beyond computer games. Many other multimedia applications, and other graphics programs that required high performance could also benefit from these enhanced facilities. Consequently, the new version of the Game SDK was renamed DirectX 2. Other versions later released were named DirectX 3, DirectX 5, DirectX 6, DirectX 7, and currently DirectX 8.1. Notice that no DirectX 4 version exists. DirectX version 8.1 SDK, released in the year 2001, is the one discussed in this book. A beta versions of DirectX 9 was released in May 2002.

The functionality of the DirectX is available to applications running in Windows 95, Windows 98, Windows Me, Windows 2000, and Windows XP. To a limited extent DirectX is also available in the various versions of Windows NT. In the more recent versions of Windows DirectX is furnished as part of the operating system software. This means that applications running under Windows are able to execute programs that use DirectX without the loading of additional drivers or other support software. Each new version of DirectX is provided with a setup utility that allows upgrading a compatible machine.

23.1.2 2D and 3D Graphics in DirectX

In previous versions of the DirectX SDK the 2D graphics interface was referred to as DirectDraw, while 3D graphics were part of the Direct3D interface. Starting with DirectX 8.1 DirectDraw and Direct3D were merged into a single interface. The Microsoft documentation for DirectX 8.1 de-emphasizes the presence of a 2D and a 3D component and refers to both of them as DirectX graphics. Furthermore, most of the DirectX 8.1 SDK documentation and tutorials are about 3D, while the 2D topics, that were previously discussed in great detail, are not included. It is difficult to ponder why 2D graphics topics were excluded from the DirectX 8.1 SDK documentation. One could guess that the 3D element of DirectX has achieved such complexity that a simple matter of space forced the documentation designers to leave out the 2D part. The decision may also be related to the fact that the DirectX 8.1 package includes the DirectX 7 documentation.

As a consequence of this attitude, Microsoft's DirectX 8.1 documentation often equates DirectX graphics with Direct3D, as if 2D graphics no longer existed in DirectX. Whatever reasons Microsoft had for leaving 2D graphics out of the SDK documentation, the fact remains that, in practical programming, 2D graphics cannot be ignored. In the first place, many graphics applications do not required 3D-level modeling or rendering: sophisticated and powerful graphics can be obtained in 2D. Often animations are easier to implement and show better performance in 2D than in 3D graphics. Many successful computer games and other high-level graphics applications are implemented in 2D. Furthermore, most 3D applications rely heavily on 2D graphics for rendering backgrounds, sprites, and other non-3D elements. Another reason for separating 2D and 3D graphics is that the learning curve for 3D graphics is quite steep. 2D provides a reasonable introduction to a complex and sometimes intimidating technology. For all these reasons, in this book we maintain the distinction between the 2D and the 3D components of DirectX.

23.1.3 Obtaining the DirectX SDK

Several versions of the DirectX SDK are available for download, at no cost, on the Microsoft web site located at:

```
http://msdn.microsoft.com/directx
```

DirectX has grown in size during its evolution. The current version (8.1 at the present time) takes up approximately 390 Mb. Downloading the SDK, even in compressed format, can take considerable time online.

23.2 DirectX 8.1 Components

The DirectX 8.1 SDK includes the following components:

- DirectX Graphics combines the DirectDraw and Direct3D components of previous versions of DirectX. This single API can be used for either 2D or 3D graphics programming. DirectX Graphics includes the Direct3DX utility library that simplifies many graphics programming tasks.
- DirectX Audio combines the DirectSound and DirectMusic components of previous DirectX versions. All audio programming is done with this single API.
- DirectPlay makes possible connecting applications over a modem link or a network.
- DirectInput provides support for input devices including joystick, mouse, keyboard, and game controllers. It also provides support for feedback game devices.
- DirectShow provides capture and playback of multimedia streams.
- DirectSetup provides a simple installation procedure for DirectX. It simplifies the updating of display and audio drivers and makes sure that there are no software or hardware conflicts.
- AutoPlay allows creating a CD ROM disk that installs automatically once inserted in the drive. AutoPlay is not unique to DirectX since it is part of the Microsoft Win32 API.

This book is concerned mostly with DirectX graphics. The other components of DirectX are discussed only incidentally.

23.3 New Features in DirectX 8

The DirectX documentation lists the following new features for the SDK:

- Integration of DirectDraw and Direct3D into a single DirectX Graphics component. This approach supposedly makes it easier to use and to support the latest graphics hardware.
- DirectMusic and DirectSound are more integrated. Wave files and other resources can now be loaded by the DirectMusic loader, and played through the DirectMusic performance, synchronized with MIDI notes.
- DirectPlay has been updated to increase its capabilities and improve its ease-of-use. DirectPlay now supports voice communication between players.
- DirectInput introduces a major new feature called action mapping. Action mapping enables you to establish a connection between input actions and input devices. The connection does not depend on the existence of particular device objects.
- DirectShow is now part of DirectX and has been updated.
- You can use the DirectX Control Panel Application to switch between the debug and retail builds of DirectInput, Direct3D, and DirectMusic.

- The DirectX 8.1 SDK includes several new sample programs with the corresponding source code and development tools.

Version 8.1 of DirectX contains the following new features:

- Added new Direct3D samples (cull, lighting, volume fog, self-shadowing and enhanced usage of D3DX in the samples).
- Continued improvement of the D3Dx documentation.
- SDK contains a graphics screensaver framework.
- A MView mesh utility, useful for previewing meshes, normals, etc.
- DirectX AppWizard for Visual C++ v6.0.
- DirectX error lookup tool providing error lookup for Directx 8.x interfaces only. There is also an error lookup function you may use in your application.

The SDK screensaver framework is modeled after the graphics sample framework. It provides multi-monitor support, a feature the standard graphics sample framework does not privide.

23.3.1 Installing the DirectX SDK

DirectX 8.1 contains an installation utility that loads and sets up the software on the target system. Microsoft recommends that any previous versions of the SDK be uninstalled before the setup program is executed, but take into account that only the most recent versions of the DirectX SDK are equipped with uninstall utilities. The SDK installs to a default folder C:\DXSDK. Certain uncommon features of the SDK directory structure are designed for compatibility with Microsoft Developers Network (MSDN) Platform SDK, which duplicates most of DirectX 8.1.

If the SDK is in a CD ROM the installation will begin automatically when the disk is recognized. If not you can execute the install application located in the DirectX main directory. Figure 23-1 shows the initial screen of the DirectX 8.1 installation program.

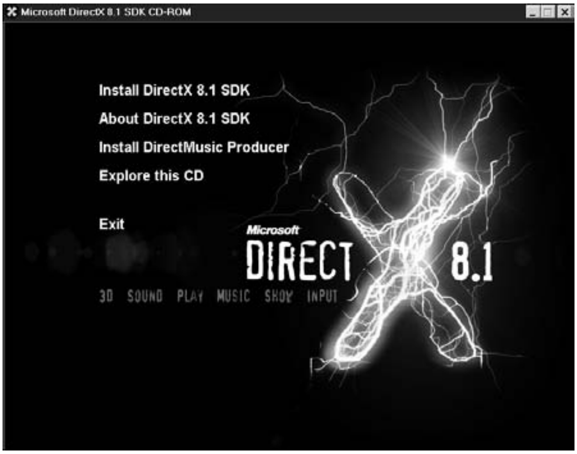
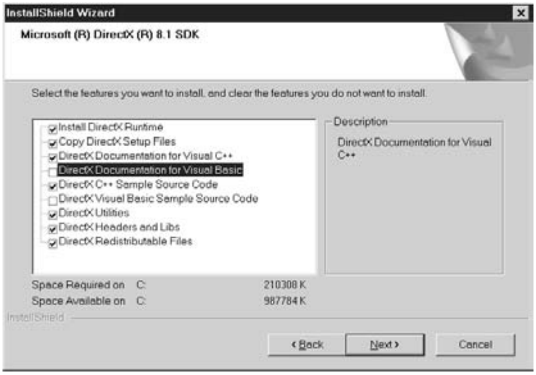

Microsoft DirectX 8.1 SDK CD-ROM
Install DirectX 8.1 SDK
About DirectX 8.1 SDK
Install DirectMusic Producer
Explore this CD
Exit
Microsoft
DIRECT X 8.1
3D SOUND PLAY MUSIC SHOW INPUT

Figure 23-1 *DirectX 8.1 Installation Main Screen*

To install the SDK you double-click on the Install DirectX 8.1 SDK option. The software then presents the Microsoft license agreement, which the user must accept, and continues by offering three installations modes: complete, custom, and runtime only. The first option installs all SDK files in your system and updates the system-level support software. The second option allows choosing the SDK components to be installed in your machine. This option displays a screen containing check boxes for each installation component that can be selected, as shown in Figure 23-3.



InstallShield Wizard

Microsoft (R) DirectX (R) 8.1 SDK

Select the features you want to install, and clear the features you do not want to install.

- ☑ Install DirectX Runtime
- ☑ Copy DirectX Setup Files
- ☑ DirectX Documentation for Visual C++
- □ DirectX Documentation for Visual Basic
- ☑ DirectX C++ Sample Source Code
- □ DirectX Visual Basic Sample Source Code
- ☑ DirectX Utilities
- ☑ DirectX Headers and Libs
- ☑ DirectX Redistributable Files

Description

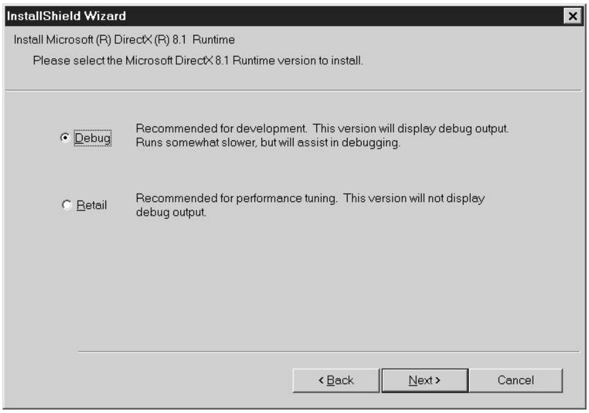
DirectX Documentation for Visual C++

Space Required on	C:	210308 K
Space Available on	C:	987784 K

Figure 23-2 *DirectX 8.1 Custom Installation Screen*

The custom installation option may allow you to save some hard-disk space by excluding components that will not be used. For example, if you do not plan to develop Visual Basic applications that use DirectX you may de-select these options, as in Figure 23-2. If options not originally installed are needed later, you may run the custom installation again.

As the DirectX installation continues, another screen is displayed which offers the option of installing the debug or the retail version of the DirectX dynamic-link libraries (DLLs). This screen is shown in Figure 23-3.



InstallShield Wizard

Install Microsoft (R) DirectX (R) 8.1 Runtime

Please select the Microsoft DirectX 8.1 Runtime version to install.

● Debug — Recommended for development. This version will display debug output. Runs somewhat slower, but will assist in debugging.

○ Retail — Recommended for performance tuning. This version will not display debug output.

Figure 23-3 *DirectX 8.1 Retail or Debug Runtime Selector*

Selecting the debug version installs both debug and retail DLLs on your system. The debug DLLs have additional code that displays error messages while your program is executing. In this case errors are described to a greater detail. On the other hand, the debug DLLs execute more slowly than the retail DLLs. Programmers working in Visual C++ can configure their system so that debug output is displayed in a Window, in a second monitor, or even in another computer. You can toggle between DirectX retail and debug system components in Direct3D and DirectInput by selecting the corresponding box in the DirectX Properties dialog box. This utility is activated by clicking the DirectX icon in the Windows Control Panel.

23.3.2 Compiler Support

DirectX 7 documentation states that the SDK is compatible with Microsoft Visual C++ version 4.2 and later, as well as with Watcom 11.0 and Borland C Builder 3 and 4. However, documented compiler support in DirectX 8.1 is limited to Visual Studio 6.0 or higher. All sample programs in DirectX 8.1 were developed with Visual Studio 6.0. Visual C++ 6.0 project files (.dsp) are included in the sample code and demonstration programs contained in the package.

23.3.3 Accessing DirectX Programs and Utilities

You may inspect the various components of the SDK by navigating through the Windows toolbar Start button, selecting Programs, then Microsoft DirectX 8.1 SDK, as shown in Figure 23-4.

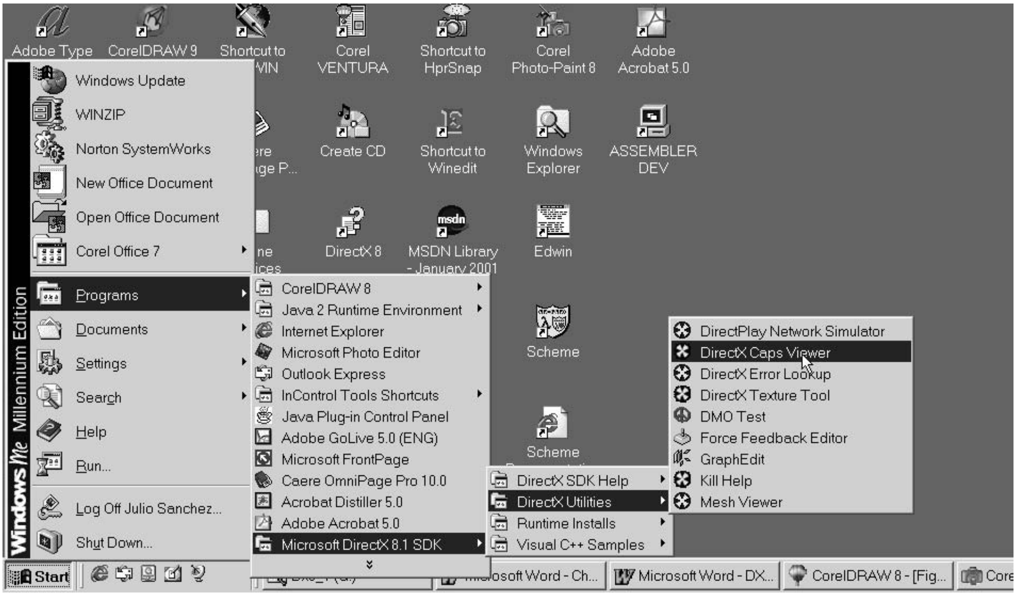
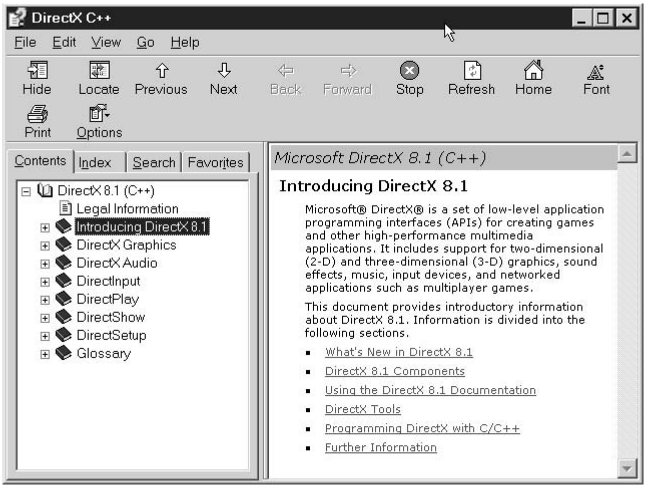


Figure 23-4 *Navigating to the DirectX 8.1 Programs and Utilities*

The DirectX executable files are signaled by the x-shaped DirectX logo or by a custom icon. You can execute the programs by clicking the corresponding icon.

If you are planning on developing DirectX software it may be a good idea to create a desktop item for the DirectX 8.1 documentation utility, which is actually the Windows HTMLHelp viewer. This can be accomplished by right-clicking and dragging the item named DirectX Documentation (Visual C++) from the program list onto the desktop. When the right mouse button is released, a menu box with several options is displayed. Select the option labeled Create Shortcut(s) Here. Figure 23-5 shows the DirectX 8.1 documentation using the HTMLHelp viewer utility.



Microsoft DirectX 8.1 (C++)

Introducing DirectX 8.1

Microsoft® DirectX® is a set of low-level application programming interfaces (APIs) for creating games and other high-performance multimedia applications. It includes support for two-dimensional (2-D) and three-dimensional (3-D) graphics, sound effects, music, input devices, and networked applications such as multiplayer games.

This document provides introductory information about DirectX 8.1. Information is divided into the following sections.

- What's New in DirectX 8.1
- DirectX 8.1 Components
- Using the DirectX 8.1 Documentation
- DirectX Tools
- Programming DirectX with C/C++
- Further Information

Figure 23-5 *DirectX 8.1 Documentation Utility*

Table 23-1 lists the directory layout of the DirectX 8.1 CD ROM.

Table 23-1

DirectX 8.1 CD ROM Directory Layout

\Bin
\DXUtils
High level DirectX applications & tools.

\AppWizard
DirectX 8.1 application Wizard that can be plugged
into Microsoft Visual C++ 6.0. The AppWizard creates a minimal
C++ template application that optionally integrates Direct3D,
DirectInput, DirectMusic, DirectSound, and DirectPlay to work.

\Doc
Reference documentation for the DirectX 8.1 APIs. This
documentation must be viewed with Windows HTMLHelp.

\Essentls
\DMusProd (DirectMusic Producer)
The authoring tool for DirectMusic. Allows composers and sound
designers to use the interactive and variable resources of
DirectMusic along with the consistent sound performance of DLS.
The DirectMusic Producer setup program and all files are located
here.

(continues)

Table 23-1

DirectX 8.1 CD ROM Directory Layout (continued)

\Extras
\Direct3D
Skinning exporter tools
\DirectShow

\DVDBoilerplate
Contains additional media that can be used with
DirectShow and the DirectShow Editing Services (DES) interfaces.
\Documentation
DirectX 7 HTMLHelp Documentation for English and Japanese.
Also contains DirectX 7 Documentation in Microsoft Word format.
\Symbols

Directories of DirectX 8.1 symbol files for Win9x, Win2000 and
WinXP (retail and debug for each).
\Include
DirectX 8.1 include files for DirectX core components.

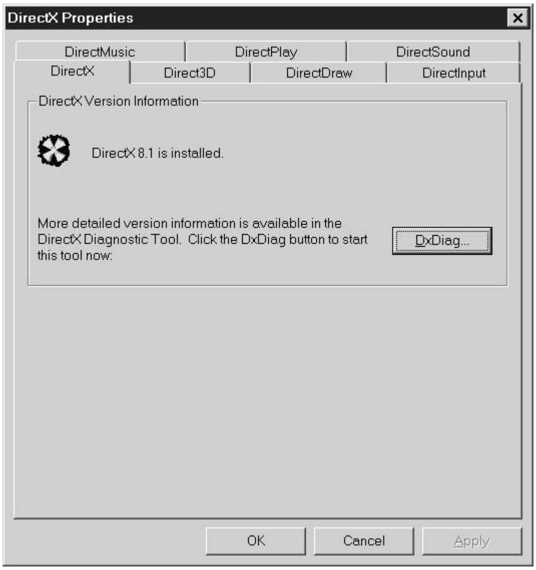
\Lib
DirectX 8.1 library files for DirectX core components.

\License
Text versions of the DirectX SDK and End User License
Agreements and the Redistributable License Agreement.
\Redist
Redistributable versions of the DirectX 8.1 Runtime.

\Samples
Sample code and sample binaries. Most samples can be
accessed from the Start menu when installed via the downloaded
InstallShield setup.
\SDKDev
Contains the runtime installs that are installed with the SDK.
They are English only and contain both debug and retail
DirectX 8.1 system components that can be "switched" between
retail and debug without reinstalling.
\Suppport
Contains support tools required for the SDK installation.
The folder can be deleted following installation.
\System32
Contains support tools required for the SDK installation. This
folder can also be deleted following installation.

23.4 Testing the Installation

The DirectX SDK contains several diagnostic tools that provide information about the DirectX components installed in the system and tests that the various DirectX components are working properly. The easiest way to access the diagnostic utility is by double-clicking on the DirectX propeller-shaped icon in the Windows Control Panel. The DirectX diagnostic program, named directx, is located in the \bin\DXUtils folder which is located in the DirectX installation directory, by default named DXSDK. Alternatively, the program can be executed by clicking the directx program icon. Figure 23-6 shows the initial screen of the DirectX Properties Dialog box.



DirectX 8.1 is installed.

More detailed version information is available in the DirectX Diagnostic Tool. Click the DxDiag button to start this tool now:

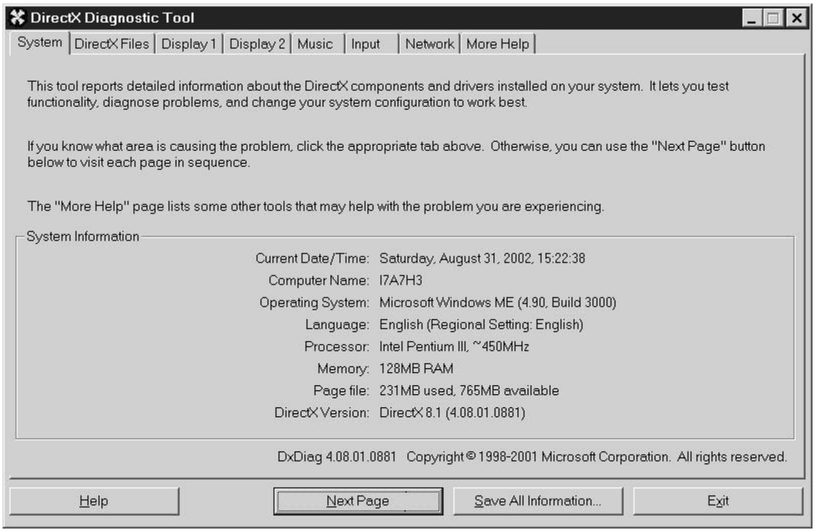
Figure 23-6 *DirectX Properties Dialog Box*

The Properties Dialog box access to the following DirectX components:

- DirectMusic
- DirectPlay
- DirectSound
- DirectX
- Direct3D
- DirectDraw
- DirectInput

You can move to the different components by clicking the tabs. The DirectX Properties Dialog contains a button labeled DxDiag... (see Figure 23-6) which activates the diagnostic function. It is the diagnostic utility that provides the most information about the DirectX API components and drivers installed on the system. It also enables you to test the system capabilities and to selectively enable and disable some hardware acceleration features. The information provided by the diagnostic tool can be saved to a text file for later reference. Clicking the DxDiag... button produces the screen shown in Figure 23-7, on the following page.

The display screen of the DirectX Diagnostic Tool utility changes according to the system configuration. The one shown in Figure 23-10 corresponds to a machine equipped with two video systems. Information regarding one of them is found in the Display1 tab, and the other one in the Display2 tab. By clicking the Next Page button you can visit each tab page in succession. Figure 23-8, on the following page, shows the Display1 screen in one of the author's machines.



This tool reports detailed information about the DirectX components and drivers installed on your system. It lets you test functionality, diagnose problems, and change your system configuration to work best.

If you know what area is causing the problem, click the appropriate tab above. Otherwise, you can use the "Next Page" button below to visit each page in sequence.

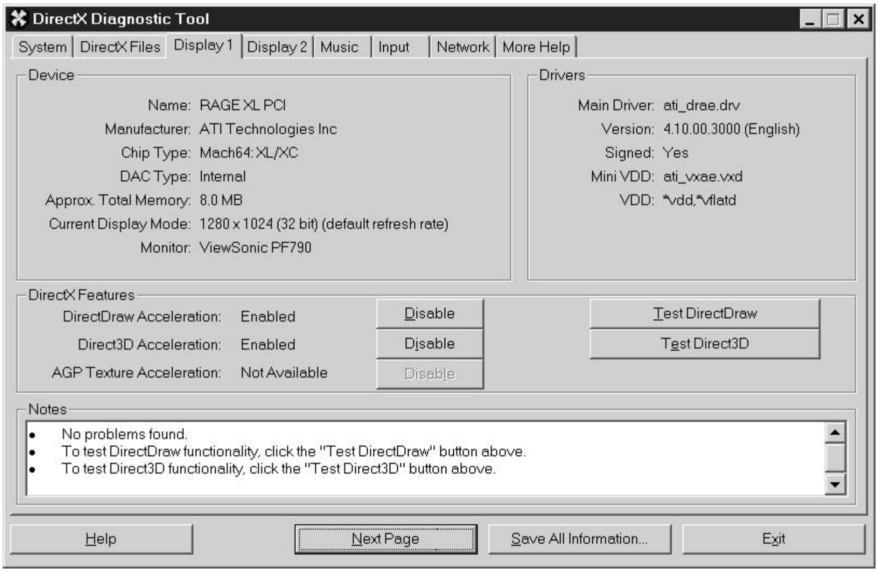
The "More Help" page lists some other tools that may help with the problem you are experiencing.

System Information

Current Date/Time: Saturday, August 31, 2002, 15:22:38
Computer Name: I7A7H3
Operating System: Microsoft Windows ME (4.90, Build 3000)
Language: English (Regional Setting: English)
Processor: Intel Pentium III, ~450MHz
Memory: 128MB RAM
Page file: 231MB used, 765MB available
DirectX Version: DirectX 8.1 (4.08.01.0881)

DxDiag 4.08.01.0881 Copyright © 1998-2001 Microsoft Corporation. All rights reserved.

Figure 23-7 *DirectX Diagnostic Utility*



Device

Name: RAGE XL PCI
Manufacturer: ATI Technologies Inc
Chip Type: Mach64: XL/XC
DAC Type: Internal
Approx. Total Memory: 8.0 MB
Current Display Mode: 1280 x 1024 (32 bit) (default refresh rate)
Monitor: ViewSonic PF790

Drivers

Main Driver: ati_drae.drv
Version: 4.10.00.3000 (English)
Signed: Yes
Mini VDD: ati_vxae.vxd
VDD: *vdd,*vflatd

DirectX Features

DirectDraw Acceleration: Enabled
Direct3D Acceleration: Enabled
AGP Texture Acceleration: Not Available

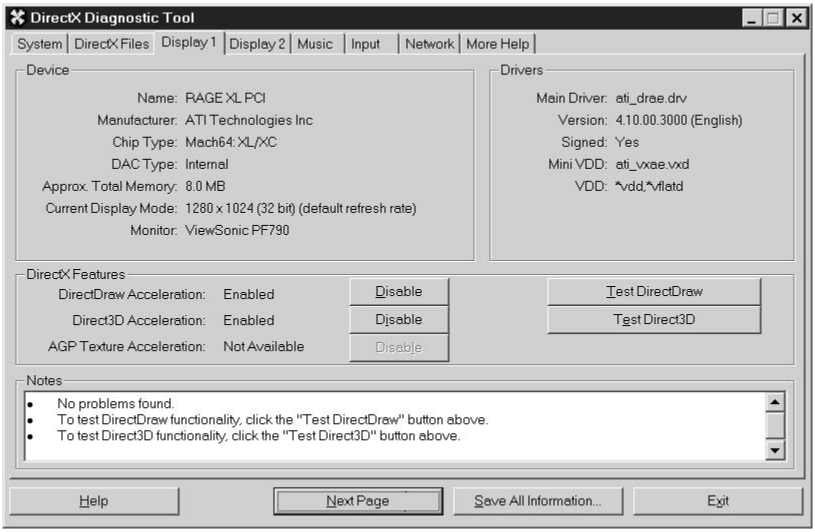
Notes

- No problems found.
- To test DirectDraw functionality, click the "Test DirectDraw" button above.
- To test Direct3D functionality, click the "Test Direct3D" button above.

Figure 23-8 *DirectX Diagnostic Utility Display Test*

The Display function of the DirectX Diagnostic Tool provides information about the display device, the installed drivers, and the DirectX features available in the hardware. It also provides tests for the supported hardware features. Notice that the machine tested does not support AGP (Accelerated Graphics Port) texture acceleration.

Clicking the corresponding buttons allows testing DirectDraw and Direct3D functionality in the hardware. Figure 23-9 shows the results of the DirectDraw test on the same machine.



Name: RAGE XL PCI
Manufacturer: ATI Technologies Inc
Chip Type: Mach64: XL/XC
DAC Type: Internal
Approx. Total Memory: 8.0 MB
Current Display Mode: 1280 x 1024 (32 bit) (default refresh rate)
Monitor: ViewSonic PF790

Main Driver: ati_drae.drv
Version: 4.10.00.3000 (English)
Signed: Yes
Mini VDD: ati_vxae.vxd
VDD: *vdd,*vflatd

DirectDraw Acceleration: Enabled
Direct3D Acceleration: Enabled
AGP Texture Acceleration: Not Available

- No problems found.
- To test DirectDraw functionality, click the "Test DirectDraw" button above.
- To test Direct3D functionality, click the "Test Direct3D" button above.

Figure 23-9 *Testing DirectDraw Functionality*

The first note in the bottom window shows that no problems were detected with the installed drivers or the hardware features. The Test Direct3D button performs similar functions for the 3D features. It is usually a good idea to run all available hardware tests, especially if the system is to be used in DirectX software development.

Chapter 24

DirectX and COM

Topics:

- Fundamentals of object orientation
- Review of C++ indirection
- COM in DirectX
- Creating the COM object
- Accessing the COM Object

The first hurdle in learning DirectX programming relates to understanding Microsoft's Component Object Model (COM). The COM is a foundation for an object oriented system, at the operating system level, which supports and promotes the reuse of interfaces. COM originated as a support for Windows object linking and embedding (OLE). The COM has often been criticized as being difficult to understand and use. But as DirectX programmers we have no choice in the matter: DirectX is based on COM.

The DirectX programmer deals with the COM only superficially. It is not necessary for the programmer to know how to implement COM functionality, but just how to use it. But even then, understanding COM requires knowledge of some of the fundamentals of object orientation and some notion of C++ indirection. We start with a review of these concepts.

24.1 Object Orientation and C++ Indirection

This section is intended as a review of some C++ concepts on which the COM is based. It can be skipped if you are already familiar the basics of object orientation, as well as with pointers, double indirection, and virtual functions.

24.1.1 Indirection Fundamentals

One of the most unique features of C and C++ is their extensive use of indirection. Other programming languages, such as Pascal, Ada, and PL/I, implement pointer variables, however, C and C++ do so in a unique way. This uniqueness is particularly evident in how C and C++ treat pointers to void, pointers to functions, pointers to objects, and pointers to pointers. In C++ the use of pointers is necessary in implementing inheritance and runtime polymorphism. Only with a thorough understanding of pointers will the C++ programmer be able to take advantage of all the power and flexibility of object orientation.

The following short program uses pointers to transfer a string from one buffer into another one.

```
#include <iostream.h>
main(){

  char buffer1[] = "This is a test";  // buffer1 is initialized
  char buffer2[20];                   // buffer2 is reserved
  char* buf1_ptr;                     // One pointer variable
  char* buf2_ptr;                     // A second pointer variable

// Set up pointers to buffer1 and buffer2. Note that since array
// names are pointer constants, we can equate the pointer variables
// to the array name. However, we cannot say: buf1_ptr = &buffer1;
  buf1_ptr = buffer1;
  buf2_ptr = buffer2;

// Proceed to copy buffer1 into buffer2
  while (*buf1_ptr) {
    *buf2_ptr = *buf1_ptr;     // Move character using pointers
    buf1_ptr++;                // Bump pointer to buffer1
    buf2_ptr++;                // Bump pointer to buffer2
    }
  *buf2_ptr = NULL;          // Place string terminator

// Display both buffers to check program operation
  cout << "\n\n\n"
       << buffer1 << "\n"
       << buffer2 << "\n\n";
  return 0;
}
```

The code has several peculiarities, for example, the statements

```
  char* buf1_ptr;
  char* buf2_ptr;
```

declare that buf1_prt and buf2_ptr are pointer variables to variables of type char. If the statements had been:

```
  char *buf1_ptr;
  char *buf2_ptr;
```

the results would have been identical since C and C++ allow this and other syntax variations. Placing the asterisk close to the data type seems to emphasize that the pointer is a pointer to a type. However, if we were to initialize several variables simultaneously we would have to place an asterisk before each variable name, either in the form:

```
  char* buf_ptr1, * buf_ptr2;
```

or in the form:

```
  char *buf_ptr1, *buf_ptr2;
```

Either syntax seems to favor the second style.

Once a pointer variable has been created, the next step is to initialize the pointer variables (in this case buf_ptr1 and buf_ptr2) to the address of the first byte of the data areas named buffer1 and buffer2. Someone familiar with the use of the & operator to obtain the address of a variable may be tempted to code:

```
buf1_ptr = &buffer1;      // INVALID FOR ARRAYS
```

However, in C and C++ an array name is an address constant. Therefore, this expression is illegal for arrays, but legal and valid for any other data type. In the case of an array we must initialize the pointer variable with an expression such as:

```
buf1_ptr = buffer1;
```

We again overload the * symbol when we need to address the characters pointed to by the pointer variable. As is the case in the expressions:

```
while (*buf1_ptr)
{
        *buf2_ptr = *buf1_ptr;
        buf1_ptr++;
        buf2_ptr++;
}
*buf2_ptr = NULL;
```

The process of accessing the value of the target variable by means of a pointer is called dereferencing. The asterisk symbol (*) is also used in dereferencing a pointer. In this case it precedes the name of the pointer variable and is sometimes called the indirection operator. For example, if prt1 is a pointer to the integer variable var1, which holds the value 22, then the following statement displays this value:

```
cout << *ptr1;
```

24.1.2 Pointers to Pointers

The concept of double indirection is familiar to most C/C++ programmers. In this case a pointer variable is used to hold the address of another pointer variable. For example:

```
int value1 = 20;
```

We can now create and initialize a pointer to the variable value1:

```
int *ptr = &value1;
```

and another pointer variable to hold the address of the first pointer:

```
int **pptr = &ptr;
```

Now the value of the variable value1 can be accessed directly or by dereferencing the pointer variable or the pointer to a pointer variable:

```
value1 = 50;
```

or

```
*ptr = 55;
```

or

```
**pptr = 60;
```

In Hungarian notation the prefix p is usually assigned to simple pointer variables and pp to pointers to pointers.

In COM functions are accessed through a pointer to an interface. To invoke the function you use essentially the same syntax that you would to invoke a pointer to a C++ function. For example, to invoke the IAnInterface::DoIt you would use the following syntax.

```
IAnInterface *pAnIface;
...
pAnIface->DoIt(...);
```

The need for a second level of indirection results from the fact that to use a function you must first obtain an interface pointer. In order to do so, you declare a variable as a pointer to the desired interface, and pass the address of the pointer variable to the method. In other words, what you pass to the method is the address of a pointer. When the method returns, the variable (of type pointer to pointer) will point to the requested interface. You use this pointer to call any of the interface's functions.

24.1.3 Pointers to Functions

The Intel x86 family of microprocessors supports indirect jumps and calls. Indirect access to code is achieved either through a register or memory operand, or through both simultaneously. One commonly used technique of indirect access to code is by means of a memory table which holds a set of addresses to various routines. An offset value is added to the address of the start of the table to determine the destination for a particular jump or call.

C++ implements code indirection by means of pointers to functions. Since a function address is its entry point, this address can be stored in a pointer and used to call the function. When these addresses are stored in an array of pointers, then the resulting structure is called a call table. Jump and call tables are sometimes called dispatch tables by C and C++ programmers.

The implementation of pointers to functions and dispatch tables in C and C++ requires a special syntax. In the first place, a pointer to a function has a type that corresponds to the data type returned by the function and is declared inside parentheses. For example, to declare a function pointer named fun_ptr, which receives two parameters of int type in the variables named x and y, and returns void, you would code:

```
void (*fun_ptr) (int x, int y);
```

In this special syntax, the parentheses have the effect of binding to the function name, not to its data type. If you were to remove the parentheses, the result would be a pointer to a function that returns type void. Note that the previous line creates a function pointer that is not yet initialized. This pointer can be set to point to any function that receives two int-type parameters and returns void. For example, if there was a function named Fun1 with these characteristics we could initialize the function pointer with the statement:

```
fun_ptr = Fun1;
```

C and C++ compilers assume that a function name is a pointer to its entry point, thus the address of (&) operator is not used. Once the function pointer is initialized, we can access the function Fun1 with the statement:

```
(*Fun1)(6, 8);
```

In this case we are passing to the function the two integer parameters, in the conventional manner.

24.1.4 Polymorphism and Virtual Functions

Run-time polymorphism is also called late or dynamic binding. This topic is at the core of object-oriented programming since it provides a powerful mechanism for achieving several very desirable properties: reusability, isolation of program defects, and the component-based architecture previously discussed. The fundamental notion of dynamic binding is that the method to be executed is determined when the program runs, not when it is compiled. Suppose a class hierarchy which includes a base class named B and several derived classes named D1, D2, and D3 respectively. Also assume that there is a method named M() in the base class, which is inherited and perhaps modified in the derived classes. We now implement a pointer named ptr to the method in the base class. In C++ we can access this method by means of the statement:

```
ptr-> M();
```

However, in dynamic binding terms this does not imply that the method of the base class is forcefully executed. Instead, which method is used depends on the object referenced by the pointer variable. If ptr is currently pointing to method M() in class D2, then it is this implementation of M() that is executed, not the one in the base class.

In most modern object-oriented languages, methods are dynamically bound by default. This is not the case with C++, where methods are statically bound by default. Dynamic binding in C++ is accomplished by means of virtual functions. A virtual function is declared in the base class and redefined in one or more derived classes. This means that the function declared virtual in the base class defines a general type of methods and serves to specify the interface. Other functions with the same name and interface can be implemented in the derived classes to override the one in the base class. If the virtual function is accessed by means of its name, it behaves as any other function. However, when a function declared virtual in the base class is accessed via a pointer, then the one executed depends on the object which the pointer is referencing.

In C++ a pointer to an object in the base class can be set to point to an object in a derived class. It is this mechanism that allows implementation of dynamic binding in C++. The following short program shows how it is accomplished:

```
//*************************************************************
// Program name: virtual_1.cpp
// C++ program to illustrate virtual functions and run-time
// polymorphism
//*************************************************************

#include <iostream.h>

//****************************
//        classes
//****************************
// Base class
class BaseClass {
public:
   virtual void DisplayMsg() {
     cout << "Method in BaseClass executing\n" ;
   }
};

// A derived class
class DerClass1 : public BaseClass {
public:
   virtual void DisplayMsg() {
     cout << "Method in DerClass1 executing\n" ;
   }
};
// A second derived class
class DerClass2 : public BaseClass {
public:
   virtual void DisplayMsg() {
     cout << "Method in DerClass2 executing\n" ;
   }
};

//****************************
//          main()
//****************************
main() {
   BaseClass *base_ptr;        // Pointer to object of base class
   BaseClass base_obj;         // Object of BaseClass
   DerClass1 der_obj1;         // Object of DerClass1
   DerClass2 der_obj2;         // Object of DerClass2

// Access object of base class using base class pointer
   base_ptr = &base_obj;       // Pointer to base class object
   base_ptr-> DisplayMsg();

// Access object of first derived class using base class pointer
   base_ptr = &der_obj1;       // Pointer to derived class object
   base_ptr-> DisplayMsg();

// Access object of second derived class using base class pointer
   base_ptr = &der_obj2;       // Pointer to derived class object
   base_ptr-> DisplayMsg();

   return 0;
}
```

When the virtual_1 program executes, the following text messages are displayed:

```
Method in BaseClass executing
Method in DerClass1 executing
Method in DerClass2 executing
```

During program execution (run-time) the base class pointer named base_ptr is first set to the base class and the base class method is executed. The coding is as follows:

```
   base_ptr = &base_obj;
   base_ptr-> DisplayMsg();
```

Next, the same pointer is reset to the first derived class and the execution statement is repeated:

```
   base_ptr = &der_obj1;
   base_ptr-> DisplayMsg();
```

The fact that the statement

```
   base_ptr-> DisplayMsg();
```

executes different methods in each case proves that the decision regarding which method executes is made at runtime, not at compile time, since two identical statements generate the same object code. The program virtual_1 uses pointers to access methods in the base and derived classes. A pointer to the base class is redirected to derived classes at run time, thus achieving dynamic binding.

The program virtual_2, listed below, uses an array of pointers to the various functions.

```
//****************************************************************
// Program name: virtual_2.cpp
// C++ program to illustrate virtual functions and run-time
// polymorphism by means of an array of pointers
//****************************************************************

#include <iostream.h>

//****************************
//        classes
//****************************
// Base class
class BaseClass {
public:
   virtual void DisplayMsg() {
     cout << "Method in BaseClass executing\n" ;
   }
};

// A derived class
class DerClass1 : public BaseClass {
public:
   virtual void DisplayMsg() {
     cout << "Method in DerClass1 executing\n" ;
   }
};
// A second derived class
class DerClass2 : public BaseClass {
```

```
public:
   virtual void DisplayMsg() {
     cout << "Method in DerClass2 executing\n" ;
   }
};
//****************************
//          main()
//****************************
main() {

   BaseClass* ptr_list[3];      // Array of 3 pointers
   BaseClass base_obj;          // Object of BaseClass
   DerClass1 der_obj1;          // Object of DerClass1
   DerClass2 der_obj2;          // Object of DerClass2

// Initialize pointer array with objects
   ptr_list[0] = &base_obj;
   ptr_list[1] = &der_obj1;
   ptr_list[2] = &der_obj2;

// Create variable to store user input
int user_input = 0;

// Prompt user for input
   cout << "\nEnter a number from 1 to 3: ";
   cin >> user_input;
// Test for invalid input
   if(user_input < 1 || user_input > 3){
     cout << "\ninvalid input\n" ;
     return 1;
   }

// Index into array of pointers using user input
   ptr_list[user_input - 1]-> DisplayMsg();
  return 0;
}
```

The program virtual_2.cpp selects the method to be executed using the user input as an offset into a pointer array, by means of the statement:

```
   ptr_list[user_input - 1]-> DisplayMsg();
```

The preceding programs (virtual_1.cpp and virtual_2.cpp) both use pointers to access methods in the base and derived classes. In the first program (virtual_1.cpp) a pointer to the base class is redirected to derived classes at run time, thus achieving dynamic binding. In the second program (virtual_2.cpp), three different pointers are stored in an array at compile time, with the statements:

```
   ptr_list[0] = &base_obj;
   ptr_list[1] = &der_obj1;
   ptr_list[2] = &der_obj2;
```

The program then requests input from the user and scales this value to use it as an offset into the pointer array. The selection is done by means of the statement:

```
   ptr_list[user_input - 1]-> DisplayMsg();
```

In the case of the program virtual_2.cpp, although the pointer to be used is not known until program execution, the selection is not based on redirecting a base class pointer at run time. Therefore, it is not a true example of dynamic binding. However, if we eliminate the virtual keyword from the code, then every valid user input brings about the execution of the base version of the DisplayMsg() method. This leads to the conclusion that the virtual keyword is doing something in the code, although it is not producing dynamic binding.

Virtual functions, by themselves, do not guarantee dynamic binding since a virtual function can be accessed by means of the dot operator. For example, if der_obj1 is an object of the class Der1Class, then the statement:

```
  der_obj1.DisplayMsg();
```

executes the corresponding method in this class. However, in this case the virtual attribute is not necessary since the class is bound directly by its object.

The mechanism for selecting among two or more functions of the same name by means of the virtual attribute is called overriding. It is different from the notion of overloaded functions, since overloaded functions must differ in their data types or number of parameters. Overridden functions, on the contrary, must have an identical interface. The prototypes of virtual functions must be identical in the base and in the derived classes. If a function with the same name is defined with a different prototype, then the compiler reverts to overloading and the function is bound statically.

The virtual keyword is not necessary in the derived classes, since the virtual attribute is inherited. For this reason the class definition for DerClass1 could have read as follows:

```
// A derived class
class DerClass1 : public BaseClass {
public:
   void DisplayMsg() {
     cout << "Method in DerClass1 executing\n" ;
   }
};
```

The implicit virtual attribute is inherited; consequently, we may have to trace through the entire inheritance tree in order to determine if a method is virtual or not. For this reason we prefer to explicitly state the virtual attribute since it should not be necessary to refer to the base class to determine the virtual or non-virtual nature of a method.

Virtual functions exist in a class hierarchy that follows the order of derivation. This concept is important since overriding a method in a base class is optional. If a derived class does not contain a polymorphic method, then the next one in reverse order of derivation is used.

24.1.5 Pure Virtual Functions

Virtual functions are implemented in the base class and possibly redefined in the derived classes. However, what would happen if a polymorphic method were not implemented in the base class? For instance, suppose that the class named BaseClass in the program virtual_1.cpp was recoded as follows:

```
class BaseClass
{
public:
   virtual void DisplayMsg();
};
```

In a DOS based C++ compiler, if the method DisplayMsg() was not implemented in the base class, the program would compile correctly but would generate a linker error. That happens because there is no address for the method DisplayMsg() in the base class since the method does not exist. Therefore, the statement:

```
   base_ptr-> DisplayMsg();
```

cannot be resolved by the linker. However, there are occasions in which there is no meaningful definition for a method in the base class. Consider the class structure shown in Figure 24-1.

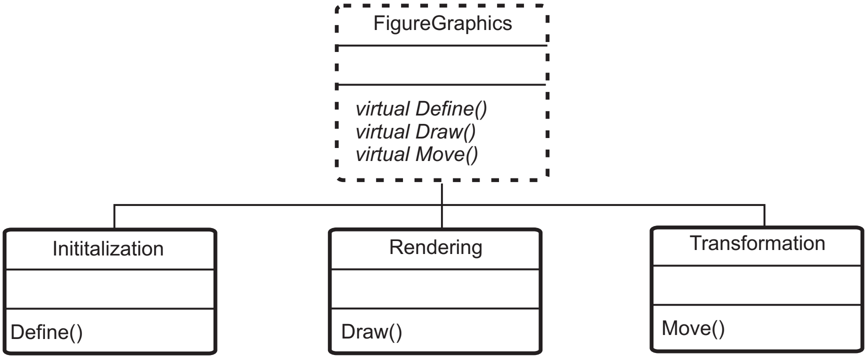


Figure 24-1 *Abstract Class Structure*

In the case of Figure 24-1 there is no possible implementation of the methods Define(), Draw(), and Move() in the base class FigureGraphics. The implementations are left to the subclasses. The method Draw() in the base class serves to define the method name and the interface, but the implementation is left for the derived class or classes: it is a pure virtual function.

In C++ a pure virtual function is declared in the following general form:

```
virtual return-type function-name(parameter-list) = 0;
```

When a function is declared in the this manner, implementation must be provided by all derived classes. A compiler error occurs if any derived class fails to provide an implementation for a pure virtual function. Note that the case of the pure virtual function is quite different from that of non-virtual functions, in which a missing implementation is automatically replaced by the closest one in reverse order of derivation.

Pure virtual functions have two organizational effects. The first one is that the base class serves to define a general interface that sets a model that all derived classes must follow. The second one is that implementation in the derived classes is automatically assured since the code does not compile otherwise.

Abstract Classes

C++ pure virtual functions furnish a mechanism whereby a base class is used to define an interface by declaring the method's parameters and return type, while one or more derived classes define implementations for the specific cases. A class that contains a pure virtual function is designated an abstract class. The abstract class model satisfies the "one interface, multiple methods" approach that is a core notion of object orientation. The programmer is able to create a class hierarchy that goes from the most general to the more specific; from conceptual abstraction to implementation details. The following short program shows a possible use of abstract classes.

```
//***************************************************************
// Program name: virtual_3.cpp
// C++ program to illustrate a pure virtual function
//***************************************************************

#include <iostream.h>

//****************************
//        classes
//****************************
// Abstract base class
class GeoFigure {
private:
   float dim1;             // First dimension
   float dim2;             // Second dimension
public:
   virtual float Area(float, float) = 0; // Pure virtual function
};

// derived class
class Rectangle : public GeoFigure {
public:
   virtual float Area(float x, float y) {
   return (x * y);
   }
};

class Triangle : public GeoFigure {
public:
   virtual float Area(float x, float y) {
   return (x * y)/2;
   }
};

class Circle : public GeoFigure {
public:
   virtual float Area(float x, float y) {
   return (x * x)* 3.1415;
   }
};

//****************************
//          main()
//****************************
```

```
main() {
   GeoFigure *base_ptr;    // Pointer to the base class

   Rectangle obj1;         // Declare objects of derived classes
   Triangle obj2;
   Circle obj3;

// Polymorphically access methods in derived classes
   base_ptr = &obj1;       // Set base class pointer to Rectangle
   cout << "\nRectangle area: " << base_ptr-> Area(5.1, 10);

   base_ptr = &obj2;       // Set base class pointer to Triangle
   cout << "\nTriangle area: " << base_ptr-> Area(3.7, 11.22);

   base_ptr = &obj3;       // Set base class pointer to Circle
   cout << "\nCircle area: " << base_ptr-> Area(3.22, 0);

   return 0;
}
```

In program virtual_3.cpp you can note that the pure virtual function in the base class defines the interface, which must be adhered to by all implementations in the derived classes. In this manner the Area() method in the class Circle must preserve the interface, which passes two parameters, although a single one suffices for calculating the area in the circle case.

In C++ it is not possible to declare an object of an abstract class, even if the class contains other methods that are not virtual. For example, we modify the class GeoFigure in the program virtual_3.cpp as follows:

```
class GeoFigure
{
private:
   float dim1;             // First dimension
   float dim2;             // Second dimension
public:
   float GetDim1() { return dim1; }
   virtual float Area(float, float) = 0; // Pure virtual function
};
```

The class now includes a nonvirtual function named GetDim1(). However, we still cannot instantiate an object of class GeoFigure, therefore the statement:

```
GeoFigure objx;            // ILLEGAL STATEMENT
```

would be rejected by the compiler. However, any method implemented in the base class can be accessed by means of a pointer to an object of a derived class, in which case the C++ rules for inheritance are followed. If a method has a unique name in the base class, then it is executed independently of the object referenced. If the classes constitute a simple inheritance hierarchy, then the selection is based on the rules for overloading. If the classes contain non-pure virtual functions, overriding takes place. If the class is an abstract class, then the derived classes must provide implementations of the method declared to be pure virtual.

Virtual Function Table (vtable)

A class with at least one pure virtual function is an abstract class. The class GeoFigure in the program virtual_3.cpp is an abstract class since it contains the pure virtual function named Area(). In Figure 9-3 the class FigureGraphics is also abstract since it contains three pure virtual functions: Define(), Draw(), and Move(). In C++ the implementation of virtual functions is relatively consistent from one compiler to another one. The mechanism takes advantage of the processor's capability of making a function call indirectly, that is, through a pointer to a function. In implementation, virtual functions use indirect addressing to ensure the following capabilities:

- Function calls are expressed in terms of member functions.
- Each class contains a table of pointers to the implementations.
- The indirect call syntax is the same as calls to any other member function.

For each class with at least one pure virtual function the compiler creates a table of function pointers. This table is known as the vtable, the VTBL, or the v-table. Each entry in the vtable contains the address of a function. For example, the class FigureGraphics in Figure 9-3 could be defined as follows:

```
class FigureGraphics {
     public:
     virtual bool Define(double, double) = 0;
     virtual bool Draw(int, int) = 0;
     virual bool Move(int, int) = 0;
};
```

The C++ compiler now constructs a table that contains a pointer definition for each virtual function in the class. This is the vtable for the class. Figure 24-2 shows the virtual function table.

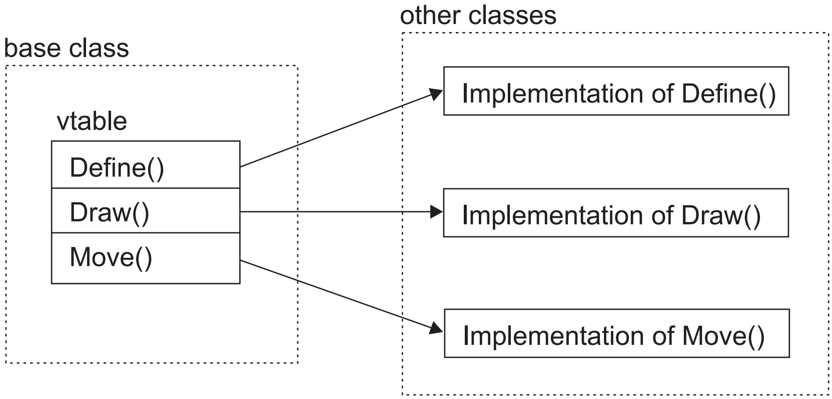


Figure 24-2 *The Virtual Function Table (vtable)*

The vtable exists at the class level. Each class contains a single vtable and every object instantiated from the class is given a reference to the class vtable. This reference, which is one of the object's data members, is implemented as a pointer. Assuming that the pointer to the vtable is named pVtable, then each object's pVtable member points to class' vtable. In regards to the class diagram in Figure 24-1 and 24-2 the code could be as follows:

```
FigureGraphics fig1;     // Declare a class object
FigureGraphics *pFigs;   // Declare a pointer to the base class
pFigs = &fig1;           // Initialize pointer to base class object
pFigs->Define();         // Use pointer to access method
```

Notice that the pointer to the base class serves as a pointer to the vtable. With some additional complications, the virtual function mechanism can be implemented in standard C. The memory layout for the COM implementation of the component-based model is identical to the one used for abstract base classes and virtual functions in C++, as you will see in the sections that follow.

24.2 COM in DirectX Programming

Microsoft's Component Object Model (COM) is a foundation for an object oriented system that attempts to improve on the C++ model. COM is described as an object model at the operating system level, which supports and promotes the reuse of interfaces. The fundamental notion of the COM relates to the idea of a component-based architecture.

24.2.1 COM Fundamentals

In order to understand the COM model consider a conventional program in which all of its elements are defined when the application is compiled and linked. This type of program is based on what has been called a monolithic architecture. If such a program requires to be updated, it must be re-compiled and re-linked. This means that the user will have to be provided with a new copy of the application software with every program update. A more effective model considers the application as a set of individual components. In this new model, called a component-based or component architecture, each program element (or component) behaves as a mini-application. Each component is, in fact, a unit of execution which is compiled and linked independently. The application itself provides the interaction between its various components. Since each component can be replaced independently, the application can be more easily customized and updated. Figure 24-3 graphically represents a monolithic and a component-based application.

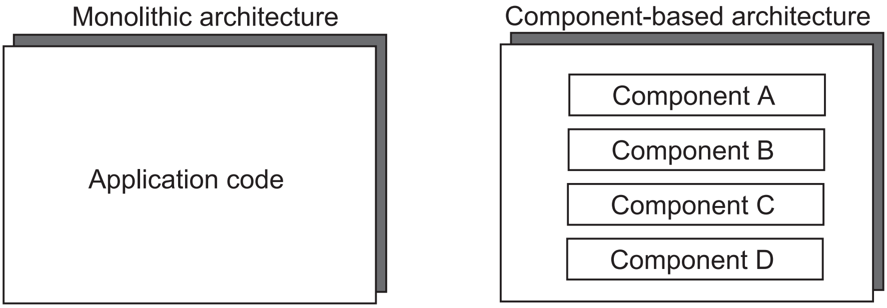


Figure 24-3 *Monolithic and Component-Based Applications*

For the COM model to work we must be able to replace a component without breaking the application. Suppose that Component C in the program shown in Figure 9-3 has become obsolete and must be updated. In this case we must be able to create a new component, say Component C New, that replaces the old Component C. There are two requirements that make possible component replacement:

- Components must link to the application at runtime.
- Components must encapsulate the implementation details.

The first requirement results from the fact that if the components were to link statically with the application, then the code would have to be re-compiled and we would be back into the monolithic model. To have replaceable components it must be possible to replace them at runtime, that is, the binding between the component and the application code must be dynamic. The second requirement is a consequence of the first one. In order to change a component dynamically, the new component must connect with the application exactly in the same manner as the old one. Otherwise the application itself would have to be changed.

One of the golden rules of object orientation is to program to an interface, not to an implementation. In the COM model encapsulation is achieved by creating components with different implementation but identical interfaces. If we refer to a program or component that uses another component as a client, then we can say that a client connects to a component through an interface. If the interface is not changed, then a new component can be designed to replace an old one without breaking the client. By the same token, another client with the same interface can use the same component. Striving to design interfaces that do not change, while hiding the implementation details, is the basic task of the COM programmer. In other words, programming to and interface, not to an implementation.

To isolate the component from the client's implementation details requires several constraints:

- The component must be language neutral. That is, clients should be able to use components coded in any computer language. This implies that a component is a binary entity that is shipped ready-to-use.
- Components must be replaceable without breaking the client.
- Components must be relocatable on a network. That is, to the client a component on a remote system appears identical to a local one.

Defining COM

COM is a protocol for building component-based applications. It defines components that can be dynmically interchanged and clients that can use these components. In this sense COM is a standard. The COM specification document, called the Component Object Model Specification, was developed jointly by Microsoft Corporation and Digital Equipment Corporation in 1995. The document is available on-line from the Microsoft Web site at

```
http://www.microsoft.com/oledev/
```

COM is not a computer language, an API, or a DLL, although COM uses DLLs to implement dynamic linking. On the other hand, COM is not only a formal specification since is does provide some component management services in the form of an API. These services are furnished as a COM library. The purpose of the COM library is to save developers time in the creation of components and clients. In short, COM is a way of writing programs with reusable, replaceable components.

24.2.2 COM Concepts in DirectX

DirectX is presented to the programmer using the COM. From C++, the COM object appears as an abstract class. Access, in this model, is by means of the pointer to the DirectX COM object. When using straight C, the function must pass the pointer to the COM object as an additional parameter. In addition, the call must include a pointer to a property of the COM object called the vtable. Since this book assumes C++ programming, we use the simpler interface to the COM.

The bulk of the Microsoft DirectX run time is in the form of COM-compliant objects. For this reason DirectX developers need to have a basic understanding of COM principles and programming techniques. There are two distinct flavors of COM programming:

- Applications that use existing COM objects.
- Applications that implement new COM objects.

Using existing COM objects by application code is straightforward and uncomplicated. Creating COM objects, on the other hand, is a more complicated matter. DirectX applications do not need to implement COM objects, but must deal with those provided by DirectX. This means that DirectX developers are usually concerned only with the easiest flavor of COM programming.

The COM Object

A COM object can be visualized as a black box that can be used by applications to perform one or more tasks. In DirectX COM objects are always implemented as DLLs. A COM object, like a conventional DLL, contains methods that an application can call to perform a specific task.

COM objects enforce stricter encapsulation that C++ objects. The public functions of a COM objects are grouped into one or more interfaces. To use a function, application code must first create the object and obtain the its interface. Typically,the interface to a COM object contains a related set of methods that provide access to a particular DirectX feature. For example, the IDirect3D8 interface contains methods that allow creating Direct3D objects, setting up the environment, and obtaining device capabilities. Once enable you obtained the interface you can access all its methods, but not those that are not part of IDirect3D8.

Although COM objects are typically contained in a DLL, you do not need to explicitly load the DLL or link to a static library in order to use a COM object. Each COM object has a unique registered identifier that is used to create the object. COM automatically loads the required DLL when the object is referenced.

The COM Interface

The concepts of object and interface are at the core of the COM. Although in casual reference we sometimes refer to an object by the name of its interface, the concepts of object and interface are unique and should not be confused. In this sense it is often said that an object exposes several interfaces. An interface is described as a group of methods that performs a set of related operations. To say that an object exposes an interface is equivalent to stating that, in order to use a particular function, you must first create the object and then obtain the interface.

All COM objects must expose the IUnknown interface, as well as at least one additional interface. Some COM objects expose many interfaces and more than one object might expose the same interface. The interface specifies the syntax of the methods and their general functionality. A highly specialized interface is usually exposed by a single object. Generally useful interfaces are often exposed by many objects. The most generally useful interface, named IUnknown, is exposed by all COM objects.

COM requires that if an object exposes an interface, it must support every method in the interface definition. For this reason you can call any function with the confidence that it exists. How a particular function is implemented may vary from object to object. For example, two object that perform the same calculation may use different algorithms to obtain the result.

The COM specification requires that a published interface must not change once. For this reason it is not possible to add a new function to an existing interface. Instead, you must create a new interface. Although it is not strictly required by the standard, common practice is to have the new interface include all the of the old interface's functions, plus the new function.

Interfaces can be implemented in several generations. Generally all generations of an interface perform essentially the same overall task, but they may differ in implementation details. Often, an object exposes every generation of interface. This allows older applications to continue using the object's older interfaces, while newer applications take advantage of the features of the newer interfaces. Family of interfaces usually have the same name, plus an integer indicating the generation. For example, the original DirectDraw interface was named IDirectDraw. This interface was later updated to IDirectDraw2, IDirectDraw4, and IDirectDraw7. Microsoft typically labels successive generations of DirectX interfaces with the corresponding version number. For this reason the integer identifier may not be a dense set.

The GUID

To insure that every interface is unique it is assigned an identifier, called the IID. Every new version or interface receives its own unique IID. Therefore the IID is permanently linked to the interface. In the COM the IID is a 16-byte (128 bit) structure called the Globally Unique Identifier (GUID). GUIDs are created so that no two GUIDs are the same. COM uses GUIDS extensively for two primary purposes:

- To uniquely identify a COM object.
- To uniquely identify a particular COM interface.

The term IID is used to request a particular interface from an object. An interface's IID will be the same, regardless of which object exposes the interface. DirectX documentation refers to objects and interfaces by a descriptive name, such as IDirect3D8. Although descriptive names are useful, there is no guarantee that another object or interface does not have the same name. The only unambiguous way to refer to a particular object or interface is by its GUID.

Although GUIDs are structures, they are often expressed as an equivalent string. The general format of the string form of a GUID is:

```
{VVVVVVVV-WWWW-XXXX-YYYY-ZZZZZZZZZZZZ}
```

In this format each letter corresponds to a hexadecimal integer. For example, the string form of the IID for the IDirect3D8 interface is:

```
{1DD9E8DA-1C77-4D40-B0CF-98FEFDFF9512}
```

In order to make the GUID identifier more difficult to mistype it is also provided with a name. The customary naming convention is to prefix either IID_ or CLSID_ to the descriptive name of the interface or object. For example, the name of the IDirect3D8 interface's IID is IID_IDirect3D8.

The HRESULT Structure

All COM methods return a 32-bit integer called an HRESULT. Although the name HRESULT seems to suggest a handle, it is essentially a structure that contains two separate pieces of information:

- Whether the method succeeded or failed.
- Information about the outcome of the operation.

The value returned as an HRESULT can normally be found in the function's documentation. Figure 24-4 shows the HRESULT bitmap.

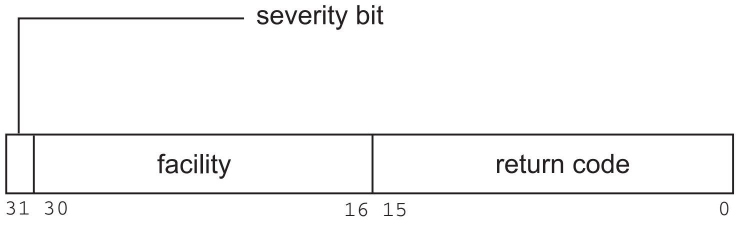


Figure 24-4 *HRESULT Bitmap*

The most significant bit of HRESULT, called the severity bit, reports whether the function succeeded or failed. The last 16 bits contain the return code, and the 15 bits of the facility field provide additional information regarding the type and origin of the return code. Applications usually need not look into these bit fields since macros are available for this purpose.

By convention, success codes have names that start with S_ , while failure codes start with the E_ prefix. The two most commonly used codes are S_OK, to indicate success, and E_FAIL, to indicate simple failure. Because COM functions can return a variety of success or failure codes, you have to be careful how you test the HRESULT value. Suppose a function is documented to return S_OK if successful. However, since a function may also return other failure or success codes it is dangerous to assume that it will always return E_FAIL if not successful. If you coded a test assuming that failure is always associated with E_FAIL, then another error code could be interpreted as a successful call. For example:

```
HRESULT  res;
. . .
if(res != E_FAIL)
  {
    // Assume success
  }
else
  {
    // Handle the failure
  }
```

Applications that need detailed information on the outcome of the function call need to test each relevant HRESULT value. To simplify processing it is recommended that applications use the macros SUCCEEDED or FAILED to test HRESULT. The SUCCEEDED macro returns TRUE for a success code and FALSE for a failure code. The FAILED macro returns TRUE for a failure code and FALSE for a success code. The following code fragment shows the use of the FAILED macro.

```
HRESULT  res;
. . .
if(FAILED(res))
  {
    //Handle failure
  }
else
  {
    //Handle success
  }
```

Table 24-1 lists some frequently used error codes.

Table 24-1

HRESULT Frequently Used Error Codes

NAME	MEANING
S_OK	Function succeeded and returns boolean TRUE.
NOERROR	Same as S_OK
S_FALSE	Function succeeded and returns boolean FALSE.
E_UNEXPECTED	Function failed unexpectedly.
E_NOTIMPL	Function not implemented
E_NOINTERFACE	Component does not support the interface. This error code is returned by the QueryInterface() call.
E_OUTOFMEMORY	Could not allocated required memory.
E_FAIL	Unspecified failure.
E_INVALIDARG	Invalid argument in function call

A few COM methods return a simple integer as an HRESULT. These methods are implicitly successful. In these cases the SUCCESS macro always returns TRUE. For example, the IUnknown::Release method decrements an object's reference count by one and returns the current reference count. In this case HRESULT always holds the current reference count.

24.2.3 The IUnknown Interface

All COM objects support an interface called IUnknown. The IUnknown interface provides DirectX objects with the ability to retrieve other interfaces and with the control of the object's lifetime. This is accomplished through the three methods of IUnknown:

- QueryInterface() allows the object to request pointers to a specific interface.
- AddRef() increments the object's reference count by 1.
- Release() decrements the object's reference count by 1.
- Reference Counting

Reference counting is a COM memory-management mechanism that allows an object to destroy itself once it is no longer used. Each COM component maintains a reference count. The AddRef() and Release() methods manage the reference count. AddRef() increments the reference count and Release() decrements it. When the reference count reaches 0, the object de-allocates itself and releases the memory it used. For example, if you create a Microsoft Direct3D object, the object's reference count is set to 1. Every time a function returns a pointer to an interface for that object, the function should call AddRef(), using the pointer as an argument. The effect of the AddRef() call is to increment the object's reference count. Each AddRef() call is matched with a call to Release(). When an object's reference count reaches 0 it is destroyed and all interfaces to it become invalid. The rules for reference counting are as follows:

- Functions that return interfaces automatically call AddRef() before returning. This means that when code obtains an interface through a COM function it does not need to call AddRef().
- When application code is finished with an interface it should call Release() using the interface pointer as an argument. This action decrements the reference count for the interface. If as the result of the call to Release() the reference count became zero, the object is automatically destroyed.
- If application code assigns an interface pointer to an interface pointer variable, it should call AddRef() in order to increment the reference count.

Using QueryInterface()

The QueryInterface() method of IUknown is used to determine whether an object supports a specific interface. Furthermore, if an object supports an interface, QueryInterface() returns a pointer to the interface. Code can then use the methods of the interface by means of the interface pointer. If QueryInterface() is successful, it calls AddRef() to increment the reference count. The application must call Release() to decrement the reference count before destroying the pointer to the interface.

24.3 Creating and Accessing the COM Object

COM objects can be created in several ways. The two most common methods used in DirectX programming are:

- Directly, by passing the object's CLSID to the CoCreateInstance function. The function will create an instance of the object, and it will return a pointer to an interface that you specify.
- Indirectly, by calling a DirectX function that creates the object for you. In this case the function creates the object and returns the interface to the object. When you create an indirectly you cannot specify which interface should be returned.

24.3.1 Creating the COM Object

When an object is created directly it must be initialized by calling the CoInitialize() function. However, the object's creation method will handle this task is the object is created indirectly. In this case the caller passes the address of a variable that is to serve as an interface pointer to the object creation method. The method then creates the object and returns an interface pointer. The following code fragment calls the IDirect3D8::CreateDevice() method to create a device object to represent a display adapter. It returns a pointer to the object's IDirect3DDevice8 interface.

```
IDirect3DDevice8 *pd3dDevice = NULL;
...
if( FAILED( pD3D->CreateDevice(D3DADAPTER_DEFAULT,
                               3DDEVTYPE_HAL,
                               hWnd,
                               D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                               &d3dpp,
                               &pd3dDevice )))
  return E_FAIL;
```

The first four parameters of the CreateDevice() method provide information needed to create the object, and the fifth parameter receives the interface pointer.

24.3.2 Using COM Objects

Once the COM object has been created, and the interface pointer has been obtained, this pointer can be used to access any of the interface's methods. The syntax is the same as that used to access a C++ method by means of a pointer. In the previous code fragment we called the CreateDevice() methods to obtain a pointer to IDirect3D8::CreateDevice. If the call succeeded, we can now use the returned interface pointer to access any method in IDirect3D8. For example, the method GetAdapterCount() of IDirect3D8 returns the number of adapters in the system in a variable of type UINT. Assuming that you have obtained a valid pointer with the CreateDevice() call, you could now get the number of display adapters as follow:

```
UINT adapters;
. . .
adapters = pd3dDevice->GetAdapterCount();
```

The COM Object's Lifetime

A COM object consumes system memory resources. When it is no longer needed, it should be destroyed so that memory can be used for other purposes. In C++ you control the object's lifetime with the new and delete operators. COM objects cannot be created or destroyed directly. The reason is that the same COM object may be used by more than one application. If one application were to destroy the object, the others may fail.

COM uses a system of reference counting to control an object's lifetime. An object's reference count is the number of times one of its interfaces has been requested. Each time an interface is requested, the reference count is incremented. When an application releases an interface reference count is decremented. The object remains in memory as long as it reference count is greater than zero. When the reference count reaches zero, the object is automatically destroyed. This mechanism ensures that code does not need to know about an object's reference count as long as object interfaces are obtained and released correctly. In other words, an object ensures its own appropriate lifetime. By the same token, when interfaces are not properly released, the reference count will never reach zero, and the object will remain in memory indefinitely. The result is usually a memory leak.

Manipulating the Reference Count

You have already seen that the appropriate processing consists of incrementing the reference count whenever a new interface pointer is obtained, and decrementing it whenever an interface is released. Recall that the reference count is incremented by a call to IUnknown::AddRef. However, applications do not usually need to explicitly call this method. If the interface pointer is obtained calling an object creation method, or by calling IUnknown::QueryInterface, the call automatically increments the reference count.

On the other hand, code must release all interface pointers, regardless of whether you or the object incremented the reference count. This is done by calling IUnknown::Release to decrement the reference count. A good programming practice is to initialize all interface pointers to NULL, and set them back to NULL when they are released. In this manner the cleanup routine can test all interface pointers. Those that are non-NULL are released before you terminate the application.

Problems with an objects reference count may originate in code that copies interface pointers and then calls AddRef(). The following code fragment, taken from the Microsoft DirectX 8 documentation, shows one possible way to handle reference counting in such cases.

```
IDirectSoundBuffer8* pDSBPrimary = NULL;
IDirectSound3DListener8* pDSListener = NULL;
IDirectSound3DListener8* pDSListener2 = NULL;
...
//Create the object and obtain an additional interface.
//The object increments the reference count.
if(FAILED(hr = g_pDS->CreateSoundBuffer( &dsbd, &pDSBPrimary, NULL )))
  return hr;
```

```
if(FAILED(hr=pDSBPrimary->QueryInterface(IID_IDirectSound3DListener8,
                                          (LPVOID *)&pDSListener)))
  return hr;

//Make a copy of the IDirectSound3DListener8 interface pointer.
//Call AddRef to increment the reference count and to ensure that
//the object is not destroyed prematurely
pDSListener2 = pDSListener;
pDSListener2->AddRef();
...
//Cleanup code. Check to see if the pointers are still active.
//If they are, call Release to release the interface.
if(pDSBPrimary != NULL)
{
  pDSBPrimary->Release();
  pDSBPrimary = NULL;
}
if(pDSListener != NULL)
{
  pDSListener->Release();
  pDSListener = NULL;
}
if(pDSListener2 != NULL)
{
  pDSListener2->Release();
  pDSListener2 = NULL;
}
```

Chapter 25

Introducing DirectDraw

Topics:

- 2D graphics in DirectX
- DirectDraw graphics fundamentals
- DirectDraw architecture
- Programming with DirectDraw

In this chapter we start discussing DirectDraw, which is the 2D component of DirectX. Although DirectDraw was merged with Direct3D in DirectX 8, COM insures that DirectDraw functionality continues to be available to applications.

25.1 2D Graphics and DirectDraw

In Chapter 23 we discussed that, in previous versions of the DirectX SDK, the 2D graphics interface was referred to as DirectDraw. Starting with DirectX 8, DirectDraw and Direct3D were merged into a single interface. However, in practical programming, 2D graphics cannot be ignored for the following reasons:

- Many graphics applications do not required 3D modeling or rendering.
- Most 3D applications use 2D graphics extensively.
- Some types of animations are easier to implement and show better performance in 2D than in 3D graphics.
- Many successful computer applications, including some successful and popular games, are implemented entirely in 2D.
- The learning curve for 3D graphics is quite steep. Starting with 2D provides a reasonable introduction to a complex and difficult technology.

DirectDraw is usually considered the most basic component of DirectX. It allows an application to access display memory as well as some of the hardware functions in the video card. The result is that a Windows program can obtain a high level of graphics performance without sacrificing device independence and while maintaining compatibility with the GDI. DirectDraw is implemented as a software interface to the card's video memory and graphics functions. Although its original intention was merely to facilitate game development under Windows, many other types of graphics applications can benefit from the higher degree of control and the performance gains that it provides.

DirectDraw has been described as a display memory manager that also furnishes access to some hardware acceleration features, as well as other graphics facilities available on the video card. Unfortunately, there is no uniform set of graphics features that all DirectDraw devices must provide. For this reason, the decision to use DirectDraw also entails the burden of accommodating varying degrees of Direct Draw functionality. DirectDraw provides services that allow querying the capabilities of a particular video card as well as the level of hardware support. Most features not supported by the hardware are emulated in software by DirectX, but at a substantial performance penalty.

A DirectDraw system implements its functionality both in hardware and in software emulation, each one with its own capabilities. Applications can query DirectDraw to retrieve the hardware and software capabilities of the specific implementation in the installed video card. DirectDraw is furnished as a 32-bit dynamic link library named DDRAW.DLL.

25.1.1 DirectDraw Features

The following are the most important features of DirectDraw:

- Direct access to video memory
- Manipulation of multiple display surfaces
- Page flipping
- Back buffering
- Clipping
- Palette management
- Video system support information

25.1.2 Advantages and Drawbacks

The following are possible advantages of using DirectDraw:

1. DirectDraw provides direct access to video memory. Accessing video memory directly allows the programmer to increase performance and obtain the highest degree of control. This feature also makes it easier to port some DOS graphics programs and routines into the Windows environment.
2. DirectDraw improves application performance by taking advantage of the hardware capabilities in the video card. For example, if the video card supports hardware blits, DirectDraw uses this feature.
3. DirectDraw provides hardware emulation to simulate features that are not supported by the hardware.

3. DirectDraw uses 32-bit flat memory addressing of video memory. This model is much easier to handle by code than one based on the Intel segmented architecture.
4. DirectDraw supports page flipping with multiple back buffers while executing in full-screen mode. This technique allows implementing very powerful animations.
5. In windowed mode, DirectDraw supports clipping, hardware-assisted overlays, image stretching, and other graphics manipulations.

The major disadvantages of DirectDraw are:

1. Programming in DirectDraw is more complicated and difficult than using the Windows GDI. Programs that do not need the additional performance or control provided by DirectDraw may find little additional justification for using it.
2. The graphics functions emulated by DirectDraw are often slower than those in the GDI.
3. Applications that rely on DirectDraw are less portable than those that do not.

25.2 Basic Concepts for DirectDraw Graphics

The following basic graphics concepts are extensively used in DirectDraw:

- Device-independent bitmaps
- Drawing surfaces
- Blitting
- Page flipping and back buffers
- Bounding rectangles

In this section we provide a brief review of these concepts, to serve as an introduction to DirectDraw. Some of these topics are covered in more detail later in the section on DirectDraw programming, later in the chapter.

25.2.1 Device-Independent Bitmaps

Windows and DirectX have adopted the device-independent bitmap (DIB) as its native graphics file format. A DIB file contains the image's dimensions, the number of color and the corresponding color values, and data describing the attributes of each pixel. The DIB file also contains some additional parameters, such as information about file compression and the image physical dimensions. DIB files usually have the .bmp file extension, although the .dib extension is also used.

The Windows APIs contain many functions that can be used in loading and manipulating DIB files. These functions can be used in DirectX applications. The following function, taken from the Ddutil.cpp file that is furnished with the DirectX SDK, combines Windows and DirectX functions to load a DIB onto a DirectX surface.

```
extern "C" IDirectDrawSurface * DDLoadBitmap(
IDirectDraw *pdd,
          LPCSTR szBitmap,
int dx,
int dy)
{
    HBITMAP             hbm;
    BITMAP              bm;
    DDSURFACEDESC       ddsd;
    IDirectDrawSurface *pdds;

    //
    //  This is the Win32 part.
    //  Try to load the bitmap as a resource.
    //  If that fails, try it as a file.
    //
    hbm = (HBITMAP)LoadImage(
            GetModuleHandle(NULL), szBitmap,
            IMAGE_BITMAP, dx, dy, LR_CREATEDIBSECTION);

    if (hbm == NULL)
        hbm = (HBITMAP)LoadImage(
                NULL, szBitmap, IMAGE_BITMAP, dx, dy,
                LR_LOADFROMFILE|LR_CREATEDIBSECTION);

    if (hbm == NULL)
        return NULL;

    //
    // Get the size of the bitmap.
    //
    GetObject(hbm, sizeof(bm), &bm);

    //
    // Now, return to DirectX function calls.
    // Create a DirectDrawSurface for this bitmap.
    //
    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize = sizeof(ddsd);
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT |DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
    ddsd.dwWidth = bm.bmWidth;
    ddsd.dwHeight = bm.bmHeight;

    if (pdd->CreateSurface(&ddsd, &pdds, NULL) != DD_OK)
        return NULL;

    DDCopyBitmap(pdds, hbm, 0, 0, 0, 0);

    DeleteObject(hbm);

    return pdds;
}
```

25.2.2 Drawing Surfaces

As the name implies, a drawing surface is a region of memory that receives video data to be displayed on the screen. In Windows programming the drawing surface is associated with the device context. You obtain access to the drawing surface when you obtain the handle to the device context by means of a function such as GetDC(). After the application obtains the handle to the device context it can draw to the screen.

In this case the Windows GDI provides an abstraction layer to allow a standard Windows applications to access the screen. But GDI was not designed for high performance graphics. GDI uses system memory to provide access to the video buffer, not the much faster video memory. In addition, GDI has no facilities to take advantage of special hardware features in the video card. This makes the GDI interface too slow for most games and multimedia applications.

DirectDraw, on the other hand, uses drawing surfaces that are defined in actual video memory. Furthermore, DirectDraw allows the programmer with the base address of the video buffer, which allows direct access to video memory. Applications that use DirectDraw can write directly to the memory on the video card, which results in very fast rendering. The fact that DirectDraw uses a 32-bit flat memory model simplifies programming.

Drawing surfaces are covered in greater detail in the context of DirectDraw programming, later in this chapter.

25.2.3 Blitting

The term blit stands for "bit block transfer." A blit consists of transferring a block of data from one location in memory to another. In computer graphics blitting usually consists of transferring an image between storage locations, from a storage location to video memory, or from video memory to a storage locations. Blits are also used in implementing sprite-based animation.

25.2.4 Page Flipping and Back Buffers

Page flipping is a technique often used in game and multimedia software. Page flipping is reminiscent of the animation achieved in some children's books that contain slightly different images on consecutive pages. By using your thumb to rapidly flip through the pages, the object and characters in the images appear to move. In software page flipping a series of DirectDraw surfaces are set up with slightly varying images. The image is animated when these surfaces are rapidly flipped to the screen.

In DirectDraw flipping techniques the first surface is usually referred to as the primary surface, while the other surfaces are called back buffers. The application blits the image to a back buffer, then flips the primary surface so that the back buffer appears on screen. While the system is displaying the image, the software is updating the back buffer with the next image. The process continues for the duration of the animation.

DirectDraw animation through page flipping can consists of a single pair of surfaces, that is, a primary surface and a single back buffer. More complicated schemes based on several back buffers allow producing more sophisticated effects.

25.2.5 Bounding Rectangles

Windows GDI uses a simplification for defining screen objects in terms of a rectangle that tightly binds it. This rectangle is called the bounding rectangle. By definition, the sides of the bounding rectangle are parallel to the sides of the screen. This allows defining the bounding rectangle by two points: one located at the top-left corner and the other one at the bottom-right corner. The Windows RECT structure provides a convenient way of storing the coordinates of the points that define the bounding rectangle. The RECT structure is defined as follows:

typedef struct tagRECT {

```
   LONG left;          // x coordinate of top-left corner
   LONG top;           // y coordinate of top left corner
   LONG right;         // x coordinate of bottom-right corner
   LONG bottom;        // y coordinate of bottom-right corner
} RECT;
```

For example, a RECT structure can be initialized as follows:

```
RECT aRect;
. . .
aRect.left = 10;
aRect.top = 20;
aRect.right  = 100;
aRect.bottom = 200;
```

Or as follows:

```
RECT aRect = {10,20,100,200};
```

In either case the left and top members are the x- and y-coordinates of a bounding rectangle's top-left corner. Similarly, the right and bottom members make up the coordinates of the bottom-right corner. Figure 25-1 shows a visualization of the bounding rectangle.

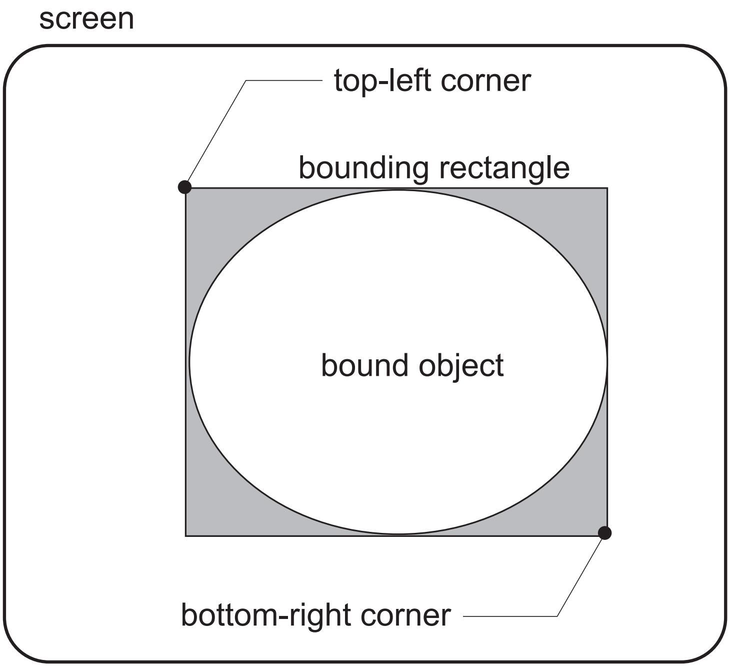


Figure 25-1 *DirectDraw Bounding Rectangle*

25.3 DirectDraw Architecture

The architecture of DirectDraw is defined by the following elements:

- The DirectDraw interface
- The DirectDraw hardware abstraction layer (HAL)
- The DirectDraw hardware emulation layer (HEL)

25.3.1 DirectDraw Interfaces

DirectDraw provides services through COM-based interfaces. The various versions of this interface are named IDirectDraw, IDirectDraw2, IDirectDraw4, and IDirectDraw7. Note that the numbers of the DirectDraw interfaces are discontinuous. IDirectDraw3, IDirectDraw5, and IDirectDraw6 do not exist, although DirectDraw3 it is erroneously mentioned in some Microsoft documents. These interfaces to DirectDraw correspond to different releases of the Game SDK and of DirectX. Since DirectDraw disappeared in DirectX 8 it is safe to assume that IDirectDraw7 will be the last implementation.

Programs can gain access to DirectDraw by means of the DirectDrawCreate() function or by the CoCreateInstance() COM function. In this book we use DirectDrawCreate() which is the easiest and more common one. Later in this chapter we discuss how a program can query at runtime which of the three DirectDraw interfaces is available.

25.4.1 DirectDraw Objects

As mentioned in Chapter 24, DirectX APIs are implemented as instances of COM objects. Communication with these objects is by means of the methods in each interface; for example, if IDirectDraw7 is the interface, the method SetDisplayMode() is accessed as follows:

```
        IDirectDraw7::SetDisplayMode
```

You already know that COM interfaces are derived from a base class called IUnknown. The following DirectDraw object types are currently defined: DirectDraw, DirectDrawSurface, DirectDrawPalette, DirectDrawClipper, and DirectDrawVideoPort. Figure 25-2, on the following page, shows the object composition of the DirectDraw interface.

The DirectDraw objects are described as follows:

- DirectDraw is the basic object of all applications. It is considered to represent the display adapter card. The corresponding COM object is named IDirectDraw. This is the first object created by a program and it relates to all other DirectDraw objects. A call to DirectDrawCreate() creates a DirectDraw object. If the call is successful, it returns a pointer to either IDirectDraw, IDirectDraw2, or IDirectDraw4 interfaces. IDirectDraw7 objects are created by calling IDirectDrawCreateEx().

- DirectDrawSurface object, sometimes called a "surface," represents an area in memory. The COM object name is IDirectDrawSurface. This object holds the image data to be displayed, or images to be moved to other surfaces. Applications usually create a surface by calling the IDirectDraw7::CreateSurface method of the DirectDraw object. The surface object interfaces are named IDirectDrawSurface, IDirectDrawSurface2, IDirectDrawSurface4, and IDirectDrawSurface7.
- DirectDrawPalette object, sometimes referred to as a "palette," represents a 16- or 256-color indexed palette. The palette object simplifies palette manipulations. It contains a series of indexed RGB triplets that describe colors associated with values within a surface. Palettes are limited to surfaces that use a pixel format of 8 bits or less. Palette objects are usually associated with corresponding surface objects, whose color attributes the palette object defines. The DirectDrawPalette objects are created by calling IDirectDraw7::CreatePalette method.
- DirectDrawClipper object, sometimes referred to as a "clipper," serves to prevent applications from drawing outside a predefined area. Clipper objects are usually convenient when a DirectDraw application is displayed in a window. In this case the clipper object prevents the application from drawing outside of its client area. A DirectDrawClipper object is created by calling IDirectDraw7::CreateClipper.
- DirectDrawVideoPort object was introduced in DirectX 5. The object represents the video-port hardware present in some systems. It allows direct access to the frame buffer without intervention of the CPU or the PCI bus.

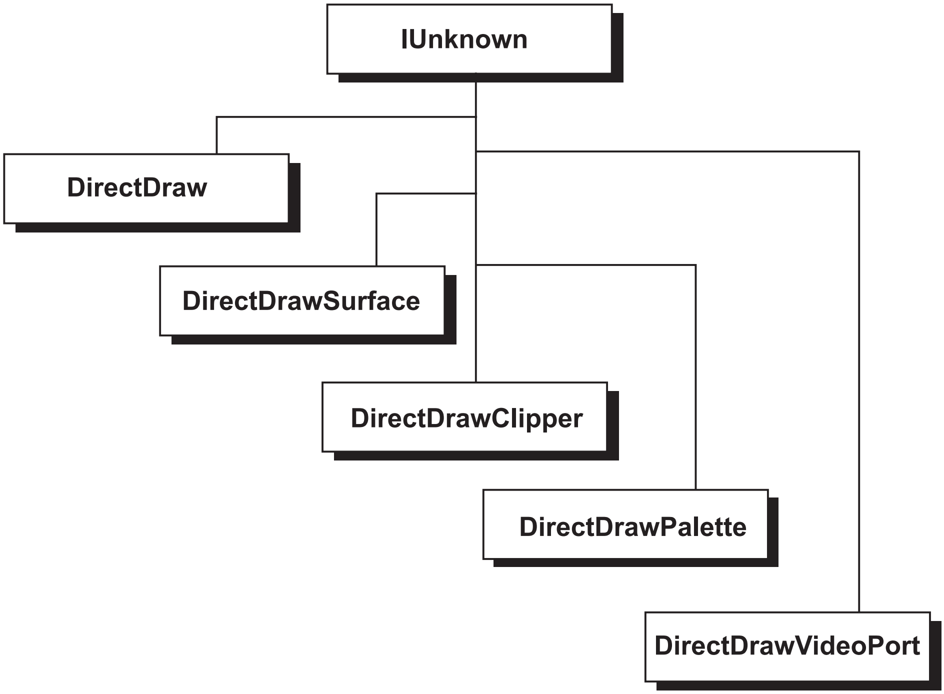


Figure 25-2 *DirectDraw Object Types*

25.4.2 Hardware Abstraction Layer (HAL)

DirectDraw ensures device independence by implementing a Hardware Abstraction Layer, or HAL. The HAL is provided by the video card manufacturer, board manufacturer, or OEM, according to Microsoft's specifications. However, applications have no direct access to the HAL, but to the interfaces exposed by DirectDraw. It is this indirect access mechanism that ensures HAL consistency and reliability.

In Windows 95/98, device manufacturer implements the HAL in both 16-bit and 32-bit code. Under Windows NT the HAL is always in 32-bit code. It can be furnished as part of card's display driver or a separate DLL. The HAL contains device-dependent code. It performs no emulation and provides no programmer accessible services. The only point of contact between an application and the HAL is when the application needs to query DirectDraw to find out what capabilities are directly supported.

25.4.3 Hardware Emulation Layer (HEL)

DirectDraw emulates in software those basic features that are not supported through the HAL. The Hardware Emulation Layer (HEL) is the part of DirectDraw that provides this functionality. Applications do not access the HEL directly. Whether a given functionality is provided through hardware features, or through emulation, is transparent to an application using DirectDraw. Code must specifically query DirectDraw to determine the origin of a given functionality. The IDirectDraw7::GetCaps() method, discussed later in this chapter, furnishes this information.

Unfortunately, some combinations of hardware-supported and emulated functions may lead to slower performance than pure emulation. DirectDraw documentation cites an example in which a display device driver supports DirectDraw but not stretch blitting. When the stretch blit function is emulated in video memory, a noticeable performance loss occurs. The reason is that video memory is often slower than system memory; therefore, the CPU is forced to wait when accessing video memory surfaces. Cases like this make evident one of the greatest drawbacks of DirectDraw, which is that applications must provide alternate processing for hardware dependencies.

25.4.4 DirectDraw and GDI

Several Windows graphics components lay between the application code and the video card hardware. Figure 25-3, on the following page, shows the relations between the various Windows graphics components.

The right-hand side of Figure 25-3 shows that an application can access the Windows video functions through the GDI, which, in turn, use the Display Device Interface. On the left-hand side an application accesses the video functions through DirectDraw. DirectDraw, in turn, uses the Hardware Abstraction Layer and the Hardware Emulation Layer to provide the necessary functionality. The horizontal arrow connecting the HAL and the DDI indicates that applications that use DirectDraw can also use the GDI functions, since both channels of video card access are open simultaneously.

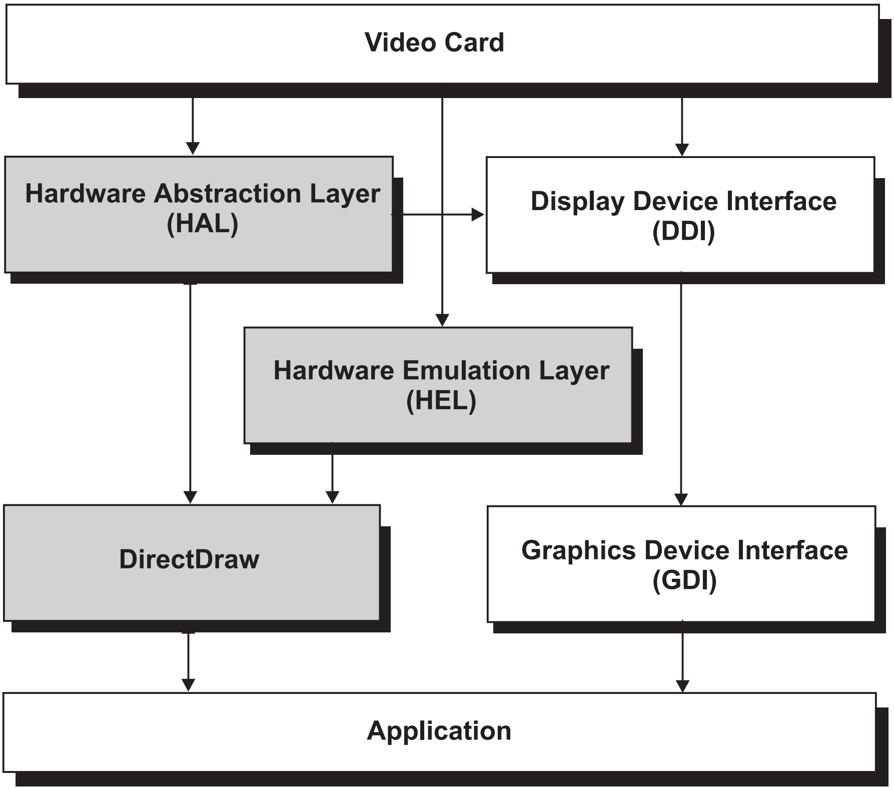

Video Card
Hardware Abstraction Layer (HAL)
Display Device Interface (DDI)
Hardware Emulation Layer (HEL)
DirectDraw
Graphics Device Interface (GDI)
Application

Figure 25-3 *Relations between Windows Graphics Components*

25.5 DirectDraw Programming Essentials

There are several topics that relate specifically to DirectDraw programming. Understanding these fundamental concepts is a prerequisite to successful DirectDraw programming. These following are core topics of DirectDraw programming:

- Cooperative levels
- Display modes
- Palettes
- Clippers

25.5.1 Cooperative Levels

Cooperative levels refers to the relationship between DirectDraw and Windows. A DirectDraw program can execute full-screen, with exclusive access to the display resources, or it can execute in a window, sharing video resources with other running programs. In this last case the DirectDraw application and the other Windows programs executing concurrently must cooperate in their use of the video resources. When a DirectDraw application requests and obtains total control of the video functions it is said to execute in exclusive mode. DirectDraw applications that do not execute in exclusive mode are usually referred to as windowed DirectDraw programs.

The SetCooperativeLevel() function is used by an application to set cooperative level. The predefined constants DDSCL_FULLSCREEN and DDSCL_EXCLUSIVE allow the application to execute full-screen and to ensure control of the display mode and the palette. In this case the DirectDraw program has almost exclusive control of the video resources. The use of this function is described later in this chapter.

DirectDraw cooperative levels have the following additional features:

- A DirectDraw application can be enabled to use a non-standard VGA resolution known as Mode X. Mode X, which executes in 320 by 240 pixels in 256 colors, was a very popular mode with DOS game programmers.
- DirectDraw applications that execute in exclusive mode can be prevented from responding to CTRL + ALT + DEL keystrokes.
- A DirectDraw application can be enabled to minimize or maximize itself.

Microsoft considers the normal cooperative level the one in which the DirectDraw application cooperates as a windowed program. However, DirectDraw applications that execute in windowed mode are not able to change the display mode or perform page flipping. Display mode control and page flipping are essential to many high-performance graphics programs, especially those that use animation. For this reason many high-performance DirectDraw programs execute in exclusive mode.

25.5.2 Display Modes

Display modes date back to the first PC graphics video system. By the time the VGA was released (1987) there were 18 different display modes. A display mode is a hardware configuration of the video system registers and hardware that enables a particular resolution and color depth. Display modes are described in terms of their pixel width, height, and bit depth. For example, VGA mode 18H has a resolution of 640-by-480-by-4. This means that it displays 640 pixel columns and 480 pixel rows in 16 colors. The last digit of the mode specification, in this case 4, is the number of bits used in the pixel color encoding. In VGA mode 18H the color range is 16, which is the maximum number of combinations of the 4 binary digits used to encode the color.

Palletized and Nonpalletized Modes

PC display modes are often classified as palletized and nonpalletized. In palletized display modes each color value is an index into an associated color table, called the palette. The bit depth of the display mode determines the number of colors in the palette. For example, in a 4-bit palletized mode, such as VGA mode 18H, each pixel attribute is a value in the range 0 to 15. This makes possible a palette with 16 entries. The actual colors displayed depend on the palette settings. The programmer can select and change the pallet colors at any time, thus selecting a sub-range of displayed colors. However, when the palette is changed, all displayed objects are shown with the new settings.

Nonpalletized display modes, on the other hand, encode pixel colors directly. In nonpalletized modes the bit depth represents the total number of color attributes that can be assigned to each pixel. There is no look up table to define the color attributes.

The higher the resolution and the color depth of a display mode, the more video memory that is required to encode the pixel data. Since not all video adapters contain the same amount of memory, not all of them support the same video modes. The DirectDraw EnumDisplayModes() function is used to list all the display modes supported by a device, or to confirm if a particular display mode is available in the video card.

Applications using DirectDraw can call the SetDisplayMode() function. The parameters passed to the call describe the dimensions, bit depth, and refresh rate of the mode to be set. A fifth parameter indicates special options for the given mode. Currently this parameter is used only to differentiate between Mode 13H, with 320 by 200 resolution and 16 colors, and VGA Mode X, also with 320 by 200 resolution but in 256 colors. Although an application can request a specific display mode resolution and bit depth, it cannot specify how the pixel depth is achieved by the hardware. After a mode is set, the application can call GetDisplayMode() to determine if the mode is palletized and to examine the pixel format. In other words, DirectDraw reserves the right to implement a particular color depth in a palletized or nonpalletized mode.

DirectDraw programs that do not execute in exclusive mode allow other applications to change the video mode. At the same time, an application can change the bit depth of the display mode, only if it has exclusive access to the DirectDraw object. DirectDraw applications that execute in exclusive mode allow other applications to allocate DirectDrawSurface objects, and to use DirectDraw and GDI services. For the same reason, applications that execute at the exclusive cooperative level are the only ones that can change the display mode or manipulate the palette.

A DirectDraw application can explicitly restore the display hardware to its original mode by calling the RestoreDisplayMode(). A DirectDraw exclusive mode application that sets the display mode by calling SetDisplayMode() can automatically restore the original display mode by calling RestoreDisplayMode().

DirectDraw supports all screen resolutions and pixel depths that are available in the card's device driver. Thus, a DirectDraw application can change to any mode supported by the display driver, including 24- and 32-bit true-color modes.

25.5.3 Surfaces

A DirectDraw surface is a linear memory area that holds image data. A surface can reside in display memory, which is located in the video card, or in system memory. Applications create a DirectDraw surface by calling the IDirectDraw7::CreateSurface() function. The call can create a single surface object, a complex surface-flipping chain, or a three-dimensional surface. The IDirectDrawSurface interface allows an application to indirectly access memory through blit functions, such as Blt() and BltFast(). In addition, a surface provides a device context to the display, which can be used with GDI functions.

IDirectDrawSurface surface functions can be used to directly access display memory. The Lock() function retrieves the address of an area of display memory and ensures exclusive access to this area. This operation is said to "lock" the surface. A primary surface is one in which the display memory area is mapped to the video display. Alternatively, a surface can refer to a nondisplayed area. In this case the surface is called an off-screen or overlay surface. Nonvisible buffers usually reside in display memory, but they can be created in system memory if DirectDraw is performing a hardware emulation, or if it is otherwise necessary due to hardware limitations. Surface objects that use a pixel depth of 8 bits or less are assigned a palette that defines the color attributes in the encoding. Figure 25-4 shows the surface-based layout of video memory.

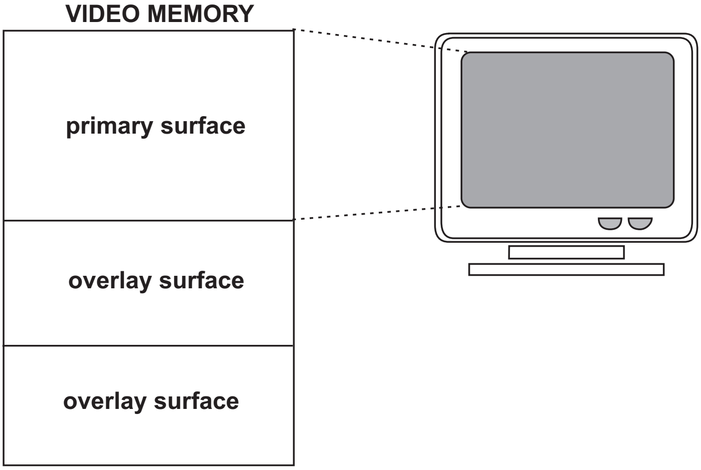


Figure 25-4 *Visualization of Primary and Overlay Surfaces*

Once a DirectDraw application receives a pointer to video memory it can use this pointer to draw on the screen, with considerable gain in control and performance. However, a program that accesses video memory directly must concern itself with many details of the video system layout that are transparent at a higher programming level. The first complicating factor is that video buffer mapping can be different in two modes with the same resolution. This possible variation is related to the fact that the video buffer is actually a storage for pixel attributes. If an attribute is encoded in 8 bits, then the buffer requires 1 byte per pixel. If a pixel attribute is stored in 24 bits, then the buffer requires 3 bytes per pixel.

Figure 25-5, on the following page, shows two video modes with different pixel depths. In the 8-bits per pixel mode the fourth memory byte is mapped to the fourth screen pixel. However, in the 24-bits per pixel mode it is the thirteenth to the fifteenth video memory bytes that are mapped to the fourth pixel. The calculations required to obtain the offset in video memory for a particular screen pixel are different in each case.

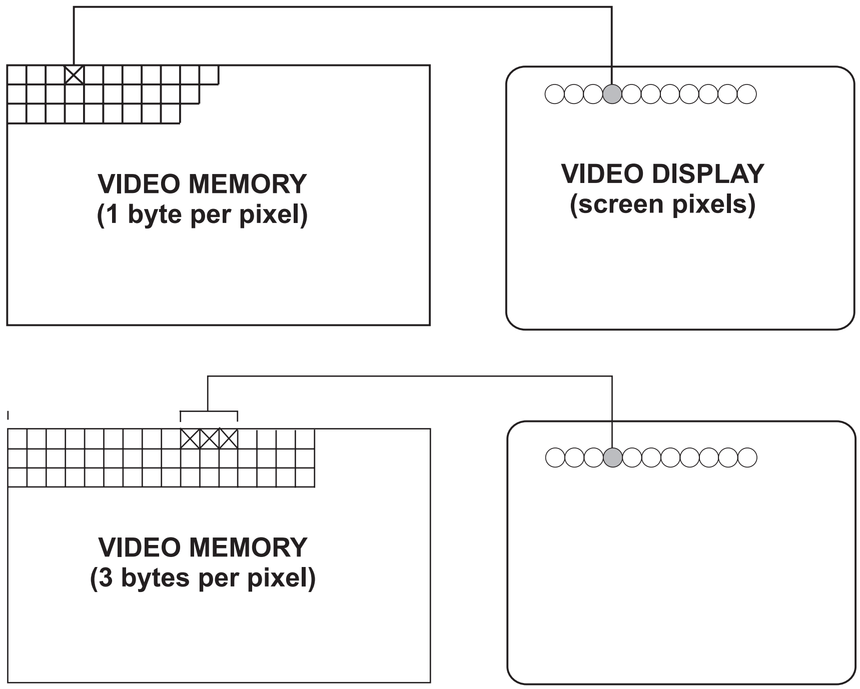


Figure 25-5 *Video Memory Mapping Variations*

There is another complication in direct access programming: in display modes the number of bytes in each video buffer row is not exactly the product of the number of pixels on the row by the number of bytes per pixel. For example, consider a row of 400 pixels in which each pixel is mapped to 3 data bytes. In this case it would be reasonable to expect that the pixel row would take up 700 bytes of video memory. However, due to video system design and performance considerations, sometimes it necessary to allocate a number of bytes in each buffer row that is a multiple of some specific number. This determines that, in some display modes, there are data areas that are not mapped to screen pixels.

Here is a real life example: a display mode with a resolution of 640 pixels per row and a color depth of 24 bits per pixel requires 1,920 bytes to store the data corresponding to a single row of screen pixels. However, the some video card designers have assigned 2,560 bytes of video buffer space for each screen row, so that the same buffer size can be used in a 32 bits per pixel mapping. The result is that in the 24-bit mode there is an area of 640 unmapped bytes at the end of each row.

This explains the difference between the terms pitch and width in regards to video buffer data. While pitch describes the actual byte length of each row in the video buffer, width refers to the number of pixels in each screen row. In programming direct access operations it is important to remember that pitch and width can have different values.

25.5.4 Palettes

A palette is a color look-up table. Palettes are a way of indirectly mapping pixel attributes. This scheme is useful when in extending the number of displayable colors in modes with limited pixel depths. For example, a display mode with 4 bits per pixel normally allows representing 16 different color attributes. Alternatively, it is possible to make each video buffer values serve as an index into a data structure called the palette. The actual pixel colors are defined in the palette. By changing the values stored in the palette the application can map many 16-color sets to the display attributes. By means of the palette mechanism the number of simultaneously displayable colors remains the same, but the actual colors mapped to the video buffer values can be changed by the application. Figure 25-6 shows how a palette provides an indirect mapping for the color attributes stored in the video buffer.

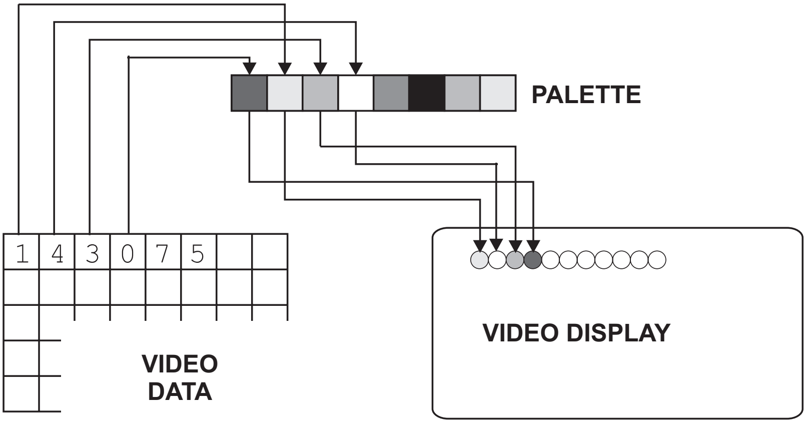


Figure 25-6 *Palette-Based Pixel Attribute Mapping*

In DirectDraw palettes are linked to surfaces. Surfaces that use a 16-bit or greater pixel format do not use palettes, since this pixel depth allows encoding rich colors directly. Therefore, the so-called real color modes (16 bits per pixel) and true color modes (24 and 32 bits per pixel) are nonpalletized. Palettes are used in modes with 1, 2, 4, and 8 bits per pixel. In these cases the palette can have 2, 4, 16, or 256 entries. In DirectDraw a palette must be attached to a surface with the same color depth. In addition, it is possible to create palettes that do not contain a color table. In these so-called index palettes, the palette values serve as an index into another palette's color table.

Each palette entry is in the form of an RGB triplet that describes the color to be used when displaying the pixel. The RGB values in the color table can be in 16- or 24-bit format. In 16 bit RGB format each palette entry is encoded in 5-6-5 form. This means that the first 5 pixels are mapped to the red attribute, the second 6 pixels to the green attribute, and the last 5 pixels to the blue attribute. This is the same mapping scheme used in the real color modes. In the 24-bit RGB palette format each of the primary colors (red, green, and blue) is mapped to 8 pixels, as in the true color modes.

An application creates a palette by calling IDirectDraw7::CreatePalette() function. At call time the application defines if the palette contains 2, 4, 16, or 256 entries and provides a pointer to a color table used in initializing the palette. If the call is successful, DirectDraw returns the address of the newly created DirectDrawPalette object. This palette object can then be used to attach the palette to a DirectDraw surface. The same palette can be attached to multiple surfaces. Once a palette is attached to a surface, an application can call the GetPalette() and SetPalette() functions to query or change the palette entries.

A type of animation is based on changing the appearance of a surface object by modifying the palette attached to the surface that contains it. By repeatedly changing the palette, the surface object can be made to appear differently without actually modifying the contents of video memory. Two different types of palette manipulations can be used to for this. The first method is based on modifying the values in a single palette. The second method is based on switching between several palettes. Since palette modifications are not hardware intensive, either method often produces satisfactory results.

Historically, the need for palettes resulted from the memory limitations of the original video systems used in the PC. In VGA the video space was on the order of a few hundred kilobytes, while the low-end PCs of today are furnished with video cards that have 2 or 4 Mb of on-board video memory. This abundance of video memory has made palettes less important. However, palletized modes allow interesting animation effects, which are achieved by manipulating the color table data. For example, an object can be made to disappear from the screen by changing to a palette in which the object attributes are the same as the background. The object can then be made to reappear by restoring the original palette.

25.5.5 Clipping

In DirectDraw clipping is a manipulation by which video output is limited to one or more rectangular-shaped regions. DirectDraw supports clipping in applications that execute in exclusive mode and windowed. The term "clippers" is often used to refer to DirectDrawClipper objects. A single bounding rectangle is sometimes used to limit the display to the application's client area. Several associated bounding rectangles are called a clip list.

The most common use for a clipper is to define the boundaries of the screen or of a rectangular window. A DirectDraw clipper can be used to define the screen area of an application so as to ensure that a bitmap is progressively displayed as it moves into this area. If a clipping area is not defined, then the blit fails because the destination drawing surface is outside the display limits. However, when the boundaries of the video display area are defined by means of a clipper, DirectDraw knows not to display outside this area and the blit succeeds. Blitting a bitmap to unclipped and clipped display areas is shown in Figure 25-7.

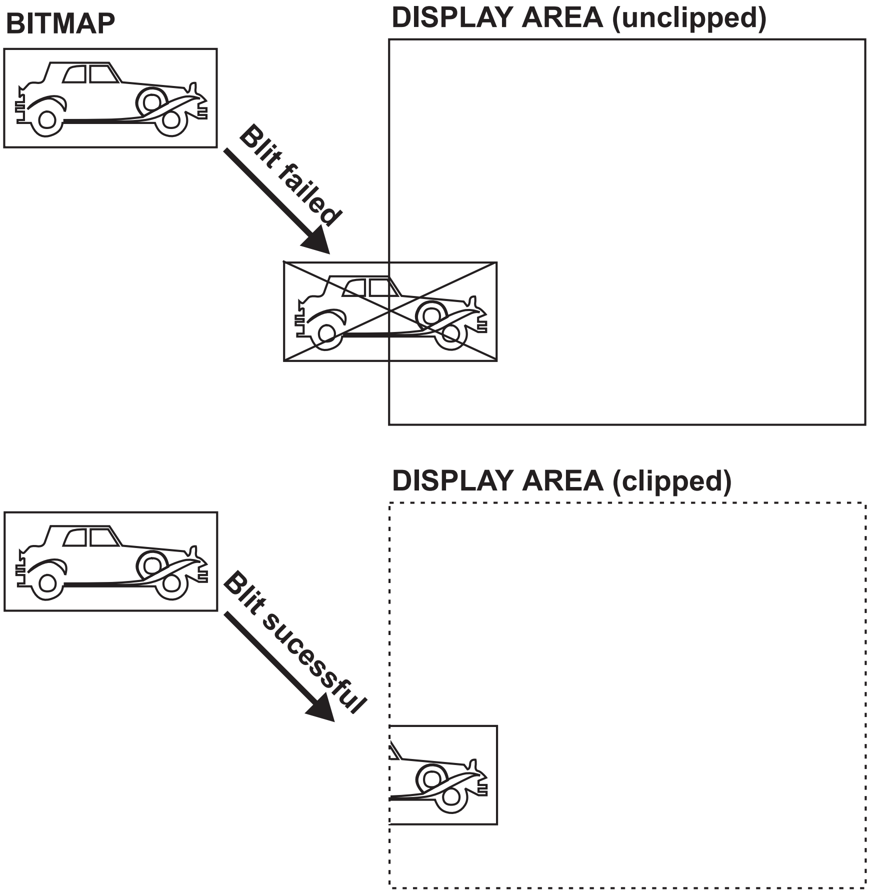


Figure 25-7 *Clipping a Bitmap at Display Time*

In Figure 25-7 shows how clipping makes it possible to display a bitmap that does not entirely fit in the display area. The top of the illustration shows a blit operation that fails because the source bitmap does to fit in the destination area. When clipping is enabled it possible to display the bitmap of the automobile as it progressively enters the screen area, instead of making it pop onto the screen all at once.

To implement clipping in DirectDraw you create a clipper with the screen rectangle as its clip list. Once the clipper is created, trimming of the bitmap is performed automatic. Clipper objects are also used to designate areas within a destination surface. If the designated areas are tagged as writable, DirectDraw automatically crops images that fall outside this area. Figure 25-8, on the following page, shows a display area with a clipper defined by two rectangles. When the text bitmap is blitted onto the screen, only those parts that fall inside the clipper are displayed. The pixel data is preserved in the screen areas not included in the clipper. In this case the clip list consists of the two rectangles for which output is validated.

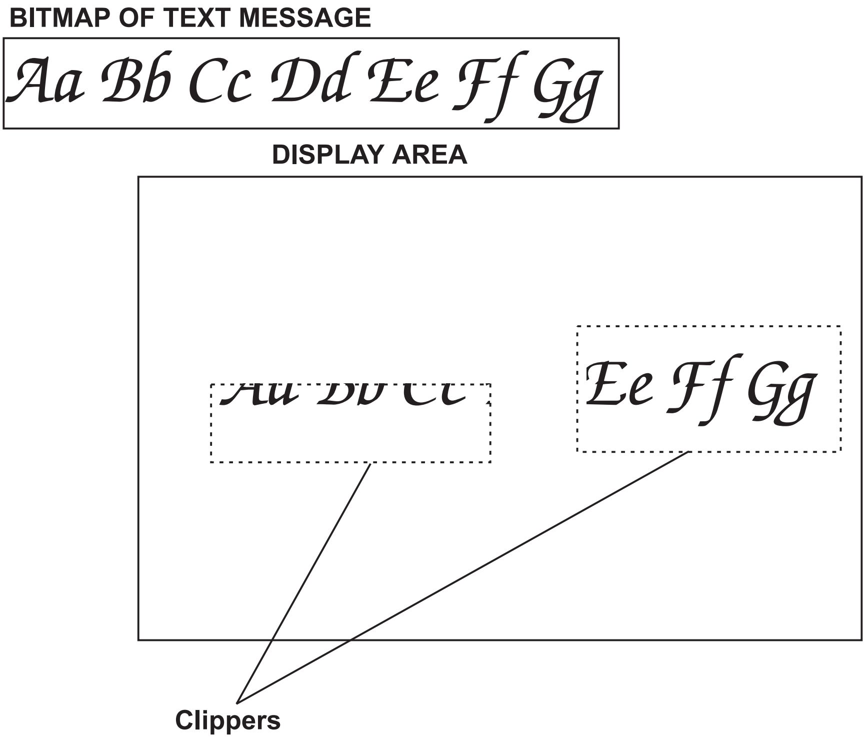


Figure 25-8 *Clipper Consisting of Two Rectangular Areas*

Chapter 26

Setting Up DirectDraw

Topics:

- Configuring Developer Studio
- Creating the DirectDraw object
- Obtaining the interface version
- Selecting the level of interaction
- Obtaining the hardware capabilities
- Obtaining and listing the display modes

DirectX applictions that use DirectDraw must first initialize the software and perform a series of configuration tests. One of the most critical elements is to determine and select how the DirectX program cooperates with concurrent Windows applications. In this chapter we describe the initialization and setup operations for DirectDraw programming and develop code that serves as a template for creating DirectDraw programs.

26.1 Set-up Operations

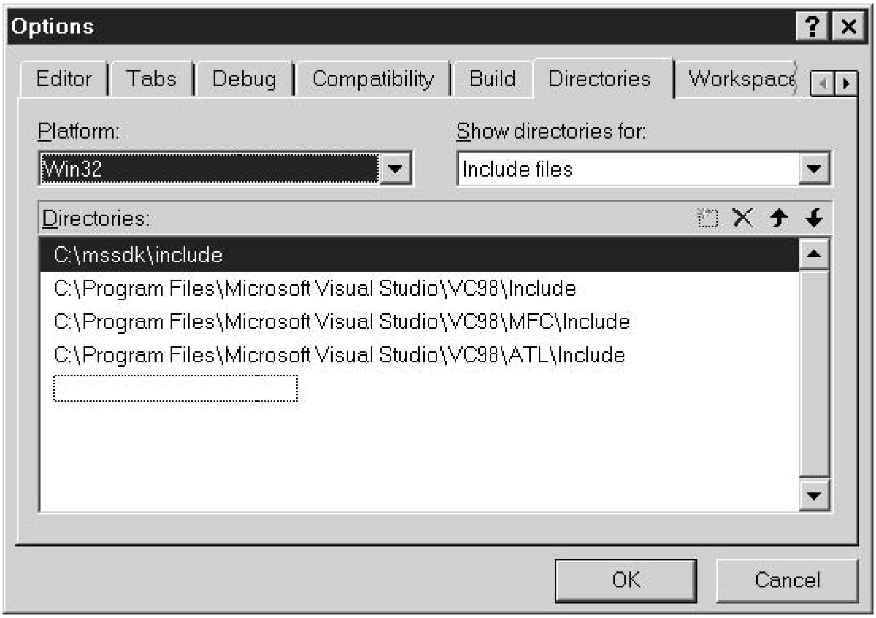
DirectDraw programs must first set up the development system so that application code can access the DirectDraw functions in DirectX. The first step is to include the DirectDraw header file, named ddraw.h, which is part of the DirectX SDK as well as the newer versions of Windows.

26.1.1 DirectDraw Header File

As newer versions of the DirectX SDK are installed, either directly or through operating system patches, it is possible to find several versions of the ddraw.h file on the same computer. The DirectX programmer needs to make sure that the software under development is using the most recent release of the DirectDraw header file. One way to ensure this is use the Windows Explorer search feature to look for all files named ddraw.h. Once the files are located, it is easy to rename or delete the older versions of ddraw.h. Usually the date stamp and the file size serve to identify the most recent one.

However, you cannot assume that the installation program for the operating system, the SDK, or the development environment will do this for you.

In addition, the ddraw.h file must be located so that it is accessible to the development software. This may require moving or copying the newest version of ddraw.h to the corresponding include directory, as well as making certain that the path in the development environment corresponds with this directory. In Visual C++ the directories searched by the development system can be seen by means of the Options command in the Tools menu. In this case the Show directories for scroll box should be set for include files. At this time you may enter, in the edit box, the path to the DirectDraw include file and libraries. The edit is located at the bottom of the Directories window. While in this window, it is a good idea to drag the box to the top of the list so that this directory is searched first. Figure 26-1 shows the Directories tab for the Include files when DirectX is installed in the default drive and path.



Options

Platform: Win32

Show directories for: Include files

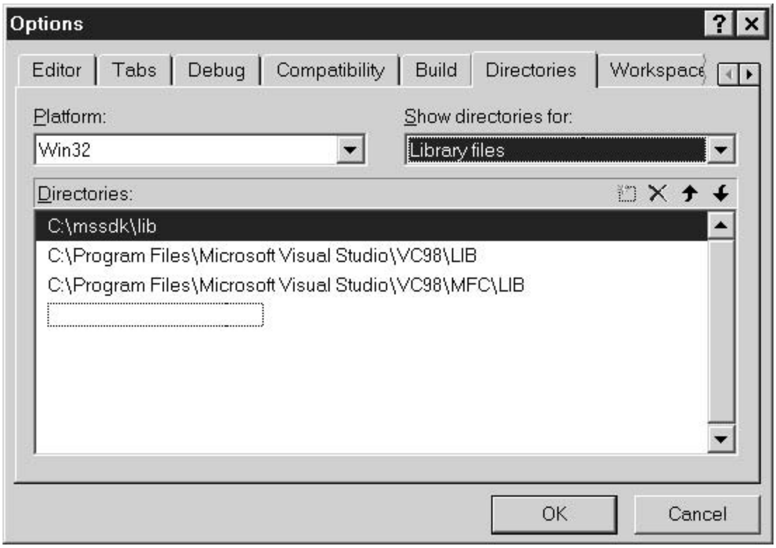
Directories:

C:\mssdk\include
C:\Program Files\Microsoft Visual Studio\VC98\Include
C:\Program Files\Microsoft Visual Studio\VC98\MFC\Include
C:\Program Files\Microsoft Visual Studio\VC98\ATL\Include

Figure 26-1 *Directories Tab (Include Files) in the Options Dialog Box*

26.1.2 DirectDraw Libraries

Another software component necessary for DirectDraw programming is the ddraw.lib library file. Here again, it is possible that duplicate versions of the software be present in the system. It is necessary that all but the most recent one be eliminated. The same process described for the Include files in the preceding section can be applied to the library files. Figure 26-2 shows the Directories tab for the library files.



Platform: Win32

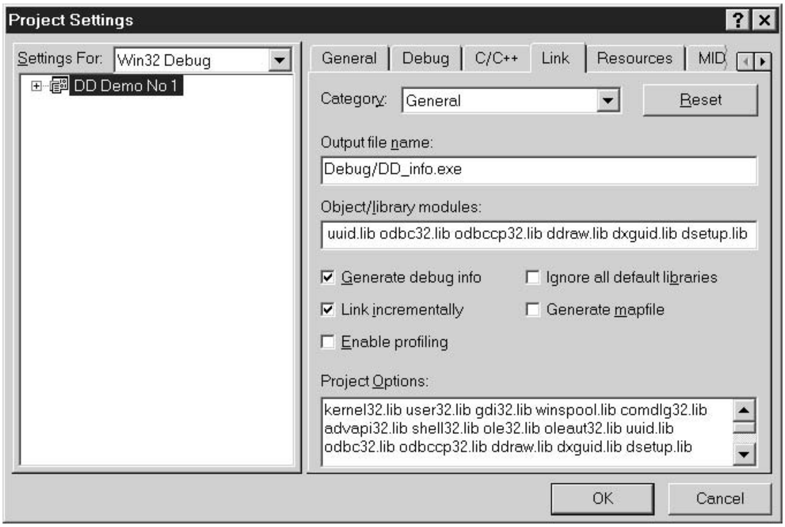
Show directories for: Library files

Directories:

C:\mssdk\lib
C:\Program Files\Microsoft Visual Studio\VC98\LIB
C:\Program Files\Microsoft Visual Studio\VC98\MFC\LIB

Figure 26-2 *Directories Tab (Library files) in the Options Dialog Box*

In addition to finding the newest version of the library, and installing it in the system's library path, Visual C++ users must also make sure that the development environment is set up to look for the DirectDraw libraries. To make sure of this you can inspect the dialog box that is displayed when the Settings command is selected in Developer Studio Project menu. The ddraw.lib and the dxguid.lib files must be listed in both the Object/Library Modules and the Project Options windows of the Link tab in the Project Settings dialog box, as shown in Figure 26-3.



Settings For: Win32 Debug

DD Demo No 1

Category: General

Output file name: Debug/DD_info.exe

Object/library modules: uuid.lib odbc32.lib odbccp32.lib ddraw.lib dxguid.lib dsetup.lib

☑ Generate debug info
□ Ignore all default libraries
☑ Link incrementally
□ Generate mapfile
□ Enable profiling

Project Options:

kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib ddraw.lib dxguid.lib dsetup.lib

Figure 26-3 *Link Tab in Developer Studio Project Settings Dialog Box*

The ddraw.lib library contains the DirectDraw functions, and the dxguid.lib has the identifiers required for accessing the various interface versions. If the Link tab in the Project Settings dialog box does not show a reference to ddraw.lib and dxguid.lib, you can manually insert the library names in the Object/library modules edit box. The library names are automatically copied to the Project Options box. Once access to the DirectDraw header file and the libraries are in place, the development system is ready for use.

26.2 Creating the DirectDraw Object

To use DirectDraw an application must first create a DirectDraw object. The DirectDraw object is actually a pointer to the DirectDraw interface as implemented in the video card. Since this pointer provides access to all other DirectDraw functions, a DirectDraw application can do little else without this object. To create the object you use the DirectDrawCreateEx() function. This function creates a DirectDraw object that supports a new set of Direct3D functions first released with DirectX 7. The function's general form is as follows:

```
HRESULT DirectDrawCreateEx (
            GUID FAR lpGUID,             // 1
            LPVOID *lplpDD,              // 2
            REFIID  iid                  // 3
            IUnknown FAR *pUnkOutter,    // 4
            );
```

The first parameter (lpGUID) is a globally unique identifier (GUID) that represents the driver to be created. If this parameter is NULL then the call refers to the active display driver. The newer versions of DirectDraw allow passing one of two flags in this parameter. The flags control the behavior of the active display, as follows:

- DDCREATE_EMULATIONONLY: DirectDraw use only emulation. Hardware support features are not be used.
- DDCREATE_HARDWAREONLY: DirectDraw object does not use emulated features. If a hardware supported features is not available the call returns DDERR_UNSUPPORTED.

The second parameter (*lplpDD) is a pointer that the call initializes if it succeeds. This is IDirectDraw7 interface pointer object returned by DirectDrawCreateEx().

The third parameter (iid) must be set to IID_DirectDraw7. Any other interface returns an error.

The fourth parameter (*pUnkOutter) is provided for future compatibility with the COM interface. At present it should be set to NULL.

The call returns DD_OK if it succeeds. If it fails, one of the following predefined constants is returned:

- DDERR_DIRECTDRAWALREADYCREATED
- DDERR_GENERIC

- DDERR_INVALIDDIRECTDRAWGUID
- DDERR_INVALIDPARAMS
- DDERR_NODIRECTDRAWHW
- DDERR_OUTOFMEMORY

On systems with multiple monitors, specifying NULL for the first parameter causes the DirectDraw object to run in emulation mode. In these systems the call must specify the device's GUID in order to use hardware acceleration.

26.2.1 Obtaining the Interface Version

You have seen that the COM requires that objects update their functionality by means of new interfaces, rather than by changing methods within existing interfaces. The purpose of this requirement is to keep existing interfaces static, so that older applications continue to be compatible with the newer interfaces.

The availability of various interfaces facilitates component updating, but it also creates some coding complications. For example, currently the DirectDraw surface object supports three different interfaces, named IDirectDrawSurface, IDirectDrawSurface2, and IDirectDrawSurface4. Consistent with the COM requirement, each interface version supports all the methods of its predecessor, and adds new ones for the new features. But there is no assurance that a host machine contains the newest version of the interface. For this reason applications must query DirectDraw to determine which interface or interfaces are available in the host, then provide alternative processing routes for each case. The situation is further complicated by the fact that, in some rare cases, a new interface may not support all the functions provided in a previous one. The result is a return to device-dependent programming that Windows was designed to avoid in the first place.

Once the DirectDraw application has used the DirectDrawCreateEx() function to obtain a pointer to the DirectDraw object, COM provides the IUnknown::QueryInterface method which allows finding out whether the object supports other interfaces. If the call succeeds, QueryInterface() returns a pointer to the interface requested as a parameter. It is through this pointer that code gains access to the methods of the new interface. If the QueryInterface() function returns any other value but S_OK call can assume that the interface is not available. Possible options in this case are to provide some sort of work-around for the missing functionality, or to abort execution if the lack of processing capabilities in the host machine cannot be remedied.

The QueryInterface() has the following general form:

```
HRESULT QueryInterface(
                       REFIID riid,            // 1
                       LPVOID* obj,            // 2
                       );
```

The first parameter (riid) is a reference identifier for the object being queried. The calling code must know this unique identifier before the call is made. The second parameter is the address of a variable that will contain a pointer to the new interface, if the call is successful. The return value is S_OK if the call succeeds or one of the following error messages if it fails:

- E_NOINTERFACE
- E_POINTER
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_OUTOFMEMORY

The DDERRR_OUTOFMEMORY error message is returned by IDirectDrawSurface2 and IDirectDrawSurface4 objects only. If, after making the call, the application determines that it does not need to use the interface, it should call the Release() function to free it.

Four IDirectDraw interfaces are implemented in DirectX . The corresponding reference identifiers are IID_DirectDraw, IID_DirectDraw2, IID_DirectDraw4, and IID_DirectDraw7. It is not recommended that an application mix methods from two or more interfaces since the results are sometimes unpredictable.

Microsoft attempts to ensure the portability of applications that commit to a specific DirectX implementation by furnishing an installation utility that upgrades the host system to the newest components. In the DirectSetup element of DirectX there are the diagnostics and installation programs, as well as the drivers and library files, that serve to update a system to the corresponding version of the SDK. DirectSetup also includes a ready-to-use installation utility that copies all the system components to the corresponding directories of the client's hard drive and performs the necessary modifications in the Windows registry. In the DirectX SDK Microsoft also provides all the required project files for a sample installation application named dinstall. The DirectX programmer can use the source code of the dinstall program as a base on which to create a customized installation utility for DirectX.

The following code fragment shows how to determine the version of the DirectDraw interface installed in the host system:

```
// Global variables for DirectDraw operations
HRESULT         DDConnect;

// Interfaces pointers
LPDIRECTDRAW    lpDD;
LPDIRECTDRAW    lpDD2;
LPDIRECTDRAW    lpDD4;
LPDIRECTDRAW    lpDD7;
int dDLevel     0;     // Implementation level
.
.
.
//*************************
//   DirectDraw Init
//*************************
DDConnect = DirectDrawCreate ( NULL,
               &lpDD0,
               NULL);
```

```
// Querying the interface to determine most recent
// version
if(DDConnect == DD_OK)
{
       DDLevel = 1;              // Store level
       lpDD = lpDD0;             // copy pointer
       DDConnect = lpDD0->QueryInterface(
                   IID_IDirectDraw2,
                   (LPVOID *) &lpDD2);
}

if(DDConnect == S_OK)
{
       DDLevel = 2;          // Update level
       lpDD0->Release();      // Release old pointer
       lpDD = (LPDIRECTDRAW) lpDD2;
       DDConnect = lpDD->QueryInterface(
                   IID_IDirectDraw4,
                   (LPVOID *) &lpDD4);
}

if(DDConnect == S_OK)
{
       DDLevel = 4;           // Update level
       lpDD2->Release();      // Release old pointer
       lpDD = (LPDIRECTDRAW) lpDD4;
       DDConnect = lpDD->QueryInterface(
                   IID_IDirectDraw7,
                   (LPVOID *) &lpDD7);
}

if(DDConnect == S_OK)
{
      DDLevel = 7;           // Update level
      lpDD4->Release();      // Release old pointer
      lpDD = (LPDIRECTDRAW) lpDD7;
}
//  Note that the pointer returned is typecast into LPDIRECTDRAW.
```

Notice in the preceding code that when a valid object is found, the preceding one is released by calling the Release() function. In Chapter 9 you saw that IUnknown contains a function named AddRef() which increments the object's reference count by 1 when an interface or an application binds itself to an object. Also that the Release() function decrements the object's reference count by 1 when it is no longer needed. When the count reaches 0, the object is automatically de-allocated. Normally, every function that returns a pointer to an interface calls AddRef() to increment the object reference count. By the same token, the application calls Release() to decrement the object reference count. When an object's reference count reaches 0, it is destroyed and all interfaces to it become invalid. In the previous sample code we need not call the AddRef() method since QueryInterface() implicitly calls AddRef() when a valid object is found. However, the code must still call Release() to decrement the reference object count and destroy the pointer to the interface.

26.2.2 Interface Version Strategies

The preceding code fragment allows determining which DirectDraw interface version is available in the host machine. If a DirectDraw interface is found, the pointer is used to make a few calls to the DirectDraw interface. But notice that this scheme works only in trivial applications, such as the DD Info project listed later in this chapter. DirectDraw implements version-specific pointers for the different interfaces, of types LPDIRECTDRAW, LPDIRECTDRAW2, LPDIRECTDRAW4, and LPDIRECTDRAW7. Any substantial DirectDraw program will almost certainly generate errors when calls are made with a pointer that is not of the version-specific type. In the preceding code we have typecast the version-specific pointers to a generic pointer of type LPDIRECTDRAW, for example, in the code fragment:

```
if(DDConnect == S_OK)
{
   DDLevel = 7;            // Update level
   lpDD4->Release();       // Release old pointer
   lpDD = (LPDIRECTDRAW) lpDD7;
}
```

The resulting pointer assumes the functionality of IDirectDraw.

Although programmers often wish to know what is the DirectX interface version installed in the host machine, writing substantial application code that runs in several possible interfaces can be a complicated matter. One problem often encountered is that function signatures change from one interface to another one. For example, a function call in IDirectDraw may take three parameters and the same function takes four parameters in IDirectDraw2. Accommodating the variations in the different interfaces usually requires a considerable amount of contingency code.

A more practical strategy is to decide beforehand which is the lowest interface version required by the application, then make sure that this version of the interface is available in the host machine. The strategy usually works since the COM architecture insures that the functionality of older versions of the interface is always maintained. For example, if you have decided that your application requires the IDirectDraw7 interface, the following function can be used to test for the presence of this interface level and to obtain the corresponding pointer.

```
// Global variable
LPDIRECTDRAW7       lpDD7;     // Pinter to IDirectDraw7
.
.
.
//*****************************************
// Name: DD7Interface()
// Desc: Created DirectDraw object and finds
//       DirectDraw7 interface
// PRE:
//       Caller's code contains pointer
//       variable:
//       LPDIRECTDRAW7   lpDD7;
//
// POST:
```

```
//   returns 0 if no DirectDraw7 interface
//   returns 1 if DirectDraw7 found
//            Caller's pointer variable is
//            initialized
//****************************************
int DD7Interface()
{

  HRESULT         DDConnect;
  LPDIRECTDRAW    lpDD;    // Pointer to DirectDraw

  DDConnect = DirectDrawCreate ( NULL,
              &lpDD,
              NULL);

  if(DDConnect != DD_OK)
        return 0;
  // Atempt to locate DirectDraw4 interface
  DDConnect = lpDD->QueryInterface(
              IID_IDirectDraw7,
           (LPVOID *) &lpDD7);
  if(DDConnect != S_OK)
        return 0;

  lpDD->Release();      // Release old pointer
      return 1;

}
```

The function DD7Interface(), listed previously is used by most of the DirectDraw sample programs listed in this book.

26.2.3 Setting the Cooperative Level

A DirectDraw application can obtain almost exclusive control over the system hardware resources, while a normal Windows application shares these resources with other programs. One of the most critical of these resources is the video system. Control over the video system is necessary for implementing some types of interactive, animated games and other high-performance graphics programs. But not all DirectDraw programs need this special functionality. Some DirectDraw applications execute in a window and behave like a normal Windows program. The SetCooperativeLevel() function is used to request a specific level of resource control and, at the same time, to establish the level of cooperation with other Windows applications.

The function SetCooperativeLevel() has slightly different implementations in the DirectDraw, DirectDraw2, DirectDraw4, and DirectDraw7 interfaces. The following discussion refers to IDirectDraw7::SetCooperativeLevel.

The basic decision that must be made at the time of calling SetCooperativeLevel() is whether the application is to run full-screen, with exclusive access to the display resources, or as a normal windowed program. DirectDraw cooperative levels also have the following effects:

1. Enable DirectDraw to use of Mode X resolutions.

2. Prevent DirectDraw from releasing exclusive control of the display or rebooting if the user pressed CTRL+ALT+DEL.
3. Enable DirectDraw to minimize or maximize the application in response to user events.

Table 26-1 lists the predefined constants that are recognized by the SetCooperativeLevel() function.

Table 26-1

Cooperative Level Symbolic Constants

FLAG	DESCRIPTION
DDSCL_ALLOWMODEX	Allows the use of Mode X display modes. This flag can only be used with the DSCL_EXCLUSIVE and DDSCL_FULLSCREEN modes.
DDSCL_ALLOWREBOOT	Allows the CTRL+ALT+DEL keystroke to function while in exclusive mode.
DDSCL_CREATEDEVICEWINDOW	DirectDraw is to create and manage a default device window for this DirectDraw object. Focus and device windows are multi-monitor functions supported by Windows 98 and Windows 2000.
DDSCL_EXCLUSIVE	Requests the exclusive level. Must be used with DDSCL_FULLSCREEN.
DDSCL_FPUPRESERVE	The calling application cares about the FPU state and does not want Direct3D to modify it in ways visible to the application. In this mode, Direct3D saves and restores the FPU state every time that it needs to modify the FPU state.
DDSCL_FPUSETUP	Indicates that the DirectDraw application will keep the math unit set up for single precision and exceptions disabled, which is the best setting for optimal Direct3D performance.
DDSCL_FULLSCREEN	The exclusive-mode owner is responsible for the entire primary surface. GDI is ignored. Must be used with DDSCL_EXCLUSIVE.
DDSCL_MULTITHREADED	Requests multithread-safe DirectDraw behavior. This causes Direct3D to take the global critical section more frequently.
DDSCL_NORMAL	Indicates a regular Windows application. Cannot be used with the DDSCL_ALLOWMODEX, DDSCL_EXCLUSIVE, or DDSCL_FULLSCREEN flags. Applications executing in this mode cannot perform page flipping or change the palette.
DDSCL_NOWINDOWCHANGES	DirectDraw is not allowed to minimize or restore the application window.
DDSCL_SETDEVICEWINDOW	The hWnd parameter is the window handle of the device window for this DirectDraw object. DDSCL_SETFOCUSWINDOW The hWnd parameter is the window handle of the focus window for the DirectDraw object. Cannot be used with the

	DDSCL_SETDEVICEWINDOW flag. Supported by Windows 98 and NT 5.0 only.

The SetCooperativevel() function's general form is as follows:

```
HRESULT SetCooperativeLevel(
                           HWND hwnd,        // 1
                           DWORD  aDword     // 2
                           );
```

The first parameter (hwnd) is the handle to the application window; however, if an application requests DDSCL_NORMAL in the second parameter, it can use NULL for the window handle. The second parameter (aDword) is one or more of the flags defined by the symbolic constants listed in Table 26-1. The function returns DD_OK if the call succeeds, or one of the following error messages:

- DDERR_EXCLUSIVEMODEALREADYSET
- DDERR_HWNDALREADYSET
- DDERR_HWNDSUBCLASSED
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_OUTOFMEMORY

The DDERR_EXCLUSIVEMODEALREADYSET message refers to the fact that only one application can request the exclusive mode. If this message is received, then there is another application that has been granted the exclusive mode and code should provide alternative processing or an exit.

Full-screen applications receive the DDERR_NOEXCLUSIVEMODE return value if they lose exclusive device access, as is the case when the user has pressed ALT+TAB to switch to another program. In this event one possible coding alternative is to call TestCooperativeLevel() function in a loop, exiting only when it returns DD_OK, indicating that exclusive mode is now available.

Applications that use the normal cooperative level (DDSCL_NORMAL flag) receive DDERR_EXCLUSIVEMODEALREADYSET if another application has taken exclusive device access. In this case a windowed application can be coded to loop until TestCooperativeLevel() returns DD_OK.

The two most common flag combinations used in the SetCooperativeLevel() call are for programs that execute in exclusive mode and those that are windowed. The following code fragment shows the call to SetCooperativeLevel() for a DirectDraw application that requests exclusive mode:

```
LPDIRECTDRAW       lpDD;      // DirectDraw object
HWND               hwnd;      // Handle to the window
...
lpDD->SetCooperativeLevel(hwnd, DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN);
```

For some unexplained reason two flags are required to set DirectDraw exclusive mode: DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN. In reality all exclusive mode applications execute full screen so the second flag is actually redundant.

To set the cooperative level to the normal mode the code can be as follows:

```
LPDIRECTDRAW      lpDD;    // DirectDraw object
...
lpDD->SetCooperativeLevel(NULL, DDSCL_NORMAL);
```

Note that exclusive mode applications pass the handle to the window parameter (hwnd) so that Windows has a way of recovering from conditions that freeze the otherwise disabled video system. This is not required for normal Windows programs that use conventional recovery procedures.

26.2.4 Hardware Capabilities

Conventional Windows applications often ignore the specific configuration of the system hardware; however, this is not the case in programs that use DirectDraw. Video systems support DirectDraw to varying degrees of hardware compatibility and to varying degrees of DirectDraw functionality. Most DirectDraw programs need to know what level of DirectDraw hardware support is available in a particular machine, as well as the amount of video memory available, before deciding if the code is compatible with the host, or how to proceed if a given functionality is not present.

In order to determine the supported hardware-accelerated features a DirectDraw application can enumerate the hardware capabilities. In general, it is safe to assume that most features that are not implemented in hardware are emulated by DirectX. Notice, however, that there are a few cases in which this is not true. It is this emulation that makes possible some degree of device independence. The IDirectDraw7::GetCaps function returns runtime information about video resources and hardware capabilities. By examining these capabilities during the initialization stage, an application can decide whether the available functionality is insufficient and abort execution, or make other adjustments, in order to provide the best possible performance over varying levels of support.

It has been documented that, in some cases, a particular combination of hardware supported features and emulation can result in worse performance than emulation alone. For example, if the device driver does not support stretch blitting from video memory, noticeable performance losses occur. The reason is that video memory is usually slower than system memory, which forces the CPU to wait when accessing video memory surfaces. For this reason, applications that use features not supported by the hardware are usually better off creating surfaces in system memory, rather than in video memory.

The GetCaps() function returns the capabilities of the device driver for the hardware (HAL) and for the hardware-emulation layer (HEL). The general form of the GetCaps() function is as follows:

```
HRESULT GetCaps(
                LPDDCAPS lpDDDriverCaps,     // 1
                LPDDCAPS lpDDHelCaps         // 2
                );
```

The first parameter (lpDDDriverCaps) is the address of a structure of type DDCAPS that is filled with the capabilities of the hardware abstraction layer (HAL), as reported by the device driver. Code can set this parameter to NULL if the hardware capabilities are not necessary. The second parameter (lpDDHelCaps) is the address of a structure, also of type DDCAPS, that is filled with the capabilities of the hardware emulation layer (HEL). This parameter can also be set to NULL if these capabilities are not to be retrieved. Code can only set one of the two parameters to NULL, otherwise the call would be trivial. If the method succeeds, the return value is DD_OK. If the method fails, the return value is one of the following error constants:

- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS

The DDCAPS structure is a large one indeed: it contains 58 members in the DirectDraw4 and DirectDraw7 versions. The structure is defined as follows:

```
typedef struct _DDCAPS {
  DWORD    dwSize;                  // size of structure DDCAPS
  DWORD    dwCaps;                  // driver-specific caps
  DWORD    dwCaps2;                 // more driver-specific caps
  DWORD    dwCKeyCaps;              // color key caps
  DWORD    dwFXCaps;                // stretching and effects caps
  DWORD    dwFXAlphaCaps;           // alpha caps
  DWORD    dwPalCaps;               // palette caps
  DWORD    dwSVCaps;                // stereo vision caps
  DWORD    dwAlphaBltConstBitDepths;
                                    // alpha bit-depth members
  DWORD    dwAlphaBltPixelBitDepths;       //   .
  DWORD    dwAlphaBltSurfaceBitDepths;     //   .
  DWORD    dwAlphaOverlayConstBitDepths;   //   .
  DWORD    dwAlphaOverlayPixelBitDepths;   //   .
  DWORD    dwAlphaOverlaySurfaceBitDepths; //   .
  DWORD    dwZBufferBitDepths;      // Z-buffer bit depth
  DWORD    dwVidMemTotal;           // total video memory
  DWORD    dwVidMemFree;            // total free video memory
  DWORD    dwMaxVisibleOverlays;    // maximum visible overlays
  DWORD    dwCurrVisibleOverlays;   // overlays currently visible
  DWORD    dwNumFourCCCodes;        // number of supported FOURCC
                                    // codes
  DWORD    dwAlignBoundarySrc;      // overlay alignment
                                    // restrictions
  DWORD    dwAlignSizeSrc;          //   .
  DWORD    dwAlignBoundaryDest;     //   .
  DWORD    dwAlignSizeDest;         //   .
  DWORD    dwAlignStrideAlign;      // stride alignment
  DWORD    dwRops[DD_ROP_SPACE];    // supported raster ops
  DWORD    dwReservedCaps;          // reserved
  DWORD    dwMinOverlayStretch;     // overlay stretch factors
  DWORD    dwMaxOverlayStretch;     //   .
  DWORD    dwMinLiveVideoStretch;   // obsolete
  DWORD    dwMaxLiveVideoStretch;   //   .
  DWORD    dwMinHwCodecStretch;     //   .
  DWORD    dwMaxHwCodecStretch;     //   .
  DWORD    dwReserved1;             // reserved
  DWORD    dwReserved2;             //   .
  DWORD    dwReserved3;             //   .
  DWORD    dwSVBCaps;               // system-to-video blit related
                                    // caps
  DWORD    dwSVBCKeyCaps;           //   .
  DWORD    dwSVBFXCaps;              //  .
  DWORD    dwSVBRops[DD_ROP_SPACE]; //  .
  DWORD    dwVSBCaps;                // video-to-system blit related caps
  DWORD    dwVSBCKeyCaps;            //  .
  DWORD    dwVSBFXCaps;              //  .
  DWORD    dwVSBRops[DD_ROP_SPACE]; //  .
  DWORD    dwSSBCaps;                // system-to-system blit related
                                     // caps
  DWORD    dwSSBCKeyCaps;            //  .
  DWORD    dwSSBCFXCaps;             //  .
  DWORD    dwSSBRops[DD_ROP_SPACE]; //  .
  DWORD    dwMaxVideoPorts;          // maximum number of live video
                                     // ports
  DWORD    dwCurrVideoPorts;         // current number of live video
                                     // ports
  DWORD    dwSVBCaps2;               // additional system-to-video
                                     // blit
                                     // caps
  DWORD    dwNLVBCaps;               // nonlocal-to-local video
                                     // memory
                                     //  blit caps
  DWORD    dwNLVBCaps2;              //  .
  DWORD    dwNLVBCKeyCaps;           //  .
  DWORD    dwNLVBFXCaps;             //  .
  DWORD    dwNLVBRops[DD_ROP_SPACE];//  .
  DDSCAPS2 ddsCaps;                  // general surface caps
DDCAPS,FAR* LPDDCAPS;
```

Most applications are only concerned with a few of the capabilities of a DirectDraw device. Table 26-2 lists some of the most often needed capabilities.

Table 26-2

Device Capabilities in the GetCaps() Function

DWCAPS	MEMBER CONSTANTS:
DDCAPS_3D	Display hardware has 3D acceleration.
DDCAPS_ALPHA	Display hardware supports alpha-only surfaces.
DDCAPS_BANKSWITCHED	Display hardware is bank-switched. Therefore it is very slow at random access operations to display memory.
DDCAPS_BLT	Display hardware is capable of blit operations.
DDCAPS_BLTCOLORFILL	Display hardware is capable of color filling with a blitter.
DDCAPS_BLTSTRETCH	Display hardware is capable of stretching during blit operations.
DDCAPS_CANBLTSYSMEM	Display hardware is capable of blitting to or from system memory.
DDCAPS_CANCLIP	Display hardware is capable of clipping with Blitting.

(continues)

Table 26-2

Device Capabilities in the GetCaps() Function (continued)

DWCAPS	MEMBER CONSTANTS:
DDCAPS_CANCLIPSTRETCHED	Display hardware is capable of clipping while stretch blitting.
DDCAPS_COLORKEY	System supports some form of color key in either overlay or blit operations.
DDCAPS_GDI	Display hardware is shared with GDI.
DDCAPS_NOHARDWARE	No hardware support.
DDCAPS_OVERLAY	Display hardware supports overlays.
DDCAPS_OVERLAYCANTCLIP	Display hardware supports overlays but cannot clip.
DDCAPS_PALETTE	DirectDraw is capable of creating and supporting DirectDrawPalette objects for more surfaces than the primary one.
DDCAPS_READSCANLINE	Display hardware is capable of returning the current scan line.
DDCAPS_ZBLTS	Supports the use of z-buffers with blit operations.
DWCAPS2 MEMBER CONSTANTS:	
DDCAPS2_VIDEOPORT	Display hardware supports live video.
DDCAPS2_WIDESURFACES	Display surfaces supports surfaces wider than the primary surface.
DWPALCAPS MEMBER CONSTANTS:	
DDPCAPS_1BIT	Supports 2-color palettes.
DDPCAPS_2BIT	Supports 4-color palettes.
DDPCAPS_4BIT	Supports 16-color palettes.
DDPCAPS_8BIT	Supports 256-color palettes.
DDPCAPS_8BITENTRIES	Specifies an index to an 8-bit color index. This field is valid only when used with the DDPCAPS_1BIT, DDPCAPS_2BIT, or DDPCAPS_4BIT capability.
DDPCAPS_ALPHA	Supports palettes that include an alpha component.
DDPCAPS_ALLOW256	Supports palettes with all 256 entries defined.

(continues)

Table 26-2

Device capabilities in the GetCaps() Function (continued)

DWCAPS	MEMBER CONSTANTS:
DDPCAPS_PRIMARYSURFACE	The palette is attached to the primary surface. Changing the palette has an immediate effect on the display unless the DDPCAPS_VSYNC capability is specified and supported.
OTHER STRUCTURE MEMBERS:	
DwVidMemTotal	Total amount of display memory.
DwVidMemFree	Amount of free display memory.
DwMaxVisibleOverlays	Maximum number of visible overlays or overlay sprites.
DwCurrVisibleOverlays	Current number of visible overlays or overlay sprites.
DwReservedCaps	Reserved. Prior to DirectX 6.0, this member contained general surface capabilities.
DwMinOverlayStretch DwMaxOverlayStretch	Minimum and maximum overlay stretch factors multiplied by 1000. For example, 1.3 = 1300.
DwSVBCaps	Driver-specific capabilities for system-memory-to-display-memory blits.
DwVSBRops	Raster operations supported for display-memory-to-system-memory blits.
DwSSBCaps	Driver-specific capabilities for system-memory-to-system-memory blits.

The following code fragment shows the processing required in order to read the hardware capabilities using DirectDraw7::GetCaps. The code reads various capabilities and displays the corresponding screen messages. After each message is displayed, the screen position is indexed by one or more lines. The project named DD Info, in the book's software package, uses similar processing.

```
DDCAPS         DrawCaps;     // DDCAPS structure
LPDIRECTDRAW   lpDD;         // DirectDraw object
.
.
.
//**********************************
// DirectDraw hardware capabilities
//**********************************
DrawCaps.dwSize = sizeof ( DrawCaps );
// Call to capabilities function
   lpDD->GetCaps (&DrawCaps, NULL );
// Video memory
```

```
    strcpy ( message, " Total Video Memory: ");
    sprintf ( message+strlen(" Total Video Memory: "),
                "%i",DrawCaps.dwVidMemTotal );
    TextOut ( hdc, text_x, text_y, message, strlen (message) );
    text_y += tm.tmHeight+tm.tmExternalLeading;
// Free video memory
   strcpy ( message, " Free Video Memory: ");
   sprintf ( message+strlen(" Free Video Memory: "),
                "%i",DrawCaps.dwVidMemFree );
    TextOut ( hdc, text_x, text_y, message, strlen (message) );
    text_y += tm.tmHeight+tm.tmExternalLeading;
    text_y += tm.tmHeight+tm.tmExternalLeading;

   // Video card hardware
    strcpy ( message,
            " Video card hardware support as follows:");
    TextOut ( hdc, text_x, text_y, message, strlen (message) );
    text_y += tm.tmHeight+tm.tmExternalLeading;
    text_x = 16;
    if (DrawCaps.dwCaps & DDCAPS_NOHARDWARE)
    {
    strcpy ( message, "  No DirectDraw hardware support available" );
        TextOut ( hdc, text_x, text_y, message, strlen (message) );
        text_y += tm.tmHeight+tm.tmExternalLeading;
        return;
    }
    if (DrawCaps.dwCaps & DDCAPS_3D)
    {
        strcpy ( message, "  3D support" );
        TextOut ( hdc, text_x, text_y, message, strlen (message) );
        text_y += tm.tmHeight+tm.tmExternalLeading;
    }
   else
    {
        strcpy ( message, "  No 3D support" );
        TextOut ( hdc, text_x, text_y, message, strlen (message) );
        text_y += tm.tmHeight+tm.tmExternalLeading;
    }
    if (DrawCaps.dwCaps & DDCAPS_BLT)
    {
        strcpy ( message, "  Hardware Bitblt support" );
        TextOut ( hdc, text_x, text_y, message, strlen (message) );
        text_y += tm.tmHeight+tm.tmExternalLeading;
    }
   else
    {
        strcpy ( message, "  No hardware Bitblt support" );
        TextOut ( hdc, text_x, text_y, message, strlen (message) );
        text_y += tm.tmHeight+tm.tmExternalLeading;
    }
    if (DrawCaps.dwCaps & DDCAPS_OVERLAY)
    {
        strcpy ( message, "  Hardware overlays supported");
        TextOut ( hdc, text_x, text_y, message, strlen (message) );
        text_y += tm.tmHeight+tm.tmExternalLeading;
    }
   else
    {
        strcpy ( message, "  No hardware overlays ");
        TextOut ( hdc, text_x, text_y, message, strlen (message) );
```

```
         text_y += tm.tmHeight+tm.tmExternalLeading;
     }

    if (DrawCaps.dwCaps & DDCAPS_CANCLIP)
     {
         strcpy ( message, "  Clipping supported in hardware");
         TextOut ( hdc, text_x, text_y, message, strlen (message) );
         text_y += tm.tmHeight+tm.tmExternalLeading;
     }
    else
     {
         strcpy ( message, "  No hardware clipping support ");
         TextOut ( hdc, text_x, text_y, message, strlen (message) );
         text_y += tm.tmHeight+tm.tmExternalLeading;
     }
     if (DrawCaps.dwCaps & DDCAPS_BANKSWITCHED)
     {
         strcpy ( message, "  Memory is bank switched");
         TextOut ( hdc, text_x, text_y, message, strlen (message) );
         text_y += tm.tmHeight+tm.tmExternalLeading;
     }
    else
     {
         strcpy ( message, "  Memory not bank switched");
         TextOut ( hdc, text_x, text_y, message, strlen (message) );
         text_y += tm.tmHeight+tm.tmExternalLeading;
     }
        if (DrawCaps.dwCaps & DDCAPS_BLTCOLORFILL)
     {
         strcpy ( message, "  Color fill Blt support");
         TextOut ( hdc, text_x, text_y, message, strlen (message) );
         text_y += tm.tmHeight+tm.tmExternalLeading;
     }
    else
     {
         strcpy ( message, "  No Blt color fill support");
         TextOut ( hdc, text_x, text_y, message, strlen (message) );
         text_y += tm.tmHeight+tm.tmExternalLeading;
     }
     if (DrawCaps.dwCaps & DDCAPS_COLORKEY)
     {
         strcpy ( message, "  Color key hardware support ");
         TextOut ( hdc, text_x, text_y, message, strlen (message) );
         text_y += tm.tmHeight+tm.tmExternalLeading;
     }
    else
     {
         strcpy ( message, "  No color key support");
         TextOut ( hdc, text_x, text_y, message, strlen (message) );
         text_y += tm.tmHeight+tm.tmExternalLeading;
     }
     if (DrawCaps.dwCaps & DDCAPS_ALPHA)
     {
         strcpy ( message, "  Alpha channels supported");
         TextOut ( hdc, text_x, text_y, message, strlen (message) );
         text_y += tm.tmHeight+tm.tmExternalLeading;
     }
    else
     {
         strcpy ( message, "  No Alpha channels support");
         TextOut ( hdc, text_x, text_y, message, strlen (message) );
         text_y += tm.tmHeight+tm.tmExternalLeading;
     }
```

26.2.5 Display Modes

The DOS concept of a display mode has a different flavor in DirectDraw programming. In DOS display modes are numbered, in DirectDraw a display mode is defined by its resolution and color depth. Therefore, a DirectDraw display mode of 640 by 480 by 8 executes with a resolution of 640 pixel rows, 480 pixel columns, and encodes the pixel attribute in 8 bits. Since 8 bits support 256 combinations, this mode supports a range of 256 colors. DirectDraw applications can obtain the available display modes. An application that executes in exclusive mode can also set a display mode and restore the previous one when it concludes.

Not all devices support all display modes. To determine the display modes supported on a given system, an application can call EnumDisplayModes. The function can be used to list all supported display modes, or to confirm that a single display mode is available in the hardware. The function's general form is as follows:

```
HRESULT EnumDisplayModes(
              DWORD dwFlags,                      // 1
              LPDDSURFACEDESC2 lpDDSurfaceDesc,   // 2
              LPVOID lpContext,                   // 3
              LPDDENUMMODESCALLBACK2 lpCallBack   // 4
              );
```

The first parameter (dwFlags) determines the function's options by means of two flags: DDEDM_REFRESHRATES and DDEDM_STANDARDVGAMODES. The first flag (DDEDM_REFRESHRATES) enumerates separately the modes that have different refresh rates, even if they have the same resolution and color depth. The second flag (DDEDM_STANDARDVGAMODES) enumerates Mode X and VGA Mode 13H as different modes. This parameter can be set to 0 to ignore both of these options.

The second parameter (lpDDSurfaceDesc) is the address of a DDSURFACEDESC2 structure. The structure is used to store the parameters of a particular display mode, which can be confirmed or not by the call. This parameter is set to NULL in order to request a listing of all available modes.

The third parameter (lpContext) is a pointer to an application-defined structure that is passed to the callback function associated with EnumDisplayModes(). This provides a mechanism whereby the application code can make local data visible to the callback function. If not used, as is most often the case, then the third parameter is set to NULL.

The fourth parameter (lpCallBack) is the address of a callback function, of prototype EnumModesCallback2(). This function is called every time a supported mode is found. Applications use this callback function to provide the necessary processing for each display mode reported by the call.

The callback function, whose address your code supplies when it calls EnumDisplayModes() must match the prototype for EnumModesCallback2(). Each time that a supported mode is found, the callback function receives control. The function's general form is:

```
HRESULT WINAPI EnumModesCallback(
                 LPDDSURFACEDESC2 lpDDSurfaceDesc2,     // 1
                 LPVOID lpContext                       // 2
                 );
```

The first parameter (lpDDSurfaceDesc2) is the address of a DDSURFACEDESC2 structure that describes the display mode. The second one (lpContext) is the address of the application-defined data structure, which may have been passed in the third parameter of the EnumDisplayModes() function call. Code can examine the values in the DDSURFACEDESC2 structure to determine the characteristics of each available display mode.

The most important members of the DDSURFACEDESC2 structure are dwWidth, dwHeight, and ddpfPixelFormat. The dwWidth and dwHeight hold the display mode's dimensions. The ddpfPixelFormat member is a DDPIXELFORMAT structure that contains information about the mode's bit depth and describes whether the display mode is palletized or not. If the dwFlags member contains the DDPF_PALETTEINDEXED1, DDPF_PALETTEINDEXED2, DDPF_PALETTEINDEXED4, or DDPF_PALETTEINDEXED8 flag, then the display mode's bit depth is 1, 2, 4, or 8 bits. In this case the pixel value is an index into the corresponding palette. If dwFlags contains DDPF_RGB, then the display mode's bit depth in the dwRGBBitCount member of the DDPIXELFORMAT structure is valid.

Applications that call EnumDisplayModes() usually do most of the processing in the EnumModesCallback2() function. For example, a program can list all the DirectDraw display modes on the screen by storing the display modes data in one or more arrays each time the callback function receives control. When execution returns to the caller, then all modes have been stored or a predetermined maximum was reached. The calling code can now read the mode data from the arrays and display the values on the screen. In this case the callback function could be coded as follows:

```
// Global variables for DirectDraw operations
HRESULT          DDConnect;
DDCAPS           DrawCaps;
// DirectDraw object
LPDIRECTDRAW   lpDD;
int            DDLevel = 0;         // DirectDraw implementation
// DirectDraw modes data
int modesCount = 0;         // Counter for DirectDraw modes
static int MAX_MODES = 60;  // Maximum number of modes
DWORD modesArray[180];      // Array for mode data
                            // 3 parameters per mode
.
.
.
//***********************************************************
// Callback function for EnumDisplayModes()
//***********************************************************
```

```
static WINAPI ModesProc(LPDDSURFACEDESC aSurface,
                        LPVOID Context)
{
    static int i;         // Index into array
    i = modesCount * 3;   // Set array pointer
   // Store mode parameters in public array
   // Note: code assumes that the dwRGBBitFormat member of
   //       the DDPIXELFORMAT structure contains valid data
   modesArray[i]     = aSurface->dwWidth;
   modesArray[i + 1] = aSurface->dwHeight;
   modesArray[i + 2] = aSurface->ddpfPixelFormat.dwRGBBitCount;
   modesCount++;          // Bump display modes counter
   // Check for maximum number of display modes
   if( modesCount >= MAX_MODES )
     return DDENUMRET_CANCEL;  // Stop mode listing

   else
     return DDENUMRET_OK;  // Continue
}
```

The callback function, named ModesProc(), uses an array of type DWORD to store the height, width, and color depth for each mode reported by DirectDraw. A public variable named modesCount keeps track of the total number of modes. In this case the calling code can be implemented in a function called DDModes, as follows:

```
//**********************************************************
// DDModes - Obtain and list DD display modes
//**********************************************************
void DDModes ( HDC hdc )
{

    TEXTMETRIC  tm;
    char        message[255];
    int j = 0;                      // Display buffer offset
    int i = 0;                      // Modes counter
    int x;                          // Loop counter

    int         text_x = 0;
    int         text_y = 0;
    int         cxChar;


    GetTextMetrics ( hdc, &tm );
    cxChar = tm.tmAveCharWidth ;


    // Test for no DirectDraw interface
    if (DDLevel == 0) {
        strcpy ( message, "No DirectDraw interface" );
        TextOut ( hdc, text_x, text_y, message, strlen (message) );
        return;
    }
    //********************************
    // if there is DirectDraw, obtain
    //    and list display modes
    //********************************
    strcpy ( message, "       DirectDraw Display Modes" );
    TextOut ( hdc, text_x, text_y, message, strlen (message) );
```

```
text_y += tm.tmHeight+tm.tmExternalLeading;

// Call EnumDisplayModes()
if(MODES_ON == 0){
    MODES_ON = 1;          // set switch
    DDConnect = lpDD->EnumDisplayModes(0, NULL, NULL, ModesProc);
}

if (DDLevel != 0) {
strcpy ( message, "       Number of display modes: ");
sprintf ( message+strlen("       Number of display modes: "),
            "%i", modesCount);
TextOut ( hdc, text_x, text_y, message, strlen (message) );
text_y += tm.tmHeight+tm.tmExternalLeading;

    // Format and display mode data
    // First column
    if(modesCount >= 20){
        for(x = 0; x < 20; x++)
        {
            if(x >= modesCount)
                break;
            j = sprintf ( message, "  %d", modesArray[i*3] );
            j += sprintf (message + j, " x %d", modesArray[i*3+1] );
            j += sprintf (message + j, " x %d", modesArray[i*3+2] );
            TextOut ( hdc, text_x, text_y, message, strlen (message) );
            text_y += tm.tmHeight+tm.tmExternalLeading;
            i++;
        }
    }
    // Done if 20 or less modes
    if(modesCount <= 20)
    return;

    // Display second column if more than 20 modes
    text_x = cxChar * 20;
    text_y = 2 * tm.tmHeight+tm.tmExternalLeading;
        for(x = 0; x < 20; x++)
        {
            if((x + 20) >= modesCount)
                break;
            j = sprintf ( message, "  %d", modesArray[i*3] );
            j += sprintf (message + j, " x %d", modesArray[i*3+1] );
            j += sprintf (message + j, " x %d", modesArray[i*3+2] );
            TextOut ( hdc, text_x, text_y, message, strlen (message) );
            text_y += tm.tmHeight+tm.tmExternalLeading;
            text_x = cxChar * 20;
            i++;
        }

    // Done if 40 or less modes
    if(modesCount <= 40)
    return;

    // Display third column if more than 40 modes
    text_x = cxChar * 40;
    text_y = 2 * tm.tmHeight+tm.tmExternalLeading;
        for(x = 0; x < 20; x++)
        {
            if((x + 40) >= modesCount)
```

```
                    break;
                j = sprintf ( message, "  %d", modesArray[i*3] );
                j += sprintf (message + j, " x %d", modesArray[i*3+1] );
                j += sprintf (message + j, " x %d", modesArray[i*3+2] );
                TextOut ( hdc, text_x, text_y, message, strlen (message) );
                text_y += tm.tmHeight+tm.tmExternalLeading;
                text_x = cxChar * 40;
                i++;
            }


        return;
    }
}
```

The processing calls EnumDisplayModes() in the statement:

```
  DDConnect = lpDD4->EnumDisplayModes(0, NULL, NULL, ModesProc);
```

The first parameter is set to 0 to indicate that no special control flags are required. Therefore, the refresh rate is not taken into consideration and mode X is not reported separately. The second parameter is NULL to indicate that no structure data for checking against available modes is used. The NULL value for the third parameter relates to the fact that no user-defined data structure is being passed to the callback function. The last parameter is the address of the callback function, in this case the ModesProc() function previously listed. When the callback function returns, the code tests for a return value of DD_OK, which indicates that the call was successful, and then proceeds to display the header messages and to convert the code data stored in ModesArray[] into ASCII strings for display.

26.3 The DD Info Project

The program named DD info.cpp, located in the DD Info project folder of the book's software package, is a demonstration of the initialization and preparatory operations for a DirectDraw application. The program starts by initializing DirectDraw. The program's menu contains commands to read and display system hardware information and to list the available display modes.

One of the first operations performed by the DD info program, which is the source for the DD Info Project, is to determine which version of the DirectDraw interface is installed in the target system. Then the code obtains and displays hardware support, and lists the available display modes in whatever DirectDraw interface is present.

Chapter 27

DirectDraw Exclusive Mode

Topics:

- Programming DirectDraw in exclusive mode
- Developing WinMain() for exclusive mode
- Initializing for DirectDraw exclusive mode
- Using GDI functions
- A DirectDraw exclusive mode template

In Chapter 26 you saw how a DirectDraw application is configured and initialized. You also learned the basics of DirectDraw architecture and developed a conventional windowed program that uses DirectDraw functions. But the fundamental purpose of DirectDraw programming is high-performance graphics. This requires a DirectDraw program that executes in exclusive mode, which is made easier if we first develop a code structure that can serve as a template for this type of application. The template must perform two critical tasks: create a WinMain() function suited for DirectDraw exclusive mode, and ensure access to the latest version of the DirectDraw interface.

27.1 WinMain() for DirectDraw

A WinMain() function DirectDraw programming in exclusive mode has some unique features, since the program needs to perform several DirectDraw-specific initializations that are not common in standard Windows. In this section we develop a template for DirectDraw exclusive mode programming that includes a suitable version of WinMain().

In addition to the usual Windows initializations, the DirectDraw-specific WinMain() must perform the following operations:

- Obtain the DirectDraw interface and store the interface pointer.
- Confirm that the desired mode is available in the host machine.
- Set the cooperative level.
- Set the display mode.

- Create the drawing surfaces. Most DirectDraw programs require at least one primary surface.
- Obtain the DirectDraw device context if the program is to execute GDI functions.

The WinMain() function for DirectDraw exclusive mode creates the program window and performs Windows and DirectDraw-specific initialization and setup. The following are the fundamental tasks to be performed by WinMain():

- Create and fill the WNDCLASS structure.
- Register the window class.
- Create the DirectDraw-compatible window.
- Set the window's show state.
- Provide a suitable message loop, according to the application type.

27.1.1 Filling the WNDCLASSEX Structure

The WINDCLASSEX structure contains window class information. There are not many differences between the WNDCLASSEX structure that is used in conventional Windows programming, and the one required for an exclusive mode DirectDraw application. One difference that can be noted is that a DirectDraw window class does not use a private device context; therefore, the CS_OWNDC constant is not present in the style member of the WNDCLASSEX structure member. In the template file the structure is filled as follows:

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

27.1.2 Registering the Window Class

The window class serves to define the characteristics of a particular window as well as the address of the window procedure. Having filled the structure members, code can now register the class, with the following call:

```
RegisterClass(&wndclass);
```

27.1.3 Creating the Window

Once the WNDCLASSEX structure has been initialized, you can proceed to create the window by means of the CreateWindowEx() function. Many combinations of parameters can be used in the call, according to the characteristics desired for the specific application window. In the case of a DirectDraw exclusive mode application, some of the predefined values are meaningless.

The extended style WS_EX_TOPMOST defines a window that is placed above all non-topmost windows. WS_EX_TOPMOST is usually the appropriate style for creating a DirectDraw exclusive mode application. The window style parameter should be WS_POPUP. If the DirectDraw application is to execute full screen, which is always the case in exclusive mode, then the horizontal and vertical origins are set to zero and the xsize and ysize parameters are filled using GetSystemMetrics(), which returns the pixel size of the entire screen area. In the template file the structure is filled as follows:

```
hWnd = CreateWindowEx(
       WS_EX_TOPMOST,        // Extended style
       szAppName,            // Application name
       "DirectDraw Demo No. 2",
       WS_POPUP,             // Window style
       0,                    // Horizontal origin
       0,                    // Vertical origin
       GetSystemMetrics(SM_CXSCREEN), // x size
       GetSystemMetrics(SM_CYSCREEN), // y size
       NULL,             // Handle of parent
       NULL,             // Handle to menu
       hInstance,        // Application instance
       NULL);            // Additional data
if (!hWnd)
   return FALSE;
```

27.1.4 Defining the Window Show State

CreateWindowEx() creates the window internally but does not display it. Code specifies how the window is to be shown by calling ShowWindow(). Conventional Windows programs first call ShowWindow() to set the show state, and then UpdateWindow() to update the client area by sending a WM_PAINT message to the window procedure. It is different in the case of a DirectDraw exclusive mode application. Since the DirectDraw interface has not been yet established, no WM_PAINT message can be sent at this point. This explains why the template file makes the call to the ShowWindow() function, but not the one to UpdateWindow(). The code is as follows:

```
ShowWindow (hwnd, iCmdShow);
```

The first parameter (hwnd) is the handle to the window returned by CreateWindowEx() function. The second parameter (iCndShow) is the window's display mode. In this first call to ShowWindow() applications must use the value received by WinMain().

27.1.5 Creating a Message Loop

At this point WinMain() can initialize DirectDraw and perhaps perform some preliminary display operations. The processing details in the case of the sample program are discussed in the following section. The last step in WinMain() is the ever-present message loop. In a standard DirectDraw exclusive mode application, the message loop is no different than the one in a conventional windows program. In the context of animation programming, later in this book, we discuss a different type of message loop. The present code is as follows:

```
  while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
  }
```

27.2 DirectDraw Initialization

Exclusive mode applications often initialize DirectDraw in WinMain(). The reason is that exclusive mode applications cannot perform display operations until they have obtained the interface, set the cooperative level, and the display mode. If display operations are to be performed by means of GDI functions, then the application must also obtain the device context. Note that DirectDraw programs can draw to the screen using both GDI and direct memory access methods.

Typically, the DirectDraw exclusive mode initialization includes the following steps:

- Obtain the current interface. In DirectX 8 this is IDirectDraw7.
- Check that the desired display mode is available in the host machine.
- Set the cooperative level and display mode.
- Create the drawing surfaces. This usually means at least a primary surface, but often other surfaces are also necessary.
- Display some initial screen text or graphics.

Screen display operations can be accomplished by means of conventional GDI functions, by direct access to video memory, by DirectDraw specific functions, or by a combination of these methods. Programs of greater complexity usually perform other initialization, setup, and initial display functions at this time. The example used in this chapter has minimal DirectDraw functionality. In the chapters that follow we develop more complex DirectDraw programs.

A preliminary issue is to provide a mechanism whereby the DirectDraw application can recover if a terminal condition is encountered during initialization and setup. In our DirectDraw template program we have included a function named DDInitFailed() that creates a message box with the corresponding diagnostic prompt. When the user acknowledges by pressing the OK button, the terminal error handler destroys the application window and returns control to the operating system. The function is coded as follows:

```
//***************************************
// Name: DDInitFailed()
// Desc: This function is called if an
//       initialization operation fails
//***************************************
HRESULT DDInitFailed(HWND hWnd, HRESULT hRet, LPCTSTR szError)
{
    char   szBuff[128];

    sprintf(szBuff, szError);
    ReleaseObjects();
    MessageBox(hWnd, szBuff, "DirectDraw Demo No. 2", MB_OK);
    DestroyWindow(hWnd);
    return hRet;
}
```

The parameters for the DDInitFailed() function are the handle to the window, the result code from the call the caused the failure, and a string pointer with the diagnostic text to be displayed in the message box. All DirectDraw initialization calls performed in the template code test for a valid result code; and if no valid code is found, they exit execution through the DDInitFailed() function. The same is true of the DirectDraw examples used in the rest of the book.

27.2.1 Obtaining the Interface Pointer

In Chapter 26 we discussed suitable strategies for managing interface pointer versions. The first processing step is to obtain the DirectDraw object that corresponds to the desired version of the interface. It is usually a good idea to store the pointer in a global variable, which can be accessed by other program elements. The function named DD7Interface() attempts to find the IDirectDraw7 object. It is coded as follows:

```
// Global data
LPDIRECTDRAW7     lpDD7;    // Pointer to IDirectDraw7
.
.
.
//*******************************************
// Name: DD7Interface()
// Desc: Created DirectDraw object and finds
//       DirectDraw7 interface
// PRE:
//       Caller's code contains pointer
//       variable:
//       LPDIRECTDRAW7   lpDD7;
//
// POST:
//   returns 0 if no DirectDraw7 interface
//   returns 1 if DirectDraw7 found
//           Caller's pointer variable
//           is initialized
//*******************************************
int DD7Interface()
{

  HRESULT         DDConnect;
  LPDIRECTDRAW lpDD;    // Pointer to DirectDraw

  DDConnect = DirectDrawCreate ( NULL,
              &lpDD,
              NULL);

  if(DDConnect != DD_OK)
       return 0;
  // Atempt to locate DirectDraw4 interface
  DDConnect = lpDD->QueryInterface(
                    IID_IDirectDraw7,
                   (LPVOID *) &lpDD7);
  if(DDConnect != S_OK)
     return 0;

  lpDD->Release();      // Release old pointer
     return 1;
}
```

The above code releases the local pointer to DirectDraw if the IDirectDraw7 pointer is found. In this manner the application code need only be concerned with releasing the object actually in use. Note that the pointer to the DirectDraw7 interface pointer is defined globally, so that is can be seen throughout the application. In WinMain() the call to the DD4Interface() function is as follows:

```
// Attempt to fetch the DirectDraw4 interface
   hRet = DD7Interface();
       if (hRet == 0)
       return DDInitFailed(hWnd, hRet,
                          "QueryInterface() call failed");
```

If the IDirectDraw7 interface is not found, then the program exits through the DDInitFailed() function previously described. In the template file the diagnostic messages simply express the name of the failed function. In your own programs you will probably substitute these text messages for more appropriate ones. For example, the failure of the QueryInterface() call can be interpreted to mean that the user needs to upgrade the host system to DirectX 7 or later version. A more detailed diagnostics may be advisable in some cases.

27.2.2 Checking Mode Availability

If the call succeeds, we have obtained a pointer to IDirectDraw7. This pointer can be used in all DirectDraw function calls. The fact that we have a pointer to the version 7 of DirectDraw does not mean that the application will execute correctly. DirectDraw programming sometimes introduces hardware dependencies that are not a found in conventional Windows code.

Video display operations in DirectDraw are dependent upon the selected display mode. Before you attempt to set a display mode, it is usually a good idea to investigate if the desired mode is available in the host system. This gives code the opportunity to select an alternative mode if the ideal one is not available, or to exit with a diagnostic message if no satisfactory display mode is found.

In Chapter 26 we used of the EnumDisplayModes() function to list the display modes available in a system. The same function can be used to query if a particular mode is available. The code used in the template file is as follows:

```
//*********************************************
// Name: ModesProc
// Desc: Callback function for EnumDisplayModes()
//*********************************************
HRESULT WINAPI ModesProc(LPDDSURFACEDESC2 aSurface,
          LPVOID Context)
{
   static int i;         // Index into array of mode data

   i = modesCount * 3;   // Set array pointer

   if( modesCount >= MAX_MODES )
    return DDENUMRET_CANCEL;  // Stop mode listing

   // Store mode parameters in public array
   // Note: code assumes that the dwRGBBitFormat member if
   //       the DDPIXELFORMAT structure contains valid data
   modesArray[i]     = aSurface->dwWidth;
   modesArray[i + 1] = aSurface->dwHeight;
   modesArray[i + 2] = aSurface->ddpfPixelFormat.dwRGBBitCount;

   modesCount++;         // Bump display modes counter
   return DDENUMRET_OK;  // Continue
}

//***************************************************************
// Name: hasDDMode
// Desc: Tests for mode availability
//***************************************************************
// PRE:
// 1. Public variable modesArray[] to store mode data
// 2. Public int variable modesCount to store number of
//    display modes
// 3. ModesProc() is an associated function that stores
//    mode data in array and count modes
//
// POST:
//  Returns 1 if mode is available
//***************************************************************

int HasDDMode(DWORD pixWidth, DWORD pixHeight, DWORD pixBits)
{

 static HRESULT DDConnect;

 // Call EnumDisplayModes()
 if(MODES_ON == 0)
 {
    MODES_ON = 1;         // set switch
    DDConnect = lpDD4->EnumDisplayModes(0,
                                        NULL,
                                        NULL,
                                        ModesProc);
 }
// Modes are now stored in modeArray[] as triplets encoding
// width, height, and pixel bit size
// Variable modesCount holds the total number of display modes
 for(int x = 0; x < (modesCount * 3); x += 3)
 {
  if(modesArray[x]==pixWidth && modesArray[x+1]==pixHeight\
   && modesArray[x+2]==pixBits)
   return 1;
 }
 return 0;
}
```

DirectDraw documentation states the EnumDisplayModes() function can be passed the address of a DDSURFACEDESC2 structure, that is then checked for a specific mode. We have found that this mode of operation is not always reliable. In order to make sure that all available modes are checked, the HasDDMode() function loads a predetermined number of modes into a global array variable, then searches the array for the desired one.

In the WinMain() template, the call to the HasDDMode() function is coded as follows:

```
// Check for available mode (640 by 480 in 24-bit color)
if (HasDDMode(640, 480, 24) == 0)
        return DDInitFailed(hWnd, hRet, "Display mode not available");
```

The code in the HasDDMode() function provides no alternative processing for the case in which the desired true-color mode is not available in the system. In your own programs you may call HasDDMode() more than once, and provide alternative processing according to the best mode found. Here again, you should note that this programming style creates device-dependencies that could bring about other complications.

27.2.3 Setting Cooperative Level and Mode

If the desired mode is available, then the code must determine the cooperative level and proceed to set the mode. Exclusive mode DirectDraw programs require the constants DDSLC_EXCLUSIVE and DDSCL_FULLSCREEN. The processing is as follows:

```
 // Set cooperative level to exclusive and full screen
    hRet = lpDD7->SetCooperativeLevel(hWnd, DDSCL_EXCLUSIVE
                                            | DDSCL_FULLSCREEN);
    if (hRet != DD_OK)
        return DDInitFailed(hWnd, hRet,
                      "SetCooperativeLevel() call failed");
    // Set the video mode to 640 x 480 x 24
    hRet = lpDD7->SetDisplayMode(640, 480, 24, 0, 0);
    if (hRet != DD_OK)
        return DDInitFailed(hWnd, hRet,
                      "SetDisplayMode() call failed");
```

27.2.4 Creating the Surfaces

In Chapter 25 you encountered the concept of DirectDraw surfaces. At that time we defined a drawing surface as an area of video memory, typically used to hold image data, and IDirectDrawSurface as a COM object in itself. Most DirectDraw7 applications require at least two types of COM objects: one, of type LPDIRECTDRAW7, is a pointer to the DirectDraw object. The second one, of type LPDIRECTDRAWSURFACE4, is a pointer to a surface. All surface-related functions use this second pointer type, while the core DirectDraw calls require the first one. Applications that manipulate several surfaces often cast a pointer for each surface. A third type of object, called a DirectDraw palette object, is necessary for programs that perform palette manipulations, while DirectDraw clipper objects are used in clipping operations.

Before accessing a DirectDraw surface you must create it by means of a call to the IDirectDraw7::CreateSurface. The call can produce a single surface object, a complex surface-flipping chain, or a three-dimensional surface. The call to CreateSurface() specifies the dimensions of the surface, whether it is a single surface or a complex surface, and the pixel format. These characteristics are previously stored in a DDSURFACEDESC2 structure whose address is included in call's parameters. The function's general form is as follows:

```
HRESULT CreateSurface(
        LPDDSURFACEDESC2 lpDDSurfaceDesc,            // 1
        LPDIRECTDRAWSURFACE7 FAR *lplpDDSurface,     // 2
        IUnknown FAR *pkUnkOutter                    // 3
        );
```

The first parameter is the address of a structure variable of type LPDDSURFACEDESC2. The CreateSurface() API requires that all unused members of the structure be set to zero. In the code sample that follows you will see how this is easily accomplished. The second parameter is the address of a variable of type LPDIRECTDRAWSURFACE7 which is set to the interface address if the call succeeds. This is the pointer used in the calls that relate to this surface. Applications often store this pointer in a global variable so that it is visible throughout the code. The third parameter is provided for future expansion of the COM. Presently, applications must set this parameter to NULL.

If the call succeeds, the return value is DD_OK. If it fails the following self-explanatory error values can be returned:

- DDERR_INCOMPATIBLEPRIMARY
- DDERR_INVALIDCAPS
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_INVALIDPIXELFORMAT
- DDERR_NOALPHAHW
- DDERR_NOCOOPERATIVELEVELSET
- DDERR_NODIRECTDRAWHW
- DDERR_NOEMULATION
- DDERR_NOEXCLUSIVEMODE
- DDERR_NOFLIPHW
- DDERR_NOMIPMAPHW
- DDERR_NOOVERLAYHW
- DDERR_NOZBUFFERHW
- DDERR_OUTOFMEMORY
- DDERR_OUTOFVIDEOMEMORY
- DDERR_PRIMARYSURFACEALREADYEXISTS

- DDERR_UNSUPPORTEDMODE

DirectDraw always attempts to create a surface in local video memory. If there is not enough local video memory available, then DirectDraw tries to use non-local video memory. Finally, if no video memory is available at all, then the surface is created in system memory. The call to CreateSurface() can explicitly request that a surface be created in a certain type of memory. This is done by means of the appropriate flags in the associated DDSCAPS2 structure. DDSCAPS2 structure is part of DDSURFACEDESC2.

The primary surface is the one currently displayed on the monitor. It is identified by the DDSCAPS_PRIMARYSURFACE flag. There is only one primary surface for each DirectDraw object. The dimensions and pixel format of the primary surface must match the current display mode. It is not necessary to explicitly enter these values when calling CreateSurface(); in fact, specifying these parameters generates an error even if they match the ones in the current display mode. In this template program we create the simplest possible surface object, which is the one that corresponds to a primary surface. The code is as follows:

```
// Global data
LPDIRECTDRAWSURFACE7       lpDDSPrimary = NULL; // DirectDraw primary
DDSURFACEDESC2             ddsd;     // DirectDraw7 surface
.
.
.
// Create a primary surface
// ddsd is a structure of type DDSRUFACEDESC2
// First, zero all structure variables using the ZeroMemory()
// function
   ZeroMemory(&ddsd, sizeof(ddsd));
// Now fill in the required members
   ddsd.dwSize = sizeof(ddsd);     // Structure size
   ddsd.dwFlags = DDSD_CAPS ;
   ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

   hRet = lpDD7->CreateSurface(&ddsd, &lpDDSPrimary, NULL);
   if (hRet != DD_OK)
       return DDInitFailed(hWnd, hRet,
                      "CreateSurface() call failed");
```

If the call succeeds, we obtain a pointer by which to access the functions that relate to DirectDraw surfaces. The pointer, named lpDDSPrimary, is stored in a global variable of type LPDIRECTDRAWSURFACE7. The surface pointer can be used to obtain a DirectDraw device context, which allows using GDI graphics in the application, or to lock the surface for direct access and retrieve its base address and pitch.

27.2.5 Using Windows GDI Functions

DirectDraw applications have access to the display functions in the GDI. As in conventional Windows programming, access to GDI requires obtaining a handle to the Windows device context. For example, an application can use a GDI function to display a message on the screen. The DirectDrawSurface7 interface contains a function called GetDC() that can be used for this purpose. This function is not the same one as GetDC() in the general Windows API. Its general form is as follows:

```
HRESULT GetDC( HDC );
```

The function's only parameter is the address of the handle to the device context which is associated with the surface. If the call succeeds it returns DD_OK. If it fails it returns one of the following error codes:

- DDERR_DCALREADYCREATED
- DDERR_GENERIC
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_INVALIDSURFACETYPE
- DDERR_SURFACELOST
- DDERR_UNSUPPORTED
- DDERR_WASSTILLDRAWING

Note that the GetDC() function uses an internal version of the IDirectDrawSurface7::Lock function to lock the surface. The surface remains locked until the IDirectDrawSurface7::ReleaseDC function is called. In the template program the code proceeds as follows:

```
static char szDDMessage1[] =
 "Hello World -- Press <Esc> to end program";
  . . .
// Display some text on the primary surface
if(lpDDSPrimary->GetDC(&hdc) != DD_OK)
 return DDInitFailed(hWnd, hRet, "GetDC() call failed");
// Display screen message
   SetBkColor(hdc, RGB(0, 0, 255));       // Blue background
   SetTextColor(hdc, RGB(255, 255, 0));   // Yellow letters
   TextOut(hdc, 120, 200, szDDMessage1, lstrlen(szDDMessage1));
   lpDDSPrimary->ReleaseDC(hdc);
```

27.3 The DD Exclusive Mode Template

The project file named DD Exclusive Mode, in the book's software pckage, contains the program DD Exclusive Mode.cpp which can be used as a template for developing simple DirectDraw applications in exclusive mode. The code contains all of the support functions previously mentioned, that is, functions to find a DirectDraw7 interface object, to test for availability of a particular display mode, to release objects, and to handle terminal errors during DirectDraw initialization. The processing consists of displaying a screen message using the text output GDI service. The code also includes a skeletal window procedure to handle keyboard input, disable the cursor, and terminate execution.

Chapter 28

Access to Video Memory

Topics:

- Programming memory-mapped video
- Using in-line assembly language
- Multi-language programming
- Developing direct access primitives
- Raster operations

The program DD Exclusive Mode, developed in Chapter 12, initializes DirectDraw, defines the cooperative level, sets a display mode, and draws text on the screen using a GDI function. The preliminary and setup operations performed by the DD Exclusive Mode program are necessary in many DirectDraw applications. However, not much is gained in performance and control by a DirectDraw application that is limited to the GDI functions. The purpose of DirectX in general, and DirectDraw specifically, is to provide a higher level of control and to improve graphics rendering speed. Neither of these are achieved using the GDI services. Before an application can enjoy the advantages that derived from the DirectDraw interface, it must gain access to video memory. Once an application has gained access to video memory, the use of low-level code to further optimize processing becomes an option. The second level of DirectDraw advantages, those that result from using the hardware features in the video card, are discussed in the chapters that follow.

28.1 Direct Access Programming

Graphics programming in DOS is based on video functions being mapped to a specific area of system memory. The DOS graphics programmer determines the base address to which the video system is mapped, and the pixel format used in the current display mode. The code then proceeds to store pixel data in this memory area and the video hardware takes care of automatically updating the display by reflecting the contents of the memory region to which it is mapped. The process is simple, although in some display modes manipulating the data can be relatively complicated.

28.1.1 Memory-Mapped Video

The greatest difficulty of programming direct access to video memory in DOS is related to the segmented architecture of the Intel CPUs. The 16-bit internal architecture of the original Intel microprocessors limits each addressable segment to 64 kilobytes. Thus, a display mode with a resolution of 640 by 480 pixels, in which each pixel attribute is stored in 1 data byte, requires 307,200 bytes (640 x 480 x 1). Since each segment is limited to 65,536 bytes, the screen memory data exceeds the span of a single segment. In fact, 307,200 data bytes require five segments for storing the pixel information. This forces the use of data splitting schemes, one of which is called memory banking. By switching the segment mapping of a hardware element called the bank, it is possible to assign several areas of system memory to the same segment. The programming appears complicated, but once the access routines are developed for a particular display mode, the code can set any screen pixel to any desired color attribute by just passing the pixel's screen column and row address and the desired color code.

Until the advent of DirectDraw, Windows graphics programmers had no way of accessing video memory directly. Even if a Windows programmer had been able to find the address to which the video display was mapped in a particular system, any attempt to access this area of memory would generate a general protection fault. DirectDraw solves both problems: it temporarily relaxes the operating system's access restriction, and it provides information about the location and mapping of the video system.

An additional advantage is that in Win32 video display area is defined in a flat memory space. Once the base address of the video buffer is stored in a 32-bit register, the entire video memory space can be accessed without any segment mapping or memory banking scheme. In this case application code uses DirectDraw functions to obtain the base address of video memory, and its bit-to-pixel mapping. With this information, the DirectDraw program can proceed to perform display operations directly and in a straightforward manner.

Hi-Color Modes

The development of SuperVGA video cards, with more video memory than the standard VGA, made possible display modes with a much richer color range than had been previously available. Modes that devote 16 bits or more for the color encoding of each screen pixel are called the hi-color modes. Although no formal designation for these mode has been defined, the 16-bit per pixel modes are usually called real-color modes, and those with 24- and 32-bits per pixel are called true-color modes.

An adapter with 4Mb of video memory, which is common in today's desktop hardware, allows several real-color and true-color modes. The screen snapshot Figure 11-4 corresponds to a video card with 4Mb of memory. In this case real-color and true color modes are available up to a resolution of 1,600 by 1,200 pixels. The graphics programmer working with current video system technology can safely assume that most PCs support real-color and true-color modes with standard resolutions. However, many laptop computers have a more limited range of video modes.

Palettes were developed mostly to increase the colors available in modes with limited pixel depth. In Windows, all display modes with a resolution of 16-bits per pixel or higher are nonpalletized. For general graphics programming the use of palette-independent display modes considerably simplifies program design and coding. Programming today's video cards, with several megabytes of display memory, there is little justification for using palletized modes. All real-color and true-color modes are, by definition, nonpalletized. Figure 28-1 shows the mapping of video memory bits to pixel attributes in a real-color mode.

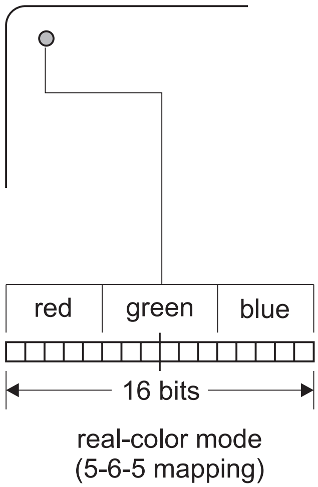


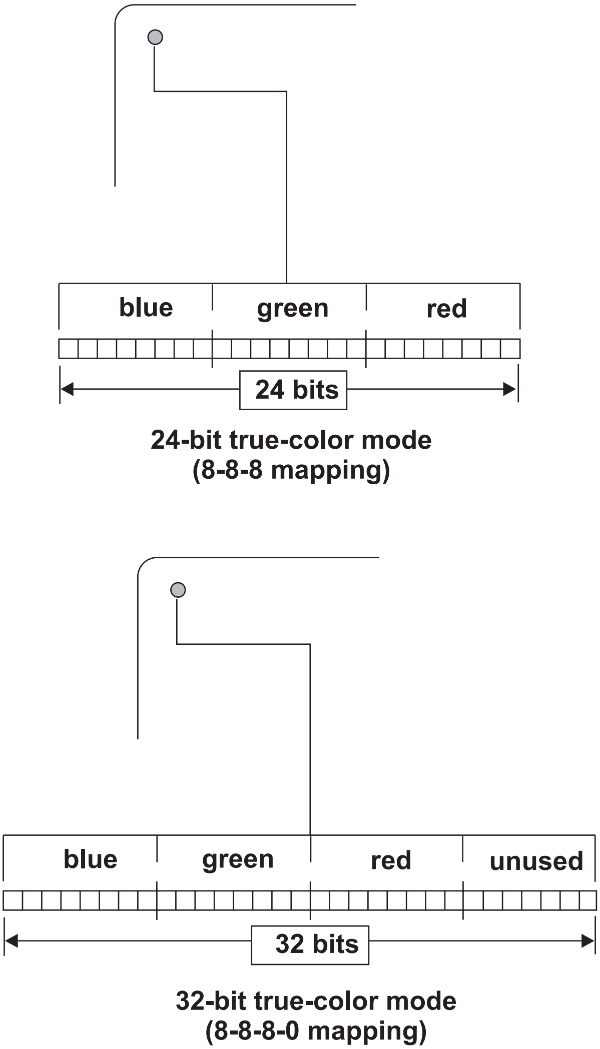
Figure 28-1 *Pixel Mapping in Real-Color Modes*

Not all real-color modes use the same mapping. The fact that 16 bits cannot be exactly divided into the three primary colors poses a problem. One possible solution is to leave 1 bit unused and map each primary color to a 5-bit field. Another option is to map the green color to a 6-bit field and the other two colors to 5-bit fields. This scheme, sometimes called a 5-6-5 mapping, is justified by the fact that the human eye is more sensitive to green than to red and blue. Figure 28-1 shows a 5-6-5 real-color mapping.

True-Color Modes

In the real color modes, the fact that the individual colors are not located at a byte boundary introduces some programming complications. To handle this uneven mapping code must perform bitwise operations to set the corresponding fields to the required values. The true-color modes, on the other hand, use 8 bits for each primary colors. This makes the direct access operations much easier to implement.

The name true-color relates to the idea that these modes are capable of reproducing a color range that is approximately equivalent to the sensitivity of the human eye. In this sense it is sometimes said that the true color modes produce a rendition that is of photographic quality. Two different mappings are used in the true-color modes, shown in Figure 28-2.



24-bit true-color mode
(8-8-8 mapping)

32-bit true-color mode
(8-8-8-0 mapping)

Figure 28-2 *Pixel Mapping in True-Color Modes*

The 24-bit mapping, shown at the top of Figure 28-2, uses three consecutive bytes to represent each of the primary colors. In the 32-bit mappings, at the bottom of Figure 28-2, there is an extra byte, which is unused, at the end of each pixel field. The reason for the unused byte in the 32-bit true-color modes relates to the 32-bit architecture of the Intel CPUs, which is also the bus width of most video cards. By assigning 4 bytes to encoding the color attribute it is possible to fill a pixel with a single memory transfer. A low-level program running in an Intel machine can store the pixel attribute in an extended machine register, such as EAX, and then transfer it into video memory with a single move instruction. By the same token, a C or C++ program can place the value into a variable of type LONG and use a pointer to LONG to move the data, also in a single operation. In other words, the 32-bit mapping scheme sacrifices memory storage space for the sake of faster rendering.

28.1.2 Locking the Surface

DirectDraw applications access the video buffer, or any surface memory area, by first calling the Lock() function. Lock() returns a pointer to the top-left corner of the rectangle that defines the surface, as well as the surface pitch and other relevant information necessary for accessing the surface. When calling Lock() the application can define a rectangular area within the surface, or the entire surface. If the surface is a primary surface, and the entire area is requested, then Lock() returns the base address of the video buffer and the number of bytes in each buffer row. This last parameter is called the surface pitch. When execution returns from the Lock() call the DirectDraw application has gained direct access to video display memory.

The Lock() function is related to the surface; therefore, it is accessed, not by the DirectDraw object, but by a surface object returned by the call to CreateSurface(). The use of the CreateSurface() function was discussed in Chapter 12. The general form of the Lock() function is as follows:

```
HRESULT Lock(
              LPRECT lpDestRect,                        // 1
              LPDDSURFACEDESC2 lpDDSurfaceDesc,         // 2
              DWORD dwFlags,                            // 3
              HANDLE hEvent                             // 4
              );
```

The first parameter is a pointer to a RECT structure that describes a rectangular area on the surface that is to be accessed directly. To lock the entire surface this parameter is set to NULL. If more than one rectangle is locked, they cannot overlap. The second parameter is the address of a structure variable of type DDSURFACEDESC2, which is filled with all the information necessary to access the surface memory directly. The information returned in this structure includes the base address of the surface, its pitch, and its pixel format. Applications should never make assumptions about the surface pitch, since this value changes according to the location of surface memory and even the version of the DirectDraw driver. The third parameter contains one or more flags that define the function's mode of operation. Table 28-1, on the following page, lists the constants currently implemented in the IDirectDrawSurface4 interface.

The DDLOCK_NOSYSLOCK flag relates to the fact that while a surface is locked DirectDraw usually holds the Win16Mutex (also known as the Win16Lock) so that gaining access to surface memory can occur safely. The Win16Mutex in effect shuts down Windows for the time that elapses between the Lock() and the Unlock() calls. If the DDLOCK_NOSYSLOCK flag is present, and the locked surface is not a primary surface, then the Win16Mutex does not take place. If a blit is in progress when Lock() is called, the function returns an error. This can be prevented by including the DDLOCK_WAIT flag, which causes the call to wait until a lock can be successfully obtained.

The fourth parameter to the Lock() call was originally documented to be a handle to a system event that is triggered when the surface is ready to be locked. The newest version of the DirectDraw documentation states that it is not used and should always be set to NULL.

Table 28-1

Flags the IDirectDrawSurface7::Lock Function

FLAG	MEANING
DDLOCK_DONOTWAIT	OIDirectDrawSurface7 interfaces, the default is DDLOCK_WAIT. Use this flag to override the default and use time when the accelerator is busy (as denoted by the DDERR_WASSTILLDRAWING return value).
DDLOCK_EVENT	Not currently implemented.
DDLOCK_NOOVERWRITE	New for DirectX 7.0. Used only with Direct3D vertex-buffer locks. This flag can be useful to append data to the vertex buffer.
DDLOCK_DISCARDCONTENTS	New for DirectX 7.0. Used only with Direct3D vertex-buffer locks. Indicates that no assumptions are made about the contents of the vertex buffer during this lock. This enables Direct3D or the driver to provide an alternative memory area as the vertex buffer. This flag is useful to clear the contents of the vertex buffer and fill in new data.
DDLOCK_OKTOSWAP	Obsolete. Replaced by DDLOCK_DISCARDCONTENTS.
DDLOCK_NOSYSLOCK	Do not take the Win16Mutex. This flag is ignored when locking the primary surface.
DDLOCK_READONLY	The surface being locked will only be read.
DDLOCK_SURFACEMEMORYPTR	A valid memory pointer to the top-left corner of the specified rectangle should be returned. If no rectangle is specified, a pointer to the top of the surface is returned. This is the default and need not be entered explicitly.
DDLOCK_WAIT	Retries lock if it cannot be obtained because a blit operation is in progress.
DDLOCK_WRITEONLY	The surface being locked will be write enabled.

Lock() returns DD_OK if it succeeds or one of the following error codes:

- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_OUTOFMEMORY
- DDERR_SURFACEBUSY
- DDERR_SURFACELOST
- DDERR_WASSTILLDRAWING

When Lock() succeeds, the application can retrieve a surface memory pointer and other necessary data and start accessing surface memory directly. Code can continue to access surface memory until a call to the Unlock() function is made. As soon as the surface is unlocked, the surface memory pointer becomes invalid. While the lock is in progress, applications cannot blit to or from surface memory. GDI functions fail when used on a locked surface.

28.1.3 Obtaining Surface Data

When the Lock() call returns DD_OK, the application can access the corresponding members of the DDSURFACEDESC2 structure variable passed as a parameter. This structure contains the data necessary for direct access. If application code knows the display mode and its corresponding pixel format, then the two data elements necessary for accessing the locked surface are its base address and the surface pitch. The base address is returned in a structure member of type LPVOID, and the surface pitch in a structure member of type LONG. Applications that plan to dereference the surface pointer typically cast it into one that matches the surface's color format. For example, a program that has set a 24-bit true-color mode is likely to access surface memory in byte-size units. In this case the pointer can be cast into a variable of type LPBYTE. On the other hand, an application executing in a 16-bit real-color mode typecasts the pointer into a LPWORD type, and one that has set a 32-bit true-color mode may typecast into an LPLONG data type.

The following code fragment shows the use of the Lock() function in a routine that fills a 50-by-50 pixel box in a 24-bit true-color video mode. The box is arbitrarily located at screen row number 80 and pixel column number 300. The pixels are filled with the red attribute by setting each third surface byte to 0xff and the other three color bytes to 0x0.

```
LONG     localPitch;    // Local variable for surface pitch
LPBYTE   localStart;    // and for buffer start
LPBYTE   lastRow;       // Storage for row start
.
.
.
// Attempt to lock the surface for direct access
if (lpDDSPrimary->Lock(NULL, &ddsd, DDLOCK_WAIT, NULL)\
  != DD_OK)
   DDInitFailed(hWnd, hRet, "Lock failed");
// Store video system data
   vidPitch = ddsd.lPitch;        // Pitch
   vidStart = ddsd.lpSurface;     // Buffer address

// ASSERT:
// Surface is locked. Global video data is as follows:
// vidPitch holds surface pitch
// vidStart holds video buffer start
// Copy to local variables typecasting void pointer
   localPitch = vidPitch;
   localStart = (LPBYTE) vidStart;

// Index to row 80
   localStart = localStart + (80 * localPitch);
// Move right 300 pixels
   localStart += (400 * 3);
```

```
// Display 50 rows, 50 times
   for(int i = 0; i < 50; i++){
      lastRow = localStart;        // Save start of row
   for(int j = 0; j < 50; j++) {
     *localStart = 0x0;            // blue attribute
     localStart++;
     *localStart = 0x0;            // green attribute
     localStart++;
     *localStart = 0xff;           // red attribute
     localStart++;
   }
 localStart = lastRow + localPitch;
}

lpDDSPrimary->Unlock(NULL);
```

28.2 In-Line Assembly Language

The maximum advantages of direct access to video memory are realized when the code is highly optimized, and the most dependable way to produce highly optimized code is by programming in 80x86 assembly language. Although an entire DirectDraw application can be coded in assembly language, this approach usually entails more difficulties and complications than can be justified by the relatively few advantages. On the other hand, most C and C++ compilers provide in-line assemblers that allow embedding assembly language code in a C or C++ program. The result is an easy-to-produce multilanguage program with the advantages of both environments. It is also possible to use an assembler program, such as MASM, to produce stand-alone assembly language modules that can be incorporated with the application at link time. Later in this chapter we discuss the development of assembly language modules that can be used by a program developed in Visual C++.

A benefit of in-line assembly is that the low-level code can reference, by name, any C++ language variable or function that is in scope. This makes it easy to access program data and processing routines. In-line assembly also avoids many of the complication and portability problems usually associated with parameter passing conventions in multi-language programming. The resulting development environment has all the advantages of high-level programming, as well as the power and flexibility of low-level code. Visual C++ and Borland C Builder support in-line assembly.

On the other hand, in-line code cannot use assembler directives to define data and perform other initialization and synchronization functions. Another advantage of stand-alone assembler modules is that the code can be easily integrated into libraries and DLL files that can be ported to other applications.

DirectDraw made assembly language coding in Windows applications an attractive option. Direct access to video memory, made possible by DirectX, opens the possibility of using assembly language to maximize performance and control. The result is a DOS-like development environment. However, in conventional GDI programming there is little justification for using low-level code.

28.2.1 The _asm Keyword

The _asm keyword is used in Visual C++ to produce assembly language instructions, one at a time, or in blocks. When the compiler encounters the _asm symbol it invokes the in-line assembler. The asembler, in turn, generates the necessary opcodes and inserts them into the object file. In this process the development system limits its action to that of an assembler program; no modification of the coding takes place and no interpretation or optimization effort is made. Thus, the programmer is certain that the resulting code is identical to the source. The fact that no separate assembly or linking is necessary considerably simplifies the development process.

Although the _asm keyword can precede a single instruction, it is more common to use it to generate a block of several assembly language lines. Braces are used to delimit the source block, as in the following example:

```
_asm
{
  ; Assembly language code follows:
    PUSH    EBX           ; EBX to stack
    MOV     EAX,vidPitch ; vidPitch is a C variable
    MOV     EBX,80        ; Constant to register
    MUL     EBX           ; Integer multiply
    POP     EBX           ; Restore EBX
}
```

The second instruction of the preceding code fragment loads a variable defined in C++ code into a machine register. Accessing high-level language variables is one of the most convenient features of in-line assembly. Assembly language code can also store results in high-level variables.

28.2.2 Coding Restrictions

There are a few rules and conventions that in-line assembly language code must follow. Perhaps the most important one is to preserve the registers used by C++. A possible source of problems is when the C++ program is compiled with the _fastcall switch or the /Gr compiler option. In these cases, arguments to functions are passed in the ECX and EDX machine registers; therefore, they must be preserved by the assembly language program section. The easiest way to avoid this concern is to make sure that programs that use in-line assembly are not compiled with either of these options. In Visual C++ the compiler options can be examined by selecting the Settings command in the Project Menu and then clicking on the C/C++ tab. The Project Options window in this dialog box shows the compiler switches and options that are active. Make sure that you inspect the settings for both the Release and the Debug options, as shown in the Settings For: scroll box.

Programs that do not use the _fastcall switch or the /Gr compiler options can assume that the four general purpose registers need not be preserved. Consequently, EAX, EBX, ECX, and EDX are free and available to the assembly language code. In regards to the pointer registers and the direction flag the Microsoft documentation is inconsistent. Some versions of the Visual C++ Programmers Guide state that ESI and EDI must be preserved, while other versions state the contrary. Regarding the direction flag, the original Microsoft C++ compilers required that the flag be cleared before returning, while the most recent manuals say that the state of the direction flag must be preserved. In view of these discrepancies, and in fear of future variations, the safest approach is to use the general purpose registers freely (EAX, EBX, ECX, and EDX) but to preserve all other machine registers and the direction flag. This means that on entry to the routine the in-line assembly code must push on the stack the registers that must be preserved, as well as the flags, and restore them before exiting the _asm block. This is the approach that we use in the book's sample code. The processing in a routine that uses the ESI and EDI registers can be as follows:

```
_asm
{
    PUSH     ESI          ; Save context
    PUSH     EDI
    PUSHF
    ; Processing operations go here
    ; .
    ; .
    ; .
    ; Exit code
    POPF             ; Restore context
    POP      EDI
    POP      ESI
}
```

28.2.3 Assembly Language Functions

Often the low-level processing routines can be conveniently located in functions that can be called by the C++ code. When the assembly code is not created by means of the on-line feature of the compiler, that is, when it is written for a separate assembler, then the assembly language routine and the C++ must interface following the calling conventions adopted by the compiler. The usual procedure is that C++ places the parameters in the stack at the time the call is made, and the assembly language routine removes them from the stack making sure that the stack integrity is preserved. In this case the assembly and the C++ code usually reside in separate files which are referenced at link time.

For applications that use in-line assembly, the inter-language protocol is considerably simplified by creating a C++ function shell to hold the assembly code. In this case the use of the stack for parameter passing becomes almost unnecessary, since the assembly code can reference the C++ variables directly. One possible drawback is that the in-line assembler does not allow the use of the data definition directives DB, DW, DD, DQ, and DT or the DUP operator. Therefore, the data elements used by the assembly language code must be defined as C++ variables. The following example is an assembly language routine to add three integers and return the sum to the calling code. The processing is contained in a C++ function shell, as follows:,

```
int SumOf3(
    int x,
    int y,
    int z)
{
   int total;        // C++ variable to hold sum
```

```
    _asm
    {
    MOV    EAX,x        ; move first parameter to accumulator
    ADD    EAX,y        ; add second parameter
    ADD    EAX,z        ; and third parameter
    MOV    total,EAX    ; store sum
    }
   return total;
}
The calling code could be as follows:
   int     aSum;            // local variable for sum
   . . .
   aSum = SumOf3(10, 20, 30);
```

This example shows that the assembly language code can access the parameters passed to the C++ function, as well as store data in a local variable that is also accessible to C++. This easy mechanism for sharing data is one of the major advantages of in-line assembly.

28.3 Multi-Language Programming

Instead of using in-line assembly language, it is possible to develop entire assembly language modules which can be accessed at link-time or at runtime. The topic of developing libraries and DLLs is outside the scope of this book. However, this section is about developing independent modules using MASM, which can later be incorporated into a C++ program. The development tools considered here are Microsoft's MASM and Visual C++ version 6.0. The resulting program is developed partly in assembly language and partly in C++. Therefore it is a case of multi-language programming.

28.3.1 Stand-Alone Assembler Modules

You have seen that one of the major limitations of in-line assembly is that it does not permit the use of assembler directives. One of the consequences of this limitation is that the programmer must declare all data as C++ variables and access this data from the in-line assembler code. Another limitation is that it is not possible to create assembly language procedures with in-line code. Therefore the assembly language routines must be defined as C or C++ functions. In many cases it is possible to work around these limitations, but not always. For example, the current version of Visual C++ does not support10-byte floating-point variables in ANSI-IEEE extended format. Applications that manipulate floating point data using the math unit of the Pentium can often benefit from the extended format. Among other reasons, for defining constants and storing temporary results. Other uses of stand-alone assembly is in creating libraries and DLLs that can be accessed by any application.

C++/Assembler Interface Functions

The transition between C++ and assembly language code is made easier by creating interface routines that receive the C++ call, format the data for the low-level code, call the low-level procedure, and re-format the results before returning to the caller. If properly designed, the interface function makes the multi-language environment transparent to the programmer. For example, imagine a low-level routine, coded in assembly language, that performs matrix multiplication. The header of the low-level procedure is as follows:

```
_MUL_MATRICES    PROC    USES esi edi ebx ebp
; Procedure to multiply two matrices (A and B) for which a matrix
; product (A * B) is defined. Matrix multiplication requires that
; the number of columns in matrix A be equal to the number of
; rows in matrix B, as follows:
;             A                   B
;           R   C               r   cr
;               |______ = ______|
;
; Example:
;
;    A=(2 by 3)          B=(3 by 4)
;    A11  A12  A13       B11  B12  B13  B14
;    A21  A22  A23       B21  B22  B23  B24
;                        B31  B32  B33  B34
;
; The product matrix (C) will have 2 rows and 4 columns
;   C=(2 by 4)
;   C11   C12   C13  C14
;   C21   C22   C23  C24
;
; In this case the product matrix is obtained as follows:
;   C11 = (A11*B11)+(A12*B21)+(A13*B31)
;   C12 = (A11*B12)+(A12*B22)+(A13*B32)
;   C13 = (A11*B13)+(A12*B23)+(A13*B33)
;   C14 = (A11*B14)+(A12*B24)+(A13*B34)
;
;   C21 = (A21*B11)+(A22*B21)+(A23*B31)
;   C22 = (A21*B12)+(A22*B22)+(A23*B32)
;   C23 = (A21*B13)+(A22*B23)+(A23*B33)
;   C24 = (A21*B14)+(A22*B24)+(A23*B34)
;
; On entry:
;      ESI --> first matrix (A)
;      EDI --> second matrix (B)
;      EBX --> storage area for products matrix (C)
;      AH = rows in matrix A
;      AL = columns in matrix A
;      CH = rows in matrix B
;      CL = columns in matrix B
;      EDX = number of bytes per entry
; Assumes:
;      Matrix C is dimensioned as follows:
;      Columns of C = columns of B
;      Rows of C = rows of A
; On exit:
;      Matrix C is the products matrix

; Note: the entries of matrices A, B, and C must be of type float
;       and of the same data format
;
.
.
.
_MUL_MATRICES    ENDP
```

The _MUL_MATRICES procedure requires that ESI and EDI point to the two multiplicand matrices stored in a valid ANSI/IEEE floating-point format. EBX points to a storage area to hold the products matrix. AH and AL holds the row/column dimensions of one source matrix, and CH and CL the dimensions of the other source matrix. EDX holds the number of bytes in each matrix entry. This is usually called the skip factor.

The interface routine uses in-line assembly and C++ code to format the data received from the caller. Code can be as follows:

```
template <class A>
bool MulMatrices(A *matA, A *matB, A *matC,
                    int rowsA, int colsA,
                    int rowsB, int colsB)
{
// Perform matrix multiplication: C = A * B using low-level code
// defined in an assembly language module
// On entry:
//       *matA and *matB are matrices to be multiplied
//       *matC is matrix for products
//       rowsA is number of the rows in matrix A
//       colsA is number of columns in the matrix A
//       rowsB is number of the rows in matrix B
//       colsB is number of columns in the matrix B
// Requires:
//       Matrices must be multiplication compatible
//       All three matrices must be of the same float
//       data type
// Asumes:
//       Matrix C dimensions are the product of the
//       columns of matrix B times the rows or matrix A
// Routine expects:
//      ESI --> first matrix (A)
//      EDI --> second matrix (B)
//      EBX --> storage area for products matrix (C)
//      AH = number of rows in matrix A
//      AL = number of columns in matrix A
//      CH = number of rows in matrix B
//      CL = number of columns in matrix B
//      EDX = horizontal skil factor
// On exit:
//      returns true if matC[] = matA[] * matB[]
//      returns false if columns of matA[] not = rows
//      of matB[]. If so, matC[] is undefined

int eSize = sizeof(matA[0]);

// Test for valid matrix sizes:
//     columns of matA[] = rows of matB[]
if(colsA != rowsB)
   return false;

_asm
{
         MOV     AH,BYTE PTR rowsA
         MOV     AL,BYTE PTR colsA
          MOV     CH,BYTE PTR rowsB
          MOV     CL,BYTE PTR colsB
          MOV     ESI,matA           // Address to ESI
          MOV     EDI,matB
          MOV     EBX,matC
          MOV     EDX,eSize          // Horizontal skip
         CALL    MUL_MATRICES
}
return true;
}
```

Notice that the interface function, called MulMatrices(), calls the low-level procedure _MUL_MATRICES. The C++ interface function uses C++ code to test that the two matrices are multiplicationcompatible, in other words, that the number of columns of the first matrix is equal to the number of rows of the second one. Code also uses the sizeof operator on one of the matrix entries to determine the skip factor. The data received as parameters by the interface function is moved to processor registers. Notice that the skip factor, now in the C++ variable eSize, is moved into the EDX register. The call to the low-level procedure is made in the statement:

```
CALL MUL_MATRICES
```

In this call the leading underscore is appended automatically by the C++ compiler. This makes it necessary to use the underscore symbol as the first character in the name of an assembly language procedures that is to be called from C++.

Once the interface function has been defined, client code can ignore all the data formatting details of the interface. The following code fragment shows the C++ data definition and calling of the MulMatrices() interface function.

```
float matA[] = { 1.0, 2.0, 4.0,
                 2.0, 6.0, 0.0};

float matB[] = { 4.0, 1.0, 4.0, 3.0,
                 0.0,-1.0, 3.0, 1.0,
                 2.0, 7.0, 5.0, 2.0};

// Product matrix has rows of matA[] * columns of matB[]
// 2 * 4 = 8
float matAB[8];
bool result;
.
.
.
result = MulMatrices(matA, matB, matAB, 2, 3, 3, 4);
if(result)
   // Success!
else
   cout << "Invalid matrix size";
```

MASM Module Format

The assembly language modules compatible with Visual C++ and Win32 must follow a specific protocol. For example, the module containing the _MUL_MATRICES procedure, mentioned previously, is defined as follows:

```
PUBLIC _MUL_MATRICES
;
        .486
        .MODEL flat
        .DATA
// Module data elements defined here
MAT_A_ROWS     DB   0    ; Rows in matrix A
```

```
.
.
.
          .CODE
// Low-level procedures coded here
_MUL_MATRICES    PROC
.
.
.
_MUL_MATRICES    ENDP
end
```

The first element in the module is the PUBLIC declaration of the procedures that are to be accessed from outside the module. The .486 directive and the .MODEL flat directive ensure that the code in Win32 compatible. In the Intel flat memory model the segment registers are unnecessary since 32-bit registers can access the entire code memory space.

The module is assembled normally. To facilitated debugging it is a good idea to use the /Zi switch. For example, a module named test_1.asm is assembled with the command:

```
masm test_1 /Zi;
```

The resulting object file must be added to the Visual C++ project in which it is to be used. This is done with Developer Studio Project/Add To Project/Files... command. At this point you may have to select the Files of type: All Files(*.*) option to make the object file visible in the dialog box. The source file can also be added to the project if it needs to be edited during development. Once added, the object and assembler source files will be listed as Resource Files in Developer Studio File View window.

C++ Module Format

The Visual C++ source files for multi-language programs must contain an external declaration of the low-level functions that are to be accessed by code. For example, if the C++ code is to call the procedure named _MUL_MATRICES, located in a separate object file, the C++ source must contain the following statement:

```
extern "C" void MUL_MATRICES();
```

The remainder of the code is conventional C++.

If you use C++ interface functions, as suggested earlier in this chapter, it may be a good idea to place the interface functions in a header file. The header file can be incorporated into the main C++ source with an #include statement.

28.3.2 Matrix Ops Project

The Matrix Ops project, located in the Chapter 13 directory in the book's software package, is a demonstration of the multi-language programming techniques discussed in this section. The project is developed as a Win32 Console Application, but the same method can be followed to create a Win32 Application. The project contains the following source files:

1. Matrix Ops.cpp is the main C++ source file.
2. Matrix Ops.h is a header file with the C++ interface functions.
3. Mat_math.asm is the assembler source.

The low-level module mat_math.asm contains several procedures to perform matrix mathematics. These include scalar addition, subtraction, multiplication, and division of rows and matrices, as well as matrix addition and multiplication. The C++ header file named Matrix Ops.h contains the interface functions for the low-level code, as well as an auxiliary function to display a matrix. All of the functions in Matrix Ops.h are defined as template functions. This is in order to facilitate its use with matrices of different numeric types. The driver source is the file named Matrix Ops.cpp. This module declares data for several test matrices and exercises all the functions in the interface module.

28.4 Direct Access Primitives

An application that uses direct access to video memory can usually benefit from a few primitive functions that perform the core processing operations. These primitives can be coded in C++ or using in-line assembly. Most primitive routines are mode specific. They assume that a particular display mode is available and has been selected. It is impossible to predict the specific functions and the number of primitives that are necessary for a particular graphics program. This depends, among other factors, on what portion of the processing is performed using direct access to video memory and on the size and scope of the application. The resulting complexity can range from a few simple routines to a full-size, stand-along graphics package. In this section we consider several direct access primitives that can be generally useful; they are:

- Lock a DirectDraw primary surface and store the video buffer's base address and pitch.
- Release a DirectDraw primary surface.
- Set an individual screen pixel at given coordinates and color attributes and to read the attributes of a screen pixel located at given coordinates.
- Lock a DirectDraw primary surface, fill a pixel rectangle, at given coordinates, dimensions, and color attributes, then release the surface.
- Lock a DirectDraw primary surface, draw a single-line box, at given coordinates, dimensions, and color attributes, then release the surface.

Many of the routines that access video memory directly must perform calculations to determine the offset of a particular pixel in the display surface. For example, a call to fill a screen rectangle passes the address of its top-left corner as parameters. The processing must convert this address, usually in column/row format, into a video memory offset. It is possible to develop a primitive function that calculates this pixel offset, but this approach introduces a call-return overhead that adversely affects performance. More often the address calculations are part of the processing routine. Therefore, before attempting to develop the direct access primitives, we take a closer look at the low-level operations necessary for calculating a pixel addresses.

28.4.1 Pixel Address Calculations

A display mode's resolution, color depth, and pitch determine the location of each pixel on the surface. For this reason, pixel address calculations are specific to a display mode. In the case of the hi-color modes, the variables that enter into the calculation of a pixel offset are the number of bytes per pixel and the surface pitch. In addition, the horizontal and vertical resolution of the display mode can be used to check for invalid input values, since it is the responsibility of direct access routines not to read or write outside of the locked surface area. Figure 28-3 shows the parameters that define the location of a screen pixel and the formula used for calculating its offset.

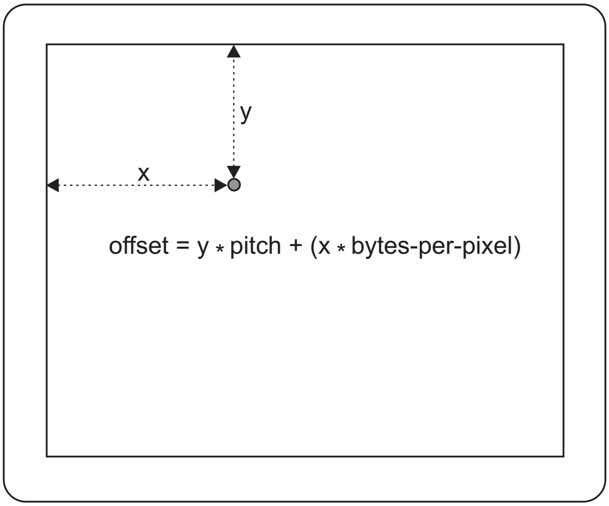


Figure 28-3 *Pixel Offset Calculation*

28.4.2 Defining the Primary Surface

You have seen that the DirectDraw Lock() function is used to lock the surface so that it can be accessed directly. The call also returns a pointer to the top-left corner of the rectangle that defines the surface, as well as its pitch. When Lock() references to the entire primary surface, the call returns the base address of the video buffer. The pitch, in this case, is the number of bytes in each screen buffer row.

In addition, the Lock() call forces Windows to relax its normal protection over video display memory. Normally, any instruction that attempts to access video memory immediately generates a protection violation exception and the application is terminated. This is important to keep in mind while designing direct access functions, since it is this feature that makes the Lock() call necessary if a previous Lock() has been released. This is true even when the video buffer address and pitch have been previously obtained and stored. On the other hand, the surface lock can be retained during more than one access to video memory. Therefore, a routine that sequentially sets several screen pixels need only call the Lock() function once. Once the pixel sequence is set, then the lock can be released. Also recall that the Lock()

call requires a pointer to an IDirectDrawSurface object, which is usually obtained by means of the CreateSurface() function. The following is a simple locking function for the entire primary surface:

```
// Global variables for surface pitch and base address
LONG        vidPitch;
LPVOID      vidStart;
.
.
.
//****************************************************************
// Name: LockSurface
// Desc: Function to lock the entire DirectDraw primary surface
//       and store the direct access parameters
//
// PRE:
// 1. First parameter is a pointer to DirectDraw surface
// 2. Video display globals have been declared as follows:
//    LONG    vidPitch; // Pitch
//    LPVOID  vidStart; // Buffer address
//
// POST:
// Returns 1 if call succeeds and 0 if it fails
//****************************************************************
int LockSurface(
    LPDIRECTDRAWSURFACE4  lpSurface)
{
    // Attempt to lock the surface for direct access
       if (lpSurface->Lock(NULL, &ddsd, DDLOCK_WAIT, NULL)\
       != DD_OK)
       return 0;                      // Lock failed
    // Store video system data
       vidPitch = ddsd.lPitch;        // Pitch
       vidStart = ddsd.lpSurface;     // Buffer address
       return 1;                      // Surface locked
}
```

28.4.3 Releasing the Surface

Developing a function to release the locked surface is also convenient. In this case the processing is based on the DirectDraw Unlock() functions. It may be a good idea to have the routine that calls the Unlock() function also reset the access variables to zero. This makes it easier to determine if a lock is being held, since a zero value is invalid for either variable. The routine itself tests one of these variables before attempting to release the lock.

```
//****************************************************************
// Name: ReleaseSurface
// Desc: Function to release locked surface
// PRE:
// 1. Parameter is pointer to locked DirectDraw surface
// 2. Video display globals as follows:
//    LONG   vidPitch; // Pitch
//    LPVOID  vidStart; // Buffer address
//****************************************************************
void ReleaseSurface(
     LPDIRECTDRAWSURFACE4  lpSurface)
{
   if(vidStart != 0) {
       lpSurface->Unlock(NULL);
       // Clear global variables
       vidPitch = 0;
       vidStart = 0;
   }
   return;
}
```

This version of the ReleaseSurface() function assumes that the object of the lock was the entire surface.

28.4.4 Pixel-Level Primitives

Pixel-level operations are the lowest-level graphics routines available, which explains why they are often considered device driver components, rather than primitives. In theory, it is possible to perform any graphics operations by using a pixel read and a pixel write routine.

The Windows GDI provides functions to set and read a single pixel. However, the GDI functions are extremely slow. Direct access, pixel-level routines execute several hundred times faster than the GDI counterparts. The pixel-level read and write primitives could be coded as follows:

```
// Global variables for surface pitch and base address
LONG        vidPitch;
LPVOID    vidStart;
.
.
.
//*****************************************************************
// Name: DASetPixel
// Desc: Assembly language code to set a single screen pixel
//       using direct access to the video buffer
//
// PRE:
// 1. Successful Lock() of surface
//    Video display globals are stored as follows:
//    LONG    vidPitch;    // Pitch
//    LPVOID    vidStart;    // Buffer address
// 2. First and second parameters are the pixel coordinates
// 3. Last three parameters are pixel RGB attributes
// 4. Assumes true color mode 640 by 480 by 28
//
// POST:
// None
//*****************************************************************
void DASetPixel(
     int xCoord,
     int yCoord,
     BYTE redAtt,
     BYTE greenAtt,
     BYTE blueAtt)
{
    _asm
     {
    PUSH   ESI               ; Save context
    PUSHF
    MOV    EAX,yCoord       ; Row number to EAX
    MUL    vidPitch;
    MOV    EBX,EAX          ; Store in EBX
    MOV    EAX,xCoord       ; x coordinate
    MOV    CX,3
    MUL    CX               ; 3 bytes per pixel
    ADD    EAX,EBX          ; move right to x coordinate
    MOV    ESI,vidStart
    ADD    ESI,EAX
  ; Load color attributes into registers
    MOV    AL,blueAtt
    MOV    DH,greenAtt
    MOV    DL,redAtt
  ; Set the pixel
    MOV    [ESI],AL       ; Set blue attribute
    INC    ESI
    MOV    [ESI],DH       ; Set green
    INC    ESI
    MOV    [ESI],DL       ; Set red
    POPF                  ; Restore context
    POP    ESI
    }
  return;
}

//***************************************************************
// Name: DAReadPixel
// Desc: Assembly language code to read a single screen pixel
//       using direct access to the video buffer
//
// PRE:
// 1. Successful Lock() of surface
//    Video display globals are stored as follows:
//    LONG    vidPitch;    // Pitch
//    LPVOID    vidStart;    // Buffer address
// 2. First and second parameters are the pixel coordinates
//    values are returned in public variables named
//    pixelRed, pixelGreen, and pixelBlue
// 3. Assumes true color mode 640 by 480 by 28
//
// POST:
// None
//***************************************************************
void DAReadPixel(
    int xCoord,
    int yCoord)
{
    _asm
    {
    PUSH    ESI          ; Save context
    PUSHF
    MOV    EAX,yCoord   ; Row number to EAX
    MUL    vidPitch
    MOV    EBX,EAX      ; Store in EBX
    MOV    EAX,xCoord   ; x coordinate
    MOV    CX,3
    MUL    CX           ; 3 bytes per pixel
    ADD    EAX,EBX      ; move right to x coordinate
    MOV    ESI,vidStart
    ADD    ESI,EAX
```

```
   ; Read and store pixel attributes
    MOV    AL,[ESI]      ; Get blue attribute
    INC    ESI
    MOV    DH,[ESI]      ; green
    INC    ESI
    MOV    DL,[ESI]      ; and red
    MOV    pixelBlue,AL     ; Store blue
    MOV    pixelGreen,DH    ; green
    MOV    pixelRed,DL      ; and red
    POPF                 ; Restore context
    POP    ESI
    }
   return;
}
```

Filling a Rectangular Area

Filling a rectangular area with a particular color attribute is such a useful manipulation that most applications that access memory directly can profit from such a primitive. To define a screen rectangle you can use the coordinates of its diagonally opposite corners, or the coordinates of one corner and the rectangle's dimensions. The following listed function adopts the second approach. In addition, the routine needs to know the values for the RGB color attributes to use in the fill. The code is as follows:

```
//***************************************************************
// Name: DARectangle
// Desc: Assembly language code to draw a rectangle on the screen
//       using direct access to the video buffer
//
// PRE:
// 1. First parameter is pointer to surface
// 2. Second and third parameters are rectangle's x and y
//    coordinates
// 3. Fourth parameter is rectangle width, in pixels
// 4. Fifth parameter is rectangle height, in pixels
// 5. Last three parameters are RGB attributes
// 6. Assumes true color mode is 640 by 480 by 24

// POST:
// Returns 1 if lock succeeded and 0 if it failed
//***************************************************************
int DARectangle(
    LPDIRECTDRAWSURFACE4 lpPrimary,
    int yCoord,
    int xCoord,
    int width,
    int height,
    BYTE redAtt,
    BYTE greenAtt,
    BYTE blueAtt)
{
 // Attempt to lock the surface for direct access
 if (!LockSurface(lpPrimary))
        return 0;                   // Lock failed

 _asm
 {
       PUSH     ESI            ; Save context
       PUSHF
       MOV      EAX,yCoord     ; Row number to EAX
       MUL      vidPitch;
       MOV      EBX,EAX        ; Store in EBX
       MOV      EAX,xCoord     ; x coordinate
       MOV      CX,3
       MUL      CX             ; 3 bytes per pixel
       ADD      EAX,EBX        ; move right to x coordinate
       MOV      ESI,vidStart
       ADD      ESI,EAX
       ; Load color attributes into registers
       MOV      AL,blueAtt
       MOV      DH,greenAtt
       MOV      DL,redAtt
       MOV      EBX,height    ; number of lines in rectangle
 NEXT_LINE:
       PUSH     ESI           ; Save start of line
       MOV      ECX,width     ; x dimension of rectangle
 SET_PIX:
       MOV      [ESI],AL      ; Set blue attribute
       INC      ESI           ; Next pixel
       MOV      [ESI],DH      ; Set green
       INC      ESI           ; Next pixel
       MOV      [ESI],DL      ; Set red
       INC      ESI           ; Next pixel
       LOOP     SET_PIX
     ; Pixel line is set
       POP      ESI
       ADD      ESI,vidPitch
       DEC      EBX
       JNZ      NEXT_LINE
       POPF                  ; Restore context
       POP      ESI
   }
 ReleaseSurface(lpPrimary);
 return 1;      // Exit
}
```

Observe that the DARectangle() calls LockSurface() and ReleaseSurface() functions previously developed. To improve performance, the function can be easily modified to call Lock() and Unlock() directly.

Box-Drawing

Drawing a box is a little more complicated than filling a rectangle. The actual processing can be based on two simple routines: one to draw a horizontal line and another one to draw a vertical line. The core routine sets up the machine registers with the necessary data and then calls the horizontal and vertical line routines to do the actual drawing. Since there are other possible uses for the vertical and horizontal line drawing operations they are coded as separate functions. The parameters to the box drawing routine are the same as those for the rectangle fill. They include the pointer to the surface, the box coordinates, its dimensions, and the color attributes. The code is as follows:

```
//***************************************************************
// Name: DABox
// Desc: Assembly language code to draw a screen box with
//       single-pixel wide lines, using direct access to the
//       video buffer
//
// PRE:
// 1. First parameter is pointer to surface
// 2. Second and third parameters are the coordinates of the
//    top-left corner of the box
// 3. Fourth parameter is box width, in pixels
// 4. Fifth parameter is box height, in pixels
// 5. Last three parameters are RGB attributes
// 6. True color mode is 640 by 480 by 24
//
// POST:
// Returns 1 if lock succeeds and 0 if it fails
//****************************************************************
int DABox(
    LPDIRECTDRAWSURFACE4 lpPrimary,
    int xCoord,
    int yCoord,
    int width,
    int height,
    BYTE redAtt,
    BYTE greenAtt,
    BYTE blueAtt)
{
  // Attempt to lock the surface for direct access
  if (!LockSurface(lpPrimary))
       return 0;                   // Lock failed

  _asm
  {
    PUSH    ESI           ; Save context
    PUSHF
    MOV     EAX,yCoord  ; Row number to EAX
    MUL     vidPitch;

    MOV     EBX,EAX       ; Store in EBX
    MOV     EAX,xCoord    ; x coordinate
    MOV     CX,3
    MUL     CX            ; 3 bytes per pixel
    ADD     EAX,EBX       ; move right to x coordinate
    MOV     ESI,vidStart
    ADD     ESI,EAX
    ; Load color attributes into registers
    MOV     AL,blueAtt
    MOV     DH,greenAtt
    MOV     DL,redAtt
    ; Draw top horizontal line
    MOV     ECX,width      ; x dimension of rectangle
    CALL    DAHorLine
    ; Draw bottom horizontal line
    PUSH    ESI            ; Save top left corner address
    PUSH    EAX            ; Save color
    PUSH    EDX
    MOV     EAX,height     ; Number of lines to EAX
    MUL     vidPitch;      ; Times the length of each line
    ADD     ESI,EAX        ; Add to start
    MOV     ECX,width      ; x dimension of rectangle
    POP     EDX            ; Restore color
    POP     EAX
    CALL    DAHorLine      ; Draw line
    POP     ESI             ; Restore start of rectangle
    ; Draw left vertical line
    MOV     EBX,vidPitch    ; Pitch to EBX
    MOV     ECX,height      ; Pixel height of vertical line
    CALL    DAVerLine
    ; Draw right vertical line
    ; ESI holds address of top-left corner
    PUSH    EAX             ; Save color
    PUSH    EDX
    MOV     EAX,width       ; Number of lines to EAX
    MOV     CX,3            ; Pixels per line
    MUL     CX
    ADD     ESI,EAX         ; Add to start
    MOV     ECX,height      ; Line y dimensions
    INC     ECX             ; One more pixel
    POP     EDX             ; Restore color
    POP     EAX
    CALL    DAVerLine       ; Draw line
    POPF                    ; Restore context
    POP     ESI
    }

    ReleaseSurface(lpPrimary);
    return 1;                // Exit
}

//****************************************************************
// Name: DAHorLine
// Desc: Assembly language support function for DABox()
//             draws a horizontal pixel line
// PRE:
//    ESI holds buffer address of start of line
//    ECX hold pixel length of line
//    AL = blue attribute
//    DH = green attribute
//    DL = red attribute
// POST:
//    ECX is destroyed
//    All others are preserved
//****************************************************************

void DAHorLine()
{
    _asm
    {
    PUSH     ESI             ; Save start of line
DRAW_HLINE:
    MOV     [ESI],AL         ; Set blue attribute
    INC     ESI
    MOV     [ESI],DH         ; Set green
    INC     ESI
    MOV     [ESI],DL         ; Set red
    INC     ESI
    LOOP    DRAW_HLINE
    POP     ESI
    }
    return;
}

//****************************************************************
```

```
// Name: DAVerLine
// Desc: Assembly language support function for DABox()
//      draws a vertical pixel line
// PRE:
//    ESI holds buffer address of start of line
//    ECX hold pixel height of line
//    EBX holds surface pitch
//    AL = blue attribute
//    DH = green attribute
//    DL = red attribute
// POST:
//    ECX is destroyed
//    All others are preserved
//****************************************************************

void DAVerLine()
{
    _asm
    {
    PUSH    ESI             ; Save start of line
DRAW_VLINE:
    PUSH    ESI             ; Save start address
    MOV     [ESI],AL        ; Set blue attribute
    INC     ESI
    MOV     [ESI],DH        ; Set green
    INC     ESI
    MOV     [ESI],DL        ; Set red
    POP     ESI             ; Restore start
    ADD     ESI,EBX         ; Index to next line
    LOOP    DRAW_VLINE
    POP     ESI
    }
  return;
}
```

28.5 Raster Operations

Direct access to video memory, combined with low-level coding, provides the programmer with all the necessary elements to develop a powerful, DOS-like, graphics toolkit. One of the many possibilities consists of using logical operations to combine object and screen data. These are sometimes called raster operations, raster ops, or mixes. A raster operation determines how two or more source images are combined to produce a destination image. Arithmetic and logical operators are used to produce the desired effect. The simplest one is to replace the destination with the source. This is what takes place when you directly write a pixel value to the video screen. When the MOV instruction writes a color value to the screen you are actually replacing the destination with the source, as follows:

```
    MOV    [ESI],AL
```

In many cases a raster operation requires a read-modify-write sequence. For example, you could increase the brightness of a specific pixel by adding a constant to its value, as follows:

```
    MOV    AL,[ESI]       ; Read pixel
    ADD    AL,20          ; Modify
    MOV    [ESI],AL       ; Write
```

The problem with this type of processing is that the read-modify-write cycle takes considerable processing time. For this reason some graphics processors perform raster operations in hardware.

The Pentium CPU has several logical operators that allow combining foreground and background data by means of a single instruction. For example, a logical AND operation can be used to combine foreground bits (object data) and background data. The result is that the background bits are preserved whenever the foreground bit is zero, and vice versa. The object data is sometimes referred to as a mask. Thus, code can overlay a white grid over an existing image by ANDing a mask consisting of 1-bits in the solid portion of the grid and 0-bits in the transparent portion. The Pentium logical opcodes are AND, OR, XOR, and NOT. For example, the following operation ANDs the value in the AL register with the screen data contained in the address pointed at by ESI:

```
AND     [ESI],AL
```

In C++ programming the bitwise operators perform a similar action, at a much greater processing cost. In the following sections we examine the XOR mix, which is one of the most useful raster graphics operations.

28.5.1 XOR Animation

Animating a screen object usually requires erasing an image from its current screen position and then redrawing it at a new location. Graphics programmers sometimes call this sequence the "save-draw-redraw" cycle. The "save" element in this sequence is determined by the fact that the original screen image must be preserved so that it can later be restored to its original form. At the same time, if the object is not erased before it is redrawn, its apparent movement leaves an undesirable image track on the display surface. You can make an object appear to move laterally, left to right, by progressively redrawing and erasing its screen image at consecutively larger x coordinates. To do this in a conventional manner we have to perform a rather complex sequence of operations:

1. Save phase: preserve the screen image data in the area where the object is to be displayed.
2. Draw phase: draw the object.
3. Redraw phase: erase the object by restoring the original screen image.

Step 1 requires reading all data in the screen area that is to be occupied by the animated object, while step 3 requires redisplaying the saved image. Both operations are time-consuming, and in computer animation, time is always in short supply.

Several hardware and software techniques have been devised for performing the save-draw-redraw cycle. In later chapters we explore DirectDraw animation techniques that are powerful and versatile. These higher-level methods are based on flipping surfaces containing images and on taking advantage of the hardware blitters that are available in most video cards. Here we are concerned with the simplest possible approach to figure animation. This technique, which is made feasible by the high performance obtained with direct access to video memory, is based on the properties of the logical exclusive or XOR operation. Although it is theoretically possible to perform XOR animation using high-level code, the most efficient and powerful technique requires assembly language.

The action of the logical XOR can be described by saying that a bit in the result is set if both operands contain opposite values. It follows that XORing the same value twice restores the original contents, as in the following case:

```
             10000001 <= original value  ----|
XOR value => 10110011                        |
             ---------                       |
             00110010  <= first result       |
XOR value => 10110011                        |
             ---------                       |
             10000001  <= final result ------|
```

XOR, like all bitwise operations, takes place on a bit-by-bit basis. In this example the final result (10000001) is the same as the original value.

Animation techniques can be based on this property of the bitwise XOR since it provides a convenient and fast way for consecutively drawing and erasing a screen object. The object is drawn on the screen by XORing it with the background data. XORing a second time erases the object and restores the original background. Therefore, the save-draw-redraw cycle now becomes an XOR-XOR cycle, which is considerably faster and simpler to implement. The XOR method is particularly useful when more than one animated object can coincide on the same screen position, since it ensures that the original screen image is automatically preserved.

There are also disadvantages to using XOR in computer animation. The most important one is that the image itself is dependant upon the background attributes. This is due to the fact that each individual pixel in the object is determined both by the XORed value and by the destination pixel. The following XOR operation produces a red object (in RGB format) on a bright white screen background:

```
                R G B
background =>   1 1 1  ( white)
 XOR value =>   0 1 1
                -------
    result =>   1 0 0  ( red )
```

However, if the same XOR operation is applied over a black background the color of the object is cyan, instead of red:

```
                R G B
background =>   0 0 0  ( black )
 XOR value =>   0 1 1
                -------
    result =>   0 1 1  ( cyan )
```

The property of the XOR operation that makes the object's color change as it moves over different backgrounds can be at times an advantage, and at times a disadvantage. For example, an object displayed by conventional methods can disappear as it moves over a background of its same color. If this object is XORed onto the screen, it remains visible over most backgrounds. On the other hand, it may happen that the color of a graphics object is an important characteristic. In this case the changes brought about by XOR display operations may not be acceptable. Figure 28-4 shows how the XOR operation changes the attributes of an object (circle) as it is displayed over different backgrounds.

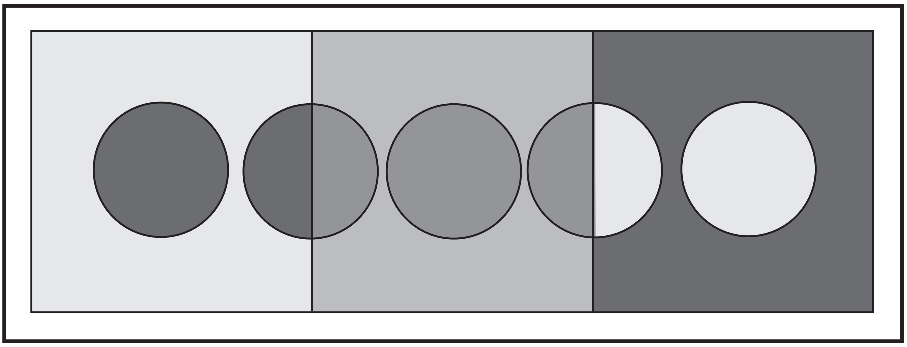


Figure 28-4 *Visualizing the XOR Operation*

28.5.2 XORing a Bitmap

One of the many possible uses of the XOR raster operation is to project a bitmap over an existing background. The graphics programmer can take advantage of the automatic draw-erase action of the XOR function to animate cursors and small sprites with minimal processing. The main drawback has already been mentioned: the object's color is partially determined by the background color. The following function XORs a variable-size bitmap onto the video display, at any desired screen location. The function assumes a true-color display mode in 640 by 480 by 24 format.

```
// Cross-shaped bitmap for demonstrating the DAXorBitmap()
// function
// 2 by 8 Bitmap      0    1    2    3    4    5    6    7
BYTE mapData[] = {  0x00,0x00,0x00,0x7f,0x7f,0x00,0x00,0x00, // 0
                    0x00,0x00,0x00,0x7f,0x7f,0x00,0x00,0x00, // 1
                    0x00,0x00,0x00,0x7f,0x7f,0x00,0x00,0x00, // 2
                    0x7f,0x7f,0x7f,0x7f,0x7f,0x7f,0x7f,0x7f, // 3
                    0x7f,0x7f,0x7f,0x7f,0x7f,0x7f,0x7f,0x7f, // 4
                    0x00,0x00,0x00,0x7f,0x7f,0x00,0x00,0x00, // 5
                    0x00,0x00,0x00,0x7f,0x7f,0x00,0x00,0x00, // 6
                    0x00,0x00,0x00,0x7f,0x7f,0x00,0x00,0x00};// 7
    . . .

//****************************************************************
// Name: DAXorBitmap
// Desc: Assembly language code to XOR a bitmap onto the screen
//       using direct access to the video buffer
//
// PRE:
// 1. First parameter is pointer to surface
// 2. Second and third parameters are x and y screen coordinates
// 3. Fourth parameter is bitmap width, in pixels
// 4. Fifth parameter is bitmap height, in pixels
// 5. Sixth parameter is pointer to bitmap
// 6. Assumes true-color mode 640 by 480 by 28
//
```

```
// POST:
// Returns 1 if call succeeds and 0 if it fails
//******************************************************************
int DAXorBitmap(
    LPDIRECTDRAWSURFACE4 lpPrimary,
    int xCoord,
    int yCoord,
    int bmWidth,
    int bmHeight,
    LPBYTE bitMapPtr)
{
    // Attempt to lock the surface for direct access
    if (!LockSurface(lpPrimary))
        return 0;                  // Lock failed

    _asm
    {
    PUSH    ESI              ; Save context
    PUSH    EDI
    PUSHF

    MOV    EAX,yCoord        ; Row number to EAX
    MUL    vidPitch;
    MOV    EBX,EAX           ; Store in EBX
    MOV    EAX,xCoord        ; x coordinate
    MOV    CX,3
    MUL    CX                ; 3 bytes per pixel
    ADD    EAX,EBX           ; move right to x-coordinate
    MOV    EDI,vidStart
    ADD    EDI,EAX
    MOV    ESI,bitMapPtr     ; Pointer to bitmap
    MOV    EBX,bmHeight      ; number of lines in bitmap
NEXT_BM_LINE:
    PUSH   EDI               ; Save start of line
    MOV    ECX,bmWidth       ; x-dimension of bitmap
XOR_PIX_LINE:
    MOV    AL,[ESI]          ; Bitmap data to AL
    XOR    [EDI],AL          ; Set blue attribute
    INC    EDI
    XOR    [EDI],AL          ; Set green
    INC    EDI
    XOR    [EDI],AL          ; Set red
    INC    EDI
    INC    ESI               ; Bitmap pointer to next byte
    LOOP   XOR_PIX_LINE
; End of line
    POP    EDI               ; Pointer to start of line
    ADD    EDI,vidPitch      ; Index to next line
    DEC    EBX               ; EBX is Lines counter
    JNZ    NEXT_BM_LINE
    ; Done!
    POPF                     ; Restore context
    POP    EDI
    POP        ESI
   }
   ReleaseSurface(lpPrimary);
   return 1;                         // Exit
}
```

In this function a single bitmap attribute is XORed with all three background colors. This keeps the bitmap small but limits the range of possible results. It would be quite easy to modify the routine so that the bitmap contains a byte value for each color attribute in a true-color mode.

28.6 Direct Access Project

The DD Access Demo project, located in the book's software package, is a demonstration of the direct access techniques discussed in this chapter. The program contains all the functions listed in this chapter, plus some other ones not mentioned in the text. It executes in exclusive, full-screen display mode. The text messages are displayed using GDI graphics and the geometrical figures using the direct access functions developed in this chapter. Color Figure 3 is a screen snapshot of the demo program. The labels lists the program functions that perform the corresponding operations.

Chapter 29

Blitting

Topics:

- Surfaces revisited
- Image transparency
- Color keys
- Using Blt() and BltFast()
- Blit-time transformations

This chapter is about a fundamental mechanism for rendering bitmaps called a blit, short for bit block transfer. The Windows GDI contains a blit function but DirectDraw provides its own versions, in the form of two functions named Blt() and BltFast(). A third variation, called BltBatch(), was announced but never implemented. Another rendering technique is called an overlay. Overlays are like a transparent media which can be placed over an image and then removed, restoring the original. Overlays have been implemented inconsistently in the video hardware, and for this reason they are not discussed.

29.1 Surface Programming

Before discussing DirectDraw blits, we must expand some of the notions related to DirectDraw surfaces. The following are the fundamental notions introduced in this section:

- The surface concept
- Primary and off-screen surfaces
- Enumerating surfaces
- Loosing and restoring surfaces

29.1.1 The DirectDraw Surface Concept

A DirectDraw surface is a linear area of video memory that holds image data. A DirectDrawSurface is a COM object in itself, with its own interface, and this interface is referenced in all surface-related operations. The current interface is IDirectDrawSurface7. Applications create a DirectDraw surface by calling the CreateSurface() function. If the call is successful it returns a pointer to the surface. In DirectX 7 this pointer is of type LPDIRECTDRAWSURFACE7. It is this pointer that is used in calling the functions of the IDirectDrawSurface7 interface. Table 29-1 lists these functions.

Table 29-1

Surface-Related Functions in DirectDraw

TYPE OR TOPIC	FUNCTION NAME
Allocating memory	Initialize()
	IsLost()
	Restore()
Attaching surfaces	AddAttachedSurface()
	DeleteAttachedSurface()
	EnumAttachedSurfaces()
	GetAttachedSurface()
Blitting	Blt()
	BltBatch() (not implemented in DirectX 6)
	BltFast()
	GetBltStatus()
Color keys	GetColorKey()
	SetColorKey()
Device contexts	GetDC()
	ReleaseDC()
Flipping	Flip()
	GetFlipStatus()
Locking surfaces	Lock()
	PageLock()
	PageUnlock()
	Unlock()
Textures	GetLOD
	GetPriority
SetLOD	PetPriority
Overlays	AddOverlayDirtyRect()
	EnumOverlayZOrders()
	GetOverlayPosition()
	SetOverlayPosition()
	UpdateOverlay()
	UpdateOverlayDisplay()
	UpdateOverlayZOrder()
Private data	FreePrivateData()
	GetPrivateData()
	SetPrivateData()
Capabilities	GetCaps()
Clipper	GetClipper()
	SetClipper()
Characteristics	ChangeUniquenessValue()
	GetPixelFormat()
	GetSurfaceDesc()
	GetUniquenessValue()
	SetSurfaceDesc()
Miscellaneous	GetDDInterface()

29.1.2 Surface Types

DirectDraw first attempts to create a surface in local video memory. If there is not enough video memory available to hold the surface, then DirectDraw tries to use non-local video memory, and finally, if no other option is available, it creates the surface in system memory. Code can also explicitly request that a surface be created in a certain type of memory by including the appropriate flags in the CreateSurface() call. A typical DirectDraw application operates on several surfaces.

The primary surface is the one visible on the monitor and it is identified by the DDSCAPS_PRIMARYSURFACE flag. There can be only one primary surface for each DirectDraw object. The size and pixel format of the primary surface matches the current display mode. For this reason, the surface dimensions, mode, and pixel depth are not specified in the CreateSurface() call for a primary surface. In fact, the call fails if these dimensions are entered, even if they match those of the display mode.

Off-screen surfaces are often used to store bitmaps, cursors, sprites, and other forms of bitmapped imagery. Off-screen surfaces can reside in video memory or in system memory. For an off-screen surface to exist in video memory the total memory on the card must exceed the memory mapped to the video display. For example, a video card with 2Mb of video memory (2,097,152 bytes), executing in mode with a resolution of 640 by 480 pixels, at a rate of 3 bytes per pixel, requires 921,600 bytes for storing the displayed image (assuming that there are no unused areas in the pixel mapping). This leaves 1,175,552 bytes of memory on the video card which can be used as off-screen memory.

A special type of off-screen surface is the back buffer. A back buffer can be created if the amount of free video memory is sufficient to store a second displayable image. In the previous example, it is possible to create one back buffer since the display area requires 921,600 bytes, and there are 1,175,552 bytes of additional video memory available on the card. Back buffers, which are frequently used in animation, are discussed in Chapter 15.

An off-screen surface is created with the CreateSurface() function. The call must specify the surface dimensions, which means that it must include the DDSD_WIDTH and DDSD_HEIGHT flags. The corresponding values must have been previously entered in the dwWidth and dwHeight members of the DDSURFACEDESC2 structure. The call must also include the DDSCAPS_OFFSCREENPLAIN flag in the DDSCAPS2 structure. If possible, DirectDraw creates a surface in display memory. If there is not enough video memory available, it creates the surface in system memory. Code can explicitly choose display or system memory by entering the DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY flags in the dwCaps member of the DDSCAPS2 structure. The call fails, returning an error, if DirectDraw cannot create the surface in the specified location. Figure 29-1, on the following page, shows different types of DirectDraw surfaces.

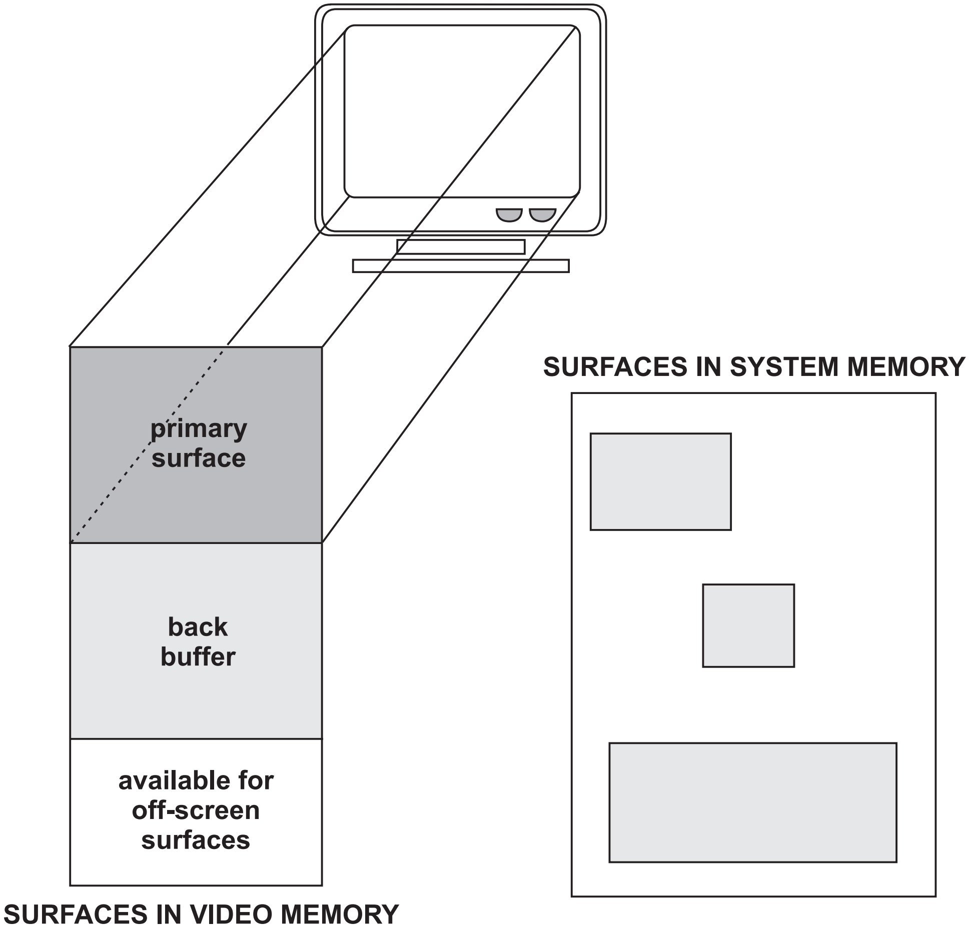


Figure 29-1 *DirectDraw Surface Types*

A surface is lost when the display mode is changed or when another application receives exclusive mode privileges. The Restore() function can be used to recreate lost surfaces and reconnect them to their DirectDrawSurface object. Applications using the IDirectDraw7 interface can restore all lost surfaces by calling RestoreAllSurfaces(). Note that restoring a surface does not reload bitmaps that may have existed before the surface was lost. It is up to the application to reconstruct the graphics on each of the surfaces.

When a surface is no longer needed it should be released by calling the Release() function. Each surface must be explicitly released, since there is no call to release all surfaces. However, if you implicitly created multiple surfaces with a single call to IDirectDraw7::CreateSurface, you need only release the front buffer. In this case, any pointers to back buffer surfaces are implicitly released and can no longer be used. Explicitly releasing a back buffer surface doesn't affect the reference count of the other surfaces in the chain.

29.1.3 Enumerating Surfaces

Applications often need to know if a surface that matches certain characteristics can be created, or may need a list of the existing surfaces and their properties. The IDirectDraw7 EnumSurfaces() function is used to enumerate surfaces. The function's general form is as follows:

```
HRESULT EnumSurfaces(
        DWORD dwFlags,                              // 1
        LPDDSURFACEDESC2 lpDDSD,                    // 2
        LPVOID lpContext,                           // 3
        LPDDENUMSURFACESCALLBACK2 lpEnumCallback    // 4
        );
```

The first parameter is a combination of a search-type flag and a matching flag. The search-type flag determines how the method searches for surfaces. Code can search for surfaces that can be created by using the description in the second parameter, or it can search for existing surfaces that already match that description. The matching flag determines whether the method enumerates all surfaces, those that match, or those that do not match the description specified in the second parameter. Table 29-2 lists the search and matching flags used in the EnumSurfaces() function.

Table 29-2

Flags in the EnumSurfaces() Function

FLAG	FUNCTION NAME
SEARCH-TYPE FLAGS:	
DDENUMSURFACES_CANBECREATED	Enumerates the first surface that can be created and that meets the specifications in the second parameter. This flag can only be used with the DDENUMSURFACES_MATCH flag.
DDENUMSURFACES_DOESEXIST	Enumerates the already existing surfaces that meet the specification in the second Parameter.
MATCHING-TYPE FLAGS:	
DDENUMSURFACES_ALL	Enumerates all of the surfaces that meet the specification in the second parameter. This Flag can only be used with the DDENUMSURFACES_DOESEXIST search type flag.
DDENUMSURFACES_MATCH	Searches for any surface that matches the specification in the second parameter.
DDENUMSURFACES_NOMATCH	Searches for any surface that does not match the specification in the second parameter.

The second parameter to EnumSurfaces() is the address of a structure variable of type DDSURFACEDESC2 that defines the characteristics of the surface. If the first parameter includes the DDENUMSURFACES_ALL flag, then this second parameter must be NULL.

The third parameter is the address of an application-defined structure that is passed to each enumeration member.

The fourth parameter is the address of a callback function, of type lpEnumSurfacesCallback, that is called every time the enumeration procedure finds a surface matching the predefined characteristics.

If the call succeeds the return value is DD_OK. If it fails, the return value may be one of the following errors:

- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS

Implementing the callback function for EnumSurfaces() is very similar to the processing described in Chapter 8 for the EnumDisplayModes() callback function. The project DD Info Demo contained in the book's software package contains sample code of the EdnumDisplayModes() callback.

Applications often need to know if a surface of certain characteristics is possible before it attempts to create it. In this case it is possible to combine the DDENUMSURFACES_CANBECREATED and DDENUMSURFACES_MATCH flags when calling EnumSurfaces(). The DDSURFACEDESC2 structure variable is initialized to contain the desired surface characteristics. If the characteristics include a particular pixel format, then the DDSD_PIXELFORMAT flag must also be present in the dwFlags member of the DDSURFACEDESC2 structure. In addition, the DDPIXELFORMAT structure in the surface description must be initialized and the flags set to the desired pixel format flags. These can be DDPF_RGB, DDPF_YUV, or both. In order to specify surface dimensions, code must include the DDSD_HEIGHT and DDSD_WIDTH flags in DDSURFACEDESC2. The dimensions are then specified in the dwHeight and dwWidth structure members. If the dimensions flags are not included, DirectDraw uses the dimensions of the primary surface.

The following code fragment shows a call to IDirectDraw7::EnumSurfaces to determine if a 640-by-480-by-24-bit RGB surface is available in the card's video memory space:

```
// Public variables
DDSURFACEDESC2      ddsd;
int                 surfCount = 0;
. . .
// Determine if a surface of 640 by 480 pixels, in 24-bits
// RGB color can be created in video memory
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize  = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS |
               DDSD_PIXELFORMAT |
               DDSD_HEIGHT |
               DDSD_WIDTH;
ddsd.ddpfPixelFormat.dwFlags = DDPF_RGB;
ddsd.ddpfPixelFormat.dwRGBBitCount = 24;
ddsd.ddsCaps.dwCaps = DDSCAPS_VIDEOMEMORY |
                      DDSCAPS_LOCALVIDMEM;
ddsd.dwHeight = 480;
ddsd.dwWidth  = 640;
lpDD7->EnumSurfaces(
       DDENUMSURFACES_CANBECREATED | DDENUMSURFACES_MATCH,
       &ddsd, NULL,
       SurfacesProc);
if (surfCount == 0)
     DDInitFailed(hWnd, hRet,
      "Surface not available");
. . .
//**********************************************
//    Callback function for EnumSurfaces()
//**********************************************
HRESULT WINAPI SurfacesProc(LPDIRECTDRAWSURFACE7 aSurfPtr,
                            LPDDSURFACEDESC2 aSurface,
                            LPVOID Context)
{
     surfCount++;
     return DDENUMRET_OK;  // Continue
}
```

Because the DDENUMSURFACES_MATCH flag is present in the call, the callback function, in this case named SurfacesProc(), receives control only if a surface can be created. In the preceding code sample each iteration of the callback function increments the variable surfCount. This variable holds the number of similar surfaces that can be created and its value is zero if no surfaces can be created. The calling routine inspects this variable to determine the results of the EnumSurfaces() call. The previous code fragment uses the DDInitFailed() function, developed in Chapter 13, to provide a terminal exit in case the surface cannot be created. In practice, an application may take another action, such as creating the surface in system memory instead of video memory. Note that the fourth parameter of EnumSurfaces() has to be typecast into a type LPDDENUMSURFACESCALLBACK2, otherwise a compiler error results.

The call to EnumSurfaces() attempts to create a temporary surface with the desired characteristics; however, code should not assume that a surface is not supported just because it is not enumerated. DirectDraw attempts to create a temporary surface with the memory constraints that exist at the time of the call. This can result in a surface not being enumerated even when the driver actually supports it.

29.1.4 Restoring Surfaces

It is possible to free surface memory associated with a DirectDrawSurface object, while the DirectDrawSurface objects representing these pieces of surface memory are not released. In this case several DirectDraw functions return DDERR_SURFACELOST. Surfaces can be lost because the display mode was changed, or because another application requested and obtained exclusive mode and freed all of the currently allocated surface memory. The DirectDraw Restore() function recreates these lost surfaces and reconnects them to their DirectDrawSurface object. If the application uses more than one surface, code can call the RestoreAllSurfaces() function to restore all surfaces at once. However, restoring a surface does not reload any imagery that may have previously existed in the surface.

29.1.5 Surface Operations

Most DirectDraw rendering operations relate to surfaces. The DirectX 7 SDK includes a program named ddtest which allows experimenting with DirectDraw options such as surfaces, blits, display modes, and capabilities, without actually writing code. Unfortunately, this program is not furnished in DirectX 8.

29.1.6 Transparency and Color Keys

In graphics programming you often need to display a new bitmap over an existing one. For example, the bitmap of an airplane is to overlay a background of mountains, sky, and clouds, contained in another bitmap. Since bitmaps are rectangular areas, the airplane bitmap is likely a rectangle that contains the image of the airplane. If we were to display the airplane by simply projecting its rectangular bitmap over the background, the result would be as shown in Color Figure 4.

The solution is to select the color of the framing rectangle of the airplane bitmap so that it is different from the colors used in drawing the airplane. The software can then be programmed to ignore the framing color while displaying the airplane bitmap. The processing logic is as follows:

- If bitmap pixel is equal to framing color, then leave the background pixel undisturbed.
- Otherwise, replace background pixel with foreground image pixel.

The effect is similar to having the image of the airplane drawn on a sheet of transparent plastic. The selection of a framing color, called the color key, plays an important role in the result. If a color key can be found that is not present in the source bitmap, then transparency is achieved perfectly. If not, some pixels of the foreground image will not be shown. The greater the color range, the easier it is to find a satisfactory color key. It is hard to imagine a 24-bit color bitmap (16.7 million colors) that will not have a single color value that is absent in the image. Color Figure 5 shows the elements of transparency using a color key.

An alternative option is based on a color key located in the background image, also called the destination. In this case the color key determines if the foreground image pixel is used or not. The logic is as follows:

- If background pixel is the color key, then use foreground pixel over background.
- Otherwise, leave background undisturbed.

The result of using a destination color key is a window on which the foreground image is displayed. Here again, the programmer must find an attribute for the color key that is not used in the background.

DirectDraw supports both source and destination color keys. Application code supplies either a single color, or a color range for source or destination color keying. Source and destination color keys can be combined on different surfaces. For example, a destination color key can be attached to a surface in order to create a window on which the mountains, sky, and clouds are visible. Then a source color key can be used in a second surface in order to display a bitmap transparently over this background. Color plate 4 shows transparency based on the simultaneous use of source and destination color keys.

Color Figure 5 shows the manipulation of three different surfaces in implementing source and destination color key transparency. Surface 1 contains a window in which a destination color key has been defined. Surface 2 is a bitmap image. Surface 3 is a sprite representing an airplane in which the background is a source color key. This sprite is transparently blitted onto the bitmap on surface 2, and then surface 2 is transparently blitted onto surface 1. The resulting image is shown at the bottom of the illustration.

29.1.7 Selecting and Setting the Color Key

A DirectDraw color key is always associated with a surface. Code can set the color keys for a surface when it is created, or afterward. To set a color key or keys when creating a surface you assign the desired color values to one or both of the ddckCKSrcBlt and ddckCKDestBlt members of the DDSURFACEDESC2 structure. When CreateSurface() is called, the color keys are automatically assigned. If the color key is to be used in blitting, one or both of DDSD_CKSRCBLT or DDSD_CKDESTBLT must be included in the dwFlags member.

The DirectDraw function SetColorKey() sets the color key for an existing DirectDrawSurface object. The function's general form is as follows:

```
HRESULT SetColorKey(
                  DWORD dwFlags,            // 1
                  LPDDCOLORKEY lpDDColorKey  // 2
                  );
```

The first parameter is a flag that determines the type of color key to be used. Table 29-3 lists the predefined constants used in this parameter.

Table 29-3

Constants Used in SetColorKey() Function

CONSTANT	ACTION
DDCKEY_COLORSPACE	The structure contains a color range. Not set if the structure contains a single color key.
DDCKEY_DESTBLT	The structure specifies a color key or color space to be used as a destination color key for blit operations.
DDCKEY_DESTOVERLAY	The structure specifies a color key or color space to be used as a destination color key for overlay operations.
DDCKEY_SRCBLT	The structure specifies a color key or color space to be used as a source color key for blit operations.
DDCKEY_SRCOVERLAY	The structure specifies a color key or color space to be used as a source color key for overlay operations.

Color keys are used in two different types of DirectDraw bitmap display operations: blits and overlays. Because the specification of overlays were never defined in

DirectX, and because they are currently supported by few video cards, we do not cover hardware overlays in this book.

The second parameter is the address of a structure variable of type DDCOLORKEY structure that contains the new color key values for the DirectDrawSurface object. If this value is NULL, then the existing color key is removed from the surface.

If the call to SetColorKey() succeeds, the function returns DD_OK. If it fails, one of the following error codes is returned:

- DDERR_GENERIC
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_INVALIDSURFACETYPE
- DDERR_NOOVERLAYHW
- DDERR_NOTAOVERLAYSURFACE
- DDERR_SURFACELOST
- DDERR_UNSUPPORTED
- DDERR_WASSTILLDRAWING

The DDCOLORKEY Structure

The color key is described in a DDCOLORKEY structure. The structure is used for either a source color key, a destination color key, or a color range. A single color key is specified when both structure members have the same value. DDCOLORKEY is defined in the Windows header files as follows:

```
typedef struct _DDCOLORKEY{
    DWORD dwColorSpaceLowValue;
    DWORD dwColorSpaceHighValue;
} DDCOLORKEY,FAR* LPDDCOLORKEY;
```

The member dwColorSpaceLowValue contains the low value (inclusive) of the color range that is to be used as the color key. The member dwColorSpaceHighValue contains the high value (also inclusive). Tthe color key is a single color, not a range, when both members are assigned the same value.

Color keys are specified using the pixel format of the surface. If a surface is palletized, then the color key is an index or a range of indexes. If the surface is a 16-bit color (hi-color), then the color key is a word-size value. If the surface's pixel format is RGB or YUV, then the color key is specified in an RGBQUAD or YUVQUAD structure, as in the following code fragments:

```
// Hi color mode is the single color key.
dwColorSpaceLowValue = 0xf011;
dwColorSpaceHighValue = 0xf011;

// RGB color 255,128,128 is the single color key.
dwColorSpaceLowValue = RGBQUAD(255,128,128);
```

```
dwColorSpaceHighValue = RGBQUAD(255,128,128);

// YUV color range used as a color key
dwColorSpaceLowValue = YUVQUAD(120,50,50);
dwColorSpaceHighValue = YUVQUAD(140,50,50);
```

The YUV format was developed to more easily compress motion video data. It is based on the physics of human vision, which makes the eye is more sensitive to brightness levels than to specific colors. The YUV acronym refers to a three-axes coordinate system. The y-axis encodes the luminance component, while the u- and v-axes encode the chrominance, or color, element. Although several different implementations of the YUV format are available, no one format is directly supported by DirectDraw.

In the third example the YUV color range extends from 120-50-50 to 150-50-50. In this case any pixel with a y value between 120 and 150, and u and v values of 50, serve as a color key. Range values for color keys are often used when working with video data or photographic images, since in this case, there are usually variations in the background color values. Art work composed with draw or paint programs often use single-key colors.

29.1.8 Hardware Color Keys

Transparency and color keys are supported by the HEL. Although you can always assume that these functions are available, support for a color key range is not required. Code should check the dwCKeyCaps member of the DDCAPS structure. The DDCAPS_COLORKEY constant of the dwCaps member identifies some form of color key support for either overlay or blit operations. The dwCKeyCaps member defines the options listed in Table 29-4.

Table 29-4

Color Key Capabilities in the dwCKeyCaps Member of DDCAPS Structure

CONSTANT	MEANING
DDCKEYCAPS_DESTBLT	Supports transparent blitting. Color key identifies the replaceable bits of the destination surface for RGB colors.
DDCKEYCAPS_DESTBLTCLRSPACE	Supports transparent blitting. Color space identifies the replaceable bits of the destination surface for RGB colors.
DDCKEYCAPS_DESTBLTCLRSPACEYUV	Supports transparent blitting. Color space identifies the replaceable bits of the destination surface for YUV colors.
DDCKEYCAPS_DESTBLTYUV	Supports transparent blitting. Color key identifies the replaceable bits of the destination surface for YUV colors.

(continues)

Table 29-4

Color Key Capabilities in DwcKeyCaps Member of DDCAPS Structure (continued)

CONSTANT	MEANING
DDCKEYCAPS_DESTOVERLAY	Supports overlaying with color keying of the replaceable bits of the destination surface being overlaid for RGB colors.
DDCKEYCAPS_DESTOVERLAYCLRSPACE	Supports a color space as the color key for the destination of RGB colors.
DDCKEYCAPS_DESTOVERLAYCLRSPACEYUV	Supports a color space as the color key for the destination of YUV colors.
DDCKEYCAPS_DESTOVERLAYONEACTIVE	Supports only one active destination color key value for visible overlay surfaces.
DDCKEYCAPS_DESTOVERLAYYUV	Supports overlaying using color keying of the replaceable bits of the destination surface being overlaid for YUV colors.
DDCKEYCAPS_NOCOSTOVERLAY	No bandwidth trade-offs for using the color key with an overlay.
DDCKEYCAPS_SRCBLT	Supports transparent blitting using the color key for the source with this surface for RGB colors.
DDCKEYCAPS_SRCBLTCLRSPACE	Supports transparent blitting using a color space for the source with this surface for RGB colors.
DDCKEYCAPS_SRCBLTCLRSPACEYUV	Supports transparent blitting using a color space for the source with this surface for YUV colors.
DDCKEYCAPS_SRCBLTYUV	Supports transparent blitting using the color key for the source with this surface for YUV colors.
DDCKEYCAPS_SRCOVERLAY	Supports overlaying using the color key for the source with this overlay surface for RGB colors.
DDCKEYCAPS_SRCOVERLAYCLRSPACE	Supports overlaying using a color space as the source color key for the overlay surface for RGB colors.
DDCKEYCAPS_SRCOVERLAYCLRSPACEYUV	Supports overlaying using a color space as the source color key for the overlay surface for YUV colors.
DDCKEYCAPS_SRCOVERLAYONEACTIVE	Supports only one active source color key value for visible overlay surfaces.
DDCKEYCAPS_SRCOVERLAYYUV	Supports overlaying using the color key for the source with this overlay surface for YUV colors.

Some hardware support color ranges only for YUV pixel data, which is usually video. The transparent background in video footage (the "blue screen" against which the subject was photographed) might not be a pure color. For this reason a color range in the key is desirable in this case.

29.2 The Blit

In the blit a rectangular block of memory bits, called the source, is transferred as a block into a rectangular memory area called the destination. If the destination of the transfer is screen memory, then the bitmapped image is immediately displayed. The source and destination bit blocks can be combined logically or arithmetically, or a unary operation can be performed on the source or the destination bit blocks.

GDI blits have extremely slow performance, thus, they are rarely used in high-quality graphics. DirectDraw contains its own blit functions, which execute considerably faster than the GDI blit. The DirectDraw blit functions are named Blt() and BltFast(). They are both associated with DirectDraw surface objects. Microsoft announced a third blit version, called BltBatch(), but it has not been implemented and probably never will.

In DirectDraw blit operations usually take place from an off-screen surface onto the back buffer or to the primary surface. Much of the processing time of a typical DirectDraw application is spent blitting imagery. Also, the performance capability, which is related to the band width of a particular blitter, determines the speed of the video output. Figure 29-2 shows the most common forms of the DirectDraw blit operation.

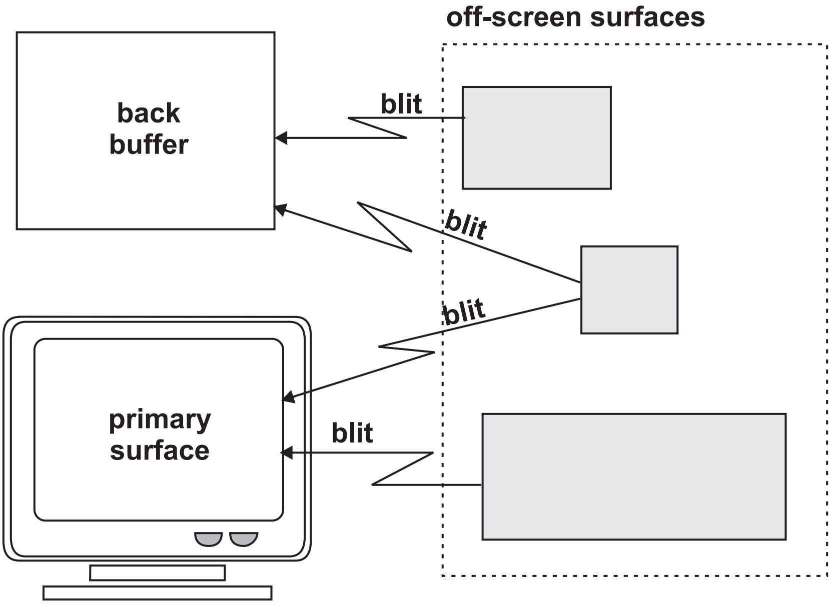


Figure 29-2 *The DirectDraw Blit.*

Both blit functions, Blt() and BltFast(), operate on a destination surface, which is referenced in the call, and receive the source surface as a parameter. It is possible for both source and destination to be the same surface. In this case DirectDraw preserves all source pixels before overwriting them. Blt() is more flexible, but BltFast() is faster, especially if there is no hardware blitter. Applications can determine the blitting capabilities of the hardware from the DDCAPS structure obtained by means of the GetCaps() function. If the dwCaps member contains DDCAPS_BLT, the hardware supports blitting.

29.2.1 BltFast()

BltFast requires a valid rectangle in the source surface. The pixels are copied from this rectangle onto the destination surface. If the entire surface is to be copied, then the source rectangle is defined NULL. BltFast() also requires x- and y-coordinates in the destination surface. The source rectangle must fit within the destination surface. If the source rectangle is larger than the destination the call fails and BltFast() returns DDERR_INVALIDRECT. BltFast() cannot be used on surfaces that have an attached clipper. Neither does it support stretching, mirroring, or other effects that can be performed with Blt().

The function's general form is as follows:

```
HRESULT BltFast(
                DWORD dwX,                              // 1
                DWORD dwY,                              // 2
                LPDIRECTDRAWSURFACE7 lpDDSrcSurface,    // 3
                LPRECT lpSrcRect,                       // 4
                DWORD dwTrans                           // 5
                );
```

The first and second parameters are the x- and y-coordinates to blit to on the destination surface. The third parameter is the address of a IDirectDrawSurface7 interface for the DirectDrawSurface object that is the source of the blit. The fourth parameter is a RECT structure that defines the upper-left and lower-right points of the rectangle on the source surface. The fifth parameter defines the type of blit, as listed in Table 29-5.

Table 29-5

Type of Transfer Constants in BltFast()

CONSTANT	ACTION
DDBLTFAST_DESTCOLORKEY	Transparent blit that uses the destination's color key.
DDBLTFAST_NOCOLORKEY	Normal copy blit with no transparency.
DDBLTFAST_SRCCOLORKEY	Transparent blit that uses the source's color key.
DDBLTFAST_WAIT	Does not produce a DDERR_WASSTILLDRAWING message if the blitter is busy. Returns as soon as the blit can be set up or another error occurs.

If the call succeeds, BltFast() returns DD_OK. If it fails it returns one of the following self-explanatory values:

- DDERR_EXCEPTION
- DDERR_GENERIC
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_INVALIDRECT
- DDERR_NOBLTHW
- DDERR_SURFACEBUSY
- DDERR_SURFACELOST
- DDERR_UNSUPPORTED
- DDERR_WASSTILLDRAWING

BltFast() always attempts an asynchronous blit if it is supported by the hardware. The function works only on display memory surfaces and cannot clip when blitting. According to Microsoft, BltFast() is 10 percent faster than the Blt() method if there is no hardware support, but there is no speed difference if the display hardware is used. Figure 29-3 is a diagram showing the parameters and operation of the BltFast() function.

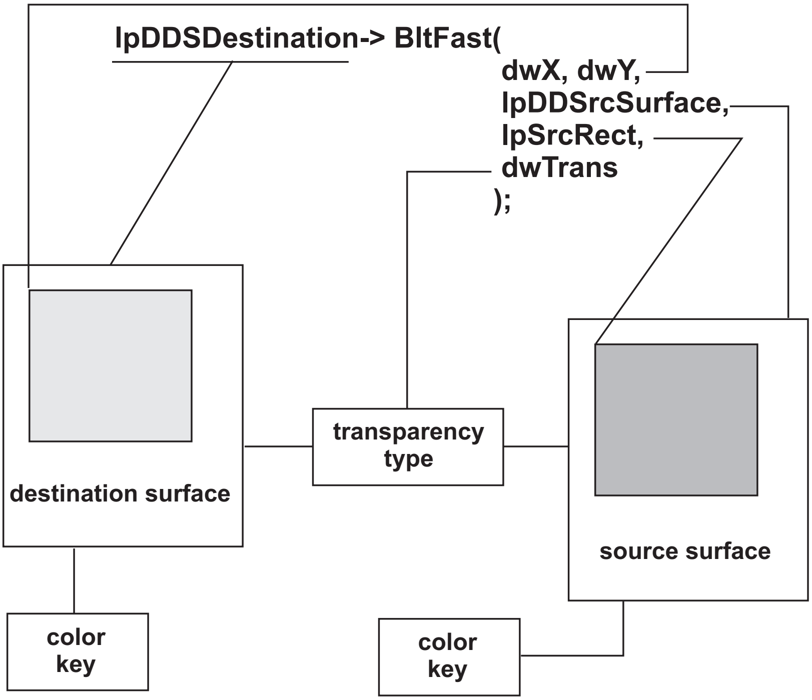


Figure 29-3 *The BltFast() Function*

29.2.2 Blt()

Like BltFast(), Blt() performs a bit block transfer from a source surface onto a destination surface, but Blt() is the more flexible and powerful of the two. Blt() allows a clipper to be attached to the destination surface, in which case clipping is performed if the destination rectangle falls outside of the surface. Blt() can also scale the source image to fit the destination rectangle. Scaling is disabled when both surfaces are of the same size. The function's general form is as follows:

```
HRESULT Blt(
          LPRECT lpDestRect,                    // 1
          LPDIRECTDRAWSURFACE7 lpDDSrcSurface,  // 2
          LPRECT lpSrcRect,                     // 3
          DWORD dwFlags,                        // 4
          LPDDBLTFX lpDDBltFx                   // 5
          );
```

The first parameter is the address of a RECT structure that defines the upper-left and lower-right points of the source rectangle. If this parameter is NULL, the entire destination source surface is used.

The second parameter is the address of the IDirectDrawSurface4 interface for the DirectDrawSurface object that is the source of the blit.

The third parameter is the address of a RECT structure that defines the upper-left and lower-right points of the source rectangle from which the blit is to take place. If this parameter is NULL, then the entire source surface is used.

The fourth parameter is one or more flags that determine the valid members of the associated DDBLTFX structure, which specifies color key information or requests a special behavior. Three types of flags are currently defined: validation flags, color key flags, and behavior flags. Table 29-6 lists the predefined constants for this parameter.

Table 29-6

Flags for the Blt() Function

FLAGS	MEANING
VALIDATION FLAGS:	
DDBLT_COLORFILL	The dwFillColor member of the DDBLTFX structure is the RGB color that fills the destination rectangle.
DDBLT_DDFX	The dwDDFX member of the DDBLTFX structure specifies the effects to use for the blit.
DDBLT_DDROPS	The dwDDROP member of the DDBLTFX structure specifies the raster operations (ROPS) that are not part of the Win32 API.
DDBLT_DEPTHFILL	The dwFillDepth member of the DDBLTFX structure is the depth value with which to fill the destination rectangle.

(continues)

Table 29-6

Flags for the Blt() Function (continued)

FLAGS	MEANING
DDBLT_KEYDESTOVERRIDE	The ddckDestColorkey member of the DDBLTFX structure is the color key for the destination surface.
DDBLT_KEYSRCOVERRIDE	The ddckSrcColorkey member of the DDBLTFX structure is the color key for the source surface.
DDBLT_ROP	The dwROP member of the DDBLTFX structure is the ROP for this blit. The ROPs are the same as those defined in the Win32 API.
DDBLT_ROTATIONANGLE	The dwRotationAngle member of the DDBLTFX structure is the rotation angle, in 1/100th of a degree units, for the surface.
COLOR KEY FLAGS:	
DDBLT_KEYDEST	The color key is associated with the destination surface.
DDBLT_KEYSRC	The color key is associated with the source surface.
BEHAVIOR FLAGS:	
DDBLT_ASYNC	Blit asynchronously in the FIFO order received. If no room is available in the FIFO hardware, the call fails.
DDBLT_WAIT	Postpones the DDERR_WASSTILLDRAWING return value if the blitter is busy, and returns as soon as the blit can be set up or another error occurs.

The fifth parameter is the address of a structure variable of type DDBLTFX that defines special effects during the blit, including raster operation codes (ROP) and override information. Because of their complexity, special effects during blit operations are discussed in a separate section. Figure 29-4, on the following page, shows the parameters and operation of the Blt() function.

If the call succeeds, the return value is DD_OK. If it fails, the return value is one of the following error codes:

- DDERR_GENERIC
- DDERR_INVALIDCLIPLIST
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_INVALIDRECT

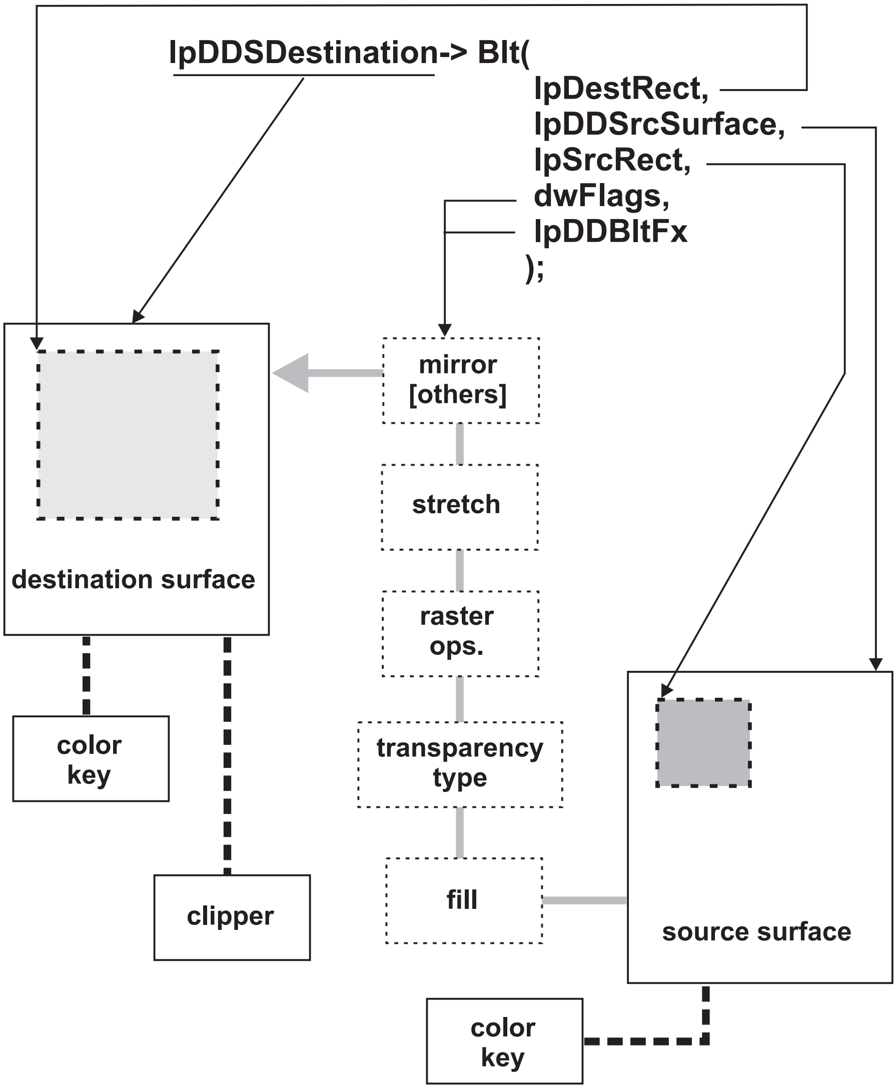


Figure 29-4 *The Blt() Function*

- DDERR_NOALPHAHW
- DDERR_NOBLTHW
- DDERR_NOCLIPLIST
- DDERR_NODDROPSHW
- DDERR_NOMIRRORHW
- DDERR_NORASTEROPHW
- DDERR_NOROTATIONHW
- DDERR_NOSTRETCHHW

- DDERR_NOZBUFFERHW
- DDERR_SURFACEBUSY
- DDERR_SURFACELOST
- DDERR_UNSUPPORTED
- DDERR_WASSTILLDRAWING

The Blt() function is capable of synchronous or asynchronous blits. Source and destination can be display memory to display memory, display memory to system memory, system memory to display memory, or system memory to system memory. The default is asynchronous. The function supports both source and destination color keys. If the source and the destination rectangles are not the same size, Blt() performs the necessary stretching. Blt() returns immediately with an error if the blitter is busy. If the code specifies the DDBLT_WAIT flag, then a synchronous blit takes place and the call waits until the blit can be set up or until another error occurs.

In the Blt() function there must be a valid rectangle in the source surface (or NULL to specify the entire surface), and a rectangle in the destination surface to which the source image is copied. Here again, NULL means the destination rectangle is the entire surface. If a clipper is attached to the destination surface, then the bounds of the destination rectangle can fall outside the surface and clipping is automatically performed. If there is no clipper, the destination rectangle must fall entirely within the surface or else the method fails with DDERR_INVALIDRECT.

29.3 Blit-Time Transformations

Several transformations can take place at blit-time. The most important ones are color fills, scaling, mirroring, and raster operations. Other effects, such as rotation, are not required by the HEL; therefore, they cannot be used if the hardware does not support them. Applications that do not require any special blit-time transformations other than scaling can pass NULL as in the fourth parameter of the Blt() function. Code can determine the hardware support for blit-time transformations by calling GetCaps().

Applications that require a particular blit-time transformation must pass the corresponding value in one of the members of the DDBLTFX structure. The appropriate flags must also be included in the fourth parameter to Blt(), which determines which members of the structure are valid. Some transformations require only setting a single flag, others require several of them.

The dwFlags member of DDBLTFX named DDBLTFX_NOTEARING can be used when blitting images directly to the front buffer. The action of this flag is to time the blit so that it coincides with the screen's vertical retrace cycle, thus minimizing the possibility of tearing. Tearing and screen update timing are discussed in the context of DirectDraw animation, in Chapter 16.

Applications that use surface color keys when calling BltFast() or Blt() must set one or both of the DDBLTFAST_SRCCOLORKEY or DDBLTFAST_DESTCOLORKEY flags in the corresponding function parameter. Alternatively, code can place the appropriate color values in the ddckDestColorkey and ddckSrcColorkey members of the DDBLTFX structure that is passed to the function in the lpDDBltFx parameter. In this case it is also necessary to set the DBLT_KEYSRCOVERRIDE or DDBLT_KEYDESTOVERRIDE flag, or both, in the dwFlags parameter. The resulting action is that the selected color keys are taken from the DDBLTFX structure rather than from the surface properties.

29.3.1 Color Fill Blit

A blit operation can be used to fill the entire surface, or a part of it, with a single color. This can be used for creating backgrounds when using a destination color key, and for clearing large screen areas. When Blt() is used to perform a color fill, the call must reference the DDBLT_COLORFILL flag. The following code fragment fills an entire surface with the color blue. Code assumes that lpDDS is a valid pointer to an IDirectDrawSurface7 interface.

```
HRESULT         ddrval;
DDBLTFX         ddbltfx;
.
.
.
ZeroMemory(&ddbltfx, sizeof(ddbltfx));
ddbltfx.dwSize = sizeof(ddbltfx);

ddbltfx.dwFillColor = ddpf.dwBBitMask; // Pure blue
ddrval = lpDDS->Blt(
                NULL,         // Destination is entire surface
                NULL,         // No source surface
                NULL,         // No source rectangle
                DDBLT_COLORFILL, &ddbltfx);
if(ddrval != DD_OK)
// Error handler goes here
```

29.3.2 Blit Scaling

The Blt() function automatically scales the source image to fit the destination rectangle. Blt() automatically re-scales the source image to fit the destination rectangle. If resizing is not required, the source and destination rectangles should be exactly the same size. Scaling must be implemented in the HEL, so it is always available. Some video cards have hardware support for scaling operations. Hardware acceleration for scaling can be detected by examining the flags that start with DDFXCAPS_BLT in the dwFXCaps member of the DDCAPS structure for the device. For example, the DDFXCAPS_BLTSTRETCHXN capability indicates integer shrinking support, and DDFXCAPS_BLTSTRETCHX arbitrary stretching support. If a device has the first flag, but not the second one, then it provides hardware support when the x-axis of the source rectangle is being multiplied by a whole number, but not when the factor is non-integral.

Applications can inspect the dwCXCaps member of the DDCAPS structure to determine if hardware support is available and of which type. Table 29-7 lists the most used predefined constants in the scaling capabilities flag.

Table 29-7

Scaling Flags for the Blt() Function

FLAG	MEANING
DDFXCAPS_BLTALPHA	Supports alpha-blended blit operations.
DDFXCAPS_BLTARITHSTRETCHY	Arithmetic operations, rather than pixel-doubling techniques, are used to stretch and shrink surfaces along the y-axis.
DDFXCAPS_BLTARITHSTRETCHYN	Arithmetic operations, rather than pixel-doubling techniques, are used to stretch and shrink surfaces along the y-axis. Stretching must be integer based.
DDFXCAPS_BLTSHRINKX	Arbitrary shrinking of a surface along the x-axis (horizontally).
DDFXCAPS_BLTSHRINKXN	Integer shrinking of a surface along the x-axis.
DDFXCAPS_BLTSHRINKY	Arbitrary shrinking of a surface along the y-axis.
DDFXCAPS_BLTSHRINKYN	Integer shrinking of a surface along the y-axis.
DDFXCAPS_BLTSTRETCHX	Arbitrary stretching of a surface along the x-axis.
DDFXCAPS_BLTSTRETCHXN	Integer stretching of a surface along the x-axis.
DDFXCAPS_BLTSTRETCHY	Arbitrary stretching of a surface along the y-axis (vertically).
DDFXCAPS_BLTSTRETCHYN	Supports integer stretching of a surface along the y-axis.

Scaling is automatically disabled when the source and destination rectangles are exactly the same size. An application can use the BltFast() function, instead of Blt(), in order to avoid accidental scaling due to different sizes of the source and destination rectangles.

Some video cards support arithmetic scaling. In this case the scaling operation is performed by interpolation rather than by multiplication or deletion of pixels. For example, if an axis is being increased by one-third, the pixels are recolored to provide a closer approximation to the original image than would be produced by doubling every third pixel on that axis. Code has little control over the type of scaling performed by the driver. The only possibility is to set the DDBLTF-X_ARITHSTRETCHY flag in the dwDDFX member of the DDBLTFX structure passed to Blt(). This flag requests that arithmetic stretching be done on the y-axis.

Arithmetic stretching on the x-axis and arithmetic shrinking are not currently supported in the DirectDraw API, but a driver may perform them on its own.

29.3.3 Blit Mirroring

Mirroring is another blit-time transformation supported by the HEL. Applications can assume that it is available even if it not supported in the hardware. Mirroring is defined in the x-axis and the y-axis of the blit rectangle. Figure 29-5 shows mirroring along either axis.

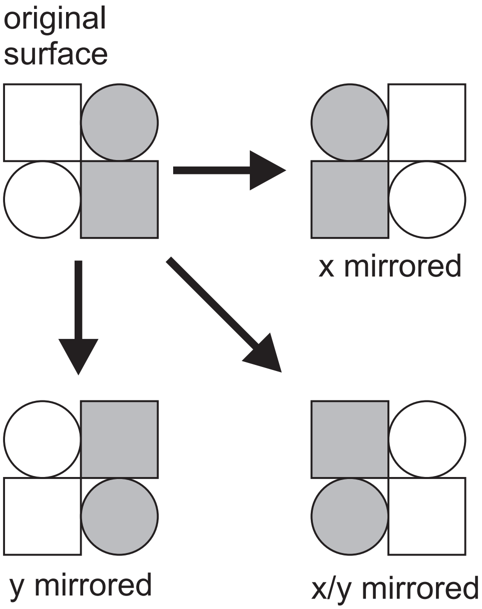


Figure 29-5 *Bit-Time Mirroring Transformations*

Table 29-8 lists the predefined constants used in mirroring transformations during Blt().

Table 29-8

Mirroring Flags for the Blt() Function

FLAGS	MEANING
DDBLTFX_MIRRORLEFTRIGHT	Mirrors on the y-axis. The surface is mirrored from left to right.
DDBLTFX_MIRRORUPDOWN	Mirrors on the x-axis. The surface is mirrored from top to bottom.

Applications sometimes need several versions of a symmetrical sprite, in which the image faces in different directions. Rather than creating a bitmap for each image, it is possible to generate them by mirroring the original. Hardware support for mirroring can be determined by the presence of the DDFXCAPS-

_BLTMIRRORLEFTRIGHT and DDFCAPS_BLTMIRRORUPDOWN identifiers in the dwFXCaps member of the DDCAPS structure.

29.3.4 Raster Operations

Blit-time transformations can include some of the standard raster operations (ROPs) used by the GDI BitBlt() functions. At present only SRCCOPY (the default), BLACKNESS, and WHITENESS are supported by the HEL. Hardware support for other raster operations can be determined by examining the DDCAPS structure. Code that uses any of the standard ROPS with the Blt method must set the corresponding flag in the dwROP member of the DDBLTFX structure. The dwDDROP member of the DDBLTFX structure is for specifying ROPs specific to DirectDraw. No such ROPs have been defined at this time.

29.4 Blit-Rendering Operations

Many types of applications rely heavily on bitmaps; these include image processing, simulations, virtual reality, artificial life, and electronic games. The real-color and true-color modes make it possible to use bitmaps to encode images with photo-realistic accuracy. The processing capabilities make possible the effective manipulation of bitmapped images. DirectDraw implements a new dimension of functionality in bitmap processing and display operations. In DirectDraw bitmap manipulations consist of four basic steps:

- Loading the bitmap into application memory
- Obtaining the bitmap data
- Moving the bitmap onto a DirectDraw surface
- Blitting the bitmap onto the video display

29.4.1 Loading the Bitmap

Loading a bitmap onto the application's memory space is an operation of GDI graphics, not actually part of DirectDraw. The demonstration program DD Bitmap Blit, in the book's software package, loads several bitmaps during WM_CREATE message processing. In this case we used Developer Studio to define the bitmaps as program resources, and then used LoadBitmap() to load them into the application's memory space. Alternatively, instead of defining the bitmap as a program resource, we can use LoadImage() to load the bitmap directly from the disk file in which it is stored. At this time we can also perform certain preliminary checks to make sure that the DirectDraw surface is compatible with the bitmap to be displayed. Note that the sample code requires that the surface be nonpalletized. GetSurfaceDesc() is used to fill a DDSURFACEDESC2 structure. The DDPIXELFORMAT structure, which is part of DDSURFACEDESC2, contains two relevant values: the flag DDPF_RGB indicates that the RGB data is valid, and the dwRGBBitCount member contains the number of RGB bits per pixel. If the DDPF_RGB flag is set and dwRGBBitCount > 15 we can assume that the surface is nonpalletized, and therefore, compatible.

Note that the LoadImage() function does not return palette information. Microsoft Knowledge Base Article Q158898 lists the function LoadBitmapFromBMPFile() which uses the DIBSection's color table to create a palette. If no color table is present, then a half-tone palette is created. The source for this function can be found in the MSDN Library that is part of Visual C++.

Once code has determined that a compatible surface is available, it can proceed to load the bitmap. The general form of the LoadImage() function is as follows:

```
HANDLE LoadImage(
                  HINSTANCE hInst,              // 1
                  LPCTSTR lpszName,             // 2
                  UINT uType,                   // 3
                  int cxDesired,                // 4
                  int cyDesired,                // 5
                  UINT fuLoad                   // 6
                  );
```

The first parameter is a handle to an instance of the module that contains the image to be loaded. In the case of an image contained in a file, this parameter is set to zero.

The second parameter is a pointer to the image to load. If it is non NULL and the sixth parameter (described later) does not include LR_LOADFROMFILE, then it is a pointer to a null-terminated string that contains the filename of the image resource.

The third parameter is the image type. It can be one of the following constants:

- IMAGE_BITMAP
- IMAGE_CURSOR
- IMAGE_ICON

The fourth and fifth parameters specify the pixel width and height of the bitmap, cursor, or icon. If this parameter is zero and the sixth parameter is LR_DEFAULTSIZE, then the function uses the SM_CXICON or SM_CXCURSOR system metric value to set the width. If this parameter is zero, and if LR_DEFAULTSIZE is present in the sixth parameter, then the function uses the actual width and height of the bitmap.

The sixth and last parameter is one or more flags represented by the predefined constants listed in Table 29-9.

LoadImage() returns the handle of the newly loaded image if the call succeeds. If the function fails, it returns NULL. Although the system automatically deletes all resources when the process that created them terminates, applications can save memory by releasing resources that are no longer needed. DeleteObject() is used to release a bitmap, DestroyIcon() for icons, and DestroyCursor() for cursor resources.

The following function is used to load a bitmap into the application's memory space and obtain its handle. In this case the code checks for a surface compatible with a nonpalletized bitmap.

Table 29-9

Predefined Constants in LoadImage() Function

<table>
<tr><th>CONSTANT</th><th>MEANING</th></tr>
<tr><td>LR_DEFAULTCOLOR</td><td>Default flag. Does nothing.</td></tr>
<tr><td>LR_CREATEDIBSECTION</td><td>When the third parameter is IMAGE_BITMAP, this flag causes the function to return a DIB section bitmap rather than a compatible bitmap. It is useful for loading a bitmap without mapping it to the colors of the display device.</td></tr>
<tr><td>LR_DEFAULTSIZE</td><td>For cursor and icons the width or height values are those specified by the system metric values, but only if the fourth and fifth parameters are set to zero. If this flag is not specified and the fourth and fifth parameters are set to zero, the function uses the actual resource size.</td></tr>
<tr><td>LR_LOADFROMFILE</td><td>Loads the image from the file specified by the second parameter. If this flag is not specified, lpszName is the name of the resource.</td></tr>
<tr><td>LR_LOADMAP3DCOLORS</td><td>Searches the color table for the image and replaces the following shades of gray with the corresponding 3D color:
<table>
<tr><th>Color</th><th>RGB value</th><th>Replaced with</th></tr>
<tr><td>Dk Gray</td><td>RGB(128,128,128)</td><td>COLOR_3DSHADOW</td></tr>
<tr><td>Gray</td><td>RGB(192,192,192)</td><td>COLOR_3DFACE</td></tr>
<tr><td>Lt Gray</td><td>RGB(223,223,223)</td><td>COLOR_3DLIGHT</td></tr>
</table></td></tr>
<tr><td>LR_LOADTRANSPARENT</td><td>Retrieves the color value of the top-left pixel in the image and replaces the corresponding entry In the color table with the default window color (COLOR_WINDOW). All pixels in the image that use that entry become the default window color. This value applies only to images that have corresponding color tables.</td></tr>
<tr><td>LR_MONOCHROME</td><td>Converts the image to black and white pixels.</td></tr>
<tr><td>LR_SHARED</td><td>Shares the image handle if the image is loaded multiple times. If LR_SHARED is not used, a second call to LoadImage for the same resource will load the image again and returns a different handle. LR_SHARED should not be used for images that have nonstandard sizes, that may change after loading, or that are loaded from a file. In Windows 95 and Windows 98 LoadImage() finds the first image with the requested resource name in the cache, regardless of the size requested.</td></tr>
<tr><td>LR_VGACOLOR</td><td>Use true VGA colors.</td></tr>
</table>

```
//***************************************************************
// Name: DDLoadBitmap
// Desc: Loads a bitmap file into memory and returns its handle
//
// PRE:
// 1. Parameter 1 is pointer to a DirectDraw surface
//    Parameter 2 is pointer to bitmap filename string
//
// POST:
// Returns handle to bitmap
//
// ERROR:
// All errors exit through DDInitFailed() function
//***************************************************************

HBITMAP DDLoadBitmap( LPDIRECTDRAWSURFACE4 lpDDS,
                      LPSTR szImage)
{
    HBITMAP         hbm;
    DDSURFACEDESC2  ddsd;

    ZeroMemory( &ddsd, sizeof( ddsd ) );
    ddsd.dwSize = sizeof( ddsd );

    if ( lpDDS->GetSurfaceDesc( &ddsd ) != DD_OK )
         DDInitFailed(hWnd, hRet,
         "GetSurfaceDesc() call failed in DDLoadBitmap()");

    // Test for compatible pixel format
    if ( ( ddsd.ddpfPixelFormat.dwFlags != DDPF_RGB ) ||
         ( ddsd.ddpfPixelFormat.dwRGBBitCount < 16 ) )
           DDInitFailed(hWnd, hRet,
          "Incompatible surface in DDLoadBitmap()");

    // Load the bitmap image onto memory
    hbm = ( HBITMAP )LoadImage( NULL, szImage,
            IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE );

    if ( hbm == NULL )
         DDInitFailed(hWnd, hRet,
          "Bitmap load failed in DDLoadBitmap()");

    return hbm;
}
```

Note that in DDLoadBitmap() all errors are considered terminal and directed through the DDInitFailed() function. This mode of operation can be changed if the code is to provide alternate processing in these cases.

29.4.2 Obtaining Bitmap Information

In order to display and manipulate a bitmap, the processing routines usually require information about its size and organization. The GDI GetObject() function is used for this purpose. This function fills a structure of type BITMAP, defined as follows:

```
typedef struct tagBITMAP {
   LONG   bmType;       // Must be zero
   LONG   bmWidth;      // bitmap width (in pixels)
   LONG   bmHeight;     // bitmap height (in pixels)
   LONG   bmWidthBytes; // bytes per scan line
   WORD   bmPlanes;     // number of color planes
   WORD   bmBitsPixel;  // bits per pixel color
   LPVOID bmBits;       // points to bitmap values array
} BITMAP;
```

The bmWidth member specifies the width, in pixels, of the bitmap, while bmHeight specifies the height, also in pixels. Both values must be greater than zero. The bmWidthBytes member specifies the number of bytes in each scan line. Windows assumes that the bitmap is word aligned; therefore, this value must be divisible by 2. The member bmPlanes specifies the number of color planes. The member bmBitsPixel specifies the number of bits required to indicate the color of a pixel. The member bmBits points to the location of the bit values for the bitmap. It is a long pointer to an array of char-size (1 byte) values.

How much of the information in the BITMAP structure is used depends on the type of bitmap processing performed by the application. The direct access operations, described earlier, allow code to manipulate bitmap data directly. In this case most of the BITMAP structure members are required in order to locate and access the bitmap data. On the other hand, applications can use high-level functions to display bitmap. Such is the case with the BitBlt() GDI function and the DirectDraw Blt() and BltFast() functions. When high-level functions are used, only the bmWidth and bmHeight members are usually necessary.

29.4.3 Moving a Bitmap to a Surface

Blit operations in DirectDraw take place between surfaces. Therefore, a useful function is one that loads a bitmap onto a surface. The function, named DDBmapToSurf(), copies a memory-resident bitmap, specified by its handle, into a DirectDraw surface.

```
//****************************************************************
// Name: DDBmapToSurf
// Desc: Moves a bitmap to a DirectDraw Surface
// PRE:
// 1. Parameter 1 is pointer to a IDirectDraw7 surface
//    Parameter 2 is handle to the bitmap
//
// POST:
// Bitmap is moved to surface
// Returns 1 if successful
// /
// ERROR:
// All errors exit through DDInitFailed() function
//****************************************************************
HRESULT DDBmapToSurf(LPDIRECTDRAWSURFACE7 pdds,
                     HBITMAP hbm)
{
    HDC                      hdcImage;
    HDC                      hdc;
    DDSURFACEDESC2           ddsd;
    HRESULT                  hr = 1;
    BOOL                     retValue;

    if (hbm == NULL || pdds == NULL)
     DDInitFailed(hWnd, hRet,
           "Invalid surface or bitmap in DDBmapToSurf");
```

```
   // Create compatible DC and select bitmap into it
   hdcImage = CreateCompatibleDC(NULL);
     if (!hdcImage)
       DDInitFailed(hWnd, hRet,
           "CreateCompatibleDC() failed in DDBmapToSurf");
   SelectObject(hdcImage, hbm);

   // Get size of surface
   ddsd.dwSize = sizeof(ddsd);
   ddsd.dwFlags = DDSD_HEIGHT | DDSD_WIDTH;
   pdds->GetSurfaceDesc(&ddsd);

   if ((hr = pdds->GetDC(&hdc)) != DD_OK)
     DDInitFailed(hWnd, hRet,
           "GetDC() failed in DDBmapToSurf");

   retValue = BitBlt(hdc, 0, 0, ddsd.dwWidth, ddsd.dwHeight,
                   hdcImage, 0, 0, SRCCOPY);
   // Release surface immediately
   pdds->ReleaseDC(hdc);

   if(retValue == FALSE)
       DDInitFailed(hWnd, hRet,
           "BitBlt() failed in DDBmapToSurf");

        DeleteDC(hdcImage);
    return hr;
}
```

29.4.4 Displaying the Bitmap

As previously mentioned, the BitBlt() GDI function provides a flexible, yet slow, mechanism for displaying bitmaps. In the case of a DirectDraw application, executing in exclusive mode, the device context must be obtained with the DirectDraw-specific version of the GetDC() function. IDirectDrawSurface7::GetDC not only returns a GDI-compatible device context, but also locks the surface for access. The following function displays a bitmap using a DirectDraw device context:

```
//****************************************************************
// Name: DDShowBitmap
// Desc: Displays a bitmap using a DirectDraw device context
//
// PRE:
// 1. Parameter 1 is pointer to a IDirectDraw7 surface
//    Parameter 2 is handle to the bitmap
//    Parameters 3 and 4 are the display location
//    Parameters 5 and 6 are the bitmap dimensions
//
// POST:
// Returns TRUE if successful
//
// ERROR:
// All errors exit through DDInitFailed() function
//****************************************************************

BOOL DDShowBitmap( LPDIRECTDRAWSURFACE7 lpDDS,
                   HBITMAP hBitmap,
                   int xLocation, int yLocation,
                   int bWidth, int bHeight)
```

```
{
    HDC                hdcImage = NULL;
    HDC                hdcSurf  = NULL;
    HDC                thisDevice = NULL;

    // Create a DC and select the image into it.
    hdcImage = CreateCompatibleDC( NULL );
    SelectObject( hdcImage, hBitmap);

    // Get a DC for the surface.
    if(lpDDS->GetDC(&hdcSurf) != DD_OK) {
        DeleteDC( hdcImage );
        DDInitFailed(hWnd, hRet,
            "GetDC() call failed in DDShowBitmap()");
        }

    // BitBlt() is used to display bitmap
    if ( BitBlt( hdcSurf, xLocation, yLocation, bWidth,
         bHeight, hdcImage, 0, 0, SRCCOPY ) == FALSE ) {
            lpDDS->ReleaseDC( hdcSurf );
            DeleteDC( hdcImage );
         // Take terminal error exit
         DDInitFailed(hWnd, hRet,
             "BitBlt() call failed in DDShowBitmap()");
         }
    // Release device contexts
       lpDDS->ReleaseDC( hdcSurf );
       DeleteDC( hdcImage );
       return TRUE;
}
```

The following code fragment shows the processing required for loading and displaying a bitmap onto the primary surface, as implemented in the project named DD Bmap Demo contained in the book's software package.

```
// Load bitmap named nebula.bmp
aBitmap = DDLoadBitmap(lpDDSPrimary, "nebula.bmp");

// Get bitmap data for displaying
GetObject(aBitmap, sizeof (BITMAP), &bMap1);

// Display bitmap
DDShowBitmap(lpDDSPrimary, aBitmap, 130, 50,
             (int) bMap1.bmWidth,
             (int) bMap1.bmHeight);
```

In Chapter 30 we examine bitmap rendering in greater detail and develop a DirectDraw windowed application that displays a bitmap.

29.5 DD Bitmap Blit Project

The DD Bitmap Blit project, in the book's software package, is a demonstration of the programming concepts and techniques discussed in this chapter. The program demonstrates the display of a bitmap on the primary surface, the creation and use of off-screen surfaces, and blitting bitmaps to and from off-screen surfaces.

Chapter 30

DirectDraw Bitmap Rendering

Topics:

- Loading a bitmap
- Obtaining bitmap data
- Moving the bitmap onto a surface
- Blitting the bitmap
- Developing a DirectDraw windowed application

Most graphics rendering consists of manipulating and displaying raster images. The color richness and high definition of today's graphics cards allow using bitmaps to encode images with photo-realistic accuracy. The hardware features of the graphics engines makes possible the effective manipulation of bitmapped images. DirectX provides a new level of functionality in bitmap processing and rendering. Applications that rely heavily on bitmaps include games, image processing, simulations, virtual reality, and artificial life.

This chapter is devoted to bitmap rendering in the context of a DirectDraw windowed application. In Chapter 16 we discuss manipulating and rendering bitmaps in exclusive mode.

30.1 Bitmap Manipulations

In DirectX bitmap manipulations consist of four basic steps:

- Loading the bitmap into memory
- Obtaining the bitmap data necessary for displaying it on the screen
- Moving the bitmap onto a surface
- Blitting the bitmap

30.1.1 Loading the Bitmap

In Chapter 29 you saw that loading a bitmap onto the application's memory space is an operation of GDI graphics. In the program DD Bitmap Blit, developed in Chapter 14 and contained in the book's software package, we load several bitmaps during WinMain() processing. The LoadBitmap() function is used to load the images into the application's memory space. Alternatively, instead of defining the bitmap as a program resource, we can use LoadImage() to load the bitmap directly from the disk file in which it is stored. At this time we can also perform certain preliminary checks to make sure that the DirectDraw surface is compatible with the bitmap to be displayed. Note that the sample code requires that the surface be nonpalletized. The GetSurfaceDesc() DirectDraw function is used to fill a DDSURFACEDESC2 structure. The DDPIXELFORMAT structure, which is part of DDSURFACEDESC2, contains two relevant values: the flag DDPF_RGB indicates that the RGB data is valid, and the dwRGBBitCount member contains the number of RGB bits per pixel. If the DDPF_RGB flag is set and dwRGBBitCount > 15 we can assume that the surface is nonpalletized, and therefore, compatible.

Note that the LoadImage() function does not return palette information. Microsoft Knowledge Base Article Q158898 lists the function LoadBitmapFromBMPFile() which uses the DIBSection's color table to create a palette. If no color table is present, then a half-tone palette is created. The source for this function can be found in the MSDN Library that is part of Visual C++.

Once code has determined that a compatible surface is available, it can proceed to load the bitmap. The general form of the LoadImage() function is as follows:

```
HANDLE LoadImage(
                  HINSTANCE hInst,                // 1
                  LPCTSTR lpszName,               // 2
                  UINT uType,                     // 3
                  int cxDesired,                  // 4
                  int cyDesired,                  // 5
                  UINT fuLoad                     // 6
                  );
```

The first parameter is a handle to an instance of the module that contains the image to be loaded. In the case of an image contained in a file, this parameter is set to zero.

The second parameter is a pointer to the image to load. If it is non NULL and the sixth parameter (described later) does not include LR_LOADFROMFILE, then it is a pointer to a null-terminated string that contains the filename of the image resource.

The third parameter is the image type. It can be one of the following constants:

- IMAGE_BITMAP
- IMAGE_CURSOR
- IMAGE_ICON

The fourth and fifth parameters specify the pixel width and height of the bitmap, cursor, or icon. If the fourth parameter is zero and the sixth parameter is

LR_DEFAULTSIZE, then the function uses the SM_CXICON or SM_CXCURSOR system metric value to set the width. If the fourth parameter is zero, and if LR_DEFAULTSIZE is present in the sixth parameter, then the function uses the actual width and height of the bitmap. The sixth and last parameter is one or more flags represented by the predefined constants listed in Table 30-1.

Table 30-1

Predefined Constants in LoadImage() Function

<table>
<tr><th>CONSTANT</th><th>MEANING</th></tr>
<tr><td>LR_DEFAULTCOLOR</td><td>Default flag. Does nothing.</td></tr>
<tr><td>LR_CREATEDIBSECTION</td><td>When the third parameter is IMAGE_BITMAP, this flag causes the function to return a DIB section bitmap rather than a compatible bitmap. It is useful for loading a bitmap without mapping it to the colors of the display device.</td></tr>
<tr><td>LR_DEFAULTSIZE</td><td>For cursor and icons the width or height values are those specified by the system metric values, but only if the fourth and fifth parameters are set to zero. If this flag is not specified and the fourth and fifth parameters are set to zero, the function uses the actual resource size.</td></tr>
<tr><td>LR_LOADFROMFILE</td><td>Loads the image from the file specified by the second parameter. If this flag is not specified, lpszName is the name of the resource.</td></tr>
<tr><td>LR_LOADMAP3DCOLORS</td><td>Searches the color table for the image and replaces the following shades of gray with the corresponding 3D color:
<table>
<tr><td>Color</td><td>RGB value</td><td>Replaced with</td></tr>
<tr><td>Dk Gray</td><td>RGB(128,128,128)</td><td>COLOR_3DSHADOW</td></tr>
<tr><td>Gray</td><td>RGB(192,192,192)</td><td>COLOR_3DFACE</td></tr>
<tr><td>Lt Gray</td><td>RGB(223,223,223)</td><td>COLOR_3DLIGHT</td></tr>
</table></td></tr>
<tr><td>LR_LOADTRANSPARENT</td><td>Retrieves the color value of the top-left pixel in the image and replaces the corresponding entry in the color table with the default window color (COLOR_WINDOW). All pixels in the image that Use that entry become the default window color. This value applies only to images that have corresponding color tables.</td></tr>
<tr><td>LR_MONOCHROME</td><td>Converts the image to black and white pixels.</td></tr>
<tr><td>LR_SHARED</td><td>Shares the image handle if the image is loaded multiple times. If LR_SHARED is not used, a second call to LoadImage for the same resource will load the image again and returns a different handle. LR_SHARED should not be used for images that Have nonstandard sizes, that may change after Loading, or that are loaded from a file. In Windows 95/98 LoadImage() finds the first image with the requested resource name in the cache, regardless of the size requested.</td></tr>
<tr><td>LR_VGACOLOR</td><td>Use true VGA colors.</td></tr>
</table>

LoadImage() returns the handle of the newly loaded image if the call succeeds. If the function fails, the return value is NULL. Although the system automatically deletes all resources when the process that created them terminates, applications can save memory by releasing resources that are no longer needed. DeleteObject() is used to release a bitmap, DestroyIcon() for icons, and DestroyCursor() for cursor resources.

The following function is used to load a bitmap into the application's memory space and obtain its handle. In this case the code checks for a surface compatible with a nonpalletized bitmap.

```
//***************************************************************
// Name: DDLoadBitmap
// Desc: Loads a bitmap file into memory and returns its handle
//
// PRE:
// 1. Parameter 1 is pointer to a DirectDraw surface
//    Parameter 2 is pointer to bitmap filename string
//
// POST:
// Returns handle to bitmap
//
// ERROR:
// All errors exit through DDInitFailed() function
//***************************************************************

HBITMAP DDLoadBitmap( LPDIRECTDRAWSURFACE7 lpDDS,
                      LPSTR szImage)
{
    HBITMAP         hbm;
    DDSURFACEDESC2  ddsd;

    ZeroMemory( &ddsd, sizeof( ddsd ) );
    ddsd.dwSize = sizeof( ddsd );

    if ( lpDDS->GetSurfaceDesc( &ddsd ) != DD_OK )
         DDInitFailed(hWnd, hRet,
         "GetSurfaceDesc() call failed in DDLoadBitmap()");

    // Test for compatible pixel format
    if ( ( ddsd.ddpfPixelFormat.dwFlags != DDPF_RGB ) ||
         ( ddsd.ddpfPixelFormat.dwRGBBitCount < 16 ) )
           DDInitFailed(hWnd, hRet,
          "Incompatible surface in DDLoadBitmap()");

    // Load the bitmap image onto memory
    hbm = ( HBITMAP )LoadImage( NULL, szImage,
            IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE );

    if ( hbm == NULL )
         DDInitFailed(hWnd, hRet,
          "Bitmap load failed in DDLoadBitmap()");

    return hbm;
}
```

Note that in DDLoadBitmap() all errors are considered terminal and directed through the DDInitFailed() function. This mode of operation can be changed if the code is to provide alternate processing in these cases.

30.1.2 Obtaining Bitmap Information

In order to display and manipulate a bitmap, the processing routines usually require information about its size and organization. The GDI GetObject() function is used for this purpose. The GetObject() function fills a structure of type BITMAP, defined as follows:

```
typedef struct tagBITMAP {
   LONG   bmType;       // Must be zero
   LONG   bmWidth;      // bitmap width (in pixels)
   LONG   bmHeight;     // bitmap height (in pixels)
   LONG   bmWidthBytes; // bytes per scan line
   WORD   bmPlanes;     // number of color planes
   WORD   bmBitsPixel;  // bits per pixel color
   LPVOID bmBits;       // points to bitmap values array
} BITMAP;
```

The bmWidth member specifies the width, in pixels, of the bitmap, while bmHeight specifies the height, also in pixels. Both values must be greater than zero. The bmWidthBytes member specifies the number of bytes in each scan line. Windows assumes that the bitmap is word aligned; therefore, this value must be divisible by 2. The member bmPlanes specifies the number of color planes. The member bmBitsPixel specifies the number of bits required to indicate the color of a pixel. The member bmBits points to the location of the bit values for the bitmap. It is a long pointer to an array of char-size (1 byte) values.

How much of the information in the BITMAP structure is used depends on the type of bitmap processing performed by the application. The direct access operations described in Chapter 13 allow code to manipulate bitmap data directly. If this is the case, then most of the BITMAP structure members are required in order to locate and access the bitmap data. On the other hand, applications can use high-level functions to display bitmap. Such is the case with the BitBlt() GDI function and the DirectDraw Blt() and BltFast() functions. When high-level functions are used, only the bmWidth and bmHeight members are usually necessary.

30.1.3 Moving a Bitmap onto a Surface

Blit operations in DirectDraw take place between surfaces. A useful function is one that loads a bitmap onto a surface. The local function, named DDBmapToSurf(), copies a memory-resident bitmap, specified by its handle, into a DirectDraw surface.

```
//*************************************************************
// Name: DDBmapToSurf
// Desc: Moves a bitmap to a DirectDraw Surface
// PRE:
// 1. Parameter 1 is pointer to a DirectDraw surface
//    Parameter 2 is handle to the bitmap
//
// POST:
// Bitmap is moved to surface
// Returns 1 if successful
// /
// ERROR:
// All errors exit through DDInitFailed() function
//****************************************************************

HRESULT DDBmapToSurf(LPDIRECTDRAWSURFACE7 pdds,
                     HBITMAP hbm)
{
    HDC                     hdcImage;
    HDC                     hdc;
    DDSURFACEDESC2          ddsd;
    HRESULT                 hr = 1;
    BOOL                    retValue;

    if (hbm == NULL || pdds == NULL)
     DDInitFailed(hWnd, hRet,
           "Invalid surface or bitmap in DDBmapToSurf");

   // Create compatible DC and select bitmap into it
   hdcImage = CreateCompatibleDC(NULL);
     if (!hdcImage)
       DDInitFailed(hWnd, hRet,
           "CreateCompatibleDC() failed in DDBmapToSurf");
   SelectObject(hdcImage, hbm);

   // Get size of surface
   ddsd.dwSize = sizeof(ddsd);
   ddsd.dwFlags = DDSD_HEIGHT | DDSD_WIDTH;
   pdds->GetSurfaceDesc(&ddsd);

   if ((hr = pdds->GetDC(&hdc)) != DD_OK)
     DDInitFailed(hWnd, hRet,
           "GetDC() failed in DDBmapToSurf");

   retValue = BitBlt(hdc, 0, 0, ddsd.dwWidth, ddsd.dwHeight,
                   hdcImage, 0, 0, SRCCOPY);
   // Release surface immediately
   pdds->ReleaseDC(hdc);

   if(retValue == FALSE)
       DDInitFailed(hWnd, hRet,
           "BitBlt() failed in DDBmapToSurf");

        DeleteDC(hdcImage);
    return hr;
}
```

30.1.4 Displaying the Bitmap

We have mentioned that the BitBlt() GDI function provides a flexible, yet slow, mechanism for displaying bitmaps. In the case of a DirectDraw application, executing in exclusive mode, the device context must be obtained with the DirectDraw-specific version of the GetDC() function. IDirectDrawSurface7::GetDC not only returns a GDI-compatible device context, but also locks the surface for access. The following local function displays a bitmap using a DirectDraw device context:

```
//****************************************************************
// Name: DDShowBitmap
// Desc: Displays a bitmap using a DirectDraw device context
//
// PRE:
// 1. Parameter 1 is pointer to a DirectDraw surface
//    Parameter 2 is handle to the bitmap
//    Parameters 3 and 4 are the display location
//    Parameters 5 and 6 are the bitmap dimensions
//
// POST:
// Returns TRUE if successful
//
// ERROR:
// All errors exit through DDInitFailed() function
//****************************************************************

BOOL DDShowBitmap( LPDIRECTDRAWSURFACE7 lpDDS,
           HBITMAP hBitmap,
           int xLocation,
           int yLocation,
           int bWidth,
           int bHeight)
{

    HDC              hdcImage = NULL;
    HDC              hdcSurf  = NULL;
    HDC              thisDevice = NULL;

    // Create a DC and select the image into it.
    hdcImage = CreateCompatibleDC( NULL );
    SelectObject( hdcImage, hBitmap);

    // Get a DC for the surface.
    if(lpDDS->GetDC(&hdcSurf) != DD_OK) {
        DeleteDC( hdcImage );
        DDInitFailed(hWnd, hRet,
            "GetDC() call failed in DDShowBitmap()");
        }

    // BitBlt() is used to display bitmap
    if ( BitBlt( hdcSurf, xLocation, yLocation, bWidth,
         bHeight, hdcImage, 0, 0, SRCCOPY ) == FALSE ) {
            lpDDS->ReleaseDC( hdcSurf );
            DeleteDC( hdcImage );
         // Take terminal error exit
         DDInitFailed(hWnd, hRet,
             "BitBlt() call failed in DDShowBitmap()");
         }
    // Release device contexts
       lpDDS->ReleaseDC( hdcSurf );
       DeleteDC( hdcImage );
       return TRUE;
}
```

The following code fragment shows the processing required for loading and displaying a bitmap onto the primary surface, as implemented in the project named DD Bitmap In Window located in the Chapter 15 folder in the book's software package.

```
// Load bitmap named hubble.bmp
aBitmap = DDLoadBitmap(lpDDSPrimary, "hubble.bmp");

// Get bitmap data for displaying
GetObject(aBitmap, sizeof (BITMAP), &bMap1);

// Display bitmap
DDShowBitmap(lpDDSPrimary, aBitmap, 130, 50,
                (int) bMap1.bmWidth,
                (int) bMap1.bmHeight);
```

30.2 Developing a Windowed Application

Exclusive mode provides the maximum power and functionality of DirectDraw. For this reason most DirectDraw applications execute in exclusive mode. But this does not preclude conventional windows programs from using DirectDraw functions in order to obtain considerable gains in performance and to perform image manipulations that are not possible in the GDI.

Running in a window usually means that the program can be totally or partially obscured by another program, that it can lose focus, that surfaces may be unbound from their memory assignments, and that the application window can be minimized or resized by the user. Most of these circumstances, which are often ignored in exclusive mode, require careful attention in windowed DirectDraw. In other words, DirectDraw programming in windowed mode restores most of the device independence that is lost in exclusive mode, which means that windowed DirectDraw code must use the conventional multitasking, message based, paradigm that is characteristic of Windows. The following are the main differences between DirectDraw programs in exclusive and non-exclusive mode:

- Exclusive mode applications usually require window style WS_POPUP, while windowed application use WS_THICKFRAME if they are resizeable. The combination WS_SYSMENU, WS_CAPTION, and WS_MINIMIZEBOX is used if the window cannot be resized by the user. WS_OVERLAPPEDWINDOW style includes WS_THICKFRAME.
- Exclusive mode programs use DDSCL_FULLSCREEN and DDSCL_EXCLUSIVE cooperative level, while windowed programs use DDSCL_NORMAL.
- Exclusive mode programs can use page flipping in implementing animation (animation techniques are covered in Chapter 16), while windowed programs have very limited flipping capabilities. This is one of the reasons why games and other animation-intensive applications usually execute in exclusive mode.
- Full-screen programs can set their own display mode, while windowed programs must operate in the current desktop display mode. By the same token, exclusive mode programs can assume a particular display mode, while windowed programs must be designed with sufficient flexibility to execute in several display modes.
- Exclusive mode applications may use clipping to produce specific graphics effects. Windowed programs often rely on clipping to facilitate interaction with other programs and with the Windows desktop.

- Exclusive mode programs can be switched to the background, but usually they cannot be minimized or resized by the user. Windowed programs can be moved on the desktop, resized, minimized, or obscured by other applications.
- Exclusive mode programs have direct control over the palette and can be designed for a particular palette. Windowed programs must use the palette manager to make changes and must accommodate palette changes made by the user or by other programs.
- Exclusive mode programs can display or hide the system cursor but cannot use system-level mouse support, as is the case with the system menu or by the buttons on the program's title bar.
- Exclusive mode programs must furnish most of the cursor processing logic. On the other hand, DirectDraw windowed applications can make use of all the cursor and cursor-related support functions in the Windows API.
- Exclusive mode applications must implement their own menus. Windowed applications can use the menu facilities in the API.

In summary, although windowed programs must address some specific issues in using DirectDraw services, they do have almost unrestricted access to the functionality of a conventional application. Thus, a DirectDraw program that executes in a windowed mode can have a title bar, resizeable borders, menus, status bar, sizing grip, scroll bars, as well as most of the other GUI components. Although there is no "standard" design for a DirectDraw windowed application, there are issues that are usually confronted by a typical DirectDraw application when executing in windowed mode. In the following sections we discuss the most important ones.

30.2.1 Windowed Mode Initialization

A DirectDraw windowed program can execute with so many variations that it is difficult to design a general template for it. The same abundance of options applies to the initialization of a windowed application. However, there are certain typical initialization steps for DirectDraw windowed applications. The project named DD WinMode Template, in the book's software package, contains a template file with minimal initializations for a DirectDraw application in windowed mode.

The first step in WinMain() processing is defining and filling the WNDCLASSEX structure variable and registering the window class. In the template file this is accomplished as follows:

```
// Defining a structure of type WNDCLASSEX
WNDCLASSEX  wndclass ;
wndclass.cbSize        = sizeof (wndclass) ;
wndclass.style         = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc   = WndProc ;
wndclass.cbClsExtra    = 0 ;
wndclass.cbWndExtra    = 0 ;
wndclass.hInstance     = hInstance ;
wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
wndclass.hbrBackground = (HBRUSH) GetStockObject
                         (WHITE_BRUSH) ;
wndclass.lpszMenuName  = szAppName;
wndclass.lpszClassName = szAppName;
```

```
 wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;
// Register the class
RegisterClassEx(&wndclass);
```

Next, the code creates the window and defines its show state. In the case of a resizeable window with the three conventional buttons and the system menu box in the title bar we can use the WS_OVERLAPPEDWINDOW style. Since it is impossible to predict in the template the window size and initial location, we have used CW_USEDEFAULT for these parameters.

```
hWnd = CreateWindowEx(0,        // Extended style
         szAppName,
         "DirectDraw Nonexclusive Mode Template",
         WS_OVERLAPPEDWINDOW,
         CW_USEDEFAULT,
         CW_USEDEFAULT,
         CW_USEDEFAULT,
         CW_USEDEFAULT,
         NULL,            // Handle of parent
         NULL,            // Handle to menu
         hInstance,       // Application instance
         NULL);           // Additional data
if (!hWnd)
    return FALSE;
ShowWindow(hWnd, nCmdShow);
```

The processing for creating a DirectDraw object and a primary surface is similar to that used in exclusive mode programming. In the template we use the same support procedures previously developed. DD7Interface() attempts to find a DirectDraw7 object and returns 1 if found and 0 if not. If the call is successful, a global pointer variable named lpDD4, of type LPDIRECTDRAW7, is initialized. DDInitFailed() provides a terminal exit for failed initialization operations. The primary surface is created by means of a call to CreateSurface(). The surface pointer is stored in the public variable lpDDSPrimary. Code is as follows:

```
     //**************************************
     // Create DirectDraw object and
     // create primary surface
     //**************************************
    // Fetch DirectDraw7 interface
     hRet = DD7Interface();
     if (hRet == 0)
         return DDInitFailed(hWnd, hRet,
                             "QueryInterface() call failed");
    // Set cooperative level to exclusive and full screen
    hRet = lpDD7->SetCooperativeLevel(hWnd, DDSCL_NORMAL);
    if (hRet != DD_OK)
        return DDInitFailed(hWnd, hRet,
                            "SetCooperativeLevel() call failed");
    //**********************************
     //    Create the primary surface
     //**********************************
     // ddsd is a structure of type DDSRUFACEDESC2
    ZeroMemory(&ddsd, sizeof(ddsd));  // Fill structure with zeros
     // Fill in other members
    ddsd.dwSize = sizeof(ddsd);
    ddsd.dwFlags = DDSD_CAPS ;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
    hRet = lpDD7->CreateSurface(&ddsd, &lpDDSPrimary, NULL);
    if (hRet != DD_OK)
        return DDInitFailed(hWnd, hRet,
                             "CreateSurface() call failed");
```

30.2.2 Clipping the Primary Surface

Clipping is the DirectDraw operation by which output is limited to a rectangular area, usually defined in a surface. DirectDraw supports clipping in both exclusive and windowed modes. Since exclusive mode applications have control over the entire client area, clipping is used mostly as a graphics output manipulation. Windowed applications, on the other hand, often share the display with the Windows desktop and with other applications. In this case clipping is often used to ensure that the application's output is limited to its own client area. Color Figure 7 shows the clipped execution of two copies of a DirectDraw application on the Windows desktop. The application is the DD Bitmap In Window program developed earlier in this chapter.

A clipper is used to define the program's screen boundaries in a DirectDraw windowed application. The clipper ensures that a graphics object is not displayed outside the client area. Failure to define a clipper may cause the blit operation to fail because the destination drawing surface could be the limits of the display surface. When the boundaries of the primary surface are defined in a clipper, then DirectDraw knows not to display outside of this area and the blit operation succeeds, as is the case in Color Figure 7. Recall that the Blt() function supports clipping but that BltFast() does not.

Pixel coordinates are stored in one or more structures of type RECT in the clip list. DirectDraw uses the clipper object to manage clip lists. Clip lists can be attached to any surface by using a DirectDrawClipper object. The simplest clip list consists of a single rectangle which defines the area within the surface to which a Blt() function outputs. Figure 30-1 shows a DirectDraw surface with an attached clipper consisting of a single rectangle.

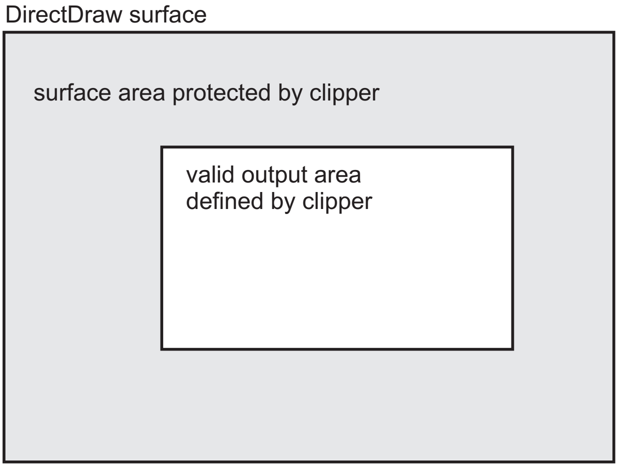


Figure 30-1 *Using a Clipper to Establish the Surface's Valid Blit Area.*

DirectDraw's Blt() function copies data to the rectangles in the clip list only. Clip lists consisting of several rectangles are often necessary in order to protect a specific surface area from output. For example, if an application requires a rectangular area in the top-center of the screen to be protected from output, it would need to define several clipping rectangles. Figure 30-2 shows this case.

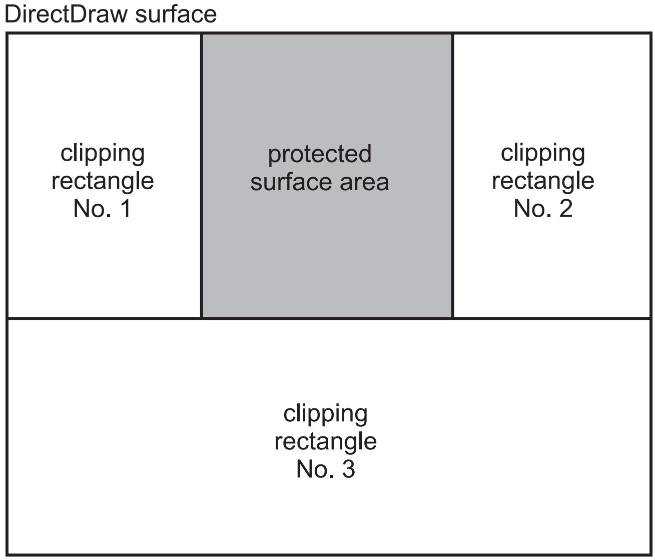


Figure 30-2 *Multiple Clipping Rectangles*

To manage a clip list, application code creates a series of rectangles and stores them in a data structure of type RGNDATA (region data), described later in this section. One of the members of RGNDATA is the RGNDATAHEADER structure, which is used to define the number of rectangles that make up the region. The function SetClipList() is called with the RGNDATA structure variable as a parameter. The SetClipList() function has the following general form:

The IDirectDrawClipper::SetClipList method sets or deletes the clip list used by the IDirectDrawSurface7::Blt, IDirectDrawSurface7::BltBatch, and IDirectDrawSurface7::UpdateOverlay methods on surfaces to which the parent DirectDrawClipper object is attached.

```
HRESULT SetClipList(
              LPRGNDATA lpClipList,          // 1
              DWORD dwFlags                  // 2
              );
```

The first parameter is the address of a valid RGNDATA structure or NULL. If there is an existing clip list associated with the DirectDrawClipper object and this value is NULL, the clip list is deleted.

The second parameter is currently not used and must be set to 0.

The function returns DD_OK if it succeeds, or one of the following error codes:

- DDERR_CLIPPERISUSINGHWND
- DDERR_INVALIDCLIPLIST
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_OUTOFMEMORY

The RGNDATA structure used with this method has the following syntax:

```
typedef struct _RGNDATA {
    RGNDATAHEADER rdh;
    char          Buffer[1];
} RGNDATA;
```

The third member of the RGNDATA structure is an RGNDATAHEADER structure that has the following syntax:

```
typedef struct _RGNDATAHEADER {
    DWORD dwSize;
    DWORD iType;
    DWORD nCount;
    DWORD nRgnSize;
    RECT  rcBound;
} RGNDATAHEADER;
```

To delete a clip list from a surface, the SetClipList() call is made using NULL for the RGNDATA parameter.

DirectDraw can automatically manage the clip list for a primary surface. Attaching a clipper to the primary surface requires several steps. In the first place, a clipper is a DirectDraw object in itself, which must be created using the DirectDraw7 interface object. The CreateClipper() function is used in this step. The function's general form is as follows:

```
HRESULT CreateClipper(
              DWORD dwFlags,                              // 1
              LPDIRECTDRAWCLIPPER FAR *lplpDDClipper,     // 2
              IInknown FAR *pUnkOuter                     // 3
              );
```

The first and third parameters are not used in current implementations: the first one should be set to zero and the third one to NULL. The second parameter is the address of a variable of type LPDIRECTDRAWCLIPPER which is set to the interface if the call succeeds; in this case the return value is DD_OK. If the call fails it returns one of the following constants:

- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_NOCOOPERATIVELEVELSET
- DDERR_OUTOFMEMORY

Once the clipper to the primary surface is created, it must be attached to the application's window. This requires a call to the SetHWnd() function. The function's general form is as follows:

```
HRESULT SetHWnd(
          DWORD dwFlags,            // 1
          HWND hWnd                 // 2
          );
```

The first parameter must be set to zero in the current implementation.

The second parameter is the handle to the window that uses the clipper object. This has the effect of setting the clipping region to the client area of the window and ensuring that the clip list is automatically updated as the window is resized, covered, or uncovered. Once a clipper is set to a window, additional rectangles cannot be added.

The clipper must be associated with the primary surface. This is done by means of a call to the IDirectDrawSurface7::SetClipper function, which has the following general form:

```
HRESULT SetClipper(
  LPDIRECTDRAWCLIPPER lpDDClipper      // 1
);
```

The function's only parameter is the address of the IDirectDrawClipper interface for the DirectDrawClipper object to be attached to the DirectDrawSurface object. If NULL, the current DirectDrawClipper object is detached.

SetClipper() returns DD_OK if it succeeds, or one of the following error codes:

- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_INVALIDSURFACETYPE
- DDERR_NOCLIPPERATTACHED

When a clipper is set to a surface for the first time, the call to SetClipper() increments the reference count. Subsequent calls do not affect the clipper's reference count. If you pass NULL as the lpDDClipper parameter, the clipper is removed from the surface, and the clipper's reference count is decremented. If you do not delete the clipper, the surface automatically releases its reference to the clipper when the surface itself is released. The application is responsible for releasing any references that it holds to the clipper when the object is no longer needed, according to the COM rules.

The SetClipper() function is primarily used by surfaces that are being overlaid, or surfaces that are blitted to the primary surface. However, it can be used on any surface.

```
The code in the template program is as follows:
//************************************
//    Create a clipper
//************************************
hRet = lpDD7->CreateClipper(0, &lpDDClipper, NULL);
if (hRet != DD_OK)
       return DDInitFailed(hWnd, hRet,
                              "Create clipper failed");
```

```
// Associate clipper with application window
hRet = lpDDClipper->SetHWnd(0, hWnd);
if (hRet != DD_OK)
        return DDInitFailed(hWnd, hRet,
                             "Clipper not linked to application window");
// Associate clipper with primary surface
hRet = lpDDSPrimary->SetClipper(lpDDClipper);
   if (hRet != DD_OK)
         return DDInitFailed(hWnd, hRet,
                             "Clipper not linked to primary surface");
```

30.3 Rendering in Windowed Mode

A simple rendering scheme in DirectDraw windowed mode programming consists of storing a bitmap in an offscreen surface and then blitting it to the primary surface. It is in the blitting stage that the windowed nature of the application introduces some constraints. The DirectDraw interface allows the program to access video memory directly, while the windowed nature of the application requires that video output be limited to the application's client area. A terminal error occurs if a windowed program attempts to display outside its own space. In GDI programming Windows takes care of clipping video output. In DirectDraw programming these restrictions must be observed and enforced by the application itself.

The most powerful rendering function for DirectDraw windowed applications is Blt(). Figure 14-7 shows some of the controls and options available in this case. DirectDraw windowed applications that use Blt() often create a destination surface clipper, and manipulate the size and position of the source and destination rectangles in order to achieve the desired effects. The BltFast() function can be used in cases that do not require clippers or other output controls that are available in Blt().

30.3.1 Rendering by Clipping

The project named DD Bitmap In Window, in the book's software package, contains two versions. Both versions display a bitmap of the Orion nebula images obtained by the Hubble Space Telescope. The first program version corresponds to the source file named DD Bitmap In Window.cpp. In this case the bitmap image is blitted to the entire primary surface and a clipper is used to restrict which portion of the image is displayed in the application's window. Color Figure 8 shows the original bitmap stretched to fill the primary surface.

The clipper, which in this case is the size of the application window, is attached to the primary surface. Color Figure 7 shows two copies of the DD Bitmap In Window program on the desktop. Each executing copy of the program displays the underlaying portion of a virtual image according to the clipper, which is automatically resized by Windows to the application's client area. This ensures that video output is limited to the application's video space.

During initialization, the wndclass.style member of the WNDCLASSEX structure is set to CS_HREDRAW and CS_VREDRAW so that the entire client area is redrawn if there is vertical or horizontal resizing. The program design calls for creating an initial application window of the same size as the bitmap. In order to obtain the bitmap dimensions, the code must load the bitmap into memory before creating the application window. The processing is as follows:

```
// Global handles and structures for bitmaps
HBITMAP      aBitmap;
BITMAP       bMap1;          // Structures for bitmap data
. . .
// Local data
RECT         progWin;        // Application window dimensions
//**************************************
//     Load bitmap into memory
 //**************************************
// Load the bitmap image into memory
aBitmap = ( HBITMAP )LoadImage( NULL, "nebula.bmp",
            IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE );
 if ( aBitmap == NULL )
      DDInitFailed(hWnd, hRet,
         "Bitmap load failed in DDLoadBitmap()");

// Get bitmap data
GetObject(aBitmap, sizeof (BITMAP), &bMap1);
// Store bitmap in RECT structure variable
progWin.left = 0;
progWin.top = 0;
progWin.right = bMap1.bmWidth;
progWin.bottom = bMap1.bmHeight;
```

The bitmap dimensions are now stored in a structure of type RECT, with the variable name progWin. But the application window is larger than the client area, since it includes the title bar and the border. It is necessary to adjust the size by calling AdjustWindowRectEx(). This function corrects the data stored in a RECT structure variable according to the application's window style. Once the size has been adjusted, code can proceed to create the window, as follows:

```
//*****************************************
//   Create a window with client area
//   the same size as the bitmap
//*****************************************
// First adjust the size of the client area to the size
// of the bounding rectangle (this includes the border,
// caption bar, menu, etc.)
AdjustWindowRectEx(&progWin,
                    WS_OVERLAPPEDWINDOW,
                    FALSE,
                    0);

hWnd = CreateWindowEx(0,        // Extended style
             szAppName,
             "DD Bitmap In Window",
             WS_OVERLAPPEDWINDOW,
             CW_USEDEFAULT,    // x of initial position
             CW_USEDEFAULT,    // y of initial position
             (progWin.right - progWin.left),  // x size
             (progWin.bottom - progWin.top),  // y size
             NULL,            // Handle of parent
             NULL,            // Handle to menu
             hInstance,       // Application instance
             NULL);           // Additional data
```

```
   if (!hWnd)
        return FALSE;
    ShowWindow(hWnd, nCmdShow);
```

In the call to CreateWindowEx() we used the default initial position and arbitrarily set the Windows dimension to that of the bitmap, the size of which is stored in the progWin structure variables. The code now proceeds to create a DirectDraw object and a primary surface in the conventional manner. Note that the cooperative level in this case is DDSCL_NORMAL.

```
     //*************************************
     // Create DirectDraw object and
     // create primary surface
     //*************************************
    // Fetch DirectDraw7 interface
     hRet = DD7Interface();
     if (hRet == 0)
          return DDInitFailed(hWnd, hRet,
                              "QueryInterface() call failed");

    // Set cooperative level to exclusive and full screen
    hRet = lpDD7->SetCooperativeLevel(hWnd, DDSCL_NORMAL);
    if (hRet != DD_OK)
        return DDInitFailed(hWnd, hRet,
                              "SetCooperativeLevel() call failed");
     // ddsd is a structure of type DDSRUFACEDESC2
    ZeroMemory(&ddsd, sizeof(ddsd));  // Fill structure with zeros
     // Fill in other members
    ddsd.dwSize = sizeof(ddsd);
    ddsd.dwFlags = DDSD_CAPS ;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

    hRet = lpDD7->CreateSurface(&ddsd, &lpDDSPrimary, NULL);
    if (hRet != DD_OK)
        return DDInitFailed(hWnd, hRet,
                              "CreateSurface() call failed");
```

It is now time to create a clipper associated with the application window and attach it to the primary surface, as described previously in this chapter. The surface element tells DirectDraw which surface to clip. The window element defines the clipping rectangle to the size of the application's client area. The processing is as follows:

```
   //**********************************
     //    Create a clipper
     //**********************************
    hRet = lpDD7->CreateClipper(0, &lpDDClipper, NULL);
    if (hRet != DD_OK)
        return DDInitFailed(hWnd, hRet,
                              "Create clipper failed");
     // Associate clipper with application window
     hRet = lpDDClipper->SetHWnd(0, hWnd);
    if (hRet != DD_OK)
          return DDInitFailed(hWnd, hRet,
                              "Clipper not linked to application window");
    // Associate clipper with primary surface
     hRet = lpDDSPrimary->SetClipper(lpDDClipper);
    if (hRet != DD_OK)
```

```
        return DDInitFailed(hWnd, hRet,
                        "Clipper not linked to primary surface");
```

Although the bitmap has been loaded, it has not yet been stored in an offscreen surface. Blt() requires that the bitmap be located on a surface, so this must be the next step. Since speed is not a factor in this program, we create the surface in system memory. This allows running several copies of the program simultaneously. The code is as follows:

```
//*************************************
//   Store bitmap in off screen surface
//*************************************
// First create an off-screen surface
// in system memory
ZeroMemory(&ddsd, sizeof(ddsd));  // Fill structure with zeros
// Fill in other members
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
                      DDSCAPS_SYSTEMMEMORY;
ddsd.dwHeight = bMap1.bmHeight;
ddsd.dwWidth = bMap1.bmWidth;
hRet = lpDD7->CreateSurface(&ddsd, &lpDDSOffscreen, NULL);
if (hRet != DD_OK)
    return DDInitFailed(hWnd, hRet,
           "Off Screen surface creation failed");
 // Move bitmap to surface using DDBmapToSurf()function
hRet = DDBmapToSurf(lpDDSOffscreen, aBitmap);
 if(hRet != DD_OK)
    return DDInitFailed(hWnd, hRet,
                     "DDBMapToSurf() call failed");
// ASSERT:
//     Bitmap is in offscreen surface -> lpDDSOffscreen
```

Finally, the bitmap stored in the offscreen surface can be blitted to the primary surface using the clipper attached to the primary surface. The Blt() call is as follows:

```
//************************************
//      Blit the bitmap
//************************************
// Update the window with the new sprite frame. Note that the
// destination rectangle is our client rectangle, not the
// entire primary surface.
 hRet = lpDDSPrimary->Blt( NULL, lpDDSOffscreen, NULL,
                 DDBLT_WAIT, NULL );
        if(hRet != DD_OK)
           return DDInitFailed(hWnd, hRet,
                 "Blt() failed");
```

Since the window is resizeable, we must also provide processing in the WM_PAINT message intercept. However, WM_PAINT is first called when the window is created; at this time the application has not yet performed the necessary initialization operations. To avoid a possible conflict we create a public switch variable, named DDOn, which is not set until the application is completely initialized. Another consideration is that the call to BeginPaint(), often included in WM_PAINT processing, automatically sets the clipping region to the application's update region. Since we are providing our own clipping, the call to BeginPaint() is undesirable. In the sample program WM_PAINT message processing is as follows:

```
   case WM_PAINT:
       if(DDOn)
          hRet = lpDDSPrimary->Blt( NULL, lpDDSOffscreen, NULL,
                   DDBLT_WAIT, NULL );
       return 0;
```

30.3.2 Blit-Time Cropping

In the preceding section we saw the first variation of the DD Bitmap In Window program. In this case the bitmap image is stretch-blitted to the entire primary surface. A clipper that was previously attached to the primary surface automatically restricts which portion of the image is displayed in the application's window. As you move the application window on the desktop, or resize it, a different portion of the bitmap becomes visible.

An alternative option, which produces entirely different results, is blitting to a destination rectangle in the primary surface which corresponds to the size of the application's client area. Because the destination of the blit is restricted to the client area there is no need for a clipper in this case, since the output is cropped by the Blt() function. Figure 30-3 graphically shows the basic operation of the two versions of the DD InWin Demo program.

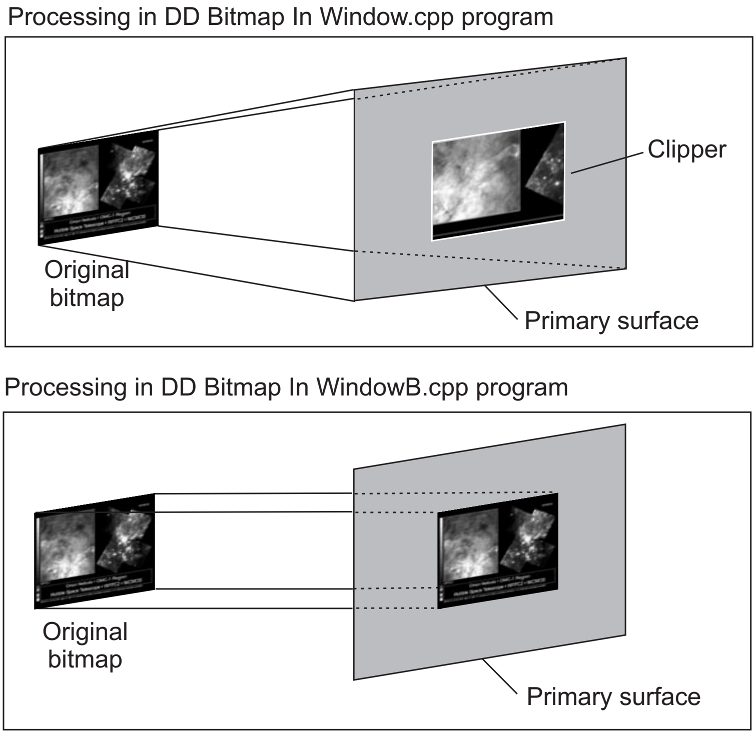

Processing in DD Bitmap In Window.cpp program
Clipper
Original bitmap
Primary surface
Processing in DD Bitmap In WindowB.cpp program
Original bitmap
Primary surface

Figure 30-3 *Comparing the Two Versions of the DD Bitmap In Window Program*

In the version DD Bitmap In WindowB.cpp the code proceeds as follows: The WNDCLASSEX structure is defined similarly as in the first version of the sample program except that, since the program window is not resizeable, the CS_HREDRAW and CS_VREDRAW window style constants are not necessary. The fixed size of the program window also determines that the code uses WS_SYSMENU, WS_CAPTION, and WS_MINIMIZEBOX as the window style constants in both AdjustWindowRectEx() and CreateWindowEx() functions. Note that a resizeable window requires the WS_THICKFRAME or WS_SIZEBOX styles. Also note that the WS_OVERLAPPEDWINDOW style, used in the first version of the sample program, includes WS_THICKFRAME and therefore also produces a resizeable window.

In the version DD Bitmap In WindowB.cpp the program window is made the same size as the original bitmap, as is the case in the first version. In the first version the size of the display area is arbitrary, since the program window is resizaeable. In the second version the bitmap is displayed identically as is it stored. Therefore, the display area must match the size of the bitmap.

Much of the initialization and setup of the second version of the program is similar to the first one. The bitmap is loaded into memory and its size is stored in the corresponding members of a RECT structure variable. The DirectDraw7 object and the primary surface are created. In this case the clipper is not attached to the primary surface since it is not used. Then the off screen surface is created and the bitmap is stored in it. Code is now ready to blit the bitmap from the offscreen surface to the primary surface, but before the blit can take place it is necessary to determine the screen location and the size of the application's client area. It is also necessary to define the destination rectangle, which is the first parameter of the Blt() function. One way to visualize the problem is to realize that, at this point, the program window is already displayed, with a blank rectangle on its client area, which is the same size as the bitmap. Also that the primary surface is the entire screen. Figure 30-4 shows the application at this stage and the dimensions necessary for locating the client area on the primary surface.

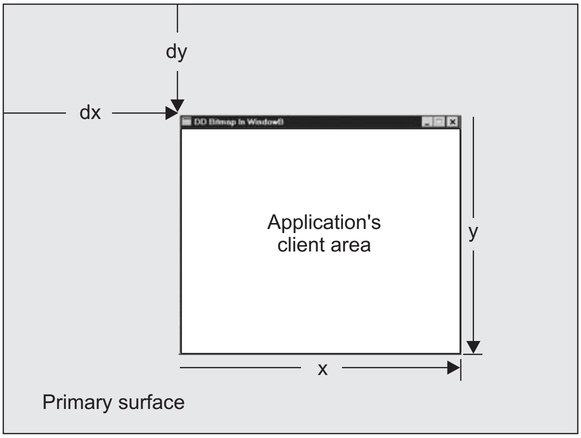


Figure 30-4 *Locating the Blt() Destination Rectangle*

The GetClientRect() API function returns the coordinates of the client area of a window. The function parameters are the handle of the target window and the address of a variable of type RECT which holds the client area dimensions. The values returned by GetClientRect() correspond to the x and y dimensions shown in Figure 30-5. Since the coordinates are relative to the application's window, the value returned by the call for the upper-left corner of the rectangle is always (0,0). This makes the left and top members of the RECT structure variable passed to the call always zero. Since you need the location of application's window in the primary surface, the code must determine the values labeled dx and dy in Figure 30-5 and add them to the coordinates stored in the RECT structure.

The ClientToScreen() function performs this operation. Its parameters are the handle to the application's window and the address of a structure of type POINT containing two coordinate values that are to be updated to screen coordinates. ClientToScreen() actually performs an addition operation on the coordinate pair: it calculates the distances labeled dx and dy in Figure 30-5 and adds these values to those stored in the structure variable. Since the POINT structure contains two members of type long, and the RECT structure contains four members, you can consider that the RECT structure member holds two structures of type POINT. The code in the sample program is as follows:

```
RECT            clientArea;           // For Blt() destination
. . .
// Obtain client rectangle and convert to screen coordinates
GetClientRect(hWnd, &clientArea);
ClientToScreen(hWnd, (LPPOINT) &clientArea.left);
ClientToScreen(hWnd, (LPPOINT) &clientArea.right);

// Blit to the destination rectangle
hRet = lpDDSPrimary->Blt( &clientArea, lpDDSOffscreen, NULL,
                          DDBLT_WAIT, NULL );
  if(hRet != DD_OK)
     return DDInitFailed(hWnd, hRet,
                    "Blt() failed");
```

Notice that the ClientToScreen() function is called twice. First, for the coordinate pair that holds the top-left corner of the client area rectangle; these are the zero values returned by GetClientRect(). Then, for the coordinate pair of the bottom-right corner of the client area rectangle, which correspond to the x and y dimensions in Figure 30-4. Similar processing must be performed in the WM_PAINT message intercept.

The project folder DD Bitmap In Window, in the Chapter 30 folder of the book's software package, contains two versions of the source program.

Chapter 31

DirectDraw Animation

Topics:

- Animation in real time
- Preventing surface tearing
- Obtaining a timed pulse
- Sprite animation
- Flipping techniques
- Multiple buffering
- Improving performance

This chapter is about real-time computer animation using the DirectDraw facility in DirectX. Before DirectX, animation in Windows was considered somewhat of an oxymoron. DirectX provides mechanisms that make possible graphics rendering at a high speed. One of these mechanisms is the storage of image data in so-called back buffers. The back buffers and the display surface can be rapidly flipped to simulate screen movement. The results are often a smooth and natural simulation of movement that can be used in computer games, simulations, and in high-performance applications.

Palette animation techniques were popular and effective in DOS programming, but the resolution and color depth of state-of-the-art video systems makes them unnecessary. Overlays, although powerful and useful, were never well defined and are supported inconsistently in the video hardware. Since overlay operations are not emulated in the HEL, they can only be used if implemented in the hardware. For these reasons neither palette animation nor overlays are discussed.

31.1 Animating in Real-Time

Computer animation is defined as the simulation of movement or lifelike actions by the manipulation of digital objects. It is a complex field on which many books have been written. Here we are concerned with real-time animation, rather than with computer-assisted techniques. Real-time animation is found in arcade machines, simulators, trainers, electronic games, multimedia applications, and in interactive programs of many kinds. In real-time animation the computing machine is both the image generator, and the display media.

Real-time animation is possible because of the physiology of the human eye. In our vision system, a phenomena is called visual retention makes the image of an object persist in the brain for a brief period of time after it no longer exists. Smooth animation is achieved by consecutively displaying images at a faster rate than our period of visual retention. The sequence of rapidly displayed images creates in our minds the illusion a moving object.

Motion picture technology uses an update rate of 24 images per second. Television is based on a slightly faster rate. In animation programming the number of images displayed in a time period is called the frame rate. The threshold rate, which is subject variations in different individuals, is that at which the animation begins to appear bumpy or jerky. In motion picture technology the threshold is about 17 images per second. In computer animations the threshold rate is considerably higher.

While the animator's principal concerns are usually speed and performance, too much speed can lead to image quality deterioration. A raster scan display system is based on scanning each horizontal row of screen pixels with an electron beam. The pixel rows are refreshed starting at the top-left screen corner of the screen and ending at the bottom-right corner, as shown in Figure 1.2. The electron beam is turned off at the end of each scan line, while the gun is re-aimed to the start of the next one. This period is called the horizontal retrace. When this process reaches the last scan line on the screen, the beam is turned off again while the gun is re-aimed to the top-left screen corner. The period of time required to re-aim the electron gun from the right-bottom of the screen to the top-left corner is known as the vertical retrace or screen blanking cycle. The fact that a CPU is capable of executing hundreds of thousands of instructions per second makes it possiblefor the image in video memory to be modified before the video system has finished displaying it. The result is a breaking of the image, known as tearing.

31.1.1 The Animator's Predicament

Computer animation is a battle against time. The animation programmer resorts to every possible trick in order to squeeze the maximum performance. Because execution speed is limited by the hardware, most of the work of the programmer-animator consists of making compromises and finding acceptable levels of undesirable effects. The animator often has to decide how small an image satisfactorily depicts the object, how much tearing is acceptable, how much bumpiness can be allowed in depicting movement, how little definition is sufficient for a certain scenery, or with how few colors can an object be realistically represented.

31.2 Timed Pulse Animation

Representing movement requires a display sequence, executed frame-by-frame, that creates the illusion of motion. Figure 31-1 shows several frames in the animation of a stick figure of a walking person.

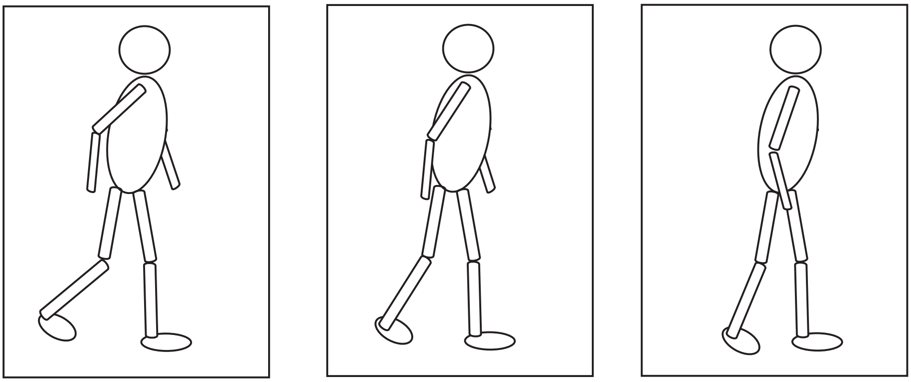


Figure 31-1 *Stick Figure Animation*

The real-time display of the frame-by-frame sequence requires a mechanism for producing a timed pulse. Windows applications have several ways of generating a timed pulse. One is based on a program loop that reads the value in a ticker register and proceeds to update the frame whenever it matches or exceeds a predefined constant. A second and more effective approach is to enable a system timer pulse, which can be intercepted in a callback function or by a window message. In the following sections we discuss both methods. Other alternatives, sometimes called high-resolution timers, are discussed in the context of performance tuning, later in this chapter.

31.2.1 The Tick Counting Method

Windows maintains a counter with the number of milliseconds elapsed since the system was started. This period, called the Windows time, is stored in a DWORD variable that can be read by code. Two identical functions allow reading this counter: GetCurrentTime() and GetTickCount(). Windows documentation states that GetCurrentTime() is now obsolete and should not be used. GetTickCount(), which takes no parameters, returns the number of milliseconds elapsed since Windows was started. Application code can determine the number of milliseconds elapsed since the last call by storing the previous value in a static or public variable, as in the following function:

```
// Public variables for counter operation
DWORD         thisTickCount;     // New ticker value
DWORD         lastTickCount;     // Storage for old value
static DWORD  TIMER_VALUE = 25;  // Constant for time lapse
.
.
.
static void UpdateFrame()
{
  thisTickCount = GetTickCount();   // Read counter
     if(( thisTickCount - lastTickCount) < TIMER_VALUE)
        return;
     else
     {
     // Frame update operations go here
```

```
        lastTickCount = thisTickCount;   // Reset tick counts
      }
  return;
}
```

In order for the ticker counter reading method to produce a smooth animation, the value in the ticker counter must be polled frequently. One possible approach is to include the frame update function call as part of the application's message loop. The processing logic can be expressed as follows: If the application is active, and no other messages are waiting to be processed, then call the frame update routine.

The PeekMessage() function checks the thread's message queue without pausing for a message to be received. The function's general form is as follows:

```
BOOL PeekMessage(
                 LPMSG lpMsg,          // 1
                 HWND hWnd,            // 2
                 UINT wMsgFilterMin,   // 3
                 UINT wMsgFilterMax,   // 4
                 UINT wRemoveMsg       // 5
                 );
```

The first parameter points to an MSG structure variable that contains message information.

The second parameter is the handle to the window whose messages are being checked. This parameter can be set to NULL to check messages for the current application.

The third and the fourth parameters are used to specify the lowest and highest value of the messages to be checked. If both parameters are set to 0, then all messages are retrieved.

The fifth parameter is one of two predefined constants: PM_REMOVE is used if the message is to be removed from the queue, and PM_NOPREMOVE otherwise. The call returns TRUE if a message is available, and FALSE if not available.

Another API function often used in message polling routines is WaitMessage(). This function, which takes no parameters, suspends thread execution and does not return until a new message is placed in the queue. The result is to yield control to other threads when the current one has nothing to do with its CPU cycles. PeekMessage() and WaitMessage() can be combined with GetMessage() in the following message polling routine:

```
MSG          msg;           // Message structure variable
int          appActive = 0; // Application active switch
                            // initialized to inactive
.
.
.
while(1)
  {
  if(PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE)) {
    if(!GetMessage(&msg, NULL, 0, 0)
      return msg.wParam;
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  else if (appActive)
  {
  // call to read ticker counter and/or update frame
  // go here
  }
  else
      WaitMessage();
  }
```

In using this sample code the application must define when to set and reset the appActive swtich. This switch determines if the frame update function is called, or if the thread just waits for another message. The method just described, that is, reading the Windows tick count inside a program loop, is usually capable of generating a faster pulse than the system timer intercept, described in the following section. On the other hand the system timer intercept is easier to implement and more consistent with the Windows multitasking environment. Therefore, the system timer intercept method is generally preferred.

31.2.2 System Timer Intercept

An alternative way of obtaining a timed pulse is by means of the Windows system timer. The SetTimer() function is used to define a time-out value, in milliseconds. When this time-out value elapses, the application gets control either at the WM_TIMER message intercept or in an application-defined callback function that has the generic name TimerProc(). Either processing is satisfactory and which one is selected is a matter of coding convenience. SetTimer() has the following general form:

```
UINT SetTimer(
              HWND hWnd,              // 1
              UINT nIDEvent,          // 2
              UINT uElapse,           // 3
              TIMERPROC lpTimerFunc   // 4
              );
```

The first parameter is the handle to the Window associated with the timer.

The second parameter is the timer number. This allows more than one timer per application. The timer identifier is passed to the WM_TIMER intercept and to the TimerProc().

The third parameter is the number of milliseconds between timer intercepts.

The fourth parameter is the address of the application's TimerProc(), if one is implemented, or NULL if processing is to be done in the WM_TIMER message intercept.

If the call succeeds, the return value is an integer identifying the new timer. Sixteen timers are available to applications, so it is a good idea to check if a timer is actually assigned to the thread. Applications must pass this timer identifier to the KillTimer() function to destroy a particular timer. If the function fails to create a timer, the return value is zero. Once a system timer has been initialized, processing usually consists of calling the application's frame update function directly, since the timer tick need not be checked in this case.

Notice that code cannot assume that system timer events will be generated at the requested rate. The only valid assumption is that the events will be produced approximately at this rate, and not more frequently. According to the Windows documentation, the minimum time between events is approximately 55 milliseconds.

31.3 Sprites

A sprite is a rather small screen object, usually animated at display time. Sprites find use in general graphics programming, but most frequently in games. Sprite animation can be simple or complex. In the first case an object represented in a single bitmap is animated by translating it to other screen positions. Alternatively, the sprite itself can perform an intrinsic action, for example, a sprite representing a rotating wheel. In complex animation both actions are performed simultaneously: a rocket moves on the screen until it reaches a point where it explodes. Sprites are typically encoded in one or more images that represent the object or its action sequence. The images can be stored in separate bitmaps, or in a single one. Figure 31-2 shows the image set of a a Pacman-like sprite.

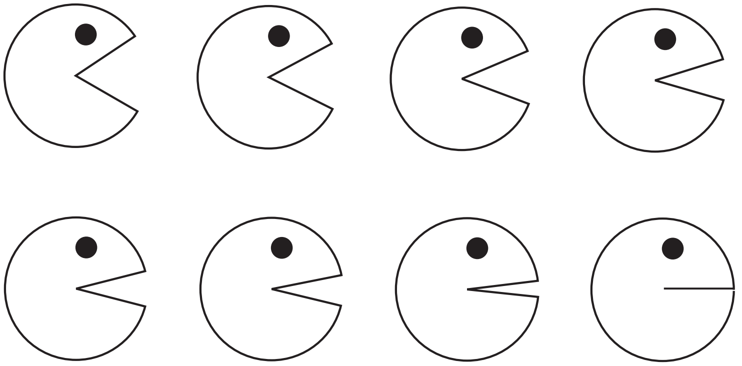


Figure 31-2 *Animation Image Set*

When the eight images in the set of Figure 31-2 are rapidly displayed, the Pacman-like sprite appears to close its mouth. If the image set is then re-displayed in reverse order, the mouth will appear to open. The 16-image sequence simulates a biting action. If the Pacman-like sprite is also moved on the screen, then the results would be a of complex sprite animation.

Sprites often use color keys in order to achieve transparency. To automate sprite source color keying some applications assume that the pixel at the top-left corner of the bitmap is the color key. Later in this chapter we discuss the use of dynamic color keys.

It is possible to encode each image of the sprite image set in separate bitmaps, but this usually leads to many disk files and complicated file access operations. A better approach is to store the entire sprite image set in a single bitmap, and then use source rectangle selection capability of either the Blt() or BltFast() functions to pick the corresponding image.

Many factors determine how a sprite is actually rendered. One of the most important ones is if the application executes in exclusive mode or windowed. Exclusive mode programs can use back buffers and flipping manipulations that considerably increase performance, while windowed programs are much more limited in the available rendering options. Other factors are the sprite's size, the number of images in the set, and the required rendering speed. Programmers often have to juggle these and other program elements in order to come up with a satisfactory animation.

31.3.2 Creating Sprites

Animated programs spend considerable resources in manipulating sprites and backgrounds. The better the image quality of these objects, the better graphics that result. Backgrounds are usually animated by panning and zooming transformations, discussed later in this chapter. In this case the programmer's effort is limited to creating a few, relatively large images. But sprites are a more complicated matter, specially if the sprite is to have internal action. In the case of sprites the individual images in the set must be tied to a common point. Perhaps the most important factor in creating good sprites is the sprite itself. For some time the creation of attractive sprites was considered some sort of black art. 3D graphics makes it possible to create solid sprites that add a new dimension to the animation.

The animator often spends a large part of his time in designing, drawing, encoding, and testing sprites. This is particularly true in 3D graphics. The process of sprite design implies several apparently contradictory decisions, for instance:

- The higher the resolution the better the image quality, but it is more difficult to animate a larger sprite.
- The more images in the sprite image set the smoother the animation, but it takes longer to display a large sprite image set.

The details of how the sprite image sets are produced is more in the realm of graphics design than in programming. The higher the quality of the drawing or paint program used, and the more experienced and talented the sprite artist, the better the resulting image set. The DD Sprite Animation project, included in the book's software package, shows two rotating, meshed gears. The image set consists of 18 images. In each image the gears are rotated by an angle of 2.5 degrees. After 18 iterations the gears have rotated through an angle of 45 degrees. Since the gears have eight teeth each, the images are symmetrical after a rotation of 45 degrees. For this reason this animation requires one-eighth the number of images that would be necessary to rotate a non-symmetrical object by the same angle. Figure 31-3 shows the image set for the DD Animation Demo program.

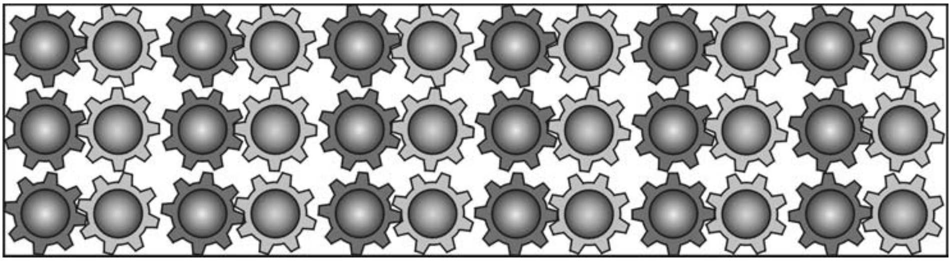


Figure 31-3 *The Sprite Image Set for the DD Sprite Animation Program*

31.3.3 Sprite Rendering

The actual display of the sprite requires obtaining a timing pulse and blitting the image onto the screen. In each case we must decide whether the rendering is done with Blt() or BltFast(), with or without transparency, using source or destination color keys, or applying any blit-time transformations. The sprite image sequence is usually stored in a single bitmap, as in Figure 31-3, but it is also possible to store several bitmaps in different disk files and then read all of these files into a single surface. In either case the result is a surface with multiple images. The program logic selects the corresponding portion of the bitmap at blit time.

Displaying partial images stored in a contiguous memory area or surface is made possible by the source area definition capabilities of both Blt() and BltFast(). A structure of type RECT can be used to store the offset of the source rectangle in the surface. If the sprite image set is stored in a rectangular bitmap, and the bitmap is then loaded onto a surface, code can then select which of the images in the set is displayed during each time-pulse iteration by assigning values to the corresponding members of the RECT structure. For example, the bitmap image set in Figure 31-3 contains a sequence of 18 individual rotations of the gears. Each of these individual bitmaps is often called an animation frame, or simply, a frame. Figure 31-4 shows the image set partitioned into six rows and three columns. The dimensions labeled x and y refer to the size of each frame in the set.

Given the pixel size of each image in the set, once the number of rows and columns in the image set are known, code can determine the coordinates of the RECT structure variable for each frame. The dotted rectangle in Figure 31-4 delimits each frame. The members of a structure variable named rect, of type RECT, are calculated using the consecutive frame number and the number of columns in the bitmap. The case illustrated shows frame number 8, of a bitmap with six columns and three rows.

In the DD Sprite Animation program, the processing has been generalized so that the code can be used to display any rectangular bitmap image set. This makes it useful for experimenting with various image sets before deciding which one is better suited for the purpose at hand. Code starts by creating global variables that define the characteristics of the image set. Code is as follows:

```
// Constants identifying the bitmap image set
static char bmapName[] = {"gears.bmp"};
static int         imageCols = 6;      // Number of image columns
static int         imageRows = 3;      // Number of rows

// Variables, constants, handles, and structure for bitmaps
int         frameCount = (imageCols * imageRows) - 1;
int         bmapXSize;      // Calculated x size of bitmap
int         bmapYSize;      // Calculated y size of bitmap
HBITMAP     aBitmap;
BITMAP      bMap1;          // Structures for bitmap data
```

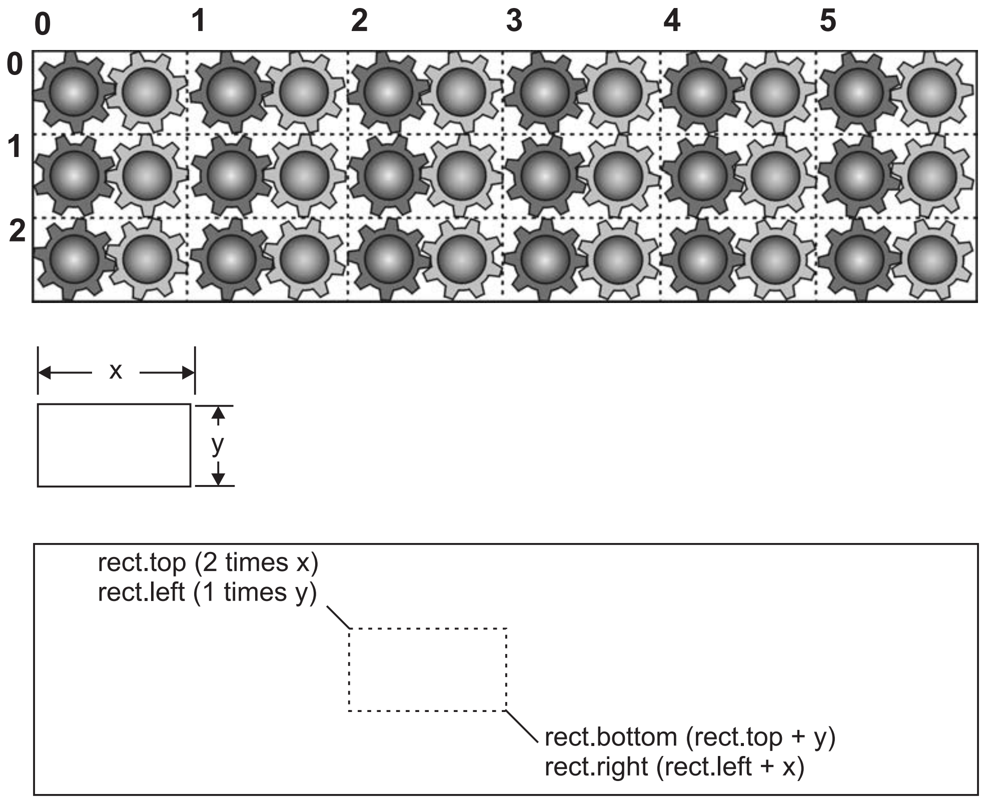


Figure 31-4 *Partitioning the Sprite Image Set*

In this case the programmer defines the name of the bitmap and states the number of image columns and rows. Code uses these values to calculate to number of frames; this number is stored in the variable frameCount. The dimensions of the bitmap are obtained after it is loaded into memory. The x dimension is stored in the variable bmapXSize and the y dimension in bmapYSize. The bitmap dimensions are also used in the sample program to define the size of the application window, all of which is shown in the following code fragment

```
//****************************************
//     Load bitmap into memory
//****************************************
// Load the bitmap image into memory
aBitmap = ( HBITMAP )LoadImage( NULL, bmapName,
          IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE );
```

```
   if ( aBitmap == NULL )
        DDInitFailed(hWnd, hRet,
         "Bitmap load failed in DDLoadBitmap()");
  // Get bitmap data
  GetObject(aBitmap, sizeof (BITMAP), &bMap1);

  // Calculate and store bitmap and image data
  bmapXSize = bMap1.bmWidth / imageCols;
  bmapYSize = bMap1.bmHeight / imageRows;

  // Store bitmap in RECT structure variable
  progWin.left = 0;
  progWin.top = 0;
  progWin.right = bmapXSize;
  progWin.bottom = bmapYSize;

  //*******************************************
  //   Create window with client area
  //   the same size as the bitmap
  //*******************************************
  // First adjust the size of the client area to the size
  // of the bounding rectangle (this includes the border,
  // caption bar, menu, etc.)

   AdjustWindowRectEx(&progWin,
             WS_SYSMENU | WS_CAPTION,
             FALSE,
             0);

   hWnd = CreateWindowEx(0,        // Extended style
             szAppName,
             "Sprite Animation Demo",
             WS_SYSMENU | WS_CAPTION,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             (progWin.right - progWin.left),
             (progWin.bottom - progWin.top),
             NULL,            // Handle of parent
             NULL,            // Handle to menu
             hInstance,       // Application instance
             NULL);           // Additional data
    if (!hWnd)
        return FALSE;
```

The actual display of the bitmap is performed by a local function named BlitSprite(). The function begins by checking the tick counter. If the difference between the old and the new tick counts is smaller than the predefined delay, execution returns immediately. If it is equal to or larger than the delay, then the offset of the next frame in the source surface is calculated and the bitmap is blitted by means of the Blt() function. In this case the frame number counter is bumped; if this is the last frame in the set, the counter is restarted. Execution concludes by updating the tick counter variable. Coding is as follows:

```
   //*****************************************
   //      update animation frame
   //*****************************************
   static void BlitSprite()
   {
      thisTickCount = GetTickCount();
      if((thisTickCount - lastTickCount) < TIMER_VALUE)
        return;
      else
      {
       // Update the sprite image with the current frame.
       bmapArea.top = ( ( frameNum / imageCols) * bmapYSize );
       bmapArea.left = ( ( frameNum % imageCols) * bmapXSize );
       bmapArea.bottom = bmapArea.top + bmapYSize;
       bmapArea.right = bmapArea.left + bmapXSize;

       hRet = lpDDSPrimary->Blt( &clientArea, lpDDSOffscreen,
              &bmapArea, DDBLT_WAIT, NULL );
           if(hRet != DD_OK)
                DDInitFailed(hWnd, hRet, "Blt() failed");
       // Update the frame counter
       frameNum++;
       if(frameNum > imageCount)
          frameNum = 0;
       lastTickCount = thisTickCount;
       return;
      }
   }
```

Color Figure 9 is a screen snapshot of the DD Animation Demo program.

31.4 Page Flipping

Page flipping is a rendering technique frequently used in multimedia applications, simulations, and computer games. The process is reminiscent of the schoolhouse method of drawing a series of images, each consecutive one containing a slight change. The figures are drawn on a paper pad. By thumbing through the package you perceive an illusion of movement. In the simplest version of computerized page flipping the programmer sets up two DirectDraw surfaces. The first one is the conventional primary surface and the other one is back buffer. Code updates the image in the back buffer and then flips the back buffer and the primary surface. The result is usually a clean and efficient animation effect. Figure 31-5, on the following page, shows the sprite animation by page flipping.

In Figure 31-5 we see that consecutive images in the animation set are moved from the image set onto the back buffer. The back buffer is then flipped with the primary surface. In this illustration the arrows represent the flip operations. The back buffers are shown in dark gray rectangles. The sequence of operations is: draw to back buffer, flip, draw to back buffer, flip, and so on.

One limitation of multiple buffering and page flipping is that it can only be used in DirectDraw exclusive mode. This is because flipping requires manipulating video memory directly, which is not possible in a windowed environment. In the DirectDraw flip operation it is the pointers to surface memory for the primary surface and the back buffers that are swapped. In other words, flipping is actually performed by switching pointers, not by physically copying data. By exception, when the back buffer cannot fit into display memory, or when the hardware does not support flipping, DirectDraw performs the flip by copying the surfaces.

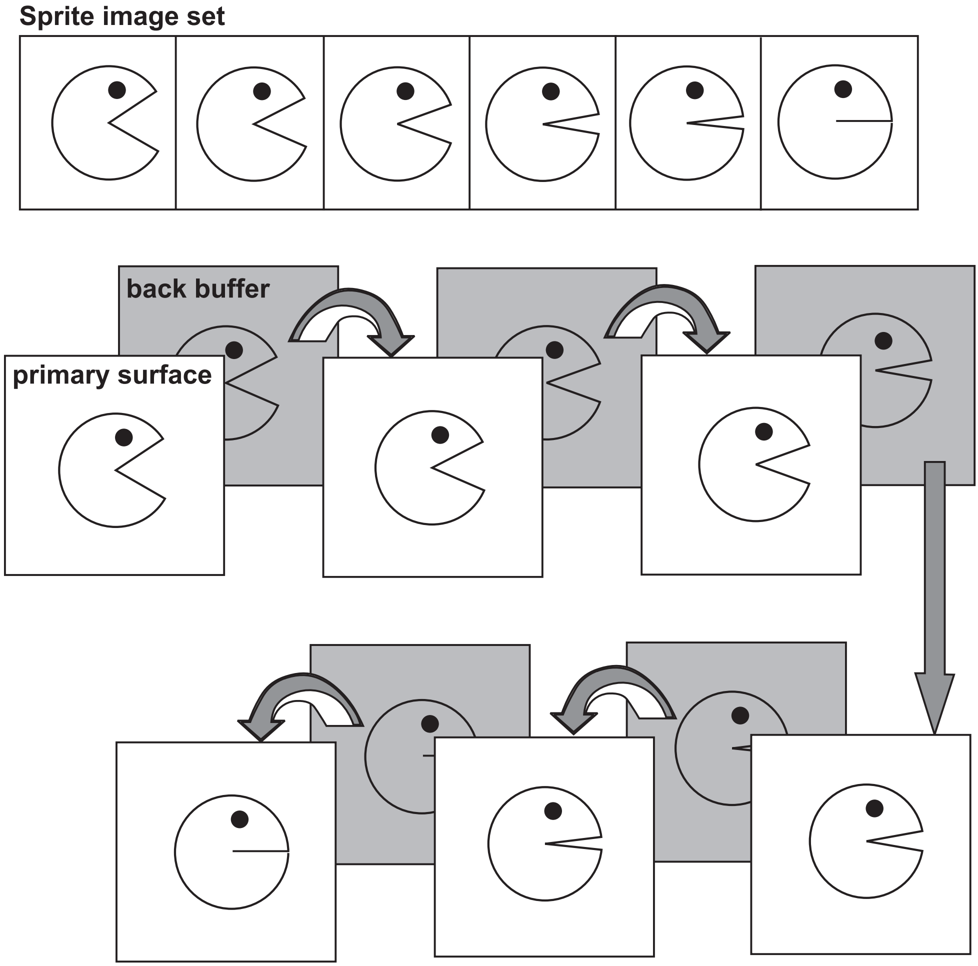


Figure 31-5 *Sprite Animation by Page Flipping*

In programming a flip-based animation you should keep in mind that code need only access the back buffer surface in order to perform the image updates. Every time the DirectDraw Flip() function is called, the primary surface becomes the back buffer and vice versa. The surface pointer to the back buffer always points to the area of video memory not displayed, and the surface pointer to the primary surface points to the video memory being displayed. If more than one back buffer is included in the flipping chain, then the surfaces are rotated in circular fashion. The case of a flipping chain with a primary surface and two back buffers is shown in Figure 31-6. In this case the flip operation rotates the surfaces as shown.

Initializing and performing flip animation consists of several well-defined steps. In most cases the following operations are necessary:

- Creating a flipping chain.

- Obtaining a back buffer pointer.
- Drawing to the back buffer.
- Flipping the primary surface and the back buffer.

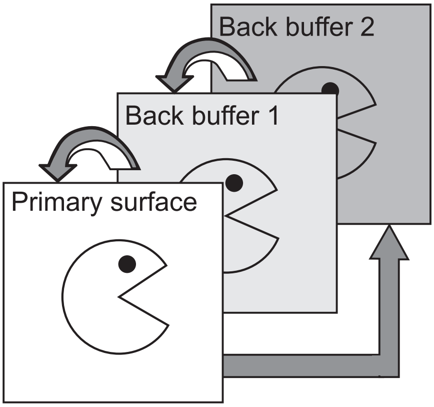


Figure 31-6 *Flipping Chain with Two Back Buffers*

The first two steps of this sequence relate to initializing the flipping surfaces, and the second two steps refer to flip animation rendering operations. Table 31-1 lists the flipping-related functions in DirectDraw.

Table 31-1

Flipping-Related DirectDraw Functions

FUNCTION	OBJECT	ACTION
CreateSurface()	IDIRECTDRAW7	Create surface and Attached back buffers.
GetAttachedSurface()	IDIRECTDRAWSURFACE7	Obtain back buffer pointer.
Flip()	IDIRECTDRAW7	Perform flipping.
GetFlipStatus()	IDIRECDRAWSURFACE7	Indicates whether a surface has concluded flipping.

The function FlipToGDISurface(), which is rarely used in practice, is not discussed in this book.

31.4.1 Flipping Surface Initialization

Any DirectDraw surface can be constructed as a flipping surface, although most commonly the flipping surfaces consist of a primary surface and at least one back buffer. The surfaces involved in the flipping are called the flipping chain. Creating the flipping chair requires two DirectDraw functions: CreateSurface() is used to create both the primary surface and the back buffer, and GetAttachedSurface() to obtain the back buffer pointer. In the case of a flipping chain, the call to CreateSurface() must include the flag DDSD_BACKBUFFERCOUNT, which defines the member dwBackBufferCount, which in turn is used to set the number of back buffers in the chain. Other flags usually listed in the call are DDSCAPS_PRIMARYSURFACE, DDSCAPS_FLIP, DDSCAPS_COMPLEX, and DDSCAPS_VIDEOMEMORY. The following code shows a call to CreateSurface() for a flipping chain consisting of a primary surface and a single back buffer:

```
  DDSURFACEDESC2      ddsd;
  .
  .
  .
 // Create the primary surface with a back buffer
    ZeroMemory(&ddsd, sizeof(ddsd));  // Fill structure with zeros
 // Fill in other members
    ddsd.dwSize = sizeof(ddsd);
    ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                          DDSCAPS_FLIP |
                          DDSCAPS_COMPLEX |
                          DDSCAPS_VIDEOMEMORY;
    ddsd.dwBackBufferCount = 1;
    hRet = lpDD7->CreateSurface(&ddsd, &lpDDSPrimary, NULL);
```

If the call to CreateSurface() returns DD_OK, then the flipping chain surfaces have been created. In order to use the flipping chain, code must first obtain the pointer to the back buffer, since the call to CreateSurface() returns only the pointer to the primary surface (in its second parameter). The GetAttachedSurface() function has the following general form:

```
HRESULT GetAttachedSurface(
           LPDDSCAPS lpDDSCaps,                                // 1
           LPDIRECTDRAWSURFACE7 FAR *lplpDDAttachedSurface     // 2
           );
```

The first parameter is a pointer to a DDSCAPS2 structure that contains the hardware capabilities of the surface.

The second parameter is the address of a variable that is to hold the pointer, of type IDIRECTDRAWSURFACE7, retrieved by the call. The retrieved surface matches the description in the first parameter. If the function succeeds, it returns DD_OK. If it fails it returns one of the following errors:

- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_NOTFOUND
- DDERR_SURFACELOST

The following code fragment obtains the back buffer surface pointer for the primary surface previously described.

```
  // Get back buffer pointer
     ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
     hRet = lpDDSPrimary->GetAttachedSurface(&ddscaps,
                          &lpDDSBackBuf);
```

If the calls to CreateSurface() and GetAttachedSurface() are successful, DirectDraw creates two attached surfaces in display memory, and the application retrieves the pointers to each of these surfaces. The pointer to the back buffer surface is used at draw time, and the pointer to the primary surface at flip time. DirectDraw automatically switches the surface pointers, transparently to application code.

31.4.2 The Flip() Function

Once the application has concluded drawing, and the frame timer count has expired, the actual rendering is performed by calling DirectDraw Flip(). The Flip() function exchanges the surface memory of the primary surface and the back buffer. If more than one back buffer is specified when the flip chain is created, then each call to Flip() rotates the surfaces in a circular manner, as shown in Figure 31-6. When DirectDraw flipping is supported by the hardware, as is the case in most current video cards, flipping consists of changing pointers and no image data is physically moved. The function's general form is as follows:

```
HRESULT Flip(
      LPDIRECTDRAWSURFACE7 lpDDSurfaceTargetOverride,     // 1
      DWORD dwFlags                                       // 2
      );
```

The first parameter, sometimes called the target override, is the address of the IDirectDrawSurface7 interface for any surface in the flipping chain. The default value for this parameter is NULL, in which case DirectDraw cycles through the flip chain surfaces in the order they are attached to each other. If this parameter is not NULL, then DirectDraw flips to the specified surface instead of the next surface in the flipping chain, thus overriding the default order. The call fails if the specified surface is not a member of the flipping chain.

The second parameter specifies one of the predefined constants that control flip options. The constants are listed in Table 31-2.

Table 31-2

DirectDraw Flip() Function Flags

FLAG	ACTION
DDFLIP_EVEN	Used only when displaying video in an overlay surface. The new surface contains data from the even field of a video signal. Cannot be used with the DDFLIP_ODD flag.
DDFLIP_INTERVAL2 DDFLIP_INTERVAL3 DDFLIP_INTERVAL4	Indicate how many vertical retraces to wait between each flip. The default is 1. DirectDraw returns DERR_WASSTILLDRAWING until the specified number of vertical retraces has occurred. If DDFLIP_INTERVAL2 is set, DirectDraw flips on every second vertical retrace cycle. If DDFLIP_INTERVAL3, on every cycle, and so on.These flags are effective only if DDCAPS2_FLIPINTERVAL is set in the DDCAPS structure for the device.

(continued)

Table 31-2

DirectDraw Flip() Function Flags (continued)

FLAG	ACTION
DDFLIP_NOVSYNC	DirectDraw performs the physical flip as close as possible to the next scan line. Subsequent operations involving the two flipped surfaces do not check to see if the physical flip has finished, that is, they do not return DDERR_WASSTILLDRAWING. This flag allows an application to perform flips at a higher frequency than the monitor refresh rate. The usual consequence is the introduction of visible artifacts. If DDCAPS2_FLIPNOVSYNC is not set in the DDCAPSstructure for the device, DDFLIP_NOVSYNC has no effect.
DDFLIP_ODD	Used only when displaying video in an overlay surface. The new surface contains data from the odd field of a video signal. This flag cannot be used with the DDFLIP_EVEN flag.
DDFLIP_WAIT	If the flip cannot be set up because the state of the display hardware is not appropriate, then the DDERR_WASSTILLDRAWING is immediately returned and no flip occurs. Setting this flag causes Flip() to continue trying if it receives the DDERR_WASSTILLDRAWING. In this case the call does not return until the flipping operation has been successfully set up, or another error, such as DDERR_SURFACEBUSY, is returned.

If the Flip() call succeeds, the return value is DD_OK. If it fails, one of the following errors is returned:

- DDERR_GENERIC
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_NOFLIPHW
- DDERR_NOTFLIPPABLE
- DDERR_SURFACEBUSY
- DDERR_SURFACELOST
- DDERR_UNSUPPORTED
- DDERR_WASSTILLDRAWING

The Flip() function can be called only for surfaces that have the DDSCAPS_FLIP and DDSCAPS_FRONTBUFFER capabilities. The first parameter is used in rare cases, that is, when the back buffer is not the buffer that should become the front buffer. In most cases this parameter is set to NULL. In its default state, the Flip() function is always synchronized with the vertical retrace cycle of the video controller. When working with visible surfaces, such as a primary surface flipping chain, Flip() function is asynchronous, except if the DDFLIP_WAIT flag is included.

Applications should check if Flip() returns with a DDERR_SURFACELOST. If so, code can make an attempt to restore the surface by means of the DirectDraw Restore() function, discussed in Chapter 15. If the restore is successful, the application loops back to the Flip() call and tries again. If the restore is unsuccessful, the application breaks from the while loop, and returns a terminal error.

31.4.3 Multiple Buffering

The call to Flip() can return before the actual flip operation takes place, because the hardware waits until the next vertical retrace to actually flip the surfaces. While the Flip() operation is pending, the back buffer directly behind the currently visible surface cannot be locked or blitted to. If code attempts to call Lock(), Blt(), BltFast(), or GetDC() while a flip is pending, the call fails and the function returns DDERR-_WASSTILLDRAWING. The effect of the surface update time on the frame rate is shown in Figure 31-7.

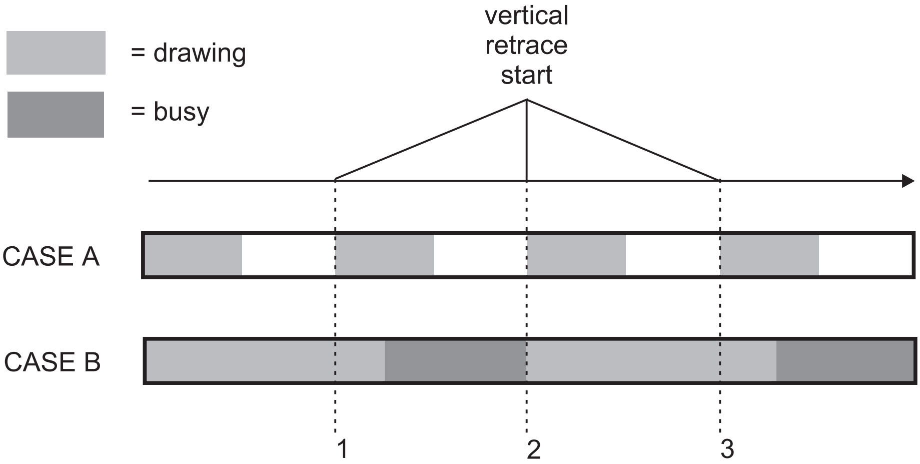


Figure 31-7 *Surface Update Time and Frame Rate*

Case A in Figure 31-7 shows an application with a relatively short surface update time. In this case the rendering is finished well before the next vertical retrace cycle starts. The result is that the image is updated at the monitor's refresh rate. In Case B the rendering time is longer than the refresh cycle. In this case, the application's frame rate is reduced to one-half the monitor's refresh rate. In such situations, any attempt to access the back buffer during the period represented by the dark gray rectangles results in DDERR_WASSTRILLDRAWING error message.

A possible way of improving the frame rate is by using two back buffers instead of a single one. With two back buffers the application can draw to the back buffer that is not immediately behind the primary surface, thereby reducing the wasted time since the blits to this rearmost surface are not subject to the DDERR_WASSTILLDRAWING error condition. This is shown in case B in Figure 31-7.

Creating a flipping chain with two or more back buffers is no great programming complication. DirectDraw takes care of flipping the surfaces and the application draws using the same back buffer pointer. The middle buffer, or buffers, are ignored by the code, which only sees the primary surface and a back buffer. The one drawback of multiple buffering is that each back buffer requires as much display memory as the primary surface. Also the law of diminishing returns applies to back buffers: the more back buffers the less increase in performance for each back buffer. Past a certain limit, adding more back buffers will actually degrade performance.

Exclusive mode applications are sometimes forced to select lower resolutions or color depths in order to make possible multiple back buffers. For example, in a video system with 2Mb of video memory, executing in 640-by-480 pixels resolution in 24-bit color can only create one back buffer, since the primary surface requires 921,600 bytes. By reducing the color depth to 16 bits, the sample application needs only 614,400 bytes for the primary surface, and it can now create two back buffers in display memory. The following code fragment shows the creation of a primary surface with two back buffers:

```
//Global variables
LPDIRECTDRAWSURFACE7        lpDDSPrimary = NULL;
LPDIRECTDRAWSURFACE7        lpDDSBackBuf = NULL;
DDSURFACEDESC2              ddsd;      // Surface description
HRESULT                     hRet;
. . .
// Create a primary surface with two back buffers
// ddsd is a structure of type DDSRUFACEDESC2
  ZeroMemory(&ddsd, sizeof(ddsd));  // Fill structure with zeros
// Fill in other members
  ddsd.dwSize = sizeof(ddsd);
  ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
  ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                        DDSCAPS_FLIP |
                        DDSCAPS_COMPLEX |
                        DDSCAPS_VIDEOMEMORY;
  ddsd.dwBackBufferCount = 2;   // Two back buffers requested
  hRet = lpDD4->CreateSurface(&ddsd, &lpDDSPrimary, NULL);
  // At this point code can examine hRet for DD_OK and
  // provide alternate processing if the surface creation
  // call failed
  // Get backbuffer pointer
  ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
  hRet = lpDDSPrimary->GetAttachedSurface(&ddscaps,
                     &lpDDSBackBuf);
  // At this point code can examine hRet for DD_OK and
  // provide alternate processing if the back buffer pointer
  // was not returned
```

31.5 Animation Programming

Exclusive mode applications that use flipping animation start by initializing DirectDraw, setting a display mode, creating the flip chain, obtaining the corresponding pointers to the front and back buffers, and setting up a timer mechanism that produces the desired beat. Once these housekeeping chores are finished, the real work can begin, which consists of rendering the imagery to the back buffer, usually by means of blits from other surfaces in video memory or off screen. The design and coding challenge in creating an animated application using DirectDraw can be broken down into two parts:

1. Assign the minimum resources that will allow the program to perform satisfactorily.
2. Make the best use of these resources in order to produce the finest and smoothest animation possible.

31.5.1 Background Animation

A typical computer game or real-time simulation often contains two different types of graphics objects: backgrounds and sprites. The backgrounds consist of larger bitmaps over which the action takes place. For example, a flight simulator program can have several background images representing different views from the cockpit. These may include landscapes, seascapes, and views of airports and runways used during takeoff and landing. A computer game that takes place in a medieval castle may use backgrounds depicting the various castle rooms and hallways where the action takes place. Sprites, on the other hand, are rather small, animated objects represented in two or three dimensions. In the flight simulator program the sprites could be other aircraft visible from the cockpit and the cabin instruments and controls that are animated during the simulation. In the computer game, the sprites could be medieval knights that do battle in the castle, as well as weapons and other objects used in the battle.

31.5.2 Panning Animation

The design and display of background images is relatively straightforward. The most difficult part consists of creating the background imagery and using clipping and blit-time rectangles to generate panning and zoom effects. The project named DD Panning Animation in the book's software package, demonstrates panning animation of a background bitmap. In the program the source rectangle has the same vertical dimension as the background bitmap, which is 480 pixels. The image bitmap is 1280 pixels wide and the source rectangle is one-half that size (640 pixels). This creates a source window that can be moved 639 pixels to the right from its original position. The white, dotted rectangle in Color Figure 10 represents the source rectangle within the background bitmap.

The program DD Panning Animation, in the book's software package, demonstrates panning animation. The logic is based on panning to right until the image right border is reached, and then reverse the panning direction until the left border is reached. The primary surface and a single back buffer are created and a clipper is installed in both surfaces. The background bitmap, in this case a mountain range, is stored in the file named image.bmp. This bitmap is twice as wide as the viewport; therefore, the source rectangle can moved horizontally within the bitmap. The panning variables and the display routine are coded as follows:

```
// Global panning animation controls
RECT         thisPan;           // Storage for source rectangle
LONG         panIteration = 0;  // panning iteration counter
LONG         panDirection = 0;  // 1 = left, 0 = right
// Constants
LONG         PAN_LIMIT_LEFT = 1;
LONG         PAN_LIMIT_RIGHT = 639;
.
```

```
.
.
//**********************************************
// Name: PanImage
// Desc: Update back with a source rectangle that
//       is a portion of the background bitmap
//       and flip to create a panning animation
//**********************************************
static void PanImage()
{

  thisTickCount = GetTickCount();
  if((thisTickCount - lastTickCount) < TIMER_VALUE)
     return;

  else
  {
    lastTickCount = thisTickCount;

  // Bump pan iteration according to direction
     if(panDirection == 1)
        panIteration--;
     else
        panIteration++;

  // Reset panning iteration counter at limits
    if(panIteration == PAN_LIMIT_RIGHT)
       panDirection = 1;           // Pan left
    if(panIteration == PAN_LIMIT_LEFT)
       panDirection = 0;

  // Set panning rectangle in source image
  thisPan.left = panIteration;
  thisPan.top = 0;
  thisPan .right = 640 + panIteration;
  thisPan.bottom = 480;

  // Blit background bitmap to back buffer
     hRet = lpDDSBackBuf->Blt(NULL,
                 lpDDSBackGrnd,
                 &thisPan,
                 DDBLT_WAIT,
                 NULL);
     if(hRet != DD_OK){
        DDInitFailed(hWnd, hRet,
           "Blt() on background failed");
     return;
     }

   // Flip surfaces
      hRet = lpDDSPrimary->Flip(NULL, DDFLIP_WAIT);
      if(hRet != DD_OK){
          DDInitFailed(hWnd, hRet,
               "Flip() call failed");
      return;
      }
   return;
   }
}
```

The local function named PanImage(), listed previously, performs the panning animation. First it bumps and checks the ticker counter. If the counter has not yet expired, execution returns immediately. Code then checks the panDirection variable. If the direction is 1, then panning is in the left-to-right cycle and the panIteration variable is decremented. If not, then panning is right-to-left and the panIteration variable is incremented. When either variable reaches the limit, as defined in the constants PAN_LIMIT_LEFT and PAN_LIMIT_RIGHT, the panning direction is reversed. A structure variable named thisPan, of type RECT, is used to define the source rectangle for the blit. The panIteration variable is used to define the offset of the source rectangle within the image bitmap. Since panning takes place on the horizontal axis only, and the display mode is defined in the code, then the image size can be hard-coded into the thisPan structure members. Once the image is blitted onto the back buffer, surfaces are flipped in the conventional manner.

31.5.3 Zoom Animation

Zooming is a background animation that can be implemented by manipulating the source or destination rectangles, or both. This is possible due to the fact that both Blt() and BltFast() perform automatic scaling when the source and destination areas are of a different size. The simplest approach to zooming animation consists of reducing the area covered by the source rectangle and letting Blt() or BltFast() perform the necessary adjustments. Color Figure 10 shows the initial and final source rectangles in a zoom animation.

The program DD Zoom Animation.cpp, in the book's software package, demonstrates zoom animation using an image of Mount Rushmore. The program action is to zoom into a bitmap by changing the position and progressively reducing the dimensions and the source rectangle. When an arbitrary maximum zoom value is reached, the process reverses and the source rectangle is made progressively larger until it is restored to the original size. As in the panning animation demo program, the primary surface and a single back buffer are created, and a clipper is installed in both surfaces. The background image, which in this case is stored in the file image.bmp, is then moved to an offscreen surface. This bitmap is the size of the viewport. In the following code fragment we show the zoom controls and display operations that are different from the panning animation, previously listed:

```
// Zoom animation controls
RECT      thisZoom;   // Storage for source rectangle
LONG      zoomIteration = 0;  // panning iteration counter
LONG      zoomDirection = 0;  // 1 = left, 0 = right
// Constants
LONG      ZOOM_LIMIT_OUT = 1;
LONG      ZOOM_LIMIT_IN = 200;
.
.
.
// Bump zoom iteration according to direction
if(zoomDirection == 1)
   zoomIteration--;
else
   zoomIteration++;

// Reset zoom iteration counter at limits
```

```
    if(zoomIteration == ZOOM_LIMIT_IN)
       zoomDirection = 1;          // Pan left
    if(zoomIteration == ZOOM_LIMIT_OUT)
       zoomDirection = 0;

    // Set zoom rectangle in source image
       thisZoom.left = zoomIteration;
       thisZoom.top = zoomIteration;
       thisZoom.right = 640 - zoomIteration;
       thisZoom.bottom = 480 - ((zoomIteration * 3)/4);
```

Notice that the dimensioning of the source rectangle for zoom animation must take into account the screen's aspect ratio, which is approximate 3:4. Therefore the y coordinate of the end point of the source rectangle is changed at a rate slower than the x coordinate. If both coordinates were reduced by the same amount, the resulting images would be stretched along this axis during the zoom.

31.5.4 Animated Sprites

Sprites are often animated. An animated sprite can be a fuel gauge on the dashboard of a race car simulation, a spaceship on a futuristic game, or a medieval warrior. Designing, encoding, and manipulating sprites require all the talents and skills of the animator. The project named DD Multi Sprite Animation, in the book's software package, demonstrates sprite animation by simultaneously moving three screen objects at different speeds. Color Figure 12 is a screen snapshot of the demonstration program. The three hot-air balloons are the sprites. During program execution the balloons rise at different speeds. The largest balloon, which appears closer to the viewer, moves up one pixel during every iteration of the frame counter. The balloon on the left moves every second iteration and the one on the right every third iteration. The background is fixed in this sample.

Controlling several sprites, simultaneously displayed, can be a challenge regarding program design and data structures, but does not present any major programming problems in DirectDraw. The program DD Multi Sprite Animation starts by creating a primary surface and two back buffers. The use of a second back buffer improves program execution in most machines. A clipper is then installed on both surfaces. The clipper makes the animated objects appear to come into the display area, and disappear from it, softly and pleasantly. The background image, which is located in the bitmap named backgrnd.bmp, is stored in an offscreen surface. This bitmap is the size of the viewport. The code creates three additional surfaces, one for each of the sprites, and moves the sprite bitmaps into these surfaces. The sprite surfaces are assigned a source color key to make the bitmap backgrounds transparent at display time. To ensure a smooth animation, all the surfaces in the sample program are located in video memory.

Sprite control in the demo program is based on a structure of type SpriteCtrl defined globally, as follows:

```
// Sprite control structure
struct SpriteCtrl
{
   LONG     startY;          // Start x coordinate
   LONG     startX;          // Start y coordinate
   LONG     bmapX;           // Width of bitmap
   LONG     bmapY;           // Height of bitmap
   LONG     iterMax;         // Maximum iteration count
   LONG     skipFactor;      // Display delay
   LONG     iteration;       // Sprite iteration counter
} Sprite1, Sprite2, Sprite3;
```

Three structure variables, named Sprite1, Sprite2, and Sprite3 are allocated, one for each animated object. The sprites are numbered left-to-right as they are displayed, as shown in Color Figure 11. Each structure variable contains the members startX and startY that define the start coordinates for each sprite. The members bmapX and bmapY store the bitmap dimensions, which are obtained as the bitmaps are loaded from their files.

The sprite animation control is performed by the last three members of the SpriteCtrl structure. The iterMax member stores the value of the iteration counter at which the sprite is repositioned to the bottom of the screen. The skipFactor member determines how many iterations are skipped at display time. This value is used to slow down the smaller balloons. Sprite1 is assigned a skipFactor of 2. Sprite2, the largest one, has skipFactor equal to 1. Sprite3, the smallest one, has a skipFactor of 3. The iteration member keeps track of the number of frame beats corresponding to each sprite. The counters are reset when the iterMax value is reached for each sprite. The range of the iteration counters are from 0 to iterMax. The code initializes the structure members for each sprite, as follows:

```
//******************************************************
//   Fill SpriteCtrl structure members for each sprite
//******************************************************
// Sprite1 is balloon bitmap in bMap1
// Resolution is 640 by 480 pixels
Sprite1.startY = 479;     // Starts at screen bottom
Sprite1.startX = 70;      // x for start position
Sprite1.bmapX = bMap1.bmWidth;
Sprite1.bmapY = bMap1.bmHeight;
Sprite1.skipFactor = 2;
Sprite1.iterMax = (480+(bMap1.bmHeight)) * Sprite1.skipFactor;
Sprite1.iteration = 50;    // Init iteration counter
// Sprite2 is balloon bitmap in bMap2
Sprite2.startY = 479;     // Starts at screen bottom
Sprite2.startX = 240;     // x for start position
Sprite2.bmapX = bMap2.bmWidth;
Sprite2.bmapY = bMap2.bmHeight;
Sprite2.skipFactor = 1;
Sprite2.iterMax = 480+(bMap2.bmHeight);
Sprite2.iteration = 50;    // Init iteration counter
// Sprite3 is balloon bitmap in bMap3
Sprite3.startY = 479;     // Starts at screen bottom
Sprite3.startX = 500;      // x for start position
Sprite3.bmapX = bMap3.bmWidth;
Sprite3.bmapY = bMap3.bmHeight;
Sprite3.skipFactor = 3;
Sprite3.iterMax = (480+(bMap3.bmHeight)) * Sprite3.skipFactor;
Sprite3.iteration = 20;    // Init iteration counter
```

During initialization, the dimensions of each sprite are read from the corresponding bmWidth and bmHeight members of the BITMAP structure for each sprite. This ensures that the code continues to work even if the size of a sprite bitmap is changed. The maximum number of iterations for each sprite is calculated by adding the number of screen pixels in the selected mode (480), to the bitmap pixel height, and multiplying this sum by the sprite's skip factor. At display time the surface with the background bitmap is first blitted to the back buffer. Then the code calls a local function, named SpriteAction(), for each sprite. The FlipImages() function is coded as follows:

```
//*************************************************
// Name: FlipImages
// Desc: Update back buffer and flip
//*************************************************

static void FlipImages()
{
   thisTickCount = GetTickCount();
   if((thisTickCount - lastTickCount) < TIMER_VALUE)
      return;
   else
   {
       lastTickCount = thisTickCount;

    // Blit background bitmap to back buffer
    hRet = lpDDSBackBuf->Blt(NULL,
                             lpDDSBackGrnd,
                             NULL,
                             DDBLT_WAIT,
                             NULL);
     if(hRet != DD_OK){
        DDInitFailed(hWnd, hRet,
                  "Blt() on background failed");
        return;
    }

  // Animate sprites. Farthest ones first
    SpriteAction(Sprite3, lpDDSBmap3);
    SpriteAction(Sprite2, lpDDSBmap2);
    SpriteAction(Sprite1, lpDDSBmap1);

 // Flip surfaces
     hRet = lpDDSPrimary->Flip(NULL, DDFLIP_WAIT);
      if(hRet != DD_OK){
        DDInitFailed(hWnd, hRet,
                    "Flip() call failed");
        return;
       }
     return;
  }
}
```

The actual display of the sprites is performed by the local function named SpriteAction(). This function receives the SpriteCtrl structure variable for the sprite being animated, and the pointer to the surface that contains the sprite image. The code checks the iteration number for the sprite against the maximum count, to determine if the iteration counter needs resetting. Then the position of the sprite is calculated by dividing the current iteration number by the skip factor. This information is stored in a RECT structure corresponding to the destination surface rectangle and the Blt() function is called. SpriteAction() code is as follows:

```
//*************************************************
// Name: SpriteAction
// Desc: Animate a sprite according
//       to its own parameters
// PRE:
//    1. Pointer to structure containing
//       sprite data
//    2. Pointer to DirectDraw surface
//       containing sprite bitmap
//************************************************
void SpriteAction(SpriteCtrl &thisSprite,
          LPDIRECTDRAWSURFACE4 lpDDSBmap)
{
    RECT        destSurf;
    LONG        vertUpdate;

   thisSprite.iteration++;

   if(thisSprite.iteration == thisSprite.iterMax)
      thisSprite.iteration = 0;

   vertUpdate = thisSprite.iteration / thisSprite.skipFactor;

   // Set coordinates for balloon1 display
   destSurf.left = thisSprite.startX;
   destSurf.top = thisSprite.startY - vertUpdate;
   destSurf.right = destSurf.left + thisSprite.bmapX;
   destSurf.bottom = destSurf.top + thisSprite.bmapY;

   // Use Blt() to blit bitmap from the off-screen surface
   // (->lpDDSBitamp), onto the back buffer (->lpDDSBackBuf)
   hRet = lpDDSBackBuf->Blt(&destSurf,
                            lpDDSBmap,
                            NULL,
                            DDBLT_WAIT | DDBLT_KEYSRC,
                            NULL);

     if(hRet != DD_OK){
        DDInitFailed(hWnd, hRet,
                  "Blt() on sprite failed");
    return;
   }
  return;
}
```

After the background bitmap and all three sprites have been blitted onto the back buffer, the code calls the Flip() function to render the results onto the primary surface.

31.6 Fine-Tuning the Animation

In computer animation the greatest concern is to produce a smooth and uniform effect, with as little bumpiness, screen tearing, and interference as possible. Today's machines, with 1000 MHz and faster CPUs, video cards with graphics coprocessors and 8, 16, or more megabytes of video memory, high-speed buses, and DirectX software, can often produce impressive animations with straightforward code. For example, the DD Multi Sprite Animation program, in the book's software package, smoothly animates three sprites, even when running in a 200-MHz Pentium machine equipped with a low-end display card and 2Mb of video memory.

In most cases the animation programmer is pushing graphics performance to its limits. If the animator finds that the code can successfully manipulate three sprites, then perhaps it can manipulate four, or even five. The rule seems to be: the more action, and the faster the action, the better the animation. In this section we discuss several topics that relate to improving program performance or to facilitating implementation.

31.6.1 High-Resolution Timers

Earlier in this chapter we explored two methods of obtaining the timed pulse that is used to produce frame updates in an animation routine. One method is based on a milliseconds counter maintained by the system, which can be read by means of the GetTickCount() function. The other one sets an interval timer that operates as an alarm clock. When the timer lapse expires the application receives control, either in a message handler intercept or a dedicated callback function. Although both methods are often used, processing based on reading the windows tick counter has considerably better resolution than the alarm clock approach. Windows documentation states that the resolution of the timer intercepts is approximately 55 milliseconds. This produces a beat of 18.2 times per second, which is precisely the default speed of the PC internal clock. In many cases this beat is barely sufficient to produce smooth and lifelike animations.

There are several ways of improving the frequency and reliability of the timed pulse. The multimedia extensions to Windows include a high-resolution timer with a reported resolution of 1 millisecond. The multimedia timer produces more accurate results because it does not rely on WM_TIMER messages posted on the queue. Each multimedia timer has its own thread, and the callback function is invoked directly regardless of any pending messages. To use the multimedia library code must include MMSYSTEM.H and make sure that WINMM.LIB is available and referenced at link time. Several timer functions are found in the multimedia library. The most useful one in animation programming is timeSetEvent(). This function starts an event timer, which runs in its own thread. A callback function, defined in the call to timeSetEvent() receives control every time the counter timer expires. The function's general form is as follows:

```
MMRESULT timeSetEvent(
                      UINT uDelay,                     // 1
                      UINT uResolution,                // 2
                      LPTIMECALLBACK lpTimeProc,       // 3
                      DWORD dwUser,                    // 4
                      UINT fuEvent                     // 5
                      );
```

The first parameter defines the event delay, in milliseconds. If this value is not in the timer's valid range, then the function returns an error. The second parameter is the resolution, in milliseconds, of the timer event. As the values get smaller, the resolution increases. A resolution of 0 indicates that timer events should occur with the greatest possible accuracy. Code can use the maximum appropriate value for the timer resolution in order to reduce system overhead. The third parameter is the address of the callback function that is called every time that the event delay counter expires. The fourth parameter is a double word value passed by Windows to the callback procedure. The fifth parameter encodes the timer event type. This parameter consists of one or more predefined constants listed in Table 31-3.

Table 31-3

Event-Type Constants in TimeSetEvent() Function

VALUE	MEANING
TIME_ONESHOT	Event occurs once, after uDelay milliseconds.
TIME_PERIODIC	Event occurs every uDelay milliseconds.
TIME_CALLBACK_FUNCTION	Windows calls the function pointed to by the third parameter. This is the default.
TIME_CALLBACK_EVENT_SET	Windows calls the SetEvent() function to set The vent pointed to by the third parameter. The fourth parameter is ignored.
TIME_CALLBACK_EVENT_PULSE	Windows calls the PulseEvent() function to pulse the event pointed to by the third parameter. The fourth parameter is ignored.

Notice that the multimedia timers support two different modes of operation. In one mode (TIME_ONESHOT) the timer event occurs but once. In the other mode (TIME_PERIODIC) the timer event takes place every time that the timer counter expires. This mode is the one most often used in animation routines. If successful, the call returns an identifier for the timer event. This identifier is also passed to the callback function. Once the timer is no longer needed, applications should call the timeKillEvent() function to terminate it.

Despite its high resolution, it has been documented that the multimedia timer can suffer considerable delays. One author states having recorded delays of up to 100 milliseconds. Applications requiring very high timer accuracy are recommended to implement the multimedia timer in a 16-bit DLL.

The WIN32 API first made available a high-resolution tick counter. These counters are sometimes called performance monitors since they were originally intended for precisely measuring the performance of coded routines. Using the high-performance monitors is similar to using the GetTickCount() function already described, but with a few special accommodations. Two Windows functions are associated with performance monitor counters: QueryPerformanceFrequency() returns the resolution of the counter, which varies according to hardware. QueryPerformanceCounter() returns the number of timer ticks since the system was started. QueryPerformanceFrequency() can also be used to determine if high-performance counters are available in the hardware, although the presence of the performance monitoring function can be assumed in any Windows 95, 98, or ME, Windows 2000, or NT machine.

The function prototypes are identical for QueryPerformanceFrequency() and QueryPerformanceCount(): the return type is of type BOOL and the only parameter is a 64-bit integer of type LARGE_INTEGER. The general forms are as follows:

```
BOOL QueryPerformanceCounter(LARGE_INTEGER*);
BOOL QueryPerformanceFrequency(LARGE_INTEGER*);
```

Although it has been stated that the performance frequency on Intel-based PCs is 0.8 microseconds, it is safer for applications to call QueryPerformanceFrequency() to obtain the correct scaling factor. The following code fragment shows this processing:

```
   _int64         TIMER_DELAY = 15;  // Milliseconds
   _int64         frequency;         // Timer frequency
   .
   .
   .
   QueryPerformanceFrequency((LARGE_INTEGER*) &frequency);
   TIMER_DELAY = (TIMER_DELAY * frequency) / 1000;
```

After executing, the TIMER_DELAY value has been scaled to the frequency of the high-resolution timer. The QueryPerformanceCounter() can now be called in the same manner as GetTickCount(), for example:

```
   _int64          lastTickCount;
   _int64          thisTickCount;
   .
   .
   .
   QueryPerformanceCounter((LARGE_INTEGER*) &thisTickCount);
   if((thisTickCount - lastTickCount) < TIMER_DELAY)
     return;
   else {
     lastTickCount = thisTickCount;
   .
   .
   .
```

The DD Multi Sprite Animation program, in the book's software package, uses a high-performance timer to produce the animation beat.

31.6.2 Dirty Rectangles

Color Figure 11 shows a background image overlayed by three sprites. During every iteration of the animation pulse, code redraws the background in order to refresh those parts of the surface that have been overwritten by the sprites. The process is wasteful since most of the background remains unchanged. In fact, only the portion of the background that was covered by the sprite image actually needs to be redrawn. Figure 31-8 shows the rectangular areas that actually need refreshing in producing the next animation iteration of the image, as shown in Color Figure 11. These are called the “dirty rectangles.”

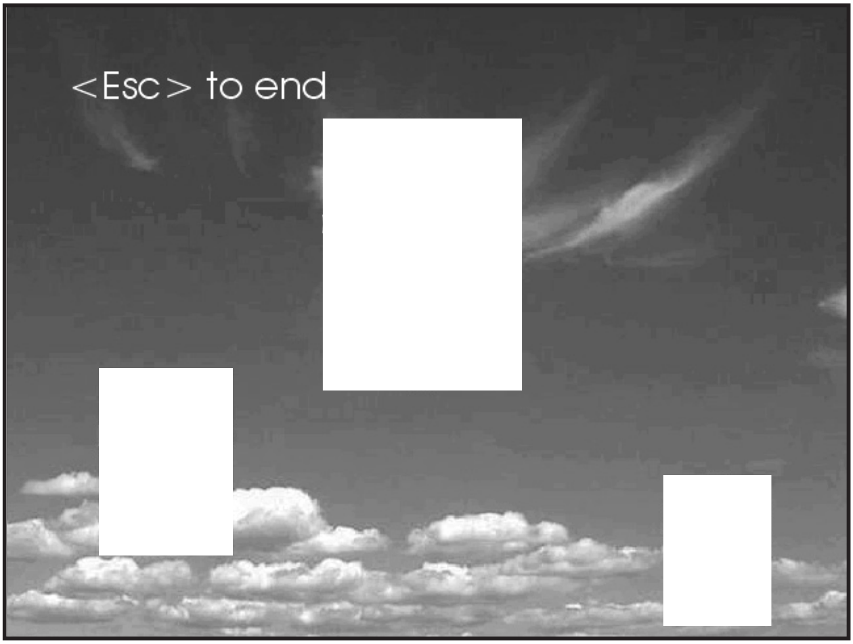


Figure 31-8 *Dirty Rectangles in Animation*

DirectDraw clipping operations can be used to identify the dirty rectangles. In this case a clip list defines the areas that require refreshing, and these are the only ones updated during the blit. The processing is simplified by the fact that the last position of the sprite, and its stored dimensions, can be used to determine the dirty rectangles.

Whether the use of dirty rectangles actually improves performance depends on several factors: the total image area covered by the dirty rectangles, the number of rectangles, the processing overhead in calculating the rectangles and creating the clip list, and, above all, the efficiency of the DirectDraw clipping operations of the particular hardware. Unfortunately, in many cases, the screen update takes longer with dirty rectangle schemes than without them. The most rational approach is to develop the animation without dirty rectangles. If the results are not satisfactory, then try the dirty rectangles technique. The comparative results can be assessed by measuring the execution time in both case. Methods for measuring performance of routines are discussed later in this section.

31.6.3 Dynamic Color Keys

It is difficult to image a sprite that can be transparently overlayed on a bitmapped background without the use of a source color key. When we create our own sprites using draw or paint programs, and these sprites are stored in 24- or 32-bit color depth bitmaps, the color key is usually known at coding time, or can be easily determined. If there is any doubt, the sprite can be loaded into a bitmap editor in order to inspect the RGB value of the background pixels. However, there are cases in which determining the color key is more difficult. One of the complicating factors with color keys occurs when the color depth of the application's video mode does not coincide with that of the sprite bitmap. This can a problem in the palletized display modes, particularly when the palette changes during execution, or in applications that use several possible video modes.

One solution is to determine the bitmap's color key dynamically, that is, at runtime. The method is based on the assumption that there is a fixed location in the bitmap which is transparent at blit time. For example, the pixel at the bitmap's upper-left corner of the sprite image rectangle is typically part of the background. Color Figure 13 shows the fixed location of the color key for one of the balloon bitmaps used in the program DD Multi Sprite Animation contained in the book's software package.

Once the relative location of a color key pixel has been determined, code can load the bitmap onto a surface, and then read the surface data at the predefined position in order to obtain the color key. The manipulation is made possible by the direct access to memory areas that is available in DirectDraw. Since the application knows the color depth of the target surface, it can read the color key directly from the surface. In this case you need not be concerned with how Windows converts a pixel value in one color depth into another one, since the code is reading the resulting color key directly. The following code is used in the DD Multi Sprite Animation program for dynamically loading the color key for Sprite1.

```
  // Video display globals
  LONG      vidPitch = 0;  // Pitch
  LPVOID    vidStart = 0;  // Buffer address
  // Color key data
  DDCOLORKEY  bColorKey;
  WORD        dynamicKey;
  .
  .
  .
 //**************************************************
 // move first balloon bitmap to off-screen surface
 //**************************************************
 // Load the bitmap into memory
 bal1Bitmap = ( HBITMAP )LoadImage( NULL, "balloon1.bmp",
                IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE );
   if ( bal1Bitmap == NULL )
        DDInitFailed(hWnd, hRet,
         "Balloon1 bitmap load failed");
 // Get bitmap dimensions to determine surface size
    GetObject(bal1Bitmap, sizeof (BITMAP), &bMap1);

 // Create the off-screen surface for bitmap in system memory
    ZeroMemory(&ddsd, sizeof(ddsd));  // Fill structure with zeros
 // Fill in other members
    ddsd.dwSize = sizeof(ddsd);
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
                          DDSCAPS_VIDEOMEMORY;
    ddsd.dwHeight = bMap1.bmHeight;
    ddsd.dwWidth = bMap1.bmWidth;
    hRet = lpDD4->CreateSurface(&ddsd, &lpDDSBmap1, NULL);
      if (hRet != DD_OK)
```

```
         return DDInitFailed(hWnd, hRet,
                           "Off screen surface1 creation failed ");

   // Move bitmap to surface using DDBmapToSurf()function
   hRet = DDBmapToSurf(lpDDSBmap1, bal1Bitmap);
     if(hRet != DD_OK)
        return DDInitFailed(hWnd, hRet,
                      "DDMapToSurf() call failed");

  //*******************************************
  //   read color key from loaded sprite
  //*******************************************
  // Attempt to lock the surface for direct access
  if (!LockSurface(lpDDSBmap1))
        return DDInitFailed(hWnd, hRet,
                      "Surface Lock failed ");
  // Surface data is stored as follows:
  //   LONG      vidPitch;  // Pitch (not used here)
  //   LPVOID    vidStart;  // Buffer address
  _asm
  {
      PUSH   ESI          ; Save context
      PUSHF
      MOV    ESI,vidStart ; Left-top pixel address
      ; Read and store pixel attributes
      MOV    AX,[ESI]        ; Get attribute
      MOV    dynamicKey,AX   ; Store value in variable
      POPF                   ; Restore context
      POP    ESI
  }
  ReleaseSurface(lpDDSBmap1);

  // Set color key for balloon1 surface using values stored
  // in variable dynamicKey
   bColorKey.dwColorSpaceLowValue = dynamicKey;
   bColorKey.dwColorSpaceHighValue = dynamicKey;
     hRet = lpDDSBmap1->SetColorKey(DDCKEY_SRCBLT, &bColorKey);
     if(hRet != DD_OK)
        return DDInitFailed(hWnd, hRet,
              "SetColorKey() for Balloon1 failed");
```

31.7 Measuring Performance

Programmers often need to know the execution time of a routine or function in order to determine if it is suitable for an animation application. Several software engineering techniques allow estimating performance and efficiency of algorithms. These methods, which are based on mathematical analysis, are usually difficult and time-consuming. Alternatively, you can often obtain the necessary performance metrics of a routine by physically measuring its execution time.

In cases in which time-of-execution ranges from several seconds to several minutes it is often possible to measure it with a stopwatch. More often the time of execution is in the milliseconds order, in which case it may be possible to use the computer's timing mechanisms to determine the time lapsed between the start and the end of a processing routine or code segment. The QueryPerformanceCounter() function, described previously, has a resolution in the order of one-millionth of a second. In order to measure the execution time of a program segment, function, or routine we need to read the tick counter at the start and the end of the processing routine, then subtract these values. The difference is the approximate execution time.

Unfortunately, there are many complicating factors that can affect the accuracy of this simple scheme. In the first place, the scheduler in a multitasking environment can interrupt a thread of execution at any time, thereby delaying it. Sometimes the unit-boundary at which a data item is located in memory affects the time required for a memory fetch operation. Another consideration relates to the occasional state of a memory cache, which can also change the fetch-time for data. This means that the measurements should be repeated as many times as practicable in order to obtain a more reliable value. Even with many repetitions the resulting numbers may not be accurate. However, for many practical programming situations the data obtained in this manner is sufficient for a decision regarding which of two or more routines is more suitable for the case at hand. The following code fragment shows measuring the execution time of two routines:

```
// Timer data
_int64          startCount;
_int64          endCount;
_int64          timeLapse1;   // First routine
_int64          timeLapse2;   // Second routine
.
.
.
// First routine starts here
QueryPerformanceCounter((LARGE_INTEGER*) &startCount);
//
// First routine code
//
QueryPerformanceCounter((LARGE_INTEGER*) &endCount);
timeLapse1 = endCount - startCount;
. . .
// Second routine starts here
QueryPerformanceCounter((LARGE_INTEGER*) &startCount);
//
// Second routine code
//
QueryPerformanceCounter((LARGE_INTEGER*) &endCount);
timeLapse2 = endCount - startCount;
```

The variables timeLapse1 and timeLapse2 now hold the number of timer ticks that elapsed during the execution of either routine. Code can display these values or a debugger can be used to inspect the variables.

Chapter 32

Direct3D Fundamentals

Topics:

- Description and origins of Direct3D
- Retained mode and immediate mode
- Direct3D and COM
- Direct3D modules
- Basic elements of retained mode
- Rendering mathematics
- File formats

In this chapter we begin our discussion of Direct3D programming with the higher-level functions, called retained mode. This chapter presents a smorgasbord of topics. The glue that holds them together is the fact that they are all necessary in order to understand and use Direct3D.

Before reading this chapter, make sure that you have grasped the material in Part I of the book, which provides the necessary background in 3D graphics.

32.1 3D Graphics in DirectX

There is some confusion regarding the scope and application of 3D graphics. One reason for this confusion is that 3D displays are not yet commercially available for the PC. Devices that render solid images, on a three-dimensional screen, are still experimental. Therefore, in a strict sense, 3D graphics do not yet exist commercially. However, systems capable of storing and manipulating images of solid objects and displaying these objects on 2D media do exist. What we call 3D graphics in today's technology is actually a 2D rendering of a 3D object.

Direct3D is the component of Microsoft's DirectX software development kit that provides support for real-time, three-dimensional graphics, as available in today's machines. 3D programming is a topic at the cutting edge of PC technology. But cutting-edge infrastructures are rarely stable. Many of its features are undergoing revisions and redesigns, and there are still some basic weaknesses and defects. Furthermore, the performance of 3D applications depends on a combination of many factors, some of which are hidden in the software layers of the development environment. In today's world 3D applications developers spend much of their time working around the system's inherent weaknesses. Scores of video cards are on the market, each one supporting its own set of features and functionality. Developing a 3D application that executes satisfactorily in most systems is no trivial task. The bright side of it is that the rewards can be enormous.

32.1.1 Origin of Direct3D

Direct3D is described as a graphics operating system, although it would be less pretentious, and perhaps more accurate, to refer to it as a 3D graphics back end. Its core function is to provide an interface with the graphics hardware, thus insulating the programmer from the complications and perils of device dependency. It also provides a set of services that enable you to implement 3D graphics on the PC. In this sense it is similar to other back ends, such as OpenGL and PHIGS. But Direct3D is also a provider of low-level 3D services for Windows. In Microsoft's plan the low-level components of Direct3D (immediate mode) serve and support its higher-level components (retained mode) and those of other 3D engines (OpenGL).

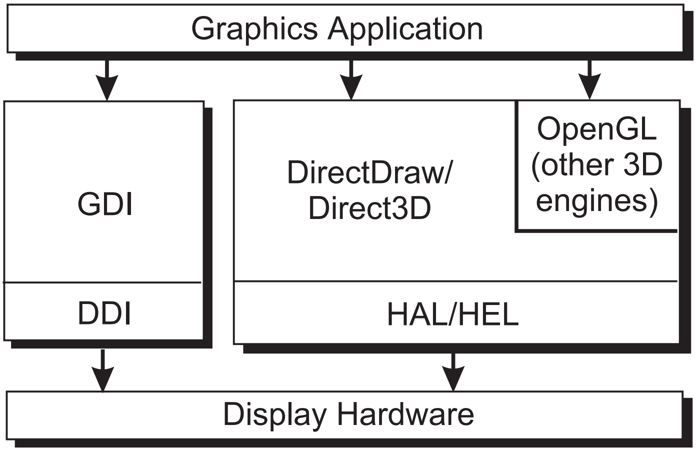
In the beginning 3D was exclusively in the realm of the graphics workstation and the mainframe computer. The standards were PHIGS (Programmer's Hierarchical Interactive Graphics Systems), and GKS (Graphical Kernel System). During the 1980s it was generally accepted that the processing power required to manipulate 3D images, with its associated textures, lightning, and shadows, was not available on the PC. However, some British game developers thought differently and created some very convincing 3D games that ran on a British Broadcasting Corporation (BBC) micro equipped with a 2MHz 6502 processor. This work eventually led Servan Keondjian, Doug Rabson, and Kate Seekings to the founding of a company named RenderMorphics and the development of the Reality Lab rendering engine for 3D graphics. Their first 3D products were presented at the SIGGRAPH '94 trade show.

In February 1995, Microsoft bought RenderMorphics and proceeded to integrate the Reality Lab engine into their DirectX product. The result was called Direct3D. Direct3D has been one of the components of DirectX since its first version, called DirectX 2. Other versions of the SDK, namely DirectX 3, DirectX 5, DirectX 6, and currently, DirectX 7, also include Direct3D. The functionality of the Direct3D is available to applications running in Windows 95/98 and Windows NT 3.1 and later. The full functionality of DirectX SDK is part of Windows 98 and will also be found in Windows NT 5.0 and Windows 2000. This means that applications running under Windows 98 and later will be able to execute programs that use Direct3D without the loading of additional drivers or other support software.

32.1.2 Direct3D Implementations

Direct3D is an application-programming interface for 3D graphics on the PC. The other major 3D API for the PC is OpenGL, which is discussed in Part IV. Figure 32-1 shows the structure of the graphics development systems under Windows.

Figure 32-1 *Windows Graphics Architecture*



Direct3D provides the API services and device independence required by developers, delivers a common driver model for hardware vendors, enables turnkey 3D solutions to be offered by personal-computer manufacturers, and makes it easy for end-users to add high-end 3D functions to their systems. Because the system requires little memory, it runs well on most of the installed base of computer systems.

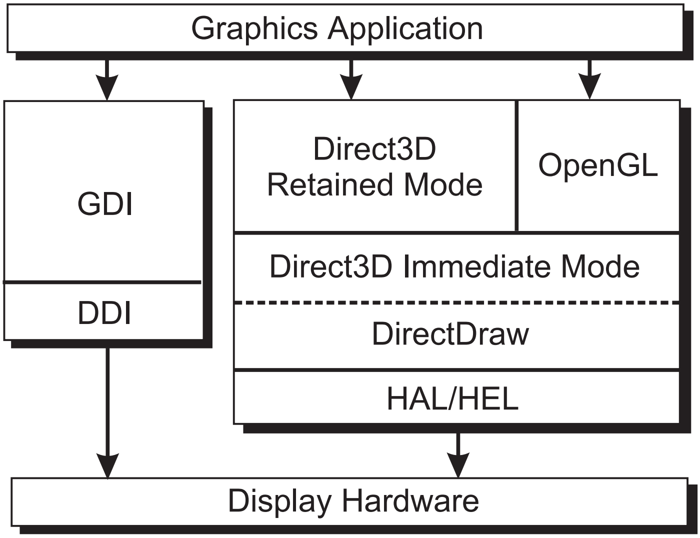
The 3D graphics services in Direct3D execute in real-time. The functions include rendering, transformations, lighting, shading, rasterization, z-buffering, textures, and transparent access to acceleration features available in the hardware. The Direct3D architecture consists of two well-defined modes: a low-level one called immediate mode, and a high-level one called retained mode. The term retained mode originally referred to the images being preserved after rendering, but this notion is no longer literally true.

32.1.3 Retained Mode

Retained mode was designed as a set of API for the high-level manipulation of 3D objects and managing 3D scenes. It is Microsoft's competition for OpenGL and other high-level 3D development environments. It is implemented as a set of interrelated COM objects that enable you to build and manipulate a 3D scene. Its intention was to make it easy to create 3D Windows applications or to add 3D capabilities to existing ones. The programmer working in retained mode can take advantage of its geometry engine, which contains advanced 3D capabilities, without having to create object databases or be concerned with internal data structures. The application uses a single call to load a predefined 3D object, usually stored in a file in .x format. The loaded object can then be manipulated within the scene and rendered in real-time. All of this is done without having to deal with the object's internals.

Retained mode is tightly coupled with DirectDraw, which serves as its rendering engine, and is built on top of the immediate mode. OpenGL and other high-level systems exist at the same level as retained mode. Figure 32-2, on the following page, shows the elements of this interface.

Figure 32-2 *DirectX Graphics Architecture*



32.1.4 Immediate Mode

Direct3D immediate mode is a layer of low-level 3D API. Its original intention was to facilitate the porting of games and other high-precision and high-performance graphics applications to the Windows operating system. It allows access to hardware features in the 3D chip and offers software rendering when the function is not available in the hardware. The intention of immediate mode is to enable applications to communicate with the 3D hardware in a device-independent manner and to provide maximum performance.

In contrast with retained mode, immediate mode does not contain a graphics engine. Code that uses immediate mode must provide its own routines to implement object and scene management. This means that the effective use of immediate mode requires considerable knowledge and skills in 3D graphics.

32.1.5 Hardware Abstraction Layer

In Figure 32-2 you see that both the Immediate and the Retained Modes of Direct3D are built on top of the Hardware Abstraction Layer (HAL). It is this software layer that insulates the programmer from the device-specific dependencies. The Hardware Emulation Layer (HEL) provides support for those features that are not present in the hardware. The combination of HEL and HAL ensure that the complete Direct3D functionality is always available.

32.1.6 DirectDraw

DirectDraw is the Windows rendering engine for 2D and 3D graphics. DirectDraw functions enable you to quickly compose images into front and back buffers, and to apply transformations, blitting, and flipping. The result is a capability of implementing smooth animation as required in computer games and other multimedia and high-performance graphics applications. DirectDraw functions can be used with images that originate in the Windows GDI, in Direct3D, or in OpenGL.

DirectDraw is implemented as an API layer that lies above the display hardware, as shown in Figure 32-2. It enables the graphics programmer to take advantage of the capabilities of graphics accelerators and coprocessors in a device-independent manner. DirectDraw is a COM-based interface.

The following are the most important connections between DirectDraw and Direct3D:

- IDirect3D, the interface to Direct3D, is obtained from a DirectDraw object by calling the QueryInterface() method.
- Direct3DDevice, the low-level interface to the 3D renderer, is similar to IDirectDrawSurface and is created by querying IDirectDrawSurface for a 3D device GUID. The 3D renderer will also render to a 2D surface and recognizes all DirectDraw 2D functions.
- IDirect3DTexture, the texture manager in Direct3D, is an extension of IDirectDrawSurface and is created by querying IDirectDrawSurface for an IID_IDirect3DTexture interface. Code can access all DirectDraw surface functions on a 2D surface.
- A Direct3D z-buffer is a DirectDraw surface created with the DDSCAPS_ZBUFFER flag. Code can use DirectDraw 2D functions in relation to z-buffers. Z-buffers are discussed later in this chapter.

32.1.7 OpenGL

OpenGL is an alternative 3D development environment that originated in graphics workstations. Its main area of application is in programs that require precise 3D image rendering, such as CAD/CAM, technical modeling and animation, simulations, scientific visualization, and others. OpenGL is part of Windows NT and is available for Windows 95 and 98. When installed, the system can execute programs that use the OpenGL APIs. Because of its high level, OpenGL appears to the programmer as an alternative to Direct3D retained mode.

32.1.8 Direct3D and COM

Like DirectDraw, Direct3D is based on Microsoft's Component Object Model (COM). COM is an object-oriented system that exists at the operating system level. In COM an interface is a group of related methods. COM's main purpose is to support and promote the reuse of interfaces. Direct3D is presented to the programmer using the Component Object Model. The COM object is a data structure that contains a pointer to the associated methods. Because it is not specific to C++, a program written in C, or even in a non-C development system, can use APIs based on the COM protocol.

There are several ways of accessing the COM interface. In C++ the COM object appears like an abstract class. In this case access is by means of the pointer to the DirectDraw COM object, which then allows code to obtain the Direct3D COM object. When programming in C the function must pass the pointer to the COM object as an additional parameter. In addition, the call must include a pointer to a property of the COM object called the vtable. In this book we use the simpler, C++ interface to the COM.

32.2 Direct3D Rendering

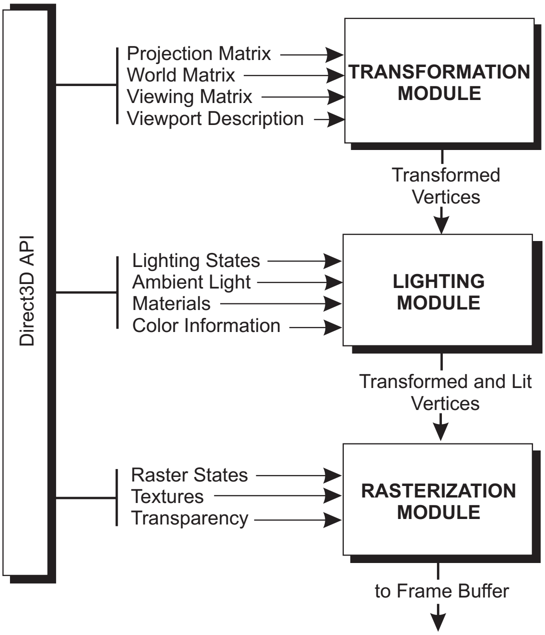
Direct3D uses a 3D rendering engine composed of three separate modules:

- Transformation Module. This module consists of four modifiable state registers: viewport, viewing matrix, world matrix, and projection matrix. It supports arbitrary projection matrices, and allows perspective and orthographic views. As the name implies, the transformation module handles the geometrical transformations. It is also called the Geometry Module.
- Lighting Module. This module calculates lighting and color information. It uses a stack-like structure to maintain a record of the current lights. It supports ambient, directional, point, and spotlight light sources and two lighting models: monochromatic and RGB.
- Rasterization Module. This module uses the output of the transformation and lighting modules to render the scene. The rasterization module is the 3D renderer in Direct3D. The scene description is based on an extensible display-list that supports both 2D and 3D primitives. Raster options such as wireframe, solid fill, and texture map are defined in this module.

Figure 32-3 shows the modules of the Direct3D rendering engine and their interaction with the other modules and with the rest of the system.

Figure 32-3 *Direct3D Rendering Modules*

Together, these three modules form the Direct3D rendering pipeline. Direct3D is furnished with one transformation module and a choice of two lighting and two



rasterization modules. This ensures greater flexibility in lighting and rendering. For example, a scene can be rendered more realistically by switching the lighting module.

32.2.1 Transformation Module

The transformation module has four state registers: the viewport, the viewing matrix, the world matrix, and the projection matrix. All four are modifiable by code. Whenever one or more of the state registers are modified, they are recombined to form a new transformation matrix. The transformation matrix defines the rotation and projection of a set of 3D vertices.

In Direct3D a display list is the name given to a set of 3D commands. The transformation module supports a number of different vertex types in the display list. The D3DTLVERTEX structure is a transformed and lit vertex that contains screen coordinates and colors. This structure contains the data and color information that is used by the lighting module. The D3DLVERTEX structure is used when the model contains data and color information only. Alternatively, the D3DHVERTEX structure is used when the application uses model-coordinate data with clipping. When this structure is used the transformations are performed in hardware. The D3DVERTEX structure is used if the hardware supports lighting. This type of vertex can be transformed and lit during rendering.

The transformation module contains two different types of methods: those that set the state and those that use the transformation module directly to act on a set of vertices. The second type of method is useful for testing bounding volumes or for acting on a set of vectors. These operations are based on the current transformation matrices. The structure used for all the direct transformation functions is D3DTRANSFORMDATA. Geometrical transformations were discussed in Chapter 3 and are revisited later in this chapter.

32.2.2 Lighting Module

The lighting module maintains a stack-like structure representing the current lights, as well as the ambient light level, and a material. When the module is used directly, the input data includes a direction vector. If the light source is positional, as is the case with point- and spotlights, then the input also contains light source position information.

The monochromatic lighting model calculates the value for each light in a shade of gray. It is also called the ramp model. The RGB model uses the color component of light sources in order to produce more realistic and pleasant results. Internal color representations are always based on a palette-based color ramp.

In the ramp mode each color is represented by an index into a look-up table that can be located either in hardware or in software. Ramp modes use either 8- or 16-bit indices. In the ramp mode the lighting module has no knowledge of the particular color; it just works with a number of shades. Because color lights are treated as white lights, the ramp mode is sometimes called the monochromatic mode. The pre-calculated color ramps are divided into two sections. The first three-quarters of the ramp are the material's diffuse color. The values of this portion of the ramp range from the ambient color to the maximum diffuse color. The last quarter of the precalculated ramp encode the maximum diffuse color to the maximum specular color of the material. At rendering time the shade value is scaled by the size of the ramp and used as an index into the look-up table.

If the material does not have a specular component, the shade is calculated using the diffuse component of the light intensity. In this case the value ranges from 0 (ambient light) to 1 (full intensity light). If the material has a specular component, then the shade calculation combines both the specular and diffuse components of the light according to the following equation

where s is the shade value, d is the diffusion, and sp is the specular value of the light.

$$s = (\tfrac{3}{4}d \times (1 - s)) + sp$$

Notice that the first term of the equation takes into account the first three quarters of the ramp, which is equivalent to the material's diffuse color. The second term takes into account the last quarter of the ramp, which corresponds with the material's diffuse or specular color value.

Whether you use the RGB or the ramp color model depends mostly on the capabilities of the hardware. Ramp color is faster in software, but the RGB model supports color lights and is as fast, or even faster, than the ramp model if there is a hardware rasterizer.

32.2.3 Rasterization Module

The rasterization module is the one that draws the triangles, lines, and points to the frame buffer. It responds only to execute calls. Instructions stored in the execute buffer determine the mode of operation of the rasterization module.

Execute buffers is just another name for display lists. They consist of self-contained, independent packets of information. The execute buffer contains a vertex list followed by an instruction stream. The instruction stream consists of individual instructions, each one containing an operation code (opcode), followed by the data. The instructions determine how the vertex list is lit and rendered. One of the most common instructions is a triangle list, which consists of a list of triangle primitives that reference vertices in the vertex list.

The size of the execute buffer is determined by the hardware. Usually, 64K is considered satisfactory. How caching is implemented by the video card influences the best size for the buffer. The GetCaps() method can be used to retrieve the buffer size.

In processing execute buffers the transformation module runs through the vertex list, generating the transformed vertices. If clipping is enabled, the corresponding clipping information is attached. If there is no vertex in view at this point, the entire buffer can be rejected. Otherwise, vertices are processed by the lighting module, which adds color to them according to the lighting instructions in the execute buffer. Finally, the rasterization module parses the instruction stream. Primitives are rendered based on the generated vertex information.

The only geometric types that can be processed by the rasterization module are triangles. The screen coordinates range from (0, 0) for the top left of the screen or window device to width –1, height –1 for the bottom right of the device. The depth values range from zero at the front of the viewing frustum to one at the back. Rasterization is performed so that if two triangles that share two vertices are rendered, no pixel along the line joining the shared vertices is rendered twice. The rasterizer culls back facing triangles by determining the winding order of the three vertices of the triangle. Only those triangles whose vertices are traversed in a clockwise orientation are rendered.

32.3 Retained Mode Programming

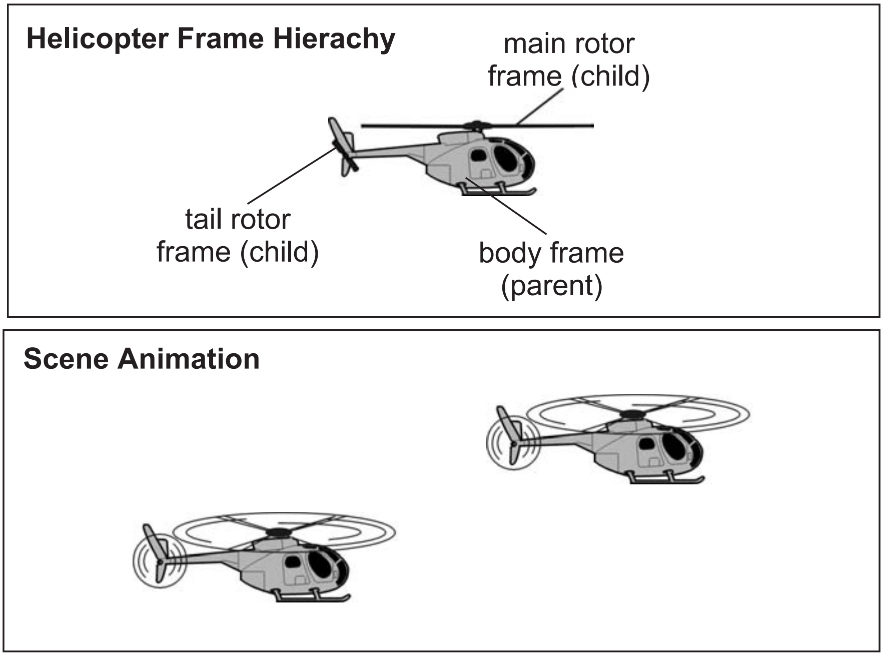
Retained mode programming consists of building 3D scenes out of components in Direct3D. The retained mode programmer does not need to be concerned with the development of geometrical primitives, or the structures of 3D objects and databases. You can load, rotate, scale, light, translate, and otherwise manipulate a 3D object, in real-time, using high-level API functions. In this section we discuss the core elements of Direct3D retained mode. These are the building blocks that we use in the following chapter to construct a Direct3D program.

32.3.1 Frames

A scene in Direct3D, sometimes called a scene graph, is a collection or hierarchy of frames. The term frame relates to the notion of a frame of reference. It should not be confused with that of single animation image, also called a frame. In retained mode the role of a frame is to serve as a container for 3D objects, such as polygon meshes, lights, and cameras. These objects have no meaning by themselves. For example, a cube cannot be rendered until it is assigned a position within a frame, relative to a light and a camera, and possibly a material, color, and texture.

Each scene contains a root or master frame and any number of child frames, each of which can have other children of its own. It is a tree-like structure in which the root frame has no parent frame and the leaf frames have no children. The root frame is the highest level element of a 3D scene. Child frames inherit their characteristics from the parent frames and are physically attached to the parent. When a frame is moved, all the objects attached to it, including its child frames, move with it. For example, a helicopter in a 3D scene may consist of several frames. One frame could model the body, another one the lift rotor blades, and a third one the steering rotor blades. In this case the rotor blades would be children frames to the helicopter body. The helicopter is made to fly by rotating the blades in the main and tail rotors and by translating the helicopter body frame. Because the rotors are child frames of the helicopter body frame, the entire machine moves as a unit. Figure 32-4 shows the frame hierarchy in this case.

Figure 32-4 *Frame Hierarchy in a Scene*



Frame hierarchies in Direct3D are not rigid. Functions are available that enable you to change the reference frame, regardless of the parent-child relationships originally established. This flexibility adds considerable power to retained mode.

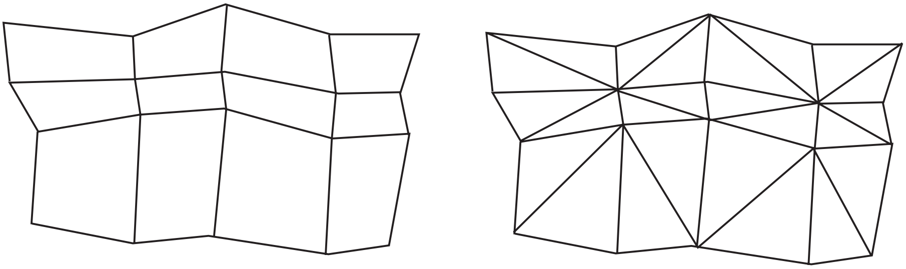
Meshes

The mesh is the principal visual object of a scene and the cornerstone of retained mode programming. Direct3D objects are made up of meshes. A mesh is described as a set of faces, each of which consists of a simple polygon. This makes a mesh equivalent to a set of polygons. Polygon meshes were discussed in Chapter 2.

The fundamental polygon type in Direct3D is the triangle. Retained mode applications can describe polygons with more than three vertices, but the system automatically translates them into triangles when rendering them. Immediate mode applications, on the other hand, are limited to triangles. Figure 32-5 shows two versions of the same mesh. The one at the top consists of 12 quadrilaterals. The one at the bottom is made up of 24 triangles.

The principal objection to modeling with nontriangular polygons is that in a polygon with more than three vertices it is possible for the vertices to lie on different planes. In addition, polygons with more than three vertices can be concave. The triangle is not only the simplest of polygons, but all the points in the surface of a triangular polygon must lie on the same plane and any line drawn from two points in a triangle is inside it. In other words, a figure defined by three vertices is coplanar and convex. The renderer requires that polygons are convex and coplanar, so triangular modeling facilitates rendering.

Figure 32-5 *Quadrilateral and Triangular Meshes*



Most graphics systems, including Direct3D, model objects by means of polygon meshes. Mesh information is stored in a database containing the vertices of each polygon and their attributes, such as color, texture, and shading. A state-of-the-art hardware-based renderer is capable of displaying hundreds of thousands to over one million of these polygons in 1 second, and at the same time applying texture, lighting and other effects.

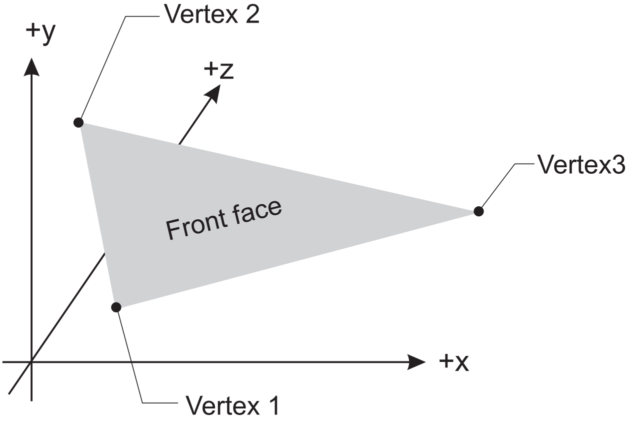
Mesh Groups

The mesh group is an organizational concept used by Direct3D immediate mode. A mesh group consists of a collection of polygons. Each group can have its own material, color, texture, and rendering quality. Groups have no names and are not supported in retained mode.

Faces

If a face is a polygon, and a mesh is a collection of faces, then building a mesh consists of building the individual faces of which it is composed. Each face is a set of vertices. If the face is a triangle, then it is defined by three vertices. A front face is one in which vertices are defined in clockwise order. Figure 32-6 shows the front face of a triangular polygon in the Direct3D's left-handed coordinate plane.

Figure 32-6 *Front Face of a Triangular Polygon*



Each face has a normal vector, perpendicular to the face. If the normal vector of a face is oriented toward the viewer, that side of the face is its front. In Direct3D, only the front side of a face is visible. For this reason, if the vertices of the polygon in Figure 32-7 had been defined in counterclockwise order, the polygon's face would not be visible at rendering time. Face normals are used in Direct3D flat shading mode. Vertex normals are used in Phong and Gouraud shading. Figure 32-7 shows the face and vertex normals of a pyramidal object modeled with triangular polygons.

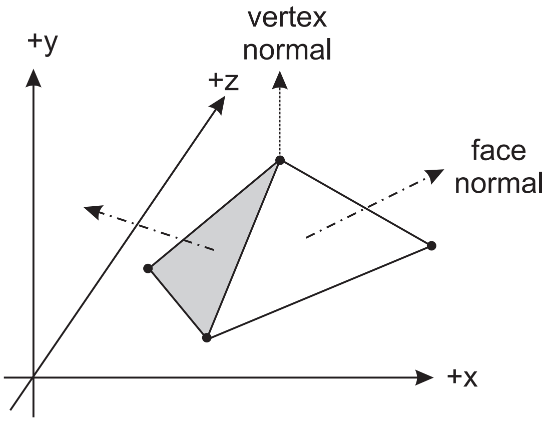


Figure 32-7 *Vertex Normals and Face Normals in a Pyramid*

32.3.2 Shading Modes

Direct3D documents three shading modes: flat, Gouraud, and Phong shading, but Phong is not currently supported. These shading algorithms were described in Chapter 4. In the flat shading mode the color of the first vertex of the polygon is duplicated across all the pixels on the object's faces. The result is that each face is rendered in a single color. Often the only way of improving the rendering is by increasing the number of polygons, which can be computationally expensive. An improvement to flat shading is called interpolative or incremental shading. In this case each polygon is rendered in more than one shade by interpolating, for the polygon interior points, the values calculated for the vertices or the edges. This type of shading algorithm is capable of producing a more satisfactory shade rendering with a smaller number of polygons. Direct3D describes two incremental shading methods, called Gouraud and Phong shading. Phong is not yet supported.

In the Gouraud and Phong shade modes, vertex normals are used to give a more satisfactory appearance to a polygonal object. In Gouraud shading, the color and intensity of the polygon edges are interpolated across the polygon face (see Figure 4-30). In Phong shading, the system calculates the appropriate shade value for each pixel. Because Gouraud shading is based on the intensity at the edges, it is possible to completely miss a highlight or a spotlight that is contained within a face. Figure 32-8 shows two possible cases in which Gouraud shading renders erroneously.

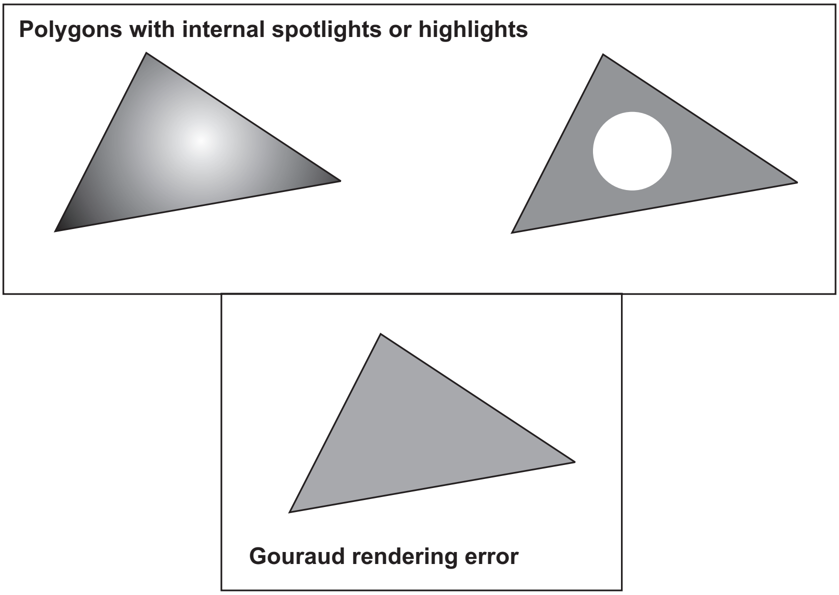


Figure 32-8 *Error in Gouraud Rendering*

Phong shading is the most effective shading algorithm in use today. This method, developed by Phong Bui-Toung is also called normal-vector interpolation. It is based on calculating pixel intensities by means of the approximated normal vector at each pixel point in the polygon. Phong shading improves the rendering of bright points and highlights that are misrendered in Gouraud shading. The one objection to Phong shading is that it takes considerably longer than Gouraud shading.

Interpolation of Triangle Attributes

At rendering time Direct3D interpolates the attributes of a triangle's vertices across the triangle face. Color, specular reflection, fog, and alpha blending attributes are interpolated. In interpolation the attributes are modified according to the current shade mode, as previously described. The interpolation of color and specular attributes depends on the color model. In the RGB model the red, green, and blue color components are used in the interpolation. In the monochromatic model only the blue component of the vertex color is taken into account. The alpha component of a color is treated as a separate interpolant. This is because device drivers can implement transparency in two different ways: by texture blending or by stippling.

32.3.3 Z-Buffers

One of the problems encountered by the renderer concerns the display of overlapping polygons. Figure 32-9, on the following page, shows three triangles located between the viewer and the display buffer. In this case the question is whether the pixel should be rendered as dark gray, white, or light gray. The answer is obviously dark gray because the dark gray polygon is the one closest to the viewer.

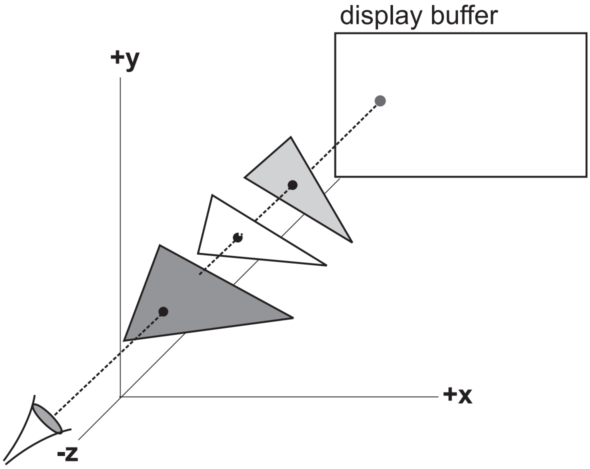


Figure 32-9 *Rendering Overlapping Triangles*

Several algorithms have been developed for eliminating hidden surfaces at rendering time. One of the best known, attributed to Catmull, is called the z-buffer or depth buffer method. Because of its simplicity of implementation and relative efficiency it has become popular in 3D graphics.

Direct3D supports the z-buffer method for solving the so-called "polygon-on-top" problem. In Direct3D the z-buffer is a rather large block of memory where the depth value for each screen pixel is stored. Initially the depth value for a pixel is that of the background. As each polygon is rendered, its depth value is examined. This is the z-order. If its depth value is less than the one in the z-buffer, then the pixel is rendered with the polygon's attribute. Otherwise it is ignored.

In Direct3D z-buffering can be turned on and off. The general rule is that z-buffering improves performance when a screen pixel is set several times in succession. The average number of times a pixel is written to is called the scene overdraw. Although overdraw is difficult to calculate exactly, it is possible to estimate it. If the scene overdraw is less than 2, then best performance is achieved by turning z-buffering off.

32.3.4 Lights

Earlier in this chapter we discussed the lighting module in Direct3D as well as the RGB and ramp color models. In processing lights the lighting module uses information about the light source, and the normal vectors of the polygon vertices, to determine how to render the light source in each pixel.

The vertex normals are calculated from the face normals of the triangles adjacent to that vertex. Face normals are perpendicular to the polygon face, as shown in Figure 32-10, on the following page. The angle between the vertex normals and the light source determines how much light intensity and color are applied to the vertex. The mathematical calculations are performed by the lighting module.

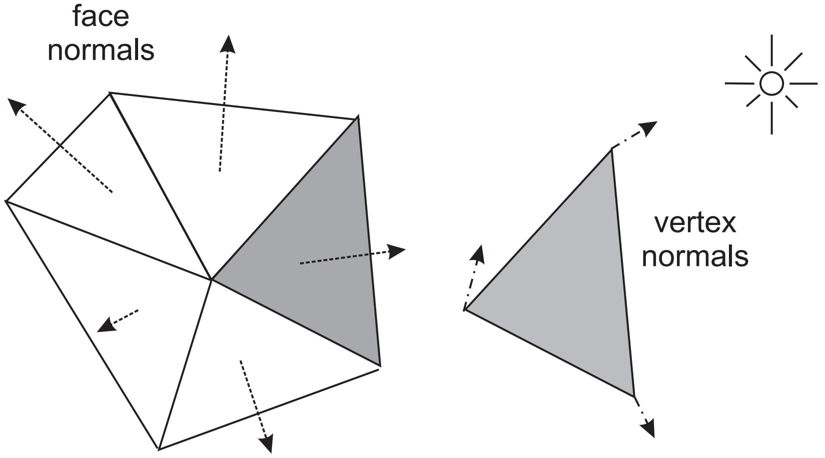


Figure 32-10 *Calculating the Vertex Normals*

Lighting effects are used to improve the visual quality of a scene. Applications can attach lights to a frame to represent a light source in a scene. In this case the attached light illuminates the objects in the scene. The position and orientation for the light is defined in the frame. Code can move and redirect a light source simply by moving and reorienting the frame to which the light is attached.

Retained mode supports five types of light sources:

- ambient
- directional
- parallel point
- point
- spotlight

Ambient Light

An ambient light source illuminates the entire scene, regardless of the orientation, position, and surface characteristics of the objects. All objects are illuminated with equal strength, therefore the position and orientation of the containing frame is inconsequential. Multiple ambient light sources can be combined within a scene.

Directional Light

A directional light source has a specific orientation, but no position. The light appears to illuminate all objects with equal intensity, as if it were at an infinite distance from the objects. Directional lighting is often used to simulate distant light sources, such as the sun. It provides maximum rendering speed.

Parallel Pint Light

The parallel point light can be considered a variation of direction light. In this case the orientation of the light is determined by the position of the light source. Whereas a directional light source has orientation, but no position, a parallel point light source has orientation and position. The parallel point light source has similar rendering-speed performance to the directional source.

Point Light

A point light source radiates light equally in all directions. This makes it necessary to calculate a new lighting vector for every face it illuminates, which makes the method more computationally expensive than a parallel point light source. One advantage of the point light source is that it produces a more faithful lighting effect. When visual fidelity is a concern, a point light source is the best option.

Spotlight

A spotlight is a cone-shaped light source with the light at the cone's vertex. Objects within the cone are illuminated. The cone produces light of two degrees of intensity, with a central brightly lit section called the umbra, and a surrounding dimly lit section called the penumbra. In Direct3D the angles of the umbra and penumbra can be individually specified. Figure 32-11 shows the umbra and the penumbra in spotlight illumination.

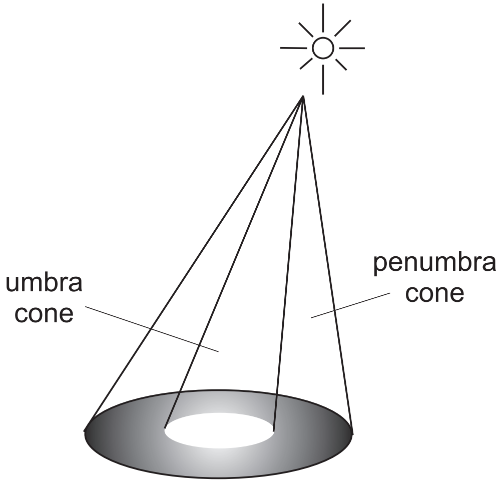


Figure 32-11 *Umbra and Penumbra in Spotlight Illumination*

32.3.5 Textures

A texture is an image, usually encoded in a 2D bitmap that can be applied to the face of a polygon to improve its visual quality. Color Figure 14 is a coffee cup to which a red marble texture has been applied.

Textures are usually stored in standard file formats, most commonly as a Windows bitmap, PCX or GIF. Although any image can be used as a texture, not all images make good textures. Textures can be scaled at the time they are applied. Each element of a texture is called a textel, which is a composite of the words texture and pixel.

In its simplest implementation, sometimes called point mapping, the rendering software looks up each pixel in a texture map and applies it to the corresponding screen pixel. In most cases point sampling produces coarse effects that are unnatural and disturbing to the viewer. Satisfactory texturing requires that the distance between the object and the viewer be taken into account at the time of applying the texture, in other words, that the texture be rendered perspectively.

The bilinear filtering method of texture rendering uses the weighted average of four texture pixels. This results in more pleasant textures than those that result from point mapping.

Direct3D supports five texture-rendering styles:

- Decals
- Texture colors
- Mipmaps
- Texture filters and blends
- Texture transparency

Decals

A decal is a texture applied directly to a scene. Decals are not rendered on a polygon face, but as an independent object. They are rectangular in shape, the rectangle facing the viewport, and they grow and shrink according to their distance from the viewer. The fact that decals always appear facing the viewer considerably limits their usefulness.

The origin point of a decal is defined as an offset from the top-left corner of the containing rectangle. The default origin is (0,0). In Direct3D an application can set and retrieve the origin of a decal. When the decal is rendered, its origin is aligned with its frame's position.

Texture colors

Direct3D code in retained mode can set and retrieve the number of colors that are used to render a texture. Applications that use the RGB color model can encode textures in 8-, 24-, and 32-bit formats. In the ramp color model textures are represented in 8 bits. However, code that uses the ramp model should be careful regarding the number of texture colors. In this mode each color requires its own lookup table. If an application uses hundreds of colors, the system must allocate and manipulate as many lookup tables.

Mipmaps

The term mipmap originates in the Latin expression multum in parvo, which can be translated literally as many in few or many objects in a small space. This texture-rendering method, sometimes referred to as MIP maps, was described by L. Williams in 1983 and has since gained considerable favor.

In Direct3D a mipmap is a set of textures representing the same image at progressively lower resolutions. Each image in the set is one-quarter the size of the preceding one, which makes the entire mipmap take up 4/3 the memory of the original image. Mipmapping provides a computationally efficient way of improving the quality of rendered textures. Each scaled image in the mipmap is called a level. The image at level 0 is at the same resolution as the original. Figure 32-12 is a diagram of the mipmap structure.

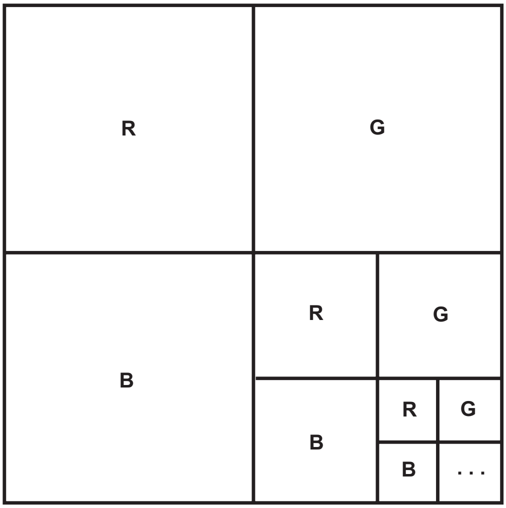


Figure 32-12 *Mipmap Structure*

Mipmaps are created by the DirectDraw interface. Each mipmap level contains its own front and back surfaces, which can be flipped in the conventional manner. When the mipmap is created, code defines the number of levels, as well as the dimensions of the level 0 mipmap. Figure 32-13 shows the DirectDraw structure of a mipmap consisting of 4 levels. In the DirectDraw implementation of mipmaps, each level consists of a front and a back surface. As is the case with all mipmaps, successive levels have one-half the resolution of the preceding one, and one-quarter the size.

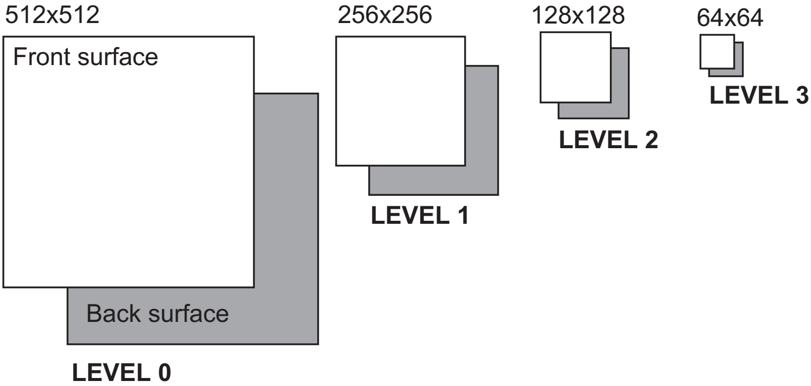


Figure 32-13 *Example of a DirectDraw Mipmap*

Texture Filters and Blends

The elements of a texture (texels) rarely correspond to individual pixels in the original image. Texture filtering is used to specify how to interpolate texels to pixels.

Direct3D supports six texture-filtering modes. They are:

- Nearest
- Linear
- Mip-nearest
- Mip-linear
- Linear-mip-nearest
- Linear-mip-linear

The nearest mode uses the texel with coordinates nearest to the desired pixel value. The result is a point filter with no mipmapping. The linear mode uses a weighted average of an area of 2-by-2 texels surrounding the desired pixel. This is equivalent to bilinear filtering with no mipmapping. In the mip-nearest mode the closest mipmap level is chosen and a point filter is applied. In the mip-linear mode the closest mipmap level is chosen and a bilinear filter is applied. The linear-mip-nearest mode uses the two closest mipmap levels, and a linear blend is used between point filtered samples of each level. In the linear-mip-linear mode the two closest mipmap levels are chosen and then combined using a bilinear filter.

Texture blending consists of combining the colors of a texture with the colors of the surface to which the texture is applied. If done correctly, the result is a translucent surface. Because texture blending can result in unexpected colors, the color white is often used for the material texture. There are a total of seven texture blending modes:

- Decal
- Modulate
- Decal-alpha
- Modulate-alpha
- Decal-mask
- Copy
- Add

Texture Transparency

Direct3D contains methods to directly produce transparent textures. In addition, immediate mode programs can take advantage of DirectDraw support for color keys to achieve transparency. By selecting a color key that contains a color or color range in the texture, the material's color will show through the texture areas within the color key range. The result is a transparent texture.

Wraps

In Direct3D a wrap is a way of applying a texture to a face or mesh. Four kinds of wraps are available:

- Flat
- Cylindrical
- Spherical
- Chrome

The flat wrap conforms to the faces of an object. The effect is sometimes compared to stretching a piece of rubber over the object. The cylindrical wrap treats the texture as if it were a sheet of paper wrapped around a cylinder. The left edge of the texture rectangle is joined to the right edge. The object is then placed in the middle of the cylinder and the texture is deformed inward onto the surface of the object. The spherical wrap is similar to the cylindrical wraps, but in this case the wrapping form is a sphere, instead of a cylinder. A chrome wrap allocates texture coordinates so that the texture appears to be reflected onto the objects. The chrome wrap takes the reference frame position and uses the vertex normals in the mesh to calculate reflected vectors, which are based on an imaginary sphere that surrounds the mesh. The resulting effect is that of the mesh reflecting whatever is wrapped on the sphere.

Texture wrapping is a complex procedure in which a two-dimensional surface is deformed to cover the surface of a three-dimensional object. The above analogies are coarse simplifications that do not take into account many of the complexities of wrapping. In practice, the results of wrapping operations are often different from what was expected. This has led some to believe that, in most cases, the complications do not justify the results. The reader interested in the more specific details on texture wrapping can refer to the article *Texture Wrapping Simplified* by Peter Donnelly that appears in Microsoft Developers Network documentation. The article includes a demonstration program for experimenting with texture wrapping operations.

32.3.6 Materials

Direct3D provides support for an object property called a material. A material determines how a surface reflects light. It has two components: an emissive property and a specular property. The emissive property determines whether the material emits light. This property is useful in modeling lamps, neon signs, or other light-emitting objects. The specular property determines if and how the material reflects light.

Code controls the emission property of a material by defining the red, green, and blue values for the emissive color. The specular property is also defined by the red, green, and blue values of the reflected light and by a power setting. The default specular color is white, but code can change it to any desired RGB value. The power setting determines the size, and consequently the sharpness, of the reflected highlights. A small highlight makes an object appear shiny or metallic. A large highlight gives a plastic appearance.

32.3.7 User Visuals

A user-visual object is an application-defined data structure that can be added to a scene and rendered, typically by means of a customized rendering module. For example, an application can add sound as a user-visual object in a scene, and then render the sound during playback. A user-visual object has no methods, but it does have a callback function that will be called by the renderer. The callback function is called twice: when the object is rendered and when the object is told to render itself. This property makes it possible for applications that execute in retained mode to use the user-visual mechanism to provide a hook into Direct3D immediate mode.

32.3.8 Viewports

The viewport contains a camera reference frame that determines which scene is to be rendered, as well as the viewing position and direction. Rendering takes place along the z-axis of the camera frame, assuming the conventional Direct3D Cartesian plane with the positive y-axis in the upward direction and the positive x-axis toward the right.

Viewing Frustum

What the camera sees from the vantage point of a particular frame is called the viewing frustum. The viewport uses a frame object as a camera. In the perspective viewing mode, the viewing frustum is a truncated pyramid with its apex at the camera position. The camera's viewing axis runs from the pyramid's apex to the center of the base, as shown in Figure 32-14.

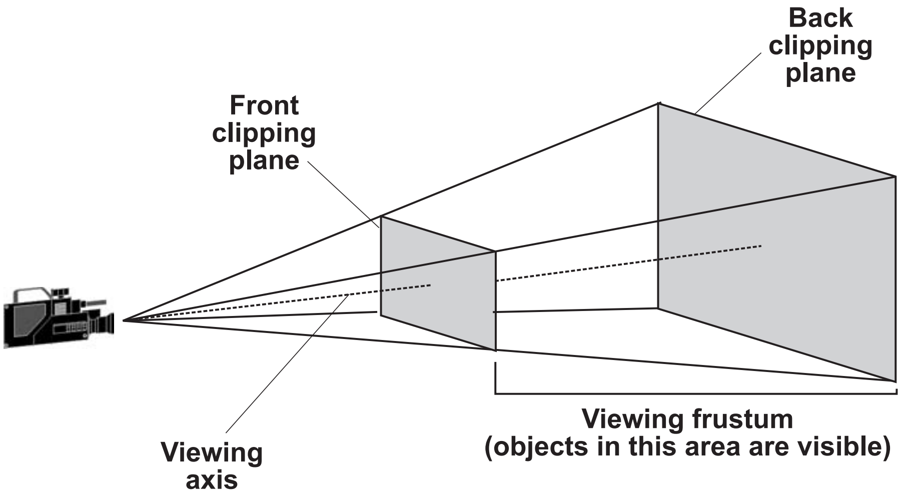


Figure 32-14 *The Viewing Frustum*

If we assume that the front clipping plane is at a distance D from the camera, and the back clipping plane is at a distance F from the front clipping plane, then the viewing angle A is determined by the formula

$$A = 2Tan^{-1}\frac{h}{D}$$

where h is one-half the height of the front clipping plane, if it is square. If the clipping plane is rectangular, then h is one half the height or the width of the front clipping plane, whichever is larger. The parameter h defines the field of view of the viewport. The above formula can be used to calculate the value of h for a specific camera angle. Figure 32-15 shows the viewport parameters.

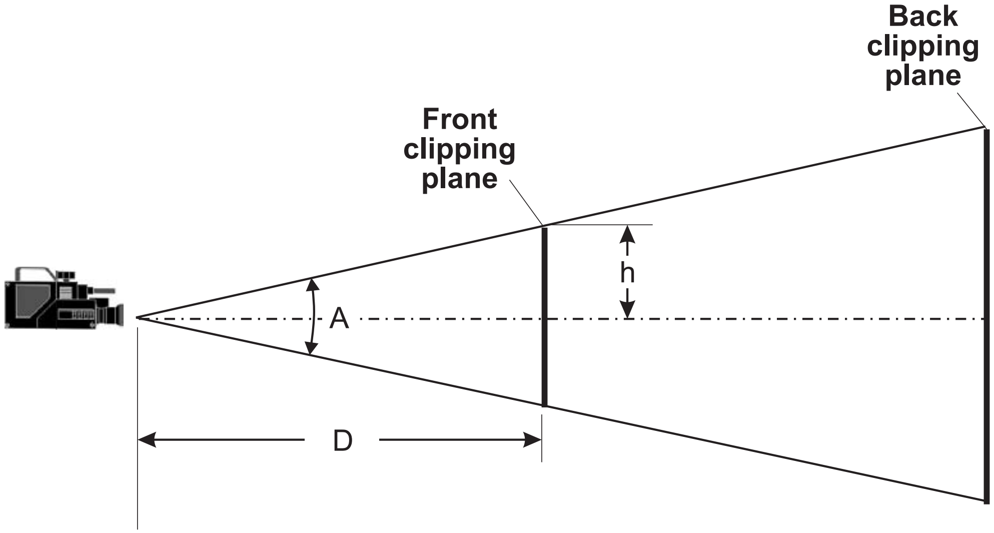


Figure 32-15 *Viewport Parameters*

Direct3D retained mode applications can set or retrieve the front and the back clipping planes, set the camera frame, as well as the viewport's field of view as defined by the parameter h in Figure 32-15. Direct3D supports two projection types: perspective and orthographic.

Transformations

In the context of Direct3D viewports, transformations are used to convert between screen and world coordinates. Direct3D transformations are based on homogenous coordinates as described in Chapter 3. The projection matrix, which is a combination of a scaling and a translation transformation, produces a four-element homogenous coordinate [x y z w]. The three-element homogeneous coordinates are derived by performing x/w, y/w, and z/w operations, where x/w and y/w are the coordinates to be used in the window and z/w is the depth. The depth ranges from 0 at the front clipping plane to 1 at the back clipping plane. The projection matrix is defined as follows:

$$\begin{bmatrix} D/hF & 0 & 0 & 0 \\ 0 & D/hF & 0 & 0 \\ 0 & 0 & F/(F-D) & 1 \\ 0 & 0 & (-F \times D)/(F-D) & 0 \end{bmatrix}$$

In the above matrix the parameters h and D are as in Figure 32-15.

Picking

Direct3D supports the selection of an object by specifying its location in the viewport. This operation, called picking, is typically based on the position of the mouse cursor. Picking is accurate to the pixel; therefore it can be used in precise object selection by technical applications. The drawback of the picking operation is that it involves considerable processing, which may introduce a visible delay in the rendering.

To pick an object, code passes the x and y screen coordinates to the corresponding method. Usually these coordinates are those of the mouse cursor at the time of the pick action. The pick function returns either the closest object in the scene, or a depth-sorted list, called the picked array, of the objects found at that location.

32.3.9 Animations and Animations Sets

In retained mode an animation provides a mechanism for adding behavior to a 3D scene. An animation set consists of one or more animations and a time reference.

An animation is defined by a set of keys, which consists of a time value, an associated scaling operation, an orientation, or a position. A Direct3D animation object defines how a transformation is modified according to the time value. The animation can be set to animate the position, orientation, and scaling of visuals, lights, or viewport objects.

Applications can define position keys, rotation keys, and scale keys in the animation. Each key references a time value in zero-based arbitrary units. For example, if an application adds a position key with a time value of 99, a new position key with a time value of 49 would occur halfway between the beginning of the animation and the first position key. An animation is driven by calling a method that sets its time component. This call sets the visual object's transformation to the interpolated position, orientation, and scale of the nearby keys. As with the methods that add animation keys, the time value is arbitrary and based on the positions of keys that the application has already added. Rotation keys in an animation are based on quaternions. Quaternions, a mathematical structure that facilitates rotation transformations, are discussed later in this chapter.

A Direct3D animation set allows animation objects to be grouped together. The result is that all the animations in the set share the same time parameter, which simplifies the playback of complex sequences. Applications can add and remove animations to and from an animation set.

32.3.10 Quaternions

Direct3D retained mode supports a mathematical structure called a quaternion that has found use in 3D animation. The quaternion is described as an extension to complex numbers that describes both an orientation and a rotation in 3D space. In Direct3D the quaternion consists of a vector that provides the orientation component and a scalar, that defines the rotation component. This can be expressed as

$$q = (s, v)$$

where s is the rotation scalar of the quaternion and v is the orientation vector.

Quaternions provide a fast alternative to the matrix operations that are typically used for 3D rotations. The quaternion can be visualized as an axis in 3D space, represented by a vector, and a rotation around that axis, represented by a scalar, as shown in Figure 32-16.

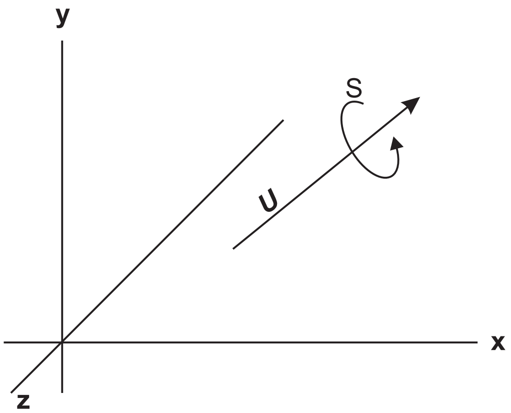


Figure 32-16 *Vector/Scalar Interpretation of the Quaternion*

Two fundamental operations can be performed on quaternions: composition and interpolation. Composition consists of combining quaternions. For example, the composition of two quaternions, q1 and q2, in reference to an object in 3D space, is interpreted to mean: rotate the object on the specified axis, by the rotation contained in quaternion q1, and then rotate the object on the specified axis by the rotation contained in quaternion q2. Quaternion interpolation is used to calculate a smooth path from one axis and orientation to another.

A common problem in computer animation is the generation of in-between frames that are necessary to simulate the smooth movement of an object from one position to another one. For example, Figure 32-25 shows images of an F-111 jet. The images at the top, called the key frames, represent the initial and final position of the aircraft in a planned animation. To simulate this movement, it is necessary to generate a set of in-between images that produce a smooth transition from the start frame to the end frame. Part of this image set, usually called the in-betweens, are shown in the lower part of Figure 32-17.

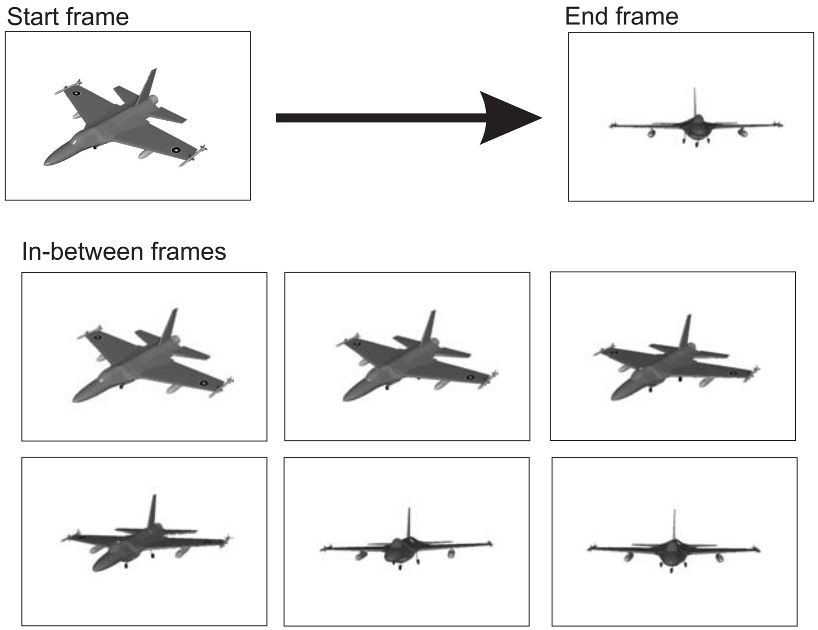


Figure 32-17 *In-Between Frames in Animation*

Rendering the in-between frames in the case of Figure 32-17 consists of performing several rotations on the image data for the F-111 aircraft. Aircraft dynamics uses three angles: the yaw refers to the vertical axis, the pitch to a horizontal axis through the wings, and the roll through the fuselage axis. These angles are shown in Figure 32-18.

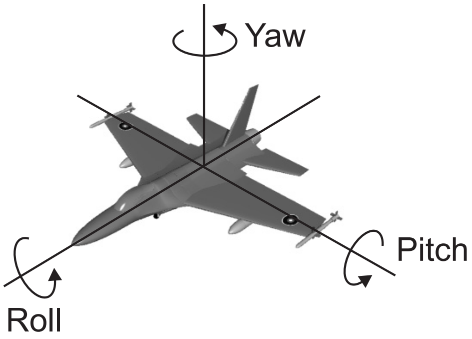


Figure 32-18 *Aircraft Dynamic Angles*

Generating the animation image set in Figure 32-18 requires rotating the aircraft along its yaw, pitch, and roll angles. Traditionally, rotations of 3D models have been by means of independent coordinates called Euler angles. This approach, although feasible, is computationally expensive because the composite rotation is based on three individual rotations along the axes.

Quaternions provide a way of changing the orientation of the aircraft by performing a single rotation, that is a composite of the three primary ones along the yaw, pitch, and roll angles. This is achieved by using composition and interpolation together. A composition is first used to go from the original to the final frame. The smooth transition from the start frame to the end frame is then achieved through interpolation. In Direct3D programming code determines an angle, called the slerp value, that defines the position for the intermediate quaternion between two vectors. For example, a slerp value of 0.5 creates a quaternion that is midway between the two input quaternions. The quaternion method provides a much simpler and computationally faster approach to calculating in-between images for animation.

32.4 Direct3D File Formats

The information that defines a 3D object must be stored in a special file format. You cannot use the conventional BMP, GIF, or TIFF file types developed for 2D bitmaps for a 3D image, although a 3D application may be capable of rendering a particular view of a 3D object into one of the 2D file formats. Several 3D formats have been developed for the PC; in fact, it seems that every 3D drawing program supports its own proprietary format. What is worse, file conversion utilities that are relatively abundant for 2D imagery are difficult to develop for the 3D formats and, consequently, not always available.

Some of the 3D file formats have gained some level of prominence, usually proportional to the muscle of its corporate sponsors. One of the first PC programs that effectively used 3D was AutoCAD, a computer-assisted design application that enjoys a lion's share of this market. The .dxf file format was designed by AutoCAD primarily for the CAD environment. Its image handling capabilities are confined to 3D faces and polylines, which makes it quite crude for 3D modeling and authoring applications. However, these limitations also imply an inherent simplicity and ease of implementation, which have made the .dxf format quite popular. In many cases the only way of moving image data between two 3D applications is by means of a .dxf file, although the results usually leave much to be desired.

One of the leading 3D image editing programs is 3D Studio. The current version is named 3D Studio MAX. The native file format for 3D Studio, named .3ds, comes close to being the industry standard at the present time. Microsoft recognized this hegemony by providing a utility, named conv3ds, that converts 3ds files into the format supported by Direct3D.

Direct3D supports a single file format called .x. It is used to store objects, meshes, textures, and animation sets. It also supports animation sets, which allow playback in real-time. The .x format supports multiple references to a single object (such as a mesh) while storing the data for the object only once per file. Earlier versions of Direct3D used a file format named .xof, which is now considered obsolete. Direct3D retained mode uses the .x format for loading objects into an application and for writing mesh information, constructed by the application, in real-time.

34.4.1 Description

The DirectX file format is a template-driven structure that allows the storage of user-defined object, meshes, textures, animations, and animation sets. The format supports multiple references to an object, such as a mesh. Multiple references allow storing data only once per file. The format provides low-level data primitives as well as application-defined higher level primitives via templates. The higher level primitives include vectors, meshes, matrices and colors.

34.4.2 File Format Architecture

The DirectX file format is context-free. Its template-driven architecture does not depend on any usage knowledge. The format is used in Direct3D retained mode to describe geometry data, frame hierarchies and animations.

Reserved Words

The following words are reserved for use by the DirectX format:

- ARRAY
- BYTE
- CHAR
- CSTRING
- DOUBLE
- DWORD
- FLOAT
- STRING
- TEMPLATE
- UCHAR
- UNICODE
- WORD

Header

The variable length header, which is compulsory, must be located at the beginning of the data stream. Table 32-1, on the following page, lists the elements in the DirectX file header.For example, the header

```
xof 0302txt
```

corresponds to a file in text format. The code "xof" refers to the old extension for the .x format and, when found in the header, indicates an .x file. The digits 0302 correspond to the .x format version number, in this case 3.2. The digits 64 indicate that floating-point numbers are encoded in 64-bit. Because no compression code is listed, the file is not compressed.

The header

```
xof 0302bin 0032
```

corresponds to an .x file in binary format, version 3.2, in which floating-point numbers are encoded in 32-bits, uncompressed.

Table 32-1

DirectX File Header

TYPE	SUB TYPE	SIZE	CONTENTS	CONTENT MEANING
Magic number (required)				
		4 bytes	"xof"	
Version number (required)				
	Major number	2 bytes	03	Major version 3
	Minor number	2 bytes	02	Minor version 2
Format type (required)				
		4 bytes	"txt "	Text file
			"bin "	Binary file
			"com "	Compressed file
				Compression type: required
If format is compressed				
		4 bytes	"lzw "	
			"zip "	
			etc...	
Float size (required)				
		4 bytes	0064	64-bit floats
			0032	32-bit floats

Comments

Comments, which are only applicable in text files, may occur anywhere in the data stream. A comment begins with either double slashes "//", or a hash character "#". The comment runs to the next newline.

```
# This is a comment.
// This is also a comment.
```

Templates

Templates are the basic element of the .x file format. A template contains no data but defines the type and order of the data objects in the file. A template is similar to a structure definition. The general template format is as follows:

```
template <template-name> {
<UUID>
<member 1>;
<member 2>;
...
<member n>;
[open/close/restricted]
[...]
}
```

The template name is a string that must not begin with a digit. The underscore character is allowed. UUID is the Windows universally unique identifier in OSF DCE format. The UUID is surrounded by angle brackets. The template members describe the data elements to which the template refers. The member format is as follows:

```
<data-type> <name>;
```

The primitive data types are listed in Table 32-2.

Table 32-2

Primitive Data Types for the .x File Format

TYPE	SIZE
WORD	16 bits
DWORD	32 bits
FLOAT	IEEE float
DOUBLE	64 bits
CHAR	8 bits
UCHAR	8 bits
BYTE	8 bits
STRING	NULL terminated string
CSTRING	Formatted C-string (currently unsupported)
UNICODE	Unicode string (currently unsupported)

The template can contain any valid data type as an array. In this case the syntax is

```
array <data-type> <name>[<dimension-size>];
```

where <dimension-size> can be either an integer or a named reference to another template member whose value is then substituted. Arrays may be n-dimensional. In this case n is determined by the number of paired square brackets trailing the statement. For example:

```
array DWORD FixedArray[24];
array DWORD VariableArray[nElements];
array FLOAT Matrix8x8[8][8];
```

Templates may be open, closed, or restricted. These elements determine which data types may appear in the immediate hierarchy of a data object. An open template has no restrictions, a closed template rejects all data types, and a restricted template allows a named list of data types.

Data

The actual data of the .x file is contained in the data objects. Data objects are formatted as follows:

```
<Identifier> [name] {
<member 1>;
...
<member n>;
}
```

The Identifier element is compulsory and must match a previously defined data type or primitive. The name element is optional. The data members can be a data object, which can be nested, a data reference to a previous data object, an integer, float, or string list, in which the individual elements are separated by semicolons.

Retained mode templates

The following templates are used by Direct3D retained mode:

- Template Name: Header
- Template Name: Vector
- Template Name: Coords2d
- Template Name: Quaternion
- Template Name: Matrix4x4
- Template Name: ColorRGBA
- Template Name: ColorRGB
- Template Name: Indexed Color
- Template Name: Boolean
- Template Name: Boolean2d
- Template Name: Material
- Template Name: TextureFilename
- Template Name: MeshFace
- Template Name: MeshFaceWraps
- Template Name: MeshTextureCoords
- Template Name: MeshNormals
- Template Name: MeshVertexColors
- Template Name: MeshMaterialList
- Template Name: Mesh
- Template Name: FrameTransformMatrix
- Template Name: Frame
- Template Name: FloatKeys
- Template Name: TimedFloatKeys
- Template Name: AnimationKey
- Template Name: AnimationOptions
- Template Name: Animation
- Template Name: AnimationSet

Chapter 33

Direct3D Programming

Topics:

- Creating a Direct3D program
- Creating the objects
- Building the scene
- Rendering the scene
- Direct3D retained mode sample program
- Windowed retained mode coding template

We introduce Direct3D retained mode by developing a simple, windowed mode application. In order to make clear the fundamentals of retained mode programming we have stripped off everything that is not essential. The result is that the processing described at this stage has minimal functionality: all we do in the code is render a static image from a file in DirectX format. The code executes by performing four clearly distinct steps:

- Initializing the software interface. That is, creating the Direct3D and DirectDraw components that are necessary to the program.
- Creating the objects. This implies creating the frames, meshes, lights, materials, and other object that serve as parts of the scene.
- Building the scene from the individual component objects.
- Rendering the scene. In this step the viewport is cleared and the frame is rendered.

Each of these steps is explained in detail and packaged in its own function. All of the coding comes together in the sample project 3DRM InWin Demo1 which is furnished in the book's software package. We also include in this chapter a coding template for windowed retained mode programming.

33.1 Initializing the Software Interface

Direct3D, as its parent DirectX, uses the Component Object Model (COM) interface specification defined by Microsoft. COM is a standard for a component-based architecture that aims at being language independent, reusable, upgradable, and transparent to application code. Whether you like or dislike COM, in Direct3D programming you have no other option than to use it.

33.1.1 The IUnknown Interface

COM interfaces are derived from a general interface called IUnknown. All other COM interfaces inherit from IUnknown, therefore IUnknown methods are always polymorphically visible to COM client code. This means that any object instantiated as a COM object has access to the methods of IUnknown. There are three relevant methods in IUnknown:

- The QueryInterface() method interrogates the object about the features it supports. If the call is successful, it returns a pointer to the interface.
- AddRef() increments the object's reference count by 1 when an interface or another application binds itself to the object. Application code rarely uses this function.
- Release() decrements the object's reference count by 1. When the count reaches 0, the object is deallocated.

The reference count is a memory management technique that enables components to self-destroy. It is based on keeping a tally of the number of interfaces allocated to a COM object. Each time an interface is allocated, the reference count is incremented. When client code is finished using an interface it decrements the reference count by calling the Release() method. If at any time the reference count goes to zero, the interface object deletes itself. The AddRef() method is normally called by the function, while the Release() method is called by your code. When QueryInterface() successfully returns a pointer to an interface, it implicitly calls AddRef to increment the reference count. This means that your application must call the Release() method before destroying the pointer to the interface.

33.1.2 Direct3DRM Object

The word "object" in the context of Direct3D is not directly related to object orientation. When you hear the word object in the context of Direct3D you should not interpret it as an "instance of a class," but in its generic and more conventional sense. Textures, cameras, viewports, meshes, and many other elements of Direct3D are loosely refered to as "objects". The common superclass of all these objects is the Direct3DRMObject. Direct3DRMObject is instantiated as a COM object and can, therefore, access the methods of the IUnknown interface.

Before an application can create the Direct3DRNObject it must first instantiate a Direct3D retained mode object. This is achieved by calling Direct3DRMCreate(). The function's general form is as follows:

```
HRESULT Direct3DRMCreate(
          LPIRECT3DRM FAR * lplpD3DRM   // Address of interface
          );
```

The function returns D3DRM_OK if it succeeds. In this case the pointer is valid and can be used to access the interface. Any other return value indicates that the function failed and that the pointer is invalid.

33.1.3 Calling QueryInterface()

The pointer returned by Direct3DRMCreate() is a COM object and can therefore access the IUnknown methods. Of these methods, QueryInterface() is the one usually called first, since it provides information regarding whether a particular COM interface is supported. The function's general form is as follows:

```
HRESULT QueryInterface(
                       REFIID riid,        // 1
                       LPVOID* obp         // 2
                       );
```

The first parameter is the reference to the unique identifier for the particular interface. It is sometimes called the interface identifier, or IID. In DirectX programming this parameter is passed to the call as a predefined constant. For example, in the DD Info Demo program developed previously, we used cascaded calls to QueryInterface() using different IIDs in order to determine the most recent version of DirectDraw supported by the system. Code is as follows:

```
   DDConnect = DirectDrawCreate ( NULL,
                &lpDD0,
                NULL);

   // Store pointer and continue if call succeeded
   if(DDConnect == DD_OK) {
        DDLevel = 1;        // Store level
        lpDD = lpDD0;       // copy pointer
   // Query the interface to determine most recent version
       DDConnect = lpDD0->QueryInterface(
                                      IID_IDirectDraw2,
                                      LPVOID *) &lpDD2);
   }
   if(DDConnect == S_OK){
        DDLevel = 2;           // Update level
        lpDD0->Release();      // Release old pointer
        lpDD = lpDD2;
        DDConnect = lpDD->QueryInterface(
                                      IID_IDirectDraw4,
                                      (LPVOID *) &lpDD4);
   }
   if(DDConnect == S_OK){
        DDLevel = 4;           // Update level
        lpDD2->Release();      // Release old pointer
        lpDD = lpDD4;
   }
```

Notice that in the above code the call to QueryInterface() is first made with the identifier IID_IDirectDraw2, then with IID_IDirectDraw4, to determine if either of these newer version of DirectDraw is available. In this case the call Returns S_OK if it succeeeds. If it fails QueryInterface returns E_NOINTERFACE or one of the following interface-specific error values listed in Table 33-1.

Table 33-1

Interface-Specific Error Values Returned by Queryinterface()

DIRECTX INTERFACE	RETURNS (COMMENT)
DirectDraw	DDERR_INVALIDOBJECT DDERR_INVALIDPARAMS DDERR_OUTOFMEMORY (IDirectDrawSurface2 only
DirectSound	DSERR_GENERIC (IDirectSound and IDirectSoundBuffer only) DSERR_INVALIDPARAM
DirectPlay	DPERR_INVALIDOBJECT DPERR_INVALIDPARAMS
Direct3D Retained Mode	D3DRM_OK (No error) DRMERR_BADALLOC (Out of memory) D3DRMERR_BADDEVICE (Device not compatible) D3DRMERR_BADFILE D3DRMERR_BADMAJORVERSION D3DRMERR_BADMINORVERSION D3DRMERR_BADOBJECT D3DRMERR_BADTYPE D3DRMERR_BADVALUE D3DRMERR_FACEUSED (Face already used in a mesh) D3DRMERR_FILENOTFOUND D3DRMERR_NOTFOUND (Object not found) D3DRMERR_UNABLETOEXECUTE

When the application is finished using the interface retrieved by a call to this method, it must call the Release() method for that interface to free it.

The COM provides two macros, named SUCCEEDED and FAILED, which are defined as follows:

```
#define SUCCEEDED(Status) ((HRESULT)(Status) >= 0)
#define FAILED(Status) ((HRESULT)(Status)<0)
```

These macros are a convenient way to check for the success or failure of any COM function without having to deal with the specific error codes. We frequently use these macros in our code samples.

In Direct3D retained mode programs the call to QueryInterface() uses the IID_IDirect3DRM identifier. The call requires a Direct3DRM object. Code usually releases the object after the interface has been validated since there is not further use for it. The following code fragment is from a function listed later in this chapter.

```
    // Create the Direct3DRM object.
    LPDIRECT3DRM   pD3DRMTemp;
    HRETURN           retval;
    ...
    retval = Direct3DRMCreate(&pD3DRMTemp);
    if (retval != D3DRM_OK)
    {
        // Display error message here
        return FALSE;
    }
    retval = pD3DRMTemp->QueryInterface(IID_IDirect3DRM3,
                               (void **)&lpD3DRM)))
    if(retval != D3DRM_OK)
    {
        pD3DRMTemp->Release();
        // Display error message here
        return FALSE;
    }
    // Release the object
    pD3DRMTemp->Release();
```

Creating the DirectDraw Clipper

We have mentioned that Direct3D is closely related to DirectDraw and uses much of its functionality. At this point we are interested in creating a DirectDraw clipper object that will determine which portion of the 3D scene is visible on the viewport. In a windowed mode application all we need to do is to create a DirectDraw clipper object and then to assign to it our application window as the clipping plane.

The DirectDraw clipper that we need for Direct3D must not be owned by a DirectDraw object. The DirectDraw API provide a function named DirectDrawCreateClipper for this purpose. The resulting objects are known as driver-independent DirectDrawClipper objects. Notice that the function DirectDrawCreateClipper() is not equivalent to IDirectDraw::CreateClipper, which creates a clipper owned by a specific DirectDraw object. The function's general form is as follows:

```
HRESULT DirectDrawCreateClipper(
          DWORD dwFlags,                              // 1
          LPDIRECTDRAWCLIPPER FAR *lplpDDClipper,     // 2
          IUnknown FAR *pUnkOuter                     // 3
          );
```

The first parameter is currently not implemented and must be set to zero. The second parameter is the address of a pointer to be filled in with the address of the new DirectDrawClipper object. The third parameter is provided for future COM features but at the present time must be set to NULL. The function returns DD_OK if successful, or one of the following error codes:

- DDERR_INVALIDPARAMS
- DDERR_OUTOFMEMORY

The object created by this function is not automatically released when an application's objects are released. They should be explicitly released by application code, although DirectDraw documentation states that they will be automatically released when the application terminates.

33.1.4 The Clip List

In the context of Direct3D windowed applications a clip list is a series of rectangles that delimit the visible areas of the surface. We have seen that a DirectDrawClipper object can be attached to any surface and that a window handle can be attached to a

DirectDrawClipper object. In this case DirectDraw updates the DirectDrawClipper clip list using the application window as a clipping plane. As the window changes, the clip list is updated.

The call to DirectDrawCreateClipper() creates the clipper but does not define the clip list. In order to do this, the application must use the pointer returned by DirectDrawCreateClipper() to call SetHWnd(). The function's general form is as follows:

```
HRESULT SetHWnd(
                DWORD dwFlags,              // 1
                HWND hWnd                   // 2
                );
```

The first parameter is currently not used and should be set to 0. The second parameter is the handle to the window that will be used as a clipping place. The call returns DD_OK if successful, or one of the following error codes:

- DDERR_INVALIDCLIPLIST
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_OUTOFMEMORY

The following code fragment creates a driver-independent DirectDrawClipper object and then attaches to it the current window as a clipping plane.

```
HRESULT         retval;
HWND            hwnd;
. . .
// Create a DirectDrawClipper object
    retval = DirectDrawCreateClipper(0, &lpDDClipper, NULL);
    if (retval != DD_OK)
    {
        // Display error message here
        return FALSE;
    }
    // Attach the program Window as a clipper
    retval = lpDDClipper->SetHWnd(0, hwnd);
    if (retval != DD_OK)
    {
        // Display error message here
        return FALSE;
    }
```

33.1.5 InitD3D() Function

The function InitD3D() in the 3DRM InWin Demo1 project, included in the book's software package, performs the processing operations described in this section. A slightly modified version of this function is included in the retained mode windowed coding template described later in this chapter. Follows a listing of this function.

```
//*******************************************************
// Name: InitD3D()
// Description: Initialize Direct3D interface
//*******************************************************
BOOL InitD3D(HWND hwnd)
```

```
{
    HRESULT retval; // Return value

    // Initialize the entire global variable structure to zero.
    memset(&globVars, 0, sizeof(globVars));

    // Create the Direct3DRM object.
    LPDIRECT3DRM pD3DRMTemp;
    retval = Direct3DRMCreate(&pD3DRMTemp);
    if (FAILED(retval))
    {
        D3DError("Failed to create Direct3DRM.");
        return FALSE;
    }
    if( FAILED( pD3DRMTemp->QueryInterface(IID_IDirect3DRM3,
                            (void **)&lpD3DRM) ) )
    {
        pD3DRMTemp->Release();
        D3DError("Direct3DRM3 interface not found" );
        return FALSE;
    }
    pD3DRMTemp->Release();

    // Create DirectDrawClipper object
    retval = DirectDrawCreateClipper(0, &lpDDClipper, NULL);
    if (FAILED(retval))
    {
        D3DError("Failed to create DirectDrawClipper object");
        return FALSE;
    }
    // Attach the program Window as a clipper
    retval = lpDDClipper->SetHWnd(0, hwnd);
    if (FAILED(retval))
    {
        D3DError("Failed to set the window handle");
        return FALSE;
    }
return TRUE;
}
```

33.2 Building the Objects

At this point in the code, the Direct3D retained mode, windowed application has performed the necessary initializations and is ready to start building the scene. In order to do this, code must first create the objects that are used in the scene. Before we tackle the details of object building there are a few housekeeping chores that need to be discussed.

To create the objects, and later, the scene itself, we need the pointer to returned in the second parameter to the QueryInterface() call discussed previously. In the code used in this chapter the pointer is publicly defined as follows:

```
LPDIRECT3DRM3 lpD3DRM = NULL;
```

By giving the pointer public visibility we are able to use it from several functions without having to pass it as a parameter in each call.

In addition to the basic Direct3D retained mode pointer (lpD3DRM) just mentioned, we also need pointers to the specific objects and devices. For example, to load a file in DirectX format into the scene we need to create a meshbuilder object using the CreateMeshBuilder() function that is available in the IDirect3DRM interface. The pointer of type LPDIRECT3DRM (stored in the named variable lpD3DRM in these examples) provide access to the interface services in IDirect3DRM. The CreateMeshBuilder() function takes as a parameter a variable of the type LPDIRECT3DMESHBUILDER3. The returned pointer is then used to access the Load() method. Other Direct3D objects, such as devices, scenes, cameras, lights, frames, materials, and meshes also require pointers to their specific interfaces. In the code samples that follow we require the following sub-set of interface-specific pointers:

- LPDIRECT3DRMDEVICE3
- LPDIRECT3DRMFRAME3
- LPDIRECT3DRMMESHBUILDER3
- LPDIRECT3DRMLIGHT
- LPDIRECT3DRMMATERIAL2

Sometimes the same pointer type is used for referencing different types of objects, for example, the type LPDIRECT3DRMFRAME3 is used to access a scene, a camera, a light, and a child frame.

Whether to make this pointers globally visible or not is mostly a matter of programming style. The most common guideline is that if the pointer will be required in several functions then it should be public. The problem with this rule is that at the time we are developing code it is often difficult to predict if a pointer will be required in other functions. Our excuse for abusing public variables in the code samples presented in this book excuse is that, today, wasting a few bytes of memory at runtime is not as important an issue as it was in the memory-starved systems of a few years ago.

Direct3D retained mode applications frequently manipulate several objects, such as frames, scenes, cameras, lights, and textures. In this case it is useful to create one or more structures that define the individual pointers and variables and then instantiate structure variables as required for different objects used in the code. An additional benefit of using structures is that all the variables in the structure can be cleared simultaneously by means of the memset buffer manipulation routine. The following global structure and variables are used in the sample code listed in this chapter and in the 3DRM InWin Demo1 program in the book's software package.

```
// Global variables
struct _globVars
{
    LPDIRECT3DRMDEVICE3   aDevice;   // Retained mode device
    LPDIRECT3DRMVIEWPORT2 aViewport; // Direct3DRM viewport
    LPDIRECT3DRMFRAME3    aScene;    // Master frame
    LPDIRECT3DRMFRAME3    aCamera;   // Camera frame
    BOOL                  isInitialized;  // All D3DRM objects
                                          // have been
                                          // been initialized.
} globVars;

LPDIRECT3DRM3 lpD3DRM = NULL;         // Direct3DRM object manager
LPDIRECTDRAWCLIPPER lpDDClipper = NULL;
                                     // DirectDrawClipper object
HWND             hWnd;
char             szXfile[] = "teapot.x" ;  // File to load
```

Notice that the template _globVars includes a boolean variable that keeps track of the application's initialization state, named isInitialized.

In addition to global variables Direct3D applications often require local ones, typically located inside the functions that perform object creation and scene building. As we will see later in this chapter, the variables used in creating objects and building a scene can have local lifetime, as long as the resulting master frame and its component object are global. In our code the master frame is stored in the global structure variable globVars.aScene, listed previously.

33.2.1 Creating the Objects

The following objects are needed to create a simple, Direct3D scene:

- A device
- A master scene frame
- A camera frame
- A viewport

The functions to be used in creating these object have in common that their names started with the word "create", for example, CreateDeviceFromClipper(), CreateFrame(), and CreateViewport. Once the objects are created they can be assembled into a master scene. A global variable, in this case the structure variable isInitialized, is used to record the fact that the master scene has been built.

Creating the Device

The term "device" in the context of Direct3D retained mode is equivalent to a "display device." It can be visualized as the video memory area to which the scene is rendered. In practice, a Direct3D device is always a DirectDraw surface. The viewport is a rectangular area within the device. We should also note that neither the device nor the viewport are equivalent to the video buffer, which is the area directly mapped to the display surface and shown on the screen.

In Direct3D the size of a device is defined when it is created and cannot be changed. In order to change the size of the device you must destroy the old device and create a new one with different dimensions. In Direct3D you can create a device from Direct3D objects, from a surface, or from a DirectDraw clipper. For the moment we will be concerned with this last method.

Since the size of the device must be defined at the time it is created, code needs to obtain the width and height of the client area. The GetClientRect() function can be used for this purpose. When the call returns, the bottom member of the RECT structure variable contains the height of the client area and the right member contains the width.

The CreateDevicFromClipper() function of IDirect3DRM2 interface allows creating a device from a DirectDraw clipper object. Previously in this chapter we called DirectDrawCreateClipper() and stored the resulting pointer in the variable lpDDClipper. This variable is now needed to create the device. CreateDeviceFromClipper() has the following general form:

```
HRESULT CreateDeviceFromClipper(
        LPDIRECTDRAWCLIPPER lpDDClipper,          // 1
        LPGUID lpGUID,                            // 2
        int width,                                // 3
        int height,                               // 4
        LPDIRECT3DRMDEVICE * lplpD3DRMDevice      // 5
        );
```

The first parameter is the address of the DirectDrawClipper object, mentioned in the preceding paragraph. The second parameter is a globally unique identifier (GUID). Normally, this parameter is set to NULL. This forces the system to search for a device with a default set of capabilities. This is the recommended way to create a device in retained mode programming, since the method always works, even if the user installs new hardware. Parameters 3 and 4 refer to the width and height of the device and usually correspond with the values obtained by the call to GetClientRect(). If the call succeeds, the fifth parameter will be filled with the address of a pointer to an IDirect3DRMDevice interface.

The call returns D3DRM_OK if successful, or an error otherwise.

The following code fragment shows creating a device using the CreateDeviceFromClipper() function

```
HWND     hwnd;        // Handle to the window
HRESULT  retval;      // Return value
RECT     rc;          // Storage for viewport dimensions
. . .

// Obtain size of client area
GetClientRect(hwnd, &rc);

retval = lpD3DRM->CreateDeviceFromClipper(lpDDClipper,
                           NULL,          // Default device
                           rc.right,
                           rc.bottom,
                           &globVars.aDevice);
if (FAILED(retval))
{
    // Display error message here
    return FALSE;
}
```

33.2.2 CreateObjects() Function

The function CreateObjects() in the 3DRM InWin Demo1 program, in the book's software package, performs the processing operations discussed in this section. Following is a code listing of this function.

```
//****************************************************************
// Name: CreateObjects()
// Description: Create the device and the scene objects
//****************************************************************
BOOL CreateObjects( HWND hwnd )
{
    // Local variables
    HRESULT retval;  // Return value
    RECT rc;         // Bounding rectangle for main window
    int width;       // Device's width
    int height;      // Device's height

    // Get client area dimensions
    GetClientRect(hwnd, &rc);
    // Create device from DirectDraw clipper
    retval = lpD3DRM->CreateDeviceFromClipper(lpDDClipper,
                       NULL, // Default aDevice
                       rc.right, rc.bottom,
                       &globVars.aDevice);
    if (FAILED(retval))
    {
        D3DError("Failed to create the D3DRM device");
        return FALSE;
    }

    // Create the master scene
    retval = lpD3DRM->CreateFrame(NULL, &globVars.aScene);
    if (FAILED(retval))
    {
        D3DError("Failed to create the master scene frame");
        return FALSE;
    }
    // Create the camera frame
    retval = lpD3DRM->CreateFrame(globVars.aScene,
                              &globVars.aCamera);
    if (FAILED(retval))
    {
        D3DError("Failed to create the camera frame");
        return FALSE;
    }

    // Create the Direct3DRM viewport using the device, the
    // camera frame, and the device's width and height.
    width = globVars.aDevice->GetWidth();
    height = globVars.aDevice->GetHeight();

    retval = lpD3DRM->CreateViewport(globVars.aDevice,
                                 globVars.aCamera, 0, 0,
                                 width, height,
                                 &globVars.aViewport);
    if (FAILED(retval))
    {
        globVars.isInitialized = FALSE;
        globVars.aDevice->Release();
        return FALSE;
    }
    // Create the scene
    if (!BuildScene(globVars.aDevice, globVars.aScene,
                    globVars.aCamera))
        return FALSE;

    // Record that global variables are initialized
    globVars.isInitialized = TRUE;
    return TRUE;
}
```

33.3 Master Scene Concepts

In Direct3D literature the notions of a scene and that of a frame sometimes overlap. A frame may have a parent frame from which it inherits all its attributes, even dynamic ones. For example, if a parent frame is rotating at a given rate, the resulting child frame rotates identically. A scene, on the other hand, is described as a frame with no parent. Some confusion results from the fact that you can create a scene (a frame with no parent) and later on associate it with a parent frame, at which time is ceases to be a scene and becomes a child frame. The CreateFrame() function of the IDirect3DRM2 interface is used for creating both frames and scenes. The function's general form is as follows:

```
HRESULT CreateFrame(
                    LPDIRECT3DRMFRAME lpD3DRMFrame,    // 1
                    LPDIRECT3DRMFRAME* lplpD3DRMFrame  // 2
                    );
```

The first parameter is the address of the frame that serves as a parent. If this parameter is NULL, then a scene is created. The second parameter is the variable that will be filled with a pointer to an IDirect3DRMFrame interface if the call succeeds.

The method returns D3DRM_OK if successful, or an error otherwise.

As previously discussed, we usually store the master scene in a global variable in order to make it visible throughout the code. The following code fragment shows the creation of a master scene.

```
// Create the master scene
retval = lpD3DRM->CreateFrame(NULL, &globVars.aScene);
if (FAILED(retval))
{
     // Display error message here
     return FALSE;
}
```

Notice that using NULL for the first parameter in the call to CreateFrame() ensures that the results are a scene, in other words, a frame with no parent.

33.3.1 The Camera Frame

In Direct3D retained mode the camera is implemented as a frame object. The camera frame determines the viewing position and direction, since the viewport renders only what is visible along the positive z-axis of the camera frame. In addition, the camera frame determines which scene is rendered. Later in this chapter we will set the camera's position. For now, we need to create the camera frame, which we do by means of the same CreateFrame() function that was used in creating the master scene in the previous section. The one difference is that the camera frame is a child frame of the master scene. Therefore, in this case, the first parameter passed to CreateFrame() refers to the master scene, and the second one to the camera frame. The following code fragment shows the processing.

```
// Create the camera frame
retval = lpD3DRM->CreateFrame(globVars.aScene,
                              &globVars.aCamera);
if (FAILED(retval))
{
    // Display error message here
    return FALSE;
}
```

33.3.2 The Viewport

The viewport defines the rectangular area into which the scene is rendered. In this sense the viewport can be described as a 2D construct that is used in rendering 3D objects. Here again we should keep in mind that they viewport is not the video buffer, and that rendering to the viewport does not display the image.

We have seen that the viewport uses the camera frame object to define which scene is rendered as well as the viewing position and direction. A viewport is defined in terms of its viewing frustum, as explained in Chapter 32. The viewport is created by calling the CreateViewport() function of the IDirect3DRM interface. The function's general from is as follows:

```
HRESULT CreateViewport(
              LPDIRECT3DRMDEVICE lpDev,                 // 1
              LPDIRECT3DRMFRAME lpCamera,               // 2
              DWORD dwXPos,                             // 3
              DWORD dwYPos,                             // 4
              DWORD dwWidth,                            // 5
              DWORD dwHeight,                           // 6
              LPDIRECT3DRMVIEWPORT* lplpD3DRMViewport   // 7
              );
```

The first parameter is the device on which the viewport is to be created. The second parameter is the camera frame that defines the position and direction of the viewport. Parameters 3 and 4 refer to the position of the viewport and parameters 5 and 6 to its dimension. All of these are expressed in device coordinates.

If the call succeeds, parameter 7 is the variable that will be filled with a pointer to an IDirect3DRMViewport interface. The call returns D3DRM_OK if successful, or an error otherwise.

The position of the viewport relative to the device frame is specific to the application's design and the proposed rendering operations. However, the size of the viewport must not be greater than that of the physical device, otherwise the call to CreateViewport() fails. To make sure that the viewport is not larger than the physical device we can use the GetWidth() and GetHeight() functions, of

IDirect3DRMDevice, to obtain the necessary dimensions. Note that the IDirect3DRMViewport interface also has GetWidth() and GetHeight() methods that retrieve the size of the viewport. At this time, since the viewport has not yet been created we must use the functions of IDirect3DRMDevice. The following code fragment shows obtaining the device size and then creating the viewport.

```
int        width;        // Storage for device size
int        height;
. . .
// Obtain device size and store in local variables
width = globVars.aDevice->GetWidth();
height = globVars.aDevice->GetHeight();
// Create the viewport
retval = lpD3DRM->CreateViewport(globVars.aDevice,
                                 globVars.aCamera,
                                 0, 0,
                                 width, height,
                                 &globVars.aViewport);
if (FAILED(retval))
{
     // Display error message here
     globVars.isInitialized = FALSE;
     globVars.aDevice->Release();
     return FALSE;
}
```

33.4 Master Scene Components

Once all the global objects have been built (in this case the device, the scene, the camera, and the viewport) we can proceed to build the master scene. In this example we assume that the mesh object is stored in a file in Directx format, and that it is located in the same directory as the executable code. In the case that we are following, the following steps are required:

- Creating a meshbuilder object and using it to load the mesh file
- Creating a child frame within the scene and adding the loaded mesh into the child frame
- Setting the camera position
- Creating the light frame
- Creating the lights used in illuminating the scene and attaching them to frames
- Creating a material and setting it in the mesh
- Setting the mesh color
- Releasing all local variables used in building the scene

In regards to this last step we must consider that in the process of building the master scene we create and use a host of Direct3D retained mode objects, such as meshes, cameras, lights, textures, and materials. Once the scene is created, the individual objects that were used in building it are no longer needed, since they have become part of the scene itself. For this reason, it is usually possible to limit the lifetime of these objects to the process of scene creation. This means that the pointers and variables required for creating the objects can have local scope and visibility. Also, that the individual objects can and should be released once they are incorporated into the scene.

33.4.1 The Meshbuilder Object

As its name implies, the meshbuilder component is a tool for building meshes. The meshbuilder itself cannot be rendered. In this chapter we use the meshbuilder object to load a mesh previously stored in a file in Directx format. The meshbuilder functions can be used to manually assemble 3D images. However, by far the most common way of creating images is by using a 3D image editor program, such as 3D Studio Max.

The first step is to create the meshbuilder object by means of the CreateMeshBuilder() function that is part of IDirect3DRM interface. The function has the following general form:

```
HRESULT CreateMeshBuilder(
        LPDIRECT3DRMMESHBUILDER* lplpD3DRMMeshBuilder  // 1
        );
```

The call's only parameter is the address of a pointer that is filled with the IDirect3DRMMeshBuilder interface if the call is successful. The function returns D3DRM_OK if it succeeds, or an error otherwise.

In the example that we are currently following we use the meshbuilder object's Load() a file in DirectX format. The file is loaded into the meshbuilder itself and takes the form of a mesh. Later in the code this mesh is stored in a frame. The Load() function has the following general form:

```
HRESULT Load(
         LPVOID lpvObjSource,                              // 1
         LPVOID lpvObjID,                                  // 2
         D3DRMLOADOPTIONS d3drmLOFlags,                    // 3
         D3DRMLOADTEXTURECALLBACK d3drmLoadTextureProc,    // 4
         LPVOID lpvArg                                     // 5
         );
```

The first parameter is the source to be loaded. It can be a file, a resource, a memory block, or a stream, depending on the source flags specified in the third parameter. The second parameter is the object name or position. This parameter depends on the identifier flags specified in the third parameter. If the D3DRMLOAD_BYPOSITION flag is specified, the second parameter is a pointer to a DWORD value that gives the object's order in the file. This parameter can be NULL. The third parameter is a flag of type D3DRMLOADOPTIONS describing the load options. Table 33-2, on the following page, lists these flags.

The fourth parameter to the Load() function is used when loading textures that require special formatting. In this case the specified callback function is called. This parameter can be NULL. The fifth parameter is the address of a data structure passed to the callback function in the fourth parameter. The function returns D3DRM_OK if successful, or an error otherwise.

Table 33-2

Flags in the D3DRMLOADOPTIONS Type

FLAG	ACTION
Flags modifying the first parameter (lpvObjSource):	
D3DRMLOAD_FROMFILE	The lpvObjSource parameter is interpreted as a string representing a local file name.
D3DRMLOAD_FROMRESOURCE	The lpvObjSource parameter is interpreted as a pointer to a D3DRMLOADRESOURCE structure.
D3DRMLOAD_FROMMEMORY	The lpvObjSource parameter is interpreted as a pointer to a D3DRMLOADMEMORY structure.
D3DRMLOAD_FROMURL	The lpvObjSource parameter is interpreted as a URL.
Flags modifying the second paramenter (lpvObjID):	
D3DRMLOAD_BYNAME	The lpvObjID parameter is interpreted as a string.
D3DRMLOAD_BYGUID	The lpvObjID parameter is interpreted as a UUID.
Other flags:	
D3DRMLOAD_FIRST	The first progressive mesh found is loaded. This is the default mode.

The following code fragment shows the creation of a meshbuilder object and its use in loading a file in DirectX format.

```
char                      szXfile[] = "teapot.x" ; // DirectX file
LPDIRECT3DRMMESHBUILDER3 meshbuilder = NULL;
HRESULT                   retval;
. . .

// Create the meshbuilder object
retval = lpD3DRM->CreateMeshBuilder(&meshbuilder);
     if (FAILED(retval))
        // Meshbuilder creation error handler goes here
       . . .
// Use meshbuilder to load a mesh from a DirectX file
retval = meshbuilder->Load(szXfile,
                           NULL,
                           D3DRMLOAD_FROMFILE,
                           NULL,
                           NULL);
if (FAILED(retval))
{
     // Load error handler goes here
. . .
```

After this code executes the file named teapot.x is converted into a mesh which becomes the meshbuilder object itself.

33.4.2 Adding a Mesh to a Frame

Currently our mesh is stored in a meshbuilder object, which cannot be rendered. The next step consists of creating a frame and loading the mesh into this frame. We have previously used CreateFrame(). We now use this same method to create a child frame. The coding is as follows:

```
LPDIRECT3DRMFRAME3          childframe = NULL;
. . .
// Create a child frame within the scene

retval = lpD3DRM->CreateFrame(aScene, &childframe);

if(FAILED(retval))

       // Error in creating frame handler goes here

. . .
```

In Direct3D a visual object, or simply a visual, is one that is displayed when the frame is in view. Meshes, textures, and even frames, can be visuals, although the most common visual is the mesh. When a texture object is labeled as a visual it becomes a decal. In this example we use the AddVisual() function, of the IDirect3DRMFrame interface, to add the mesh to the child frame as a visual. AddVisual() has the following general form:

```
HRESULT AddVisual(
                 LPDIRECT3DRMVISUAL lpD3DRMVisual       // 1
                 );
```

The function's only parameter is the address of a variable that represents the Direct3DRMVisual object to be added to the frame.

The call returns D3DRM_OK if successful, or an error otherwise. The following code fragment shows adding the mesh to the child frame.

```
// Add mesh into the child frame as a visual
retval = childframe->AddVisual(
                               (LPDIRECT3DRMVISUAL)meshbuilder);
if(FAILED(retval))
{
    // Failed AddVisual() error handler goes here
}
```

Notice that we used the pointer returned by the CreateFrame() call, which in this case is the variable childframe, of type LPDIRECT3DRMFRAME3, to access the AddVisual() function. The meshbuilder object is passed as a parameter and the result is that the mesh is added to the frame, and therefore, to the scene.

33.4.3 Setting the Camera Position

Previously in this chapter we created the camera as a global object. The camera object was stored in the variable named aCamera, of type LPDIRECT3DRMFRAME3, which is a member of the globVars structure. The camera object was created with the following statement:

```
// Create the camera frame
```

```
retval = lpD3DRM->CreateFrame(globVars.aScene,
                              &globVars.aCamera);
```

We have seen that the camera frame determines which scene is rendered and the viewing position and direction. In Direct3D the viewport renders only what is visible along the positive z-axis of the camera frame, with the up direction being in the direction of the positive y-axis.

When a child frame is created, it is positioned at the origin of the parent frame, that is, at coordinates (0,0,0). Applications can call the SetPosition() function of the IDirect3DRMFrame interface, to set the position of a frame relative to a reference point in the parent frame. To position the camera in its parent frame (the scene) we call SetPosition() using the variable aCamera as an interface reference. The general from of the SetPosition() function is as follows:

```
HRESULT SetPosition(
                 LPDIRECT3DRMFRAME lpRef,             // 1
                 D3DVALUE rvX,                        // 2
                 D3DVALUE rvY,                        // 3
                 D3DVALUE rvZ                         // 4
                 );
```

The first parameter is the address of the parent frame that is used as a reference. The second, third, and fourth parameters are the x, y, and z coordinates of the new position for the child frame. The call returns D3DRM_OK if successful, or an error otherwise.

The camera position determines what, if anything, is visible when the scene is rendered. For example, changing the position of the camera along the z-axis makes the objects in the scene appear larger or smaller (see Figure 33-1). The default position of the camera frame at the scene origin may be so close to the viewing frustum that a small portion of the object is visible. The following code fragment shows positioning of the camera frame so that it is located –7 units along the z-axis.

```
retval = aCamera->SetPosition(aScene,
                              D3DVAL(0),          // x
                              D3DVAL(0),          // y
                              -D3DVAL(7)          // z
                              );
if (FAILED(retval))
// Camera position error handler goes here
```

There is no default lighting in Direct3D retained mode. The objects in a scene without lights are invisible. In order to illuminate the scene, code must create the light frame and position it in relation to the parent frame. Once this is done, one or more lights can be added to the light frame and the scene illuminated. This means that we will be dealing with two different types of objects: the light frame object, which is of type LPDIRECT3DRMFRAME3, and one or more lights, which are of type LPDIRECT3DRMLIGHT.

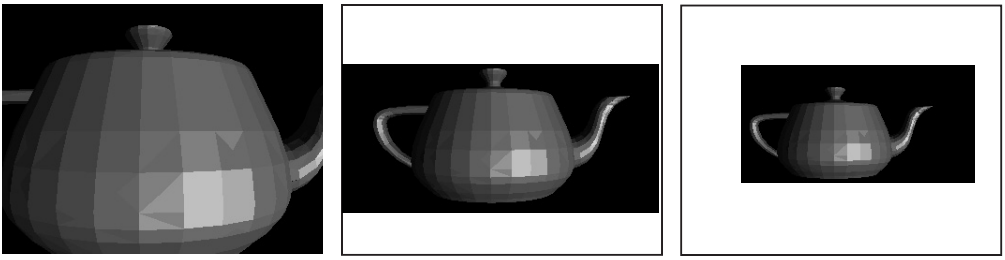


Figure 33-1 *Changing the Camera Position along the z-axis*

33.4.4 Creating and Positioning the Light Frame

We start by creating the light frame which is attached to the scene as a parent frame. Here again we use the CreateFrame() function, which is part of the IDirect3DRM3 interface. The following code fragment shows the processing.

```
LPDIRECT3DRMFRAME3 lights = NULL;
. . .
// Create a light frame as a child of the scene frame
retval = lpD3DRM->CreateFrame(aScene, &lights);
if(FAILED(retval))
{
     // Light frame creation error handler goes here
}
```

To set the position of the light frame we use the SetPosition() function of IDirect3DRMFrame interface, as in the following code fragment.

```
// Position the light frame within the scene
retval = lights->SetPosition(aScene,
                             D3DVAL(5),           // x
                             D3DVAL(0),           // y
                             -D3DVAL(7));         // z
if(FAILED(retval))
{
    // Light frame positioning error handler goes here
}
```

The position of the light frame is often related to the position of the camera frame. Since our camera frame was located at coordinates (0,0,-7), we position the light frame at the same y and z coordinates as the camera, but at a greater x coordinate. The result is that the light or lights placed in this frame will appear to come from the right of the camera and at the same vertical level (y coordinate) and distance from the object (z coordinate).

33.4.5 Creating and Setting the Lights

Now that we have a light frame, we are able to create one or more lights. There are two methods in the IDirect3DRM interface that allow creating lights: CreateLight() and CreateLightRGB(). CreateLight() requires that we specify the light color by referring a structure of type D3DCOLOR, which is obtained by calling the macros D3DRGB or D3DRGBA. CreateLightRGB() allows defining the light color directly. Because it is easier to code, we will use CreateLightRGB() in the examples in this chapter. The function's general form is as follows:

```
HRESULT CreateLightRGB(
                D3DRMLIGHTTYPE ltLightType,         // 1
                D3DVALUE vRed,                      // 2
                D3DVALUE vGreen,                    // 3
                D3DVALUE vBlue,                     // 4
                LPDIRECT3DRMLIGHT* lplpD3DRMLight   // 5
                );
```

The first parameter is one of the lighting types defined in the D3DRMLIGHTYPE enumerated type. Table 33-3 lists the constants that enumerate the different light types.

Table 33-3

Enumerator Constants in D3DRMLIGHTTYPE

CONSTANT	DESCRIPTION
D3DRMLIGHT_AMBIENT	Light is an ambient source
D3DRMLIGHT_POINT	Light is a point source
D3DRMLIGHT_SPOT	Light is a spotlight source
D3DRMLIGHT_DIRECTIONAL	Light is a directional source
D3DRMLIGHT_PARALLELPOINT	Light is a parallel point source

The second, third, and fourth parameters are the RGB color values for the light. They are expressed in a D3DVALUE type, which is Direct3D's designation for a float data type. The valid range is 0.0 to 1.0. A value of 0.0 indicates the maximum dimness and a value of 1,0 the maximum brightness. The fifth parameter is the address that will be filled by a pointer to an IDirect3DRMLight interface. The call returns D3DRM_OK if successful, or an error otherwise.

The following code fragment shows creating a parallel point source light with a slight bluish tint.

```
 LPDIRECT3DRMLIGHT light1 = NULL;
. . .
// Create a bright parallel point light
// Color values are as follows:
// 0.0 = totally dim and 1.0 = totally bright
retval = lpD3DRM->CreateLightRGB(D3DRMLIGHT_PARALLELPOINT,
                               D3DVAL(0.8),   // Red intensity
                               D3DVAL(0.8),   // Green intensity
                               D3DVAL(1.0),   // Blue intensity
                               &light1);
if(FAILED(retval))
{
     // Light creation error handler goes here
}
```

With the above call to CreateLightRGB() we have created a parallel point type light of a specific intensity and color composition. This light is stored in a variable of type LPDIRECT3DRMLIGHT, in this case named light1. But not until the light is attached to a frame will it illuminate the scene. The light frame created in the preceding section can be used at this time. The code is as follows:

```
// Add light to light frame
retval = lights->AddLight(light1);
if(FAILED(retval))
{
     // Light-to-frame attachment error handler goes here
}
```

Often the visual quality of a scene improves considerably if a dim, ambient light is added. Non-ambient lights (directional, parallel-point, point, and spot lights) are usually attached to a frame so that the light source can be positioned within the scene. Ambient light sources have no position and, therefore, it is inconsequential to which frame they are attached. Most often we attach ambient lights to the master scene frame.

You create an ambient light using the same CreateLightRGB() or CreateLight() method used for a non-ambient light. In this case the enumerator constant passed in the first parameter (see Table 33-3) is D3DRMLIGHT_AMBIENT. For ambient lights the values for the red, green, and blue component are usually in the lower part of the range. Once created, the ambient light can be attached to any frame or to the master scene. Either option produces identical results since the light uniformly illuminates the scene independent of its position. The following code fragment shows creating a dim, ambient light and attaching it to the master scene.

```
LPDIRECT3DRMLIGHT light2 = NULL;
. . .
// Create a dim, ambient light and attach it to the scene frame,
retval = lpD3DRM->CreateLightRGB(D3DRMLIGHT_AMBIENT,
                                 D3DVAL(0.1),    // Red value
                                 D3DVAL(0.1),    // Green value
                                 D3DVAL(0.1),    // Blue value
                                 &light2);
if(FAILED(retval))
{
     // Ambient light creation error handler goes here
}
// Attach ambient light to scene frame
retval = aScene->AddLight(light2);
     if(FAILED(retval))
{
     // Light attachment error handler goes here
}
```

Increasing the intensity of the ambient light often results in washed-out images. Color Figure 15 shows three versions of a teapot images in which the intensity of the ambient light has been increased from 0.1 to 0.8 for all three primary colors.

33.4.6 Creating a Material

The material property of an object determines how it reflects light. Two properties are associated with a material: emissive and specular. The emissive property of a material makes it appear to emit light and the specular property determines the sharpness of the reflected highlights thus making the surface appear hard and metallic or soft and plastic. The value of the specular property is defined by a power setting which determines the sharpness of the reflected highlights. A specular value of 5 gives a metallic appearance and higher values give a more plastic appearance.

Applications set the emissive property of a material using the SetEmissive() method of the IDirect3DRMMaterial interface. The function's general form is as follows:

```
HRESULT SetEmissive(
                    D3DVALUE *lpr,            // 1
                    D3DVALUE *lpg,            // 2
                    D3DVALUE *lpb             // 3
                    );
```

The function's three parameters are the intensity settings for the red, green, and blue components of the emitted light. The valid range for each color is 0.0 to 1.0. The function returns D3DRM_OK if it succeeds or an error otherwise.

The emissive property is useful in simulating self-luminous objects such as neon lights, radioactivity, or ghostly characters. The specular property of a material is more commonly used than the emissive property. The specular property has a power and a color component. The color component is set with the SetSpecular() function if the IDirect3DRMMaterial interface. The general form for this function is as follows:

```
HRESULT SetSpecular(
                    D3DVALUE r,        // 1
                    D3DVALUE g,        // 2
                    D3DVALUE b         // 3
                    );
```

The three parameters correspond to the value of the RGB color components for the specular highlights. The function returns D3DRM_OK if it succeeds, or an error otherwise.

The power setting for the specular property of a material can be defined when the material is created or afterwards. In the first case you use the CreateMaterial() method of the IDirect3DRM interface. To change the specular power of an existing material you can use the SetSpecular() method of IDirect3DRMMaterial interface. CreateMaterial() has the following general form:

```
HRESULT CreateMaterial(
             D3DVALUE vPower,                           // 1
             LPDIRECT3DRMMATERIAL * lplpD3DRMMaterial    // 2
             );
```

The first parameter is the sharpness of the reflected highlights, with a value of 5 corresponding to a metallic appearance. The second parameter is the address that will be filled with a pointer to an IDirect3DRMMaterial interface. The function returns D3DRM_OK if it succeeds, or an error otherwise.

Once a material is created it must be attached to a mesh or to a specific face of a mesh. Retained mode provides two related functions, both of which are named SetMaterial(). The function SetMaterial() of the IDirect3DRMFace interface attaches the material to a specific face of a mesh. The SetMaterial() function of the IDirect3DRMMeshBuilder interface attaches the material to all the faces of a mesh. The latter function has the following general form:

```
HRESULT SetMaterial(
        LPDIRECT3DRMMATERIAL2 lpIDirect3DRMmaterial   // 1
        );
```

The function's only parameter is the address of IDirect3DRMMaterial interface for the Direct3DRMMeshBuilder object, which is of type LPDIRECT3DMATERIAL2. The function returns D3DRM_OK if it succeeds, or an error otherwise.

The following code fragment shows creating a material and assigning to it a specular power of 0.8. After the material is created, it is attached to an existing mesh.

```
LPDIRECT3DRMMATERIAL2 material1 = NULL;
. . .
// Create a material setting its specular property
retval = lpD3DRM->CreateMaterial(D3DVAL(8.0), &material1);
if(FAILED(retval))
{
     // Failed material creation error handler goes here
}
// Set the material on the mesh
retval = meshbuilder->SetMaterial(material1);
if(FAILED(retval))
{
     // Material attachment error handler goes here
}
```

33.4.7 Setting the Mesh Color

Meshes have no natural color. If we attempt to render a mesh without setting it to a color attribute the result is an image in shades of gray, as shown in the top part of Color Figure 16. Retained mode includes several methods to set the color of objects, all of which are named SetColorRGB(). One of these methods belongs to the Direct3DRMFace interface and is used to set the color of a mesh face. A second SetColorRGB() function is part of IDirect3DRMFrame interface and serves to set the color of a mesh contained in a mesh. In this case the material mode is set to D3DRMMATERIAL_FROMFRAME. A third SetColorRGB() method is used to set the color of a light. The fourth one belongs to the IDirect3DRMMeshBuilder interface and is used to set all the faces of a mesh to a particular color attribute. This version of the SetColorRGB() function has the following general form:

```
HRESULT SetColorRGB(
                    D3DVALUE red,         // 1
                    D3DVALUE green,       // 2
                    D3DVALUE blue         // 3
                    );
```

The three parameters of this function determines the red, green, and blue color components of the mesh. The function returns D3DRM_OK if it succeeds, or an error otherwise.

The following code fragment shows using the SetColorRGB() function referenced by a meshbuilder object. In this case the color is set to bright green.

```
LPDIRECT3DRMMESHBUILDER3 meshbuilder = NULL;
. . .
// Set the mesh color (bright green in this case).
```

```
retval = meshbuilder->SetColorRGB(D3DVAL(0.0),   // red
                                  D3DVAL(0.7),   // green
                                  D3DVAL(0.0));  // blue

if(FAILED(retval))
{
   // Mesh color setting error handler goes here
}
```

The lower image in Color Figure 16 shows the object rendered after the mesh is assigned the color value (0.0,0.7,0.0).

33.4.8 Clean-Up Operations

Once the master scene has been built (usually by creating a meshbuilder and a mesh, loading the mesh into a child frame, setting the camera position, creating and positioning the lights, and creating the mesh material and color) we can proceed to release all the local objects used in the process. The individual objects are preserved in the scene and will be rendered on the screen. The Release() function of the IUnknown interface, mentioned earlier in this chapter, is used to deallocate the individual object and reduce the object count by one. The function's general form is as follows:

```
ULONG Release();
```

The function returns the new reference count in a variable of type ULONG. The COM object deallocates itself when its reference count reaches 0.

In reference to the code samples listed in this section, the clean-up operation is in the following code fragment:

```
// Local variables
LPDIRECT3DRMFRAME3 lights             = NULL;
LPDIRECT3DRMMESHBUILDER3 meshbuilder = NULL;
LPDIRECT3DRMLIGHT light1              = NULL;
LPDIRECT3DRMLIGHT light2              = NULL;
LPDIRECT3DRMMATERIAL2 material1       = NULL;
. . .
// Release local objects
lights->Release();
meshbuilder->Release();
light1->Release();
light2->Release();
material1->Release();
```

33.4.9 Calling BuildScene()

The BuildScene() Function in the 3DRM InWin Demo1 program in the book's software package, performs all of the processing operations discussed in this section. Following is a code listing of this function.

```
//*****************************************************************
// Name: BuildScene()
// Description: Create the scene
//*****************************************************************
BOOL BuildScene( LPDIRECT3DRMDEVICE3 aDevice,
                 LPDIRECT3DRMFRAME3  aScene,
                 LPDIRECT3DRMFRAME3  aCamera )
{
```

```
// Local varaibles
LPDIRECT3DRMFRAME3 lights            = NULL;
LPDIRECT3DRMMESHBUILDER3 meshbuilder = NULL;
LPDIRECT3DRMFRAME3 childframe        = NULL;
LPDIRECT3DRMLIGHT light1             = NULL;
LPDIRECT3DRMLIGHT light2             = NULL;
LPDIRECT3DRMMATERIAL2 material1      = NULL;
HRESULT retval;

// Create the meshbuilder object
retval = lpD3DRM->CreateMeshBuilder(&meshbuilder);
  if (FAILED(retval))
      goto ERROR_EXIT;

// Use meshbuilder to load a mesh from a DirectX file
retval = meshbuilder->Load(szXfile,            // Source
                           NULL,
                           D3DRMLOAD_FROMFILE, // Options
                           NULL, NULL);
if (FAILED(retval))
{
    D3DError("Failed to load file.");
    goto DIRECT_EXIT;
}

// Create a child frame within the aScene.
retval = lpD3DRM->CreateFrame(aScene, &childframe);
if(FAILED(retval))
    goto ERROR_EXIT;

// Add mesh into the child frame as a visual
retval = childframe->AddVisual(
                           (LPDIRECT3DRMVISUAL)meshbuilder);
if(FAILED(retval))
    goto ERROR_EXIT;
// Set up the camera frame position
retval = aCamera->SetPosition(aScene,
                              D3DVAL(0),      // x
                              D3DVAL(0),      // y
                              -D3DVAL(7));    // z
if (FAILED(retval))
{
    D3DError("Failed to position the camera in the frame.");
    goto DIRECT_EXIT;
}
// Create a light frame as a child of the scene frame
retval = lpD3DRM->CreateFrame(aScene, &lights);
  if(FAILED(retval))
    goto ERROR_EXIT;

// Position the light frame within the scene
retval = lights->SetPosition(aScene,
                             D3DVAL(5),       // x
                             D3DVAL(0),       // y
                             -D3DVAL(7));     // z
if(FAILED(retval))
    goto ERROR_EXIT;

// Create a bright, parallel point light
// Color values are as follows:
```

```
// 0.0 = totally dim and 1.0 = totally bright
retval = lpD3DRM->CreateLightRGB(D3DRMLIGHT_PARALLELPOINT,
                         D3DVAL(0.8),   // Red intensity
                         D3DVAL(0.8),   // Green intensity
                         D3DVAL(1.0),   // Blue intensity
                         &light1);
if(FAILED(retval))
    goto ERROR_EXIT;

// Add light to light frame
retval = lights->AddLight(light1);
if(FAILED(retval))
    goto ERROR_EXIT;

// Create a dim, ambient light and attach it to the scene
// frame,
retval = lpD3DRM->CreateLightRGB(D3DRMLIGHT_AMBIENT,
                                   D3DVAL(0.2),    // red
                                   D3DVAL(0.2),    // green
                                   D3DVAL(0.2),    // blue
                                   &light2);
if(FAILED(retval))
    goto ERROR_EXIT;

retval = aScene->AddLight(light2);
   if(FAILED(retval))
        goto ERROR_EXIT;

// Create a material setting its specular property
retval = lpD3DRM->CreateMaterial(D3DVAL(8.0), &material1);
if(FAILED(retval))
    goto ERROR_EXIT;

// Set the material on the mesh
retval = meshbuilder->SetMaterial(material1);
if(FAILED(retval))
    goto ERROR_EXIT;

// Set the mesh color (bright green in this case).
retval = meshbuilder->SetColorRGB(D3DVAL(0.0),   // red
                                 D3DVAL(0.7),    // green
                                 D3DVAL(0.0));   // blue

if(FAILED(retval))
    goto ERROR_EXIT;

//******************************
// Function succeeds. Clean up
//******************************
childframe->Release();
lights->Release();
meshbuilder->Release();
light1->Release();
light2->Release();
material1->Release();
return TRUE;

//******************************
//      Error exits
//******************************
```

```
    ERROR_EXIT:
    D3DError("Failure building the scene");
    DIRECT_EXIT:
    childframe->Release();
    lights->Release();
    meshbuilder->Release();
    light1->Release();
    light2->Release();
    material1->Release();
    return FALSE;
}
```

33.5 Rendering Operations

To render is to convert image data into an actual image. In all the processing operations performed so far in this chapter, all we have done is manipulate data. Nothing has been shown on the screen, or even formatted into a displayable construct.

In Chapter 32 you saw that Direct3D rendering takes place on three separate modules, called the transformation, lighting, and rasterization modules. But when programming in retained mode, the individual modules are not visible. Instead, the rendering operation is conceptualized as consisting of four functions:

- The Move() function of the IDirect3DRMFrame interface applies the rotations and velocities to all the frames in the hierarchy.
- The Clear() function of the IDirect3DRMViewport interface clears the viewport to the current background color.
- The Render() function, of the IDirect3DRNFrame, renders the scene into the viewport.
- The Update() function of the IDirect3DRMDevice interface copies the rendered image to the display surface.

33.5.1 Clearing the Viewport

In Direct3D retained mode the viewport is one of the objects of the IDirect3DRM interface. It is defined as a rectangular area in the device space. The viewport extent is always measured in device units, which are pixels for the screen device. The viewport origin is the offset of the viewport within the device space. Previously in this chapter we created a viewport using the CreateViewport() function of the lpD3DRM interface. At that time we assigned the viewport to a device frame and a camera frame. We also defined the viewport origin by means of its position in the device frame, as well as its extent.

Clearing the viewport is accomplished by calling the Clear() function if IDirect3DRMViewport. The function's general form is as follows:

```
HRESULT Clear();
```

No parameters are necessary since the viewport to be cleared is the one calling the function, as in the following code fragment:

```
// Global Structure
struct _globVars
{
```

```
. . .
    LPDIRECT3DRMVIEWPORT2 aViewport;   // Direct3DRM viewport
. . .
} globVars;

// Clear the viewport.
retval = globVars.aViewport->Clear(D3DRMCLEAR_ALL);
if (FAILED(retval))
{
    // Viewport clearing error handler goes here
}
```

32.5.2 Rendering to the Viewport

In Chapter 32 a scene is organized in a tree-like structure that consists of a root, or master frame, and any number of child frames. Child frames inherit their characteristics from the parent frames to which they are physically attached. When a frame is moved, all the child frames move with it. The parent frame and its child frames are known as a frame hierarchy. In retained mode this frame hierarchy can we changed by code.

The Render() function of the IDirect3DRMViewport interface renders a frame hierarchy to a given viewport. The call renders the visual on a given frame and all of its child frames. Frames above it on the hierarchy are not rendered or affected. This mode of operation is sometimes described as being "state based", which means that the state of the renderer is determined by the part of the frame tree currently being traversed. The general form of the Frame() function is as follows:

```
HRESULT Render(
               LPDIRECT3DRMFRAME lpD3DRMFrame      // 1
               );
```

The function's only parameter is the address of the variable that represents the Direct3DRMFrame object at the top of the frame hierarchy to be rendered. The function returns D3DRM_OK if it succeeds, or an error otherwise. The following code fragment shows a call to the Render() function.

```
// Global Structure
struct _globVars
{
. . .
    LPDIRECT3DRMVIEWPORT2 aViewport;    // Direct3DRM viewport
    LPDIRECT3DRMFRAME3    aScene;       // Master frame
. . .
} globVars;
. . .
// Render the scene
retval = globVars.aViewport->Render(globVars.aScene);
if (FAILED(retval))
{
     // Rendering failure error handler goes here
}
```

In this case since the argument of the Render() call is the master frame, which determines that all other frames attached to the master frame are rendered.

33.5.3 Updating the Screen

We have now rendered the scene to the viewport, but nothing yet shows on the video display. For this to happen we must call the Update() function of the IDirect3DRMDevice interface. Update() copies the image in the viewport to the display surface. It also provides a system-level tick, called the heartbeat. This tick was discussed in the context of DirectDraw animation. The general form of the Update() function is as follows:

```
HRESULT Update();
```

No parameters are necessary since the device is referenced in the call. Each time Update() is called, the system optionally sends execution to an application-defined callback function. Applications define the callback function by means of the AddUpdateCallback() function of the IDirect3DRMDevice interface. The callback function is convenient when the application needs to update scene data during each beat of the renderer. The Update() function returns D3DRM_OK if it succeeds, or an error otherwise.

33.5.4 RenderScene() Function

The RenderScene() function that is part of the 3DRM InWin Demo1 program in the book's software package performs the processing operations discussed in this section. Follows a code listing of this function.

```
//***************************************************************
// Name: RenderScene()
// Description: Clear the viewport, render the frame, and
//              update the window.
//***************************************************************
static BOOL RenderScene()
{
    HRESULT retval;

    // Clear the viewport.
    retval = globVars.aViewport->Clear(D3DRMCLEAR_ALL);
    if (FAILED(retval))
    {
        D3DError("Clearing viewport failed.");
        return FALSE;
    }
    // Render the aScene to the viewport.
    retval = globVars.aViewport->Render(globVars.aScene);
    if (FAILED(retval))
    {
        D3DError("Rendering scene failed.");
        return FALSE;
    }
    // Update the window.
    retval = globVars.aDevice->Update();
    if (FAILED(retval))
    {
        D3DError("Updating device failed.");
        return FALSE;
    }
    return TRUE;
}
```

33.6 Sample Project 3DRM InWin Demo1

The project named 3DRM InWin Demo1 contained in the Chapter 33 subfolder in the book's software package, demonstrates the basic retained mode operations discussed in this chapter. The program displays a file in DirectX format. The filename is contained in a global string and can be edited by the user. The file furnished in the workspace directory is named “teapot.x”. This is one of the 3D files that comes with the DirectX SDK. Rendering is static since no animation is attempted at this point. Color Figure 17 is a screen snapshot of the 3DRM InWin Demo1 program.

To facilitate reuse we have grouped the processing into four functions:

1. InitD3D() initializes the retained mode interface and creates a DirectDraw clipper object based on the application window.
2. CreateObjects() creates the device and objects that form the 3D scene.
3. BuildScene() uses the objects created in the previous step to build the application's main frame.
4. RenderScene() renders the scene to the viewport and displays it.

The functions were discussed in detail and are listed in previous sections of this Chapter.

33.6.1 Windowed Retained Mode Coding Template

The project directory 3DRM InWin Template, located in the Chapter 33 directory in the book's software package, contains a template program that could be useful in the initial stages of developing a Direct3D retained mode, windowed application. To use it you can copy the template file named 3DRM InWin Template.cpp to your own workspace. Then rename the file and edit it to suit your application. Alternatively you can copy or rename the entire directory. When using the template file make sure that you have referenced the libraries dxguid.lib, ddraw.lib, d3drm.lib, and winmm.lib. To include these libraries you must edit the Object/Libraries modules windows on the Link tab of Developer Studio Project Settings dialog box.

Appendix A

Windows Structures

This appendix contains the structures mentioned in the text. Structures are listed in alphabetical order.

```
BITMAP
   typedef struct tagBITMAP {  /* bm */
       int     bmType;
       int     bmWidth;
       int     bmHeight;
       int     bmWidthBytes;
       BYTE    bmPlanes;
       BYTE    bmBitsPixel;
       LPVOID  bmBits;
   };

BITMAPCOREHEADER
   typedef struct tagBITMAPCOREHEADER { // bmch
           DWORD   bcSize;
           WORD    bcWidth;
           WORD    bcHeight;
           WORD    bcPlanes;
           WORD    bcBitCount;
   } BITMAPCOREHEADER;

BITMAPCOREINFO
   typedef struct _BITMAPCOREINFO {    // bmci
           BITMAPCOREHEADER  bmciHeader;
           RGBTRIPLE         bmciColors[1];
   } BITMAPCOREINFO;

BITMAPFILEHEADER
   typedef struct tagBITMAPFILEHEADER { // bmfh
           WORD    bfType;
           DWORD   bfSize;
           WORD    bfReserved1;
           WORD    bfReserved2;
           DWORD   bfOffBits;
   } BITMAPFILEHEADER;

BITMAPINFO
   typedef struct tagBITMAPINFO { // bmi
```

```
      BITMAPINFOHEADER bmiHeader;
      RGBQUAD          bmiColors[1];
   } BITMAPINFO;

BITMAPINFOHEADER
   typedef struct tagBITMAPINFOHEADER{ // bmih
      DWORD  biSize;
      LONG   biWidth;
      LONG   biHeight;
      WORD   biPlanes;
      WORD   biBitCount
      DWORD  biCompression;
      DWORD  biSizeImage;
      LONG   biXPelsPerMeter;
      LONG   biYPelsPerMeter;
      DWORD  biClrUsed;
      DWORD  biClrImportant;
   } BITMAPINFOHEADER;

CHOOSECOLOR
   typedef struct {   // cc
       DWORD        lStructSize;
       HWND         hwndOwner;
       HWND         hInstance;
       COLORREF     rgbResult;
       COLORREF*    lpCustColors;
       DWORD        Flags;
       LPARAM       lCustData;
       LPCCHOOKPROC lpfnHook;
       LPCTSTR      lpTemplateName;
   } CHOOSECOLOR;

COLORADJUSTMENT
   typedef struct  tagCOLORADJUSTMENT {    /* ca */
       WORD  caSize;
       WORD  caFlags;
       WORD  caIlluminantIndex;
       WORD  caRedGamma;
       WORD  caGreenGamma;
       WORD  caBlueGamma;
       WORD  caReferenceBlack;
       WORD  caReferenceWhite;
       SHORT caContrast;
       SHORT caBrightness;
       SHORT caColorfulness;
       SHORT caRedGreenTint;
   } COLORADJUSTMENT;

CREATESTRUCT
   typedef struct tagCREATESTRUCT { // cs
       LPVOID    lpCreateParams;
       HINSTANCE hInstance;
       HMENU     hMenu;
       HWND      hwndParent;
       int       cy;
       int       cx;
       int       y;
       int       x;
       LONG      style;
       LPCTSTR   lpszName;
       LPCTSTR   lpszClass;
       DWORD     dwExStyle;
   } CREATESTRUCT;
```

```
DDBLTFX
   typedef struct _DDBLTFX{
       DWORD  dwSize;
       DWORD  dwDDFX;
       DWORD  dwROP;
       DWORD  dwDDROP;
       DWORD  dwRotationAngle;
       DWORD  dwZBufferOpCode;
       DWORD  dwZBufferLow;
       DWORD  dwZBufferHigh;
       DWORD  dwZBufferBaseDest;
       DWORD  dwZDestConstBitDepth;
   union
   {
           DWORD                dwZDestConst;
           LPDIRECTDRAWSURFACE  lpDDSZBufferDest;
   };
       DWORD   dwZSrcConstBitDepth;
   union
   {
           DWORD                dwZSrcConst;
           LPDIRECTDRAWSURFACE  lpDDSZBufferSrc;

   };
       DWORD  dwAlphaEdgeBlendBitDepth;
       DWORD  dwAlphaEdgeBlend;
       DWORD  dwReserved;
       DWORD  dwAlphaDestConstBitDepth;
   union
   {
           DWORD                dwAlphaDestConst;
           LPDIRECTDRAWSURFACE  lpDDSAlphaDest;
   };
       DWORD  dwAlphaSrcConstBitDepth;
   union
   {
           DWORD                dwAlphaSrcConst;
           LPDIRECTDRAWSURFACE  lpDDSAlphaSrc;
   };
   union
   {
           DWORD                 dwFillColor;
           DWORD                 dwFillDepth;

           LPDIRECTDRAWSURFACE  lpDDSPattern;
   };
   DDCOLORKEY  ddckDestColorkey;
   DDCOLORKEY  ddckSrcColorkey;
   } DDBLTFX,FAR* LPDDBLTFX;
```

```
DDCAPS
   typedef struct _DDCAPS{
       DWORD    dwSize;
       DWORD    dwCaps;
       DWORD    dwCaps2;
       DWORD    dwCKeyCaps;
       DWORD    dwFXCaps;
       DWORD    dwFXAlphaCaps;
       DWORD    dwPalCaps;
       DWORD    dwSVCaps;
       DWORD    dwAlphaBltConstBitDepths;
       DWORD    dwAlphaBltPixelBitDepths;
       DWORD    dwAlphaBltSurfaceBitDepths;
       DWORD    dwAlphaOverlayConstBitDepths;
       DWORD    dwAlphaOverlayPixelBitDepths;
       DWORD    dwAlphaOverlaySurfaceBitDepths;
       DWORD    dwZBufferBitDepths;

       DWORD    dwVidMemTotal;
       DWORD    dwVidMemFree;
       DWORD    dwMaxVisibleOverlays;
       DWORD    dwCurrVisibleOverlays;
       DWORD    dwNumFourCCCodes;
       DWORD    dwAlignBoundarySrc;
       DWORD    dwAlignSizeSrc;
       DWORD    dwAlignBoundaryDest;
       DWORD    dwAlignSizeDest;
       DWORD    dwAlignStrideAlign;
       DWORD    dwRops[DD_ROP_SPACE];
       DDSCAPS  ddsCaps;
       DWORD    dwMinOverlayStretch;
       DWORD    dwMaxOverlayStretch;
       DWORD    dwMinLiveVideoStretch;

       DWORD    dwMaxLiveVideoStretch;
       DWORD    dwMinHwCodecStretch;
       DWORD    dwMaxHwCodecStretch;
       DWORD    dwReserved1;
       DWORD    dwReserved2;
       DWORD    dwReserved3;
       DWORD    dwSVBCaps;
       DWORD    dwSVBCKeyCaps;
       DWORD    dwSVBFXCaps;
       DWORD    dwSVBRops[DD_ROP_SPACE];
       DWORD    dwVSBCaps;
       DWORD    dwVSBCKeyCaps;
       DWORD    dwVSBFXCaps;
       DWORD    dwVSBRops[DD_ROP_SPACE];
       DWORD    dwSSBCaps;
       DWORD    dwSSBCKeyCaps;

       DWORD    dwSSBCFXCaps;
       DWORD    dwSSBRops[DD_ROP_SPACE];
       DWORD    dwReserved4;
       DWORD    dwReserved5;
       DWORD    dwReserved6;
   } DDCAPS,FAR* LPDDCAPS;

DDCOLORKEY
   typedef struct _DDCOLORKEY{
       DWORD  dwColorSpaceLowValue;
       DWORD  dwColorSpaceHighValue;
   } DDCOLORKEY,FAR* LPDDCOLORKEY;

DDPIXELFORMAT
   typedef struct _DDPIXELFORMAT{
       DWORD  dwSize;
       DWORD  dwFlags;
       DWORD  dwFourCC;
   union
   {
           DWORD  dwRGBBitCount;
           DWORD  dwYUVBitCount;
           DWORD  dwZBufferBitDepth;
           DWORD  dwAlphaBitDepth;
   };
   union
   {
           DWORD  dwRBitMask;
           DWORD  dwYBitMask;
   };
   union
   {
           DWORD  dwGBitMask;
           DWORD  dwUBitMask;
   };
   union
   {
           DWORD  dwBBitMask;
           DWORD  dwVBitMask;
   };
   union
   {
           DWORD  dwRGBAlphaBitMask;

           DWORD  dwYUVAlphaBitMask;
   };
   } DDPIXELFORMAT, FAR* LPDDPIXELFORMAT;
```

```
DDSCAPS2
   typedef struct _DDSCAPS2 {
       DWORD    dwCaps;   // Surface capabilities
       DWORD    dwCaps2;  // More surface capabilities
       DWORD    dwCaps3;  // Not currently used
       DWORD    dwCaps4;  // .
   } DDSCAPS2, FAR* LPDDSCAPS2;
```

```
DDSURFACEDESC2
   typedef struct _DDSURFACEDESC2 {
       DWORD         dwSize;
       DWORD         dwFlags;
       DWORD         dwHeight;
       DWORD         dwWidth;
       union
       {
           LONG      lPitch;
           DWORD     dwLinearSize;
       } DUMMYUNIONNAMEN(1);
       DWORD         dwBackBufferCount;
       union
       {
           DWORD     dwMipMapCount;
           DWORD     dwRefreshRate;
       } DUMMYUNIONNAMEN(2);
       DWORD         dwAlphaBitDepth;
       DWORD         dwReserved;
       LPVOID        lpSurface;
       DDCOLORKEY    ddckCKDestOverlay;
       DDCOLORKEY    ddckCKDestBlt;
       DDCOLORKEY    ddckCKSrcOverlay;
       DDCOLORKEY    ddckCKSrcBlt;
       DDPIXELFORMAT ddpfPixelFormat;
       DDSCAPS2      ddsCaps;
       DWORD         dwTextureStage;
   } DDSURFACEDESC2, FAR* LPDDSURFACEDESC2;

DIBSECTION
   typedef struct tagDIBSECTION {
       BITMAP              dsBm;
       BITMAPINFOHEADER    dsBmih;
       DWORD               dsBitfields[3];
       HANDLE              dshSection;
       DWORD               dsOffset;
   } DIBSECTION;

DIDATAFORMAT
   typedef struct {
       DWORD dwSize;
       DWORD dwObjSize;
       DWORD dwFlags;
       DWORD dwDataSize;
       DWORD dwNumObjs;
       LPDIOBJECTDATAFORMAT rgodf;
   } DIDATAFORMAT;

DIDEVCAPS
   typedef struct {
       DWORD dwSize;
       DWORD dwDevType;
       DWORD dwFlags;
       DWORD dwAxes;
       DWORD dwButtons;
       DWORD dwPOVs;
   } DIDEVCAPS;

DIDEVICEINSTANCE
   typedef struct {
       DWORD dwSize;
       GUID  guidInstance;
       GUID  guidProduct;
       DWORD dwDevType;
       TCHAR tszInstanceName[MAX_PATH];
       TCHAR tszProductName[MAX_PATH];
   } DIDEVICEINSTANCE;

DIDEVICEOBJECTDATA
   typedef struct {
       DWORD dwOfs;
       DWORD dwData;
       DWORD dwTimeStamp;
       DWORD dwSequence;
   } DIDEVICEOBJECTDATA;

DIJOYSTATE
```

```
   typedef struct DIJOYSTATE {
       LONG    lX;
       LONG    lY;
       LONG    lZ;
       LONG    lRx;
       LONG    lRy;
       LONG    lRz;
       LONG    rglSlider[2];
       DWORD   rgdwPOV[4];
       BYTE    rgbButtons[32];
   } DIJOYSTATE, *LPDIJOYSTATE;

DIJOYSTATE2
   typedef struct DIJOYSTATE2 {
       LONG    lX;
       LONG    lY;
       LONG    lZ;
       LONG    lRx;
       LONG    lRy;
       LONG    lRz;
       LONG    rglSlider[2];
       DWORD   rgdwPOV[4];
       BYTE    rgbButtons[128];
       LONG    lVX;
       LONG    lVY;
       LONG    lVZ;
       LONG    lVRx;
       LONG    lVRy;
       LONG    lVRz;
       LONG    rglVSlider[2];
       LONG    lAX;
       LONG    lAY;
       LONG    lAZ;
       LONG    lARx;
       LONG    lARy;
       LONG    lARz;
       LONG    rglASlider[2];
       LONG    lFX;
       LONG    lFY;
       LONG    lFZ;
       LONG    lFRx;
       LONG    lFRy;
       LONG    lFRz;
       LONG    rglFSlider[2];
   } DIJOYSTATE2, *LPDIJOYSTATE2;

DIMOUSESTATE
   typedef struct {
       LONG lX;
       LONG lY;
       LONG lZ;
       BYTE rgbButtons[4];
   } DIMOUSESTATE;

DIPROPDWORD
   typedef struct {
       DIPROPHEADER     diph;
       DWORD            dwData;
   } DIPROPDWORD;
```

```
DIPROPHEADER
   typedef struct {
       DWORD    dwSize;
       DWORD    dwHeaderSize;
       DWORD    dwObj;
       DWORD    dwHow;
   } DIPROPHEADER;

DIPROPRANGE
   typedef struct {
       DIPROPHEADER diph;
       LONG         lMin;
       LONG         lMax;
   } DIPROPRANGE;

   DISPLAY_DEVICE
   typedef struct _DISPLAY_DEVICE {
       DWORD  cb;
       WCHAR  DeviceName[32];
       WCHAR  DeviceString[128];
       DWORD  StateFlags;
   } DISPLAY_DEVICE, *PDISPLAY_DEVICE, *LPDISPLAY_DEVICE;

DSETUP_CB_UPGRADEINFO
   typedef struct _DSETUP_CB_UPGRADEINFO
   {
       DWORD UpgradeFlags;
   } DSETUP_CB_UPGRADEINFO;

LOGBRUSH
   typedef struct tag LOGBRUSH { /* lb */
      UINT     lbStyle;
      COLORREF lbColor;
      LONG     lbHatch;
   } LOGBRUSH;

LOGPEN
   typedef struct tagLOGPEN {  /* lgpn */
       UINT     lopnStyle;
       POINT    lopnWidth;
       COLORREF lopnColor;
   } LOGPEN;

LV_KEYDOWN
   typedef struct tagLV_KEYDOWN {
       NMHDR hdr;
       WORD wVKey;
       UINT flags;
   } LV_KEYDOWN;

MONITORINFO
   typedef struct tagMONITORINFO {
       DWORD  cbSize;
       RECT   rcMonitor;
       RECT   rcWork;
       DWORD  dwFlags;
   } MONITORINFO, *LPMONITORINFO;

MONITORINFOEX
   typedef struct tagMONITORINFOEX {
       DWORD  cbSize;
       RECT   rcMonitor;
       RECT   rcWork;
       DWORD  dwFlags;
       TCHAR  szDevice[CCHDEVICENAME]
   } MONITORINFOEX, *LPMONITORINFOEX;

MSG
   typedef struct tagMSG {     // msg
       HWND   hwnd;
       UINT   message;
       WPARAM wParam;
       LPARAM lParam;
       DWORD  time;
       POINT  pt;
   } MSG;

NMHDR
   typedef struct tagNMHDR {
       HWND hwndFrom;
       UINT idFrom;
       UINT code;
   } NMHDR;

PAINTSTRUCT
   typedef struct tagPAINTSTRUCT { // ps
       HDC  hdc;
       BOOL fErase;
       RECT rcPaint;
       BOOL fRestore;
       BOOL fIncUpdate;
       BYTE rgbReserved[32];
   } PAINTSTRUCT;

POINT
   typedef struct tagPOINT {
      LONG x;
      LONG y;
   } POINT;

   RECT
   typedef struct tagRECT {
      LONG left;
      LONG top;
      LONG right;
      LONG bottom;
   } RECT;

RGBQUAD
   typedef struct tagRGBQUAD { // rgbq
       BYTE    rgbBlue;
       BYTE    rgbGreen;
       BYTE    rgbRed;
       BYTE    rgbReserved;
   } RGBQUAD;

RGBTRIPLE
   typedef struct tagRGBTRIPLE { // rgbt
       BYTE rgbtBlue;
       BYTE rgbtGreen;
       BYTE rgbtRed;
   } RGBTRIPLE;


RGNDATA
   typedef struct _RGNDATA { /* rgnd */
       RGNDATAHEADER rdh;
       char          Buffer[1];
   } RGNDATA;

RGNDATAHEADER
   typedef struct _RGNDATAHEADER { // rgndh
       DWORD dwSize;
       DWORD iType;
       DWORD nCount;
       DWORD nRgnSize;
       RECT  rcBound;
   } RGNDATAHEADER;

SCROLLINFO
   typedef struct tagSCROLLINFO {  // si
       UINT cbSize;
       UINT fMask;
       int  nMin;
       int  nMax;
       UINT nPage;
       int  nPos;
       int  nTrackPos;
   }   SCROLLINFO;
   typedef SCROLLINFO FAR *LPSCROLLINFO;

SIZE
   typedef struct tagSIZE {
       int cx;
       int cy;
   } SIZE;

TBBUTTON
   typedef struct _TBBUTTON { \\ tbb
       int iBitmap;
       int idCommand;
       BYTE fsState;
       BYTE fsStyle;
       DWORD dwData;
       int iString;
   } TBBUTTON, NEAR* PTBBUTTON, FAR* LPTBBUTTON;
   typedef const TBBUTTON FAR* LPCTBBUTTON;

TEXTMETRICS
   typedef struct tagTEXTMETRIC {  /* tm */
       int  tmHeight;
       int  tmAscent;
       int  tmDescent;
       int  tmInternalLeading;
       int  tmExternalLeading;
       int  tmAveCharWidth;
       int  tmMaxCharWidth;
       int  tmWeight;
       BYTE tmItalic;
       BYTE tmUnderlined;
       BYTE tmStruckOut;
       BYTE tmFirstChar;
       BYTE tmLastChar;
       BYTE tmDefaultChar;
       BYTE tmBreakChar;
       BYTE tmPitchAndFamily;
       BYTE tmCharSet;
       int  tmOverhang;
       int  tmDigitizedAspectX;
       int  tmDigitizedAspectY;
   } TEXTMETRIC;

TOOLINFO
   typedef struct {  // ti
       UINT      cbSize;
       UINT      uFlags;
       HWND      hwnd;
       UINT      uId;
       RECT      rect;
       HINSTANCE hinst;
       LPTSTR     lpszText;
   } TOOLINFO, NEAR *PTOOLINFO, FAR *LPTOOLINFO;

WNDCLASSEX
   typedef struct _WNDCLASSEX {    // wc
       UINT    cbSize;
       UINT    style;
       WNDPROC lpfnWndProc;
       int     cbClsExtra;
       int     cbWndExtra;
       HANDLE  hInstance;
       HICON   hIcon;
       HCURSOR hCursor;
       HBRUSH  hbrBackground;
       LPCTSTR lpszMenuName;
       LPCTSTR lpszClassName;
       HICON   hIconSm;
   } WNDCLASSEX;
```

Appendix B

Ternary Raster Operation Codes

This appendix describes the ternary raster-operation codes used by The Windows GDI and DirectX. These codes determine how the bits in a source are combined with those of a destination, taking into account a particular pattern.

The following abbreviations are used for the ternary operands and the boolean functions:

- D = destination bitmap
- P = pattern (determined by current brush)
- S = sSource bitmap
- & = bitwise AND
- ~ = bitwise NOT (inverse)
- | = bitwise OR
- ^ = bitwise exclusive OR (XOR)

The most commonly used raster operations have been given special names in the Windows include file, windows.h. The following table, taken from Developers Studio help files, lists all 256 ternary raster operations.

RASTER OP.	ROP CODE	BOOLEAN OPERATION	COMMON NAME
00	00000042	0	BLACKNESS
01	00010289	~(P|S|D)	-
02	00020C89	~(P|S)&D	-
03	000300AA	~(P|S)	-
04	00040C88	~(P|D)&S	-
05	000500A9	~(P|D)	-
06	00060865	~(|P~(S^D))	-
07	000702C5	~(P|(S&D))	-
08	00080F08	~P&S&D	-
09	00090245	~(P|(S^D))	-
0A	000A0329	~P&D	-

RASTER OP.	ROP CODE	BOOLEAN OPERATION	COMMON NAME
0B	000B0B2A	~P(|(S&~D))	-
0C	000C0324	~P&S	-
0D	000D0B25	~P|(~S&D))	-
0E	000E08A5	~P|~(S|D)	-
0F	000F0001	~P	-
10	00100C85	P&~(S|D)	-
11	001100A6	~(S|D)	NOTSRCERASE
12	00120868	~(S|~(P^D))	-
13	001302C8	~(S|(P&D))	-
14	00140869	~(D|~(P^S))	-
15	001502C9	~(D|(P&S))	-
16	00165CCA	P^(S^(D&~(P&S)))	-
17	00171D54	~(S^((S^P)&(S^D)))	-
18	00180D59	(P^S)&(P^D)	-
19	00191CC8	~(S^D&~(P&S)))	-
1A	001A06C5	P^(D|(S&P))	-
1B	001B0768	~(S^(D&(P^S)))	-
1C	001C06CA	P^(S|(P&D))	-
1D	001D0766	~(D^(S&(P^D)))	-
1E	001E01A5	P^(S|D)	-
1F	001F0385	~(P&(S|D))	-
20	00200F09	P&~S&D	-
21	00210248	~(S|(P^D))	-
22	00220326	~S&D	-
23	00230B24	~(S|(P&~D))	-
24	00240D55	(S^P)&(S^D)	-
25	00251CC5	~(P^(D&~(S&P)))	-
26	002606C8	S^(D|((P&S))	-
27	00271868	S^(D|~(P^S))	-
28	00280369	D&(P^S)	-
29	002916CA	~(P^(S^(D|(P&S))))	-
2A	002A0CC9	D&~(P&S)	-
2B	002B1D58	~(S^((S^P)&(P&D)))	-
2C	002C0784	S^(P&(S|D))	-
2D	002D060A	P^(S|~D)	-
2E	002E064A	P^(S|(P^D))	-
2F	002F0E2A	~(P&(S|~D))	-
30	0030032A	P&~S	-
31	00310B28	~(S|(~P&D))	-
32	00320688	S^(P|S|D)	-
33	00330008	~S	NOTSRCCOPY
34	003406C4	S^(P|(S&D))	-
35	00351864	S^(P|~(S^D))	-
36	003601A8	S^(P|D)	-
37	00370388	~(S&(P|D))	-
38	0038078A	P^(S&(P|D))	-
39	00390604	S^(P|~D)	-
3A	003A0644	S^(P^(S^D))	-
3B	003B0E24	~(S&(P|~D))	-
3C	003C004A	P^S	-
3D	003D18A4	S^(P|~(S|D))	-
3E	003E1B24	S^(P|(~S&D))	-
3F	003F00EA	~(P&S)	-
40	00400F0A	P&S&~D	-
41	00410249	~(D|(P^S))	-
42	00420D5D	(S^D)&(P^D)	-

RASTER OP.	ROP CODE	BOOLEAN OPERATION	COMMON NAME
	00431CC4	~(S^(P&~(S&D)))	-
44	00440328	S&~D	SRCERASE
45	00450B29	~(D|(P&~S))	-
46	004606C6	D^(S|(P&D))	-
47	0047076A	~(P^(S&((P^D)))	-
48	00480368	S&(P^D)	-
49	004916C5	~(P^(D^(S|(P&D))))	-
4A	004A0789	D^(P&(S|D)	-
4B	004B0605	P^(~S|D)	-
4C	004C0CC8	S&~(P&D)	-
4D	004D1954	~(S^((P^S)|(S^D)))	-
4E	004E0645	P^(D|(P^S))	-
4F	004F0E25	~(P&(~S|D))	-
50	00500325	P&~D	-
51	00510B26	~(D|(~P&S))	-
52	005206C9	D^(P|(S&D))	-
53	00530764	~(S^(P&(S^D)))	-
54	005408A9	~(D|~(P|S))	-
55	00550009	~D	DSTINVERT
56	005601A9	D^(P|S)	-
57	00570389	~(D&(P|S))	-
58	00580785	P^(D&(P|S))	-
59	00590609	D^(P|~S)	-
5A	005A0049	P^D	PATINVERT
5B	005B18A9	D^(P|~(S|D))	-
5C	005C0649	D^(P|(S^D))	-
5D	005D0E29	~(D&(P|~S))	-
5E	005E1B29	D^(P|(S&~D))	-
5F	005F00E9	~(P&D)	-
60	00600365	P&(S^D)	-
61	006116C6	~(D^(S^(P|(S&D))))	-
62	00620786	D^(S&(P|D))	-
63	00630608	S^(~P|D)	-
64	00640788	S^(D&(P|S))	-
65	00650606	D^(~P|S)	-
66	00660046	S^D	SRCINVERT
67	006718A8	S^(D|~(P|S))	-
68	006858A6	~(D^(S^(P|~(S|D))))	-
69	00690145	~(P^(S^D))	-
6A	006A01E9	D^(P&S)	-
6B	006B178A	~(P^(S^(D&(S|P))))	-
6C	006C01E8	S^(P&D)	-
6D	006D1785	~(P^(D^(S&(P|D))))	-
6E	006E1E28	S^(D&(P|~S))	-
6F	006F0C65	~(P&~(S^D))	-
70	00700CC5	P&~(S&D)	-
71	00711D5C	~(S^((S^D)&(P^D)))	-
72	00720648	S^(D|(P^S))	-
73	00730E28	~(S&(~P|D))	-
74	00740646	D^(S|(P^D))	-
75	00750E26	~(D&(~P|S))	-
76	00761B28	S^(D|(P&~S))	-
77	007700E6	~(S&D)	-
78	007801E5	P^(S&D)	-
79	00791786	~(D^(S^(P&(S|D))))	-
7A	007A1E29	D^(P&(S|~D))	-
7B	007B0C68	~(S&~(P^D))	-

RASTER OP.	ROP CODE	BOOLEAN OPERATION	COMMON NAME
7C	007C1E24	S^(P&(~S|D))	-
7D	007D0C69	~(D&~(S^P))	-
7E	007E0955	(P^S)|(S^D)	-
7F	007F03C9	~(P&S&D)	-
80	008003E9	P&S&D	-
81	00810975	~((P^S)|(S^D))	-
82	00820C49	~(P^S)&D	-
83	00831E04	~(S^(P&(~S|D)))	-
84	00840C48	S&~(P^D)	-
85	00851E05	~(P^(D&(~P|S)))	-
86	008617A6	D^(S^(P&(S|D)))	-
87	008701C5	~(P^(S&D))	-
88	008800C6	S&D	SRCAND
89	00891B08	~(S^(D|(P&~S)))	-
8A	008A0E06	(~P|S)&D	-
8B	008B0666	~(D^(S|(P^D)))	-
8C	008C0E08	S&(~P|D)	-
8D	008D0668	~S(^(D|(P^S)))	-
8E	008E1D7C	S^((S^D)&(P^D))	-
8F	008F0CE5	~(P&~(S&D))	-
90	00900C45	P&~(S^D)	-
91	00911E08	~(S^(D&(P|~S)))	-
92	009217A9	D^(P^(S&(P|D)))	-
93	009301C4	~(S^(P&D))	-
94	009417AA	P^(S^(D&(P|S)))	-
95	009501C9	~(D^(P&S))	-
96	00960169	P^S^D	-
97	0097588A	P^(S^(D|~P|S)))	-
98	00981888	~(S^(D|~(P|S)))	-
99	00990066	~(S^D)	-
9A	009A0709	(P&~S)^D	-
9B	009B07A8	~(S^(D&(P|S)))	-
9C	009C0704	S^(P&~D)	-
9D	009D07A6	~(D^(S&(P|D)))	-
9E	009E16E6	(S^(P|(S&D)))^D	-
9F	009F0345	~(P&(S^D))	-
A0	00A000C9	P&D	-
A1	00A11B05	~(P^(D|(~P&S)))	-
A2	00A20E09	(P|~S)&D	-
A3	00A30669	~(D^(P|(S^D)))	-
A4	00A41885	~(P^(D|~(P|S)))	-
A5	00A50065	~(P^D)	-
A6	00A60706	(~P&S)^D	-
A7	00A707A5	~(P^(D&(P|S)))	-
A8	00A803A9	(P|S)&D	-
A9	00A90189	~((P|S)^D)	-
AA	00AA0029	D	-
AB	00AB0889	~(P|S)|D	-
AC	00AC0744	S^(P&(S^D))	-
AD	00AD06E9	~(D^(P|(S&D)))	-
AE	00AE0B06	(~P&S)|D	-
AF	00AF0229	~P|D	-
B0	00B00E05	P&(~S|D)	-
B1	00B10665	~(P^(D|(P^S)))	-
B2	00B21974	S^((P^S)|(S^D))	-
B3	00B30CE8	~(S&~(P&D))	-
B4	00B4070A	P^(S&~D)	-

RASTER OP.	ROP CODE	BOOLEAN OPERATION	COMMON NAME
B5	00B507A9	~(D^(P&(S|D)))	-
B6	00B616E9	D^(P^(D|(P&D)))	-
B7	00B70348	~(S&(P^D))	-
B8	00B8074A	P^(S&(P^D))	-
B9	00B906E6	~(D^(S|(P&D)))	-
BA	00BA0B09	(P&~S)|D	-
BB	00BB0226	~S|D	MERGEPAINT
BC	00BC1CE4	S^(P&~(S&D))	-
BD	00BD0D7D	~((P^D)&(S^D))	-
BE	00BE0269	(P^S)|D	-
BF	00BF08C9	~(P&S)|D	-
C0	00C000CA	P&S	MERGECOPY
C1	00C11B04	~(S^(P|(~S&D)))	-
C2	00C21884	~(S^(P|~(S|D)))	-
C3	00C3006A	~(P^S)	-
C4	00C40E04	S&(P|~D)	-
C5	00C50664	~(S^(P|(S^D)))	-
C6	00C60708	S^(~P&D)	-
C7	00C707AA	~(P^(S&(P|D)))	-
C8	00C803A8	S&(P|D)	-
C9	00C90184	~(S^(P|D))	-
CA	00CA0749	D^(P&(S^D))	-
CB	00CB06E4	~(S^(P|(S&D)))	-
CC	00CC0020	S	SRCCOPY
CD	00CD0888	S|~(P|D)	-
CE	00CE0B08	S|(~P&D)	-
CF	00CF0224	S|~P	-
D0	00D00E0A	~(^(S|(P^D)))	-
D1	00D1066A	P^(~S&D)	-
D2	00D20705	~(S^(P&(S|D)))	-
D3	00D307A4	S^((P^S)&(P^D))	-
D4	00D41D78	(~(D&~(P&S))	-
D5	00D50CE9	P^(S^(D|(P&S)))	-
D6	00D616EA	~(D&(P^S))	-
D7	00D70349	~(D&(P&S))	-
D8	00D80745	P^(D&(P^S))	-
D9	00D906E8	~(S^(D|(P&S)))	-
DA	00DA1CE9	D^(P&~(S&D))	-
DB	00DB0D75	~((P^S)&(S^D))	-
DC	00DC0B04	S|(P&~D)	-
DD	00DD0228	S|~D	-
DE	00DE0268	S|(P^D)	-
DF	00DF08C8	S|~(P&D)	-
E0	00E003A5	P&(D|S)	-
E1	00E10185	~(P^(S|D))	-
E2	00E20746	D^(S&(P^D))	-
E3	00E306EA	~(P^(S|(P&D)))	-
E4	00E40748	S^(D&(P^S))	-
E5	00E506E5	~(P^(D|(P&S)))	-
E6	00E61CE8	S^(D&~(P&S))	-
E7	00E70D79	~((P^S)&(P^D))	-
E8	00E81D74	S^((P^S)&*S^D))	-
E9	00E95CE6	~(D^(S^(P&~(S&D))))	-
EA	00EA02E9	(P&S)|D	-
EB	00EB0849	~(P^S)|D	-
EC	00EC02E8	S|(P&D)	-
ED	00ED0848	S|(~(P^D)	-

RASTER OP.	ROP CODE	BOOLEAN OPERATION	COMMON NAME
EE	00EE0086	S|D	SRCPAINT
EF	00EF0A08	~P|S|D	-
F0	00F00021	P	PATCOPY
F1	00F10885	P|(~(S|D)	-
F2	00F20B05	P|(~S&D)	-
F3	00F3022A	P|~S	-
F4	00F40B0A	P|(S&~D)	-
F5	00F50225	P|~D	-
F6	00F60265	P|(S^D)	-
F7	00F708C5	P|(~(S&D)	-
F8	00F802E5	P|(S&D)	-
F9	00F90845	P|~(S^D)	-
FA	00FA0089	P|D	-
FB	00FB0A09	P|~S|D	PATPAINT
FC	00FC008A	P|S	-
FD	00FD0A0A	P|S|~D	-
FE	00FE02A9	P|S|D	-
FF	00FF0062	1	WHITENESS

Bibliography

Arnheim, Rudolf. *Art and Visual Perception.* Berkeley, CA: University of California Press, 1974.

Artwick, Bruce A. *Applied Concepts in Microcomputer Graphics.* Englewood Cliffs: Prentice-Hall, 1984.

Bargen, Bradley and Peter Donnelly. *Inside DirectX.* Microsoft Press, 1998.

Box, Don. *Essential COM*. Addison-Wesley, 1998.

Bronson, Gary. *A First Book of C++*. West Publishing Company, 1995.

Cluts, Nancy Winnick. *Programming the Windows 95 User Interface.* Microsoft Press, 1995.

Coelho, Rohan and Maher Hawash. *DirectX, RDX, RSX, and MMX Technology.* Addison-Wesley, 1998.

Conger, James L. *Windows API Bible: the Definite Programmer's Reference.* Waite Group, 1992.

Conrac Corporation. *Raster Graphics Handbook.* New York: Van Nostrand Reinholt, 1985.

Cooper, Alan. *About Face: Essentials of User Interface Design.* IDG Books, 1995.

Egerton, P A. and W. S. Hall. *Computer Graphics: Mathematical First Steps.* Prentice Hall, 1999.

Doty, David B. *Programmer's Guide to the Hercules Graphics Cards.* Reading, MA: Addison-Wesley, 1988.

Ezzell, Ben and Jim Blaney. *Windows 98 Developer's Handbook.* Sybex, 1998.

Ferraro, Richard F. *Programmer's Guide to EGA and VGA Cards.* Reading, MA: Addison-Wesley, 1988.

Foley, James D., Andries van Damm, Steven K. Feiner, and John P Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 1997.

Glidden, Rob. *Graphics Programming with Direct3D.* AddisonWesley, 1997.

Giambruno, Mark. *3D Graphics & Animation.* New Riders, 1997.

Giesecke, Frederick E et al. *Engineering Graphics.* Fourth Edition. Macmillan, 1987.

Harrington, Steven. *Computer Graphics: A Programming Approach.* New York: McGraw-Hill, 1983.

Hearn, Donald and M. Pauline Baker. *Computer Graphics.* Prentice-Hall, 1986.

Hearn, Donald, and M. Pauline Baker. *Computer Graphics: C Version.* Second Edition. Prentice-Hall, 1997.

Hoggar, S.G. *Mathematics for Computer Graphics.* Cambridge, 1992.

IBM Corporation. *Technical Reference, Personal Computer.* Boca Raton: IBM, 1984.

IBM Corporation. *Technical Reference, Personal System/2.* Boca Raton: IBM, 1987.

IBM Corporation. *Personal System/2 and Personal Computer BIOS Interface Technical Reference.* Boca Raton: IBM, 1987.

IBM Corporation. *Technical Reference, Options and Adapters.* Boca Raton: IBM, 1986.

IBM Corporation. *Technical Reference, Options and Adapters. XGA Video Subsystem.* Boca Raton: IBM, 1986.

IBM Corporation. *XGA Video Subsystem Hardware User's Guide.* Boca Raton: IBM, 1990.

IBM Corporation. *Personal System/2 Hardware Interface Technical Reference - Video Subsystems.* Boca Raton: IBM, 1992.

Kawick, Mickey. *Real-Time Strategy Game Programming using DirectX 6.0.* Wordware, 1999.

Kernigham, Brian W. and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 1978.

Kliewer, Bradley Dyck. *EGA/VGA A Programmer's Reference Guide.* New York: McGraw-Hill, 1988.

Kold, Jason. *Win32 Game Developer's Guide with DirectX 3.* Waite Group Press, 1997.

Kovach, Peter J. *The Awesome Power of Direct3D/DirectX.* Manning, 1998.

Mandelbrot, Benoit B. *The Fractal Geometry of Nature.* W.T. Freeman and Co., 1982.

Microsoft Corporation. *Programmer's Guide to Microsoft Windows 95.* Microsoft Press, 1995.

Micosoft Corporation. *The Windows Interface Guidelines for Software Design.* Microsoft Press, 1995.

Microsoft Corporation. *DirectX 5 SDK documentation.* 1998.

Microsoft Corporation. *DirectX 6 SDK documentation.* 1999.

Microsoft Corporation. *DirectX 7 SDK documentation.* 1999.

Microsoft Corporation. *DirectX 8 SDK documentation.* 2000.

Minasi, Mark. *Secrets of Effective GUI Design.* Sybex, 1994.

Moller, Thomas, and Eric Haines. *Real-Time Rendering.* A. K. Peters Ltd., 1999.

Morris, Charles W *Signs, Language and Behaviors.* George Braziller, 1955.

O'Rourke, Michael. *Principles of Three Dimensional Computer Animation.* Norton, 1998.

Petzold, Charles. *Programming Windows. Fifth Edition.* Microsoft Press, 1999.

Pokorny, Cornel K. and Curtis F. Gerald. *Computer Graphics: The Principles Behind the Art and Science.* Irvine, CA: Franklin, Beedle & Associates .

Ratner, Peter. *3-D Human Modeling and Animation.* Wiley, 1998.

Rector, Brent E. and Joseph M. Newcomer. *Win32 Programming.* Addison-Wesley, 1997.

Redmond, Frank E. III. *DCOM- Microsoft Distributed Component Object Model.* IDG Books, 1997.

Rimmer, Steve. *Windows Bitmapped Graphics.* McGraw-Hill, 1993.

Rimmer, Steve. *Supercharged Bitmapped Graphics.* New York: McGraw-Hill, 1992.

Rimmer, Steve. *SuperVGA Graphics Programming Secrets.* New York: McGraw-Hill, 1993.

Richter, Jake, and Bud Smith. *Graphics Programming for the 8514/A.* Redwood City, CA: M & T Books, 1990.

Ritcher, Jeffrey. *Advanced Windows.* Third Edition. Microsoft Press, 1997.

Rogerson, Dale. *Inside COM. Microsoft Press,* 1997.

Root, Michael and James Boer. *DirectX Complete.* McGraw-Hill, 1999.

Salmon, Rod and Mel Slater. *Computer Graphics: Systems and Concepts.* AddisonWesley, 1987.

Sanchez, Julio and Maria P. Canton. *High Resolution Video Graphics.* McGraw-Hill, 1994.

Sanchez, Julio and Maria P. Canton. *Space Image Processing.* CRC Press, 1999.

Sanchez, Julio and Maria P Canton. *Windows Graphics Programming.* M & T Books, 1999.

Sanchez, Julio and Maria P. Canton. *DirectX 3D Graphics Programming Bible.* M & T Books. 2000.

Schildt, Herbert. *C++ The Complete Reference. Second Edition.* McGraw-Hill, 1995.

Schildt, Herbert. *Windows 98 Programming from the Ground Up.* Osborne, 1998.

Simon, Richard. *Win32 Programming API Bible.* Waite Group Press, 1996.

Stein, Michael L., Eric Bowman, and Gregory Pierce. *Direct3D Professional Reference.* New Riders, 1997.

Sutty, George and Steve Blair. *Advanced Programmer's Guide to SuperVGAs.* New York: Simon & Schuster, 1990.

Thompson, Nigel. *3D Graphics Programming for Windows 95.* Microsoft Press, 1996.

Timmins, Bret. *DirectDraw Programmming.* M & T Books, 1996.

Trujillo, Stan. *High Performance Windows Graphics Programming.* Coriolis Group Books, 1998.

Trujillo, Stan. *Cutting-Edge Direct3D Programming.* Coriolis Group Books, 1996.

VESA. *Super VGA BIOS Extension,* June 2, 1990. San Jose, CA: VESA, 1990.

VESA. *Super VGA Standard, Version 1.2,* October 22, 1991. San Jose, CA: VESA, 1991.

VESA. *XGA Extensions Standard, Version 1.0,* May 8, 1992. San Jose, CA: VESA, 1992.

Walmsley, Mark. *Graphics Programming in C++*. Springer, 1998.

Watt, Alan, and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice.* Addison-Wesley, 1992.

Watt, Alan, and Fabio Policarpo. *The Computer Image.* Addison-Wesley, 1998.

Walnum, Clayton. *Windows 95 Game SDK Strategy Guide.* Que, 1995.

Young, Michael J. *Introduction to Graphics Programming for Windows 95: Vector Graphics Using C++*. AP Professional, 1996.

Zaratian, Beck. *Microsoft Visual C++ Owner's Manual. Version 5.0.* Microsoft Press, 1997.