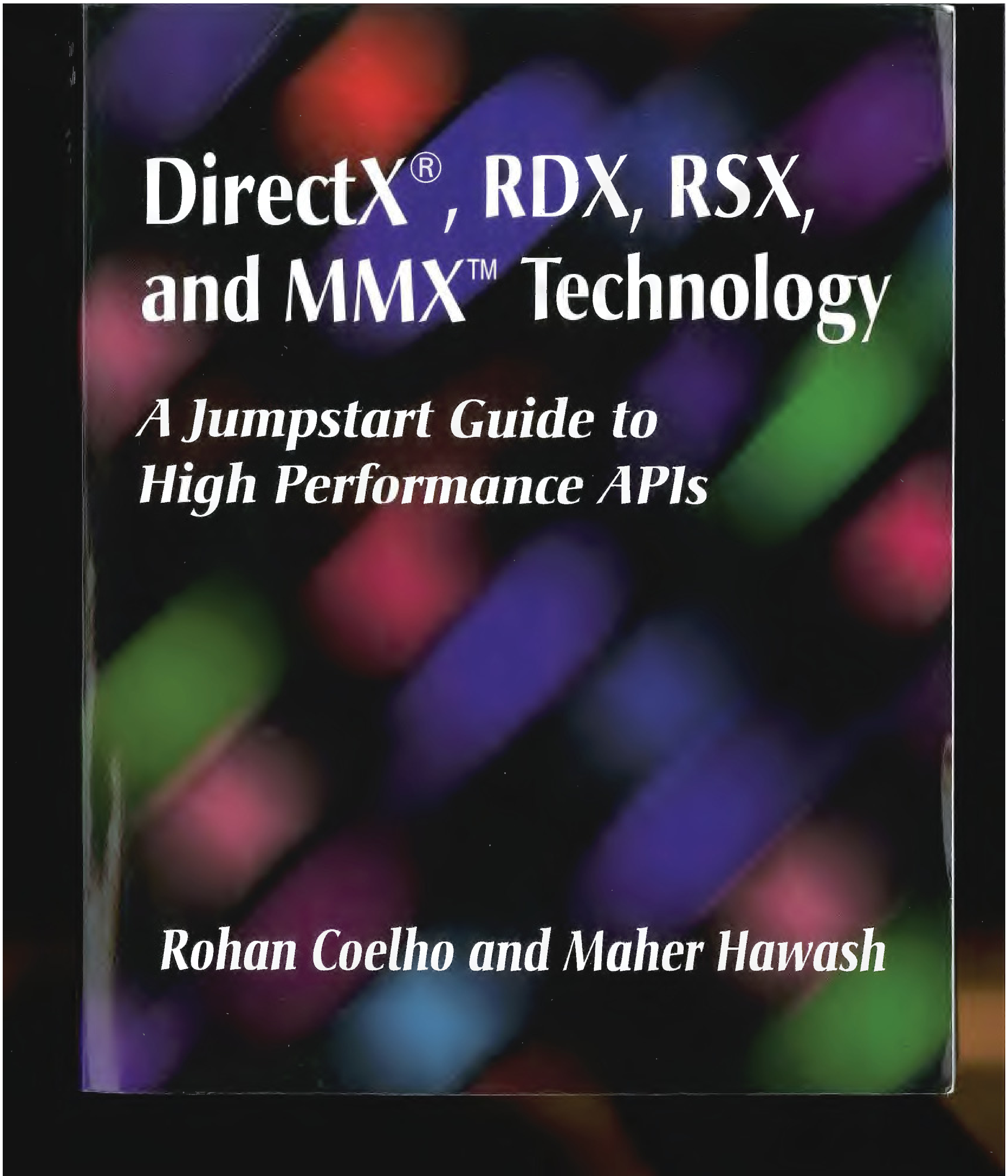


# DirectX®, RDX, RSX, and MMX™ Technology

*A Jumpstart Guide to High Performance APIs*

**Rohan Coelho and Maher Hawash**

Dept 6A10

# DirectX®, RDX, RSX, and MMX™ Technology

# DirectX®, RDX, RSX, and MMX™ Technology

## A Jumpstart Guide to High Performance APIs

Rohan Coelho
and
Maher Hawash

ADDISON-WESLEY DEVELOPERS PRESS
**An imprint of Addison Wesley Longman, Inc.**

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

*Library of Congress Cataloging-in-Publication Data*

Coelho, Rohan.
DirectX®, RDX, RSX, and MMX™ technology : a jumpstart guide to high performance APIs / Rohan Coelho and Maher Hawash.
p. cm.
Includes index.
ISBN 0-201-30944-0
1. Multimedia systems. 2. DirectX. 3. Intel Realistic display mixer. 4. RSX (CompComputer file : Digital Equipment Corporation) 5. MMX technology. I. Hawash, Maher. II. Title.
QA76.575.C64 1998
006.7'768--dc21 97-33102
CIP

Sponsoring Editor: Mary Treseler
Project Manager: John Fuller
Cover design: Chris Norum
Text design: Vicki Hochstedler
Set in 11-point Minion by Octal Publishing

1 2 3 4 5 6 7 8 9—MA—0100999897
*First printing, December 1997*

To my parents, Mofeed and Sameeha;
To my wife Lisa, my son Jared, and baby on its way;
To my nephew Ahmad and the rest of my family;
I dedicate this book. —Maher

To my immediate family: Dad, Mom, Gail, Carmen, and Sarah;
To my extended family, blood relatives and others;
And to several others, significant but unnamed;
Thanks for touching my life. —Rohan

Special thanks to
Emilie Lengel and Gerald Holzhammer
For believing in us.

# Contents

30

382

# Preface

386

## Why Read This Book?

### There's Lots of New Stuff to Learn

In the past few years, the pace of technology growth has been exhilarating. Microsoft launched Windows 95. Intel debuted the Pentium, Pentium Pro, and MMX technology processors. Netscape burst the Internet pipe with a new class of applications and architectures. These companies and others paraded out a slew of new multimedia architectures. And you've never before felt so lost in space.

Maybe you're familiar with programming for Windows 95 and now want to deliver Windows 95 multimedia applications, and you're wondering where to start. Or maybe you've programmed multimedia for DOS/Windows 3.1, and now you're scrambling to learn Windows 95, learn the new computing environment, and then learn to deliver high-performance multimedia in this environment.

Well, several new architectures have been introduced to help you deliver high-performance multimedia under Windows 9x,[1] such as DirectDraw

1. Windows 9x stands for both Windows 95 and the upcoming Windows 98.

DirectSound*, Direct3D*, DirectShow*, RealMedia*, Realistic Sound Experience (3D RSX), Realistic Display Mixer (RDX), and so forth. But now you've got to learn these new architectures, and you've got this steep learning curve on your hands.

On the hardware frontier, the power of personal computers has increased at a dramatic pace—both in processor and peripheral power. The Pentium, Pentium Pro, Pentium II, and MMX technology processors, the accelerated graphics port (AGP) bus, and the various graphics hardware accelerators are recent hardware advancements that affect multimedia performance. Surely your applications would sizzle if you mastered these advancements. But mastering these advancements only increases the learning curve.

And, of course, the Internet adds yet another dimension to the puzzle. The new programming space includes Internet browsers and their plug-ins; programming languages such as Java, HTML, and VRML; Internet architectures such as ActiveX, RealMedia, and a huge list of applications such as Internet Phones and Chat Worlds. More to learn, more to wade through, more time to spend.

## Lightening the Learning Burden

As multimedia developers, we constantly investigate, evaluate, or learn these new technologies. Our typical sources are technical reference manuals and sample applications. With so many recent products, we've got a huge quantity of material to wade through. When time is precious, as it invariably is, just getting started can be an overwhelming problem. Spending time getting started eats away from time allocated for finishing touches and product testing. And overall quality suffers when we've spent too much time just getting up to speed.

Wouldn't it be nice if there were a simple way to *just get started?* To grasp the bare essentials and leave the esoteric stuff for on-the-job training (those need-to-know moments)? To steer clear of performance pitfalls? Well, do we have a deal for you. We, the authors, have been involved in various aspects of multimedia development on the PC for five long years. Through our employment at Intel and through our relationships with Microsoft and other key players, we've had the privilege to influence the architectures of processors, peripherals, platforms, and software components toward the betterment of multimedia on the PC. During that time, we've done our fair share of defining, reviewing, and implementing numerous multimedia architectures, both software and hardware.

With this book, we hope to use our internal vantage point to give you a jump start to high-performance multimedia development for Windows 9x. We'd like to help you cut to the chase; focus on the bare necessities; stick to the essentials; and jump-start a variety of offerings. What's more, we're hoping to take you a step beyond getting started—to extracting performance.

We hope to provide you with a quick start to a wide spectrum of multimedia advancements for Windows 9x. We hope to answer questions like *Where do I start? What do I really need? How little can I get away with? How do I get it to run faster?*

A dose of caution: there's more than one way to get jump-started and more than one way to extract performance. We'll share our experiences with you, show you "a" way. We hope you'll come away with some tricks, of course, but more important, we hope you'll come away with a thought process—an approach.

We've tried to maintain a light flavor. We hope you'll have some fun along the way.

INTRODUCTION

# Organization and Conventions

**WHY READ THIS CHAPTER?**

Since we're talking about the organization of the chapters, it's only appropriate to note that all chapters start with the question above: "Why Read This Chapter?" Our purpose is to present you with a summary of what we intend to cover in the chapter. We recommend that you read the segment to see if what you will get is what you want.

This chapter shows you how we arranged the book, to help you get the most benefit out of it. In the following pages, we

- describe who we wrote the book for,
- show you how we present our material,
- outline the organization of the book, providing overviews of each chapter,
- show some conventions we use to highlight information, and
- list the tools that you'll need when working with the companion CD.

## I.1 About the Book

When we started to outline the material for this book, we quickly recognized that we would be covering a lot of ground. We struggled with what to present and what to ignore. We asked ourselves, "What kind of a book would we have wanted when we started doing whatever we started?"

### I.1.1 Where We're Coming From

Because of our roles at Intel, we've had the good fortune to work on Windows multimedia architectures right from their infancy. In our work we applied both our architectural and our CPU optimization skills, and we used them across a wide range of multimedia avenues.

Of late, we'd been called upon to help a number of software companies with their multimedia problems. Intel funded and continues to fund these software activities, in the interest of encouraging overall PC sales by promoting new uses for the PC; and in the interest of boosting demand for newer, higher-performance PCs, by promoting CPU-intensive applications.

To address multimedia performance issues, we would typically optimize critical sections of the assembly code. However, when the performance bottlenecks are at the system level, we would have to demonstrate the use of (or even develop) appropriate Windows multimedia architectures.

And this led us to think that we could write a book to offer the same thing to a larger audience, to help others get started on a number of different multimedia architectures, to help others extract a lot of performance from the PC multimedia architecture.

### I.1.2 Where We're Not Venturing

We can't claim to be *The Experts* in PC multimedia. The field is too big, and there are too many excellent software engineers out there for us to presume such a status. Nonetheless we feel we've been down some paths before and can share that experience with you, to get you started.

We didn't want to delve deeply into the gory details of any single architecture; that's what the reference documents are for. Instead, we decided it would be better to get you started with the architectures, and we're sure that your application needs will steer your further learning.

On the flip side, with the breadth of architectures we wanted to cover, we knew we would have to skip basic concepts to do the architectures any justice. So we've presumed some prerequisite knowledge and targeted the book to reasonably experienced programmers. We also narrowed our selections to focus on recent/emerging advancements so as to avoid merely putting a fresh spin on previously published information.

### I.1.3 Who Should Read This Book

OK, so who did we think we could help? It was clear to us that our readers would

- already know how to program under Windows,
- understand multimedia concepts and terminology,
- be familiar with programming with C, C++, and for some sections, even assembly language (Intel Architecture), and
- appreciate, or even prefer, a hands-on learning approach (like to learn by being pointed in the right direction and then be free to find their own way around).

## I.2 Chapter Organization

Armed with a clearer picture of our identity and our readers, we were able to outline our approach. On the one hand, we wanted to get our readers started quickly on the latest multimedia architectures. On the other hand, we wanted to show them how to extract high performance on Intel Architecture multimedia PCs. Ergo, *we have decided to provide simple samples!*

We have partitioned the book into six major parts. Each part focuses on a specific area of multimedia, with its chapters sequentially building on each other. We specifically tried to use the same or similar samples within each part. There are a total of twenty-three chapters in the book. We concentrated on making each chapter brief, less than thirty pages each, so that wordiness wouldn't dilute our subject matter. We deliberately chose the compact format to improve retention (make it less likely for readers to forget what was said before).

Let's take a closer look at what we cover in each of the parts/chapters.

### Part I: Surveying Multimedia

**Chapter 1 Overview of Media on the PC.** This chapter gives just a small overview of current multimedia architectures on the PC. We give a brief pass on the Graphics Device Interface (GDI), DirectDraw, DirectSound, Direct3D, DirectShow, Realistic Display Mixer (RDX), and Realistic Sound Experience (3D RSX).

**Chapter 2 Processor Architecture Overview.** Here we approach media from a hardware perspective. We give a high-level architectural overview of the Pentium, Pentium Pro, the Pentium processor with MMX technology, and the Pentium II processors. We also touch on the system point of view and why it is essential to optimize for the system as well as for the processor.

## Part II: Sprites, Backgrounds, and Primary Surfaces

**Chapter 3 Simple Sprites in GDI.** This chapter introduces the concept of transparent sprites and backgrounds under Windows. We show you how to draw backgrounds and transparent sprites using GDI.

**Chapter 4 Sprites with DirectDraw Primary Surfaces.** We take our sprite to the next level with a DirectDraw Primary surface. We show you how to create a Primary surface to get direct access to the display screen. We then rewrite the sprite to be drawn onto a Primary surface and compare its performance with the GDI implementation.

**Chapter 5 Hardware Acceleration via DirectDraw.** Here we show you how to implement our beloved sprite using hardware Bltters on graphics adapters. We then show you how to use Page Flipping hardware to minimize the cost of double-buffering incurred in the Primary surface implementation. Finally, we compare the performance gain of this implementation with the Primary surface implementation.

**Chapter 6 RDX: High-Performance Mixing with a High-Level API.** Realistic Display Mixer (RDX) provides a high-level mixing interface without sacrificing performance. RDX uses hardware acceleration if available; otherwise it uses assembly code tuned for various processor flavors. We show you how to implement sprites with RDX, and we compare the performance of this implementation to GDI and DirectDraw implementations.

## Part III: Making the Media Mix

**Chapter 7 Video under Windows.** This chapter introduces current multimedia architectures under Windows, including Multimedia Command Interface (MCI), Video for Windows (VFW), QuickTime for Windows (QTW), and ActiveMovie.

**Chapter 8 DirectShow Filters.** We start with an overview of the DirectShow filter graph architecture and show you how to use the graph editor to manipulate filters. We then show you how to build source, transform, and rendering filters, and explain how the connection mechanism works. Next we discuss filter registration, custom interfaces, and filter property pages.

**Chapter 9 DirectShow Applications.** Building on the previous chapter, we show you how to use filters from an application. We show you how to build a filter graph directly using the DirectShow COM interface and the DirectShow control interface. We then show you how to access custom interfaces and property pages.

**Chapter 10 Mixing Sprites, Backgrounds, and Videos.** In this chapter we show you how to use RDX to access DirectShow filters. We also explain how simple it can be to overlay a sprite on top of a video and even a video on top of another video.

**Chapter 11 Streaming down the Superhighway with RealMedia.** In this chapter we look at the latest architecture from RealNetworks, which is a cross-platform architecture. We'll show you how to build custom File-Format and Rendering plug-ins, which allow you to stream custom data types over the Internet. We'll also show you how to use RealMedia audio services.

## Part IV: Playing and Mixing Sound with DirectSound and RSX 3D

**Chapter 12 Audio Mixing with DirectSound.** We start the chapter with an overview of Microsoft's DirectSound. Then we show you how to play a simple WAV file. We then teach you how to mix two sound files and how to control the format of the final output—after mixing.

**Chapter 13 Realistic 3D Sound Experience: RSX 3D.** RSX provides a high-level programming model optimized for the Intel Architecture. We start the chapter with an overview of Intel's RSX 3D audio, and then we show you how to play one or more WAV files with it. We then give you an overview of RSX's 3D sound model and show you how to achieve a realistic sound experience with it.

## Part V: Welcome to the Third Dimension

**Chapter 14 An Introduction to Direct3D.** We kick off our 3D section with background on 3D on the PC and an overview of Microsoft's Direct3D. Then we discuss Direct3D's modes and its Immediate mode architecture. The main purpose of this chapter is to give you the bare minimum code needed to render a triangle in Direct3D's Immediate mode.

**Chapter 15 Embellishing Our Triangle with Backgrounds, Shading, and Textures.** In this chapter we add some bells and whistles to the default triangle we helped you create in the previous chapter. We work through shading options, texture mapping, and Z-Buffering, and we also render Direct3D-based backgrounds.

**Chapter 16 Understanding and Enhancing Direct3D Performance.** In previous chapters we focused on getting our application running. In this chapter we focus on how fast Direct3D performs. We then use the Ramp driver to increase render performance and measure our improvements.

**Chapter 17 Mixing 3D with Sprites, Backgrounds, and Videos.** We next look at integrating 3D with the media we worked with in previous parts (sprites, backgrounds, and videos). We walk you through displaying a 3D object in a 2D world, and we render a texture-mapped triangle with a video as a texture source.

## Part VI: Processors and Performance Optimization

**Chapter 18 The Pentium Family.** In the first chapter of this part we give you an architectural overview of the Pentium, Pentium Pro, and MMX technology processors. But first we define some of the terms and concepts that are used throughout Part VI. We then give you the 10,000-foot view of these processors so that you will begin to see how they differ from one another. Finally, we show you how to distinguish between the different flavors of these processors.

**Chapter 19 The Pentium Processor.** This chapter gives you a detailed view of the internal components of the Pentium processor and shows you what's important so that each component can attain optimal performance. We then analyze the assembly sprite from Part II for performance problems and show you how to fix them.

**Chapter 20 The Pentium Processor with MMX Technology.** Here we introduce the MMX technology instruction set, registers, and data types. We also outline the MMX scheduling rules and show you how to use them. We rewrite the sprite sample using MMX technology instructions and analyze it for performance bottlenecks.

**Chapter 21 VTune and Other Performance Optimization Tools.** Since hand tuning is a tedious and time-consuming process, we introduce VTune, a tool to help you analyze your code and pinpoint performance issues with ease. We show you how to use VTune to analyze the MMX sprite sample from the previous chapter. Then we show you how to use the hot-spot system monitor and the static and dynamic analyzers. We also teach you how to use the Time Stamp Counter (TSC) and the internal Pentium event counters, and the PMonitor event counter library.

**Chapter 22 The Pentium II Processor.** In this chapter you'll get some exposure to the Pentium II processor, the latest processor from Intel. We list new processor features and point out optimization issues specific to this processor. We introduce you to the use of the Write Combining memory type so that you can achieve better graphics performance.

**Chapter 23 Memory Optimization: Know Your Data.** We dedicate this chapter to system issues. Knowing where your data comes from and where it goes to is essential for achieving overall application performance and multimedia throughput. In this chapter we discuss the L1 and L2 caches, the PCI bus, and how to organize your writes to memory in the most efficient manner.

**Epilogue: The Finale.** In the last pages of the book we describe what we will see in the future in terms of faster processors, tighter multimedia architectures, the Internet, advances in 3D, and multimedia.

**Web Site: The Annex.** Two additional chapters on the latest technologies from Microsoft, **DirectShow Capture** and **Direct3D Draw Primitives,** are available on our Web Site. Access it with the following URL:

http://www.awl.com/cseng/titles/0-201-30944-0

## I.3 Conventions Used in This Book

When we started writing the book, we experimented with a few ideas of how to convey our material without being too detailed. We decided to settle on a few conventions based on feedback that we received from our reviewers. Even though these conventions might seem obvious when you read the rest of this chapter, it might still be advantageous to browse through the next couple of pages.

### I.3.1 Part Map

At the beginning of each part, we have inserted a part listing that shows the highlights of each chapter in the part.

### 1.3.2 Chapter Prologue and Epilogue

As we mentioned before, at the beginning of each chapter, we ask and answer the question "*Why read this chapter?*" by summarizing the material covered in that chapter. At the end of each chapter we reiterate what we have covered and what you should have learned from the chapter.

### 1.3.3 Code Listings

*Side Note:* A note about the text next to it

All our code is inserted between a thick and a thin rule and appears in a different font, as shown below. We use bubbles and side comments to highlight key points in the code and to present our material compactly. Extra special information merits a gray-shaded background.

```
CCodeText::SubliminalMessage()
{
     if (YouHaveNotReadThisBook)            ◊ Side comment.
         BuyThisBook();
     else
         BuyItForSomeoneElse();
}
```

Bubble: Highlights information about the line pointed to.

Extra special bubble, merits a gray shading.

Notice also that if we have to repeat a portion of the code, we use the **bold** font for the newly added code.

```
DOLLAR CCodeText::SubliminalMessage()
{
     if (YouHaveNotReadThisBook)
         BuyThisBook();
     else
         BuyItForSomeoneElse();

     return SomeDollarValue;
}
```

We also use plenty of icons to emphasize a point or to point out something that's not obvious. We use a star to point out the best result from a procedure and a CD icon to alert readers to when they might want to experiment with the CD that accompanies this book. Notes, and cautions are also set apart from regular text for emphasis.

Finally, notice that the code listings in the book lack a lot of the error checking code, but the code on the companion CD has all the error checking code. We decided to do this for better clarity when we describe the material in this book, and leaving out error checking code reduces clutter.

### I.3.4 Coding Style

With reference to our coding style, for the code in the companion CD,

- We decided to use C++ to implement our code since it allows us to easily build the code of one chapter on the code of a previous chapter.
- For better performance, we avoid using local class declarations within our functions. Class variables declared on the stack are allocated and initialized every time the function is called, which could result in a negative impact on performance.
- We use macros for error checking so that we can easily change error reporting schemes while still retaining source file and line information.
- We use assembly language when we discuss performance optimization issues for the processor.

### I.3.5 Material on the CD

To make it easier to browse through the material on the CD, we use a web browser approach similar to what you use on the Internet. Once you insert the CD in the CD ROM drive, the *AutoPlay* feature of Windows 95 and Windows NT launches your default Internet browser[1] and displays the home page of the CD.

If the web page is not automatically displayed when you insert the CD, you can manually run the batch file *AutoRun.Bat* from the root directory of the CD. Make sure that you have a web browser installed.

### I.3.6 Material on the Internet

From the homepage of the CD, you can go to our web site on Addison Wesley's web server. On that server, you'll:

- Find two more chapters of the latest technologies from Microsoft, *DirectShow Capture* and *Direct3D Draw Primitives.*

- See the latest feedback and discussions of issues related to our book.

1. Internet Explorer 3.01 or Netscape 3.0 or later are required.

## I.4 On Our Measurements

We have designed the book with a strong performance overtone. We're constantly measuring the performance of implementation path and looking for better options. Our measurements were performed on a machine equipped with a Pentium processor with MMX technology, an S3 Trio64V+ graphics adapter with 24 MB of VRAM, and 32 MB of EDO memory.

The performance of any implementation is extremely data sensitive. A particular implementation may outweigh other options given an input data set, but change the data set or vary the output configuration and the option may not do quite as well. Over the course of this book, you will be shown comparisons of different implementation choices. We hope to give you a flavor of various costs as well. But ultimately you should use *your application,* with its own algorithms, data sets, and target configurations, as your decision-making yardstick.

## I.5 Tools Used in This Book

Finally, here is a list of the tools that you'll need to build the sample code on the CD:

| Tool | Version | Where to Find It |
|---|---|---|
| Visual C++ Compiler | 5.0 | Buy it |
| Macro Assembler | 6.11d | Buy it |
| DirectDraw SDK | 3.0 | MSDN (or with compiler) |
| Direct3D SDK | 3.0 | MSDN (or with compiler) |
| DirectSound SDK | 3.0 | MSDN (or with compiler) |
| DirectShow SDK | 2.0 | DirectShow SDK |
| Intel VTune | 2.4 | Evaluation copy on CD |
| Realistic 3D Sound Experience | 2.1 | on CD and http://www.intel.com |
| Realistic Display Mixer | 3.0 | on CD and http://www.intel.com |
| RealMedia SDK | beta 6.0 | at http://www.real.com |

PART I

# Surveying Multimedia

WE'D LIKE TO EXTEND AN ACKNOWLEDGEMENT TO TOM'S PANCAKE HOUSE WHERE THE FIRST SPARK OF THIS BOOK WAS LIT. DARLA, CHRISTA, DIANE AND GREG. THE SERVICE IS FRIENDLY, THE FOOD IS SUPERB, AND YOU'RE GUARANTEED TO GET SOME LIP; TO LE JOI CAFÉ WHERE HSU LI WHIPS UP SOME DELICIOUS MOCHAS; AND TO PATRICIA MOORE, ANITA, ANN BRYANT, SUSAN REICHERT, NAVEEN SACHDEV, NAJI HAMDAN, DONALD FROOM, A FEW OF THE SAINTS AT SAINT VINCENT'S HOSPITAL OF PORTLAND.

CHAPTER 1

# Overview of Media on the PC

**WHY READ THIS CHAPTER?**

This chapter introduces the current multimedia software architectures available on the PC. In this book we're only concerned with media architectures running on Windows 98 and Windows NT. We'll give you an overview of the following architectures and show you how they relate to each other:

- GDI, DirectDraw, and RDX;
- MCI, VFW, QTW, DirectShow, and RealMedia;
- WAVE, DirectSound, and RSX; and
- Direct3D

## 1.1 Background

Graphics hardware on the PC has evolved from monochrome CGA graphics standards through EGA, VGA, and Super VGA to the graphics cards of today, which offer custom display formats and custom graphics acceleration hardware.

Similarly, audio hardware on the PC has evolved from the lowly PC speaker through separate 8-bit, 11-kHz Mono audio cards to today's audio chip sets; chip sets integrated right on the motherboard offering 16-bit, 44-kHz, stereo formats and possibly some audio digital signal processor features.

Multimedia software developers have had to keep pace with this evolution by writing individual software modules for each device that they wanted to support. These applications had total control over the PC from the keyboard to the monitor. This "closeness" to the hardware allowed software developers to be in total control of the overall performance of their multimedia applications. But this device dependence imposed an expensive development and maintenance burden on multimedia software developers. It also slowed the adoption of advances in graphics and audio hardware.

## 1.2 Graphics Device Independence

With the introduction of windowed operating systems like Microsoft Windows and IBM OS/2, software developers were given a uniform programming interface that abstracted their applications from graphics hardware. Their applications could paint the screen, within a dedicated window boundary, without directly accessing the graphics hardware. Instead, the operating system accessed the hardware through device drivers. The hardware-independent interface under Microsoft Windows is known as the Graphics Device Interface (GDI); see Figure 1-1.

GDI relieved software developers of the burden of catering to each of the various graphics adapters. It also enabled hardware graphics vendors to provide hardware acceleration (such as Block Transfers, or Bltters) and to seamlessly provide the acceleration to applications through device drivers.

Although the GDI library provided a host of 2D drawing and windowing commands, it did not provide support for multimedia applications.

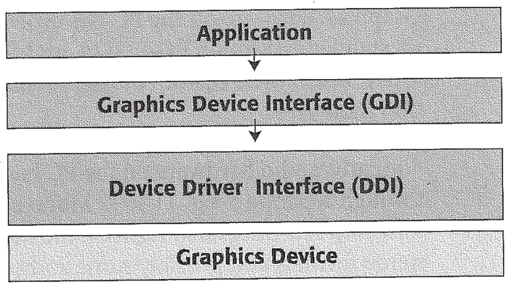

Application
Graphics Device Interface (GDI)
Device Driver Interface (DDI)
Graphics Device

FIGURE 1-1 Graphics device independence via Microsoft Windows GDI.

## 1.3 Motion Video under Windows

In their first attempt at multimedia architectures, Microsoft defined the Media Control Interface (MCI) as the first multimedia interface for Windows. MCI provided a VCR-like command interface (Play, Stop, Pause, Seek, and so forth) to enable the playback of motion video, digitized audio, VCRs and audio CD players. MCI also defined an installable device interface to allow multimedia devices to be integrated into the Windows environment.

MCI, however, did not provide any means for capturing and editing motion video. So Microsoft introduced the Video for Windows (VFW) architecture, which included tools for video capture and editing and provided an architecture for capture and compression hardware, for installable codecs (compression-decompression), and for full-motion video playback (see Figure 1-2).

VFW was a significant step forward and was a launching pad for Windows multimedia applications. It spurred the development of codecs such as Intel's Indeo Video and Radius's Cinepak. The weaknesses of the initial VFW release were inadequate synchronization between audio and video tracks and poor overall graphics performance.

Around the same time, Apple ported part of its QuickTime development environment from the Macintosh to Windows, creating QuickTime for Windows (QTW). QTW supported only audio-video playback; capture, compression, and editing were supported only on the Macintosh. Yet QTW won some favor because it had better overall performance and better synchronization mechanisms than did VFW.

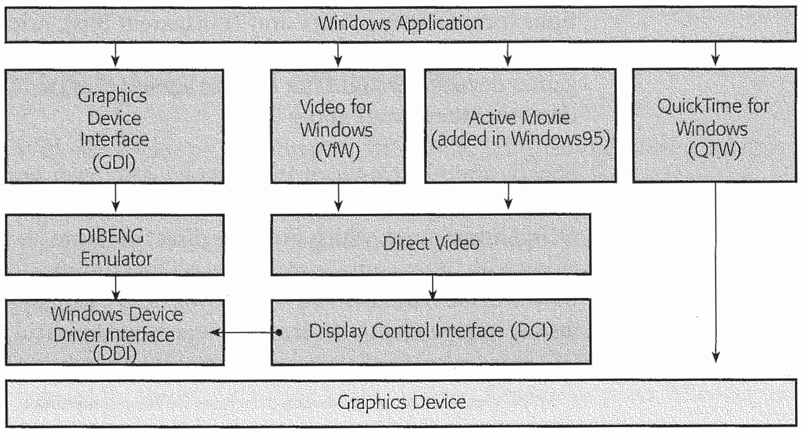

Windows Application
Graphics Device Interface (GDI)
Video for Windows (VfW)
Active Movie (added in Windows95)
QuickTime for Windows (QTW)
DIBENG Emulator
Direct Video
Windows Device Driver Interface (DDI)
Display Control Interface (DCI)
Graphics Device

**FIGURE 1-2** Video architecture under Windows 3.1.

The overhead of GDI's device-independent layer was proving too costly for graphics-intensive multimedia applications. Apple's QTW improved video performance by developing custom device drivers for various graphics devices, essentially ignoring GDI. Simultaneously, Microsoft and Intel jointly published a standard interface for graphics intensive applications—the Display Control Interface (DCI). With DCI applications could write directly to the video screen. DCI also gave users access to some video acceleration features that had not been adequately supported by GDI, namely arbitrary stretching and video-friendly YUV color formats. With DCI, full-screen, full-motion video became a reality.

At the end of 1996, Microsoft introduced the first release of ActiveMovie, targeted as a replacement for VFW. ActiveMovie addressed VFW's synchronization issues and added support for the Motion Picture Encoding Group (MPEG) class of algorithms.[1]

By the time this book is published, Microsoft will have introduced the next generation of ActiveMovie called DirectShow, which adds support for capture and compression and is integrated into the DirectX Software Development Kit (SDK). Around the same time, Apple will have released QTW Version 3.0, adding capture and compression. RealNetworks will also join the fray of multimedia architecture providers by introducing their Real Media Architecture (RMA), a multimedia streaming architecture for remote playback environments (primarily the Internet).

## 1.4 Multimedia Gaming under Windows 95

Although DCI accelerated motion video provided direct access to video memory, it did not offer direct access to graphics hardware for 2D operations (primarily Page Flips and Transparent Blts). Additionally, Windows lacked a DCI equivalent for audio devices. As a result of these shortcomings games developers could not achieve the levels of performance under Windows that they could under DOS.

Shortly after the release of Windows 95, Microsoft introduced the DirectX Software Development Kit (SDK), containing DirectDraw, the successor to DCI; DirectSound, which provides direct access to audio hardware devices; along with other components such as DirectInput and DirectPlay. With DirectX and Windows 95's AutoPlay features, games developers now had a device-independent platform that was more powerful than DOS. With

1. MPEG uses bidirectional prediction techniques for video compression.

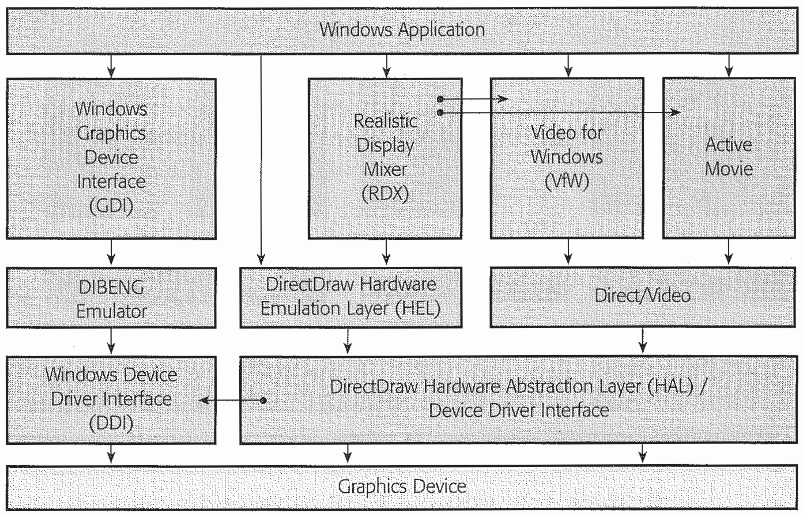

Windows Application
Windows Graphics Device Interface (GDI)
Realistic Display Mixer (RDX)
Video for Windows (VfW)
Active Movie
DIBENG Emulator
DirectDraw Hardware Emulation Layer (HEL)
DirectVideo
Windows Device Driver Interface (DDI)
DirectDraw Hardware Abstraction Layer (HAL) / Device Driver Interface
Graphics Device

**FIGURE 1-3** 2D graphics and video architectures under Windows 95.

these improvements, the Windows 95 PC established itself as a powerful gaming platform. (See Figure 1-3.)

DirectDraw is a low device-level interface. With it developers can go back to working with some amount of device dependence. Intel introduced Realistic Display Mixer (RDX), a higher-level interface that abstracts a set of multimedia objects. RDX uses hardware acceleration for these objects whenever it is available. In the absence of acceleration, RDX executes assembly code, which is hand tuned for various flavors of Intel processors. As a result, the high-level interface offers high performance video and 2D while still providing device independence.

## 1.5 3D Video Architectures on the PC

With the release of Windows NT, Microsoft launched their own port of OpenGL to the Windows NT platform. Windows NT was targeted as a high-end workstation—similar to Silicon Graphics' and Sun Microsystems'. But OpenGL was extremely slow under Windows NT since it required a huge number of calculations to determine object geometry, lighting, and shading.

Later, other companies introduced general-purpose 3D solutions specifically tailored for the PC, including Reality Labs by Rendermorphics, BRender by

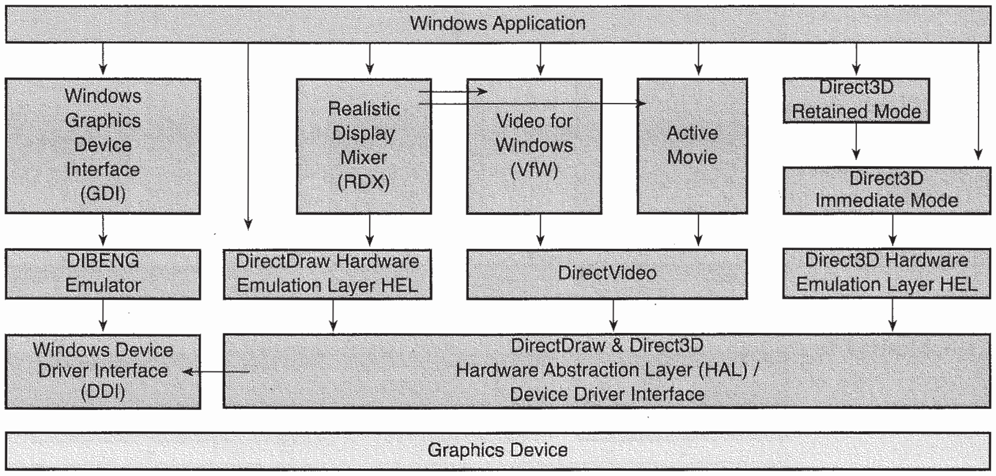

Windows Application
Windows Graphics Device Interface (GDI)
Realistic Display Mixer (RDX)
Video for Windows (VfW)
Active Movie
Direct3D Retained Mode
Direct3D Immediate Mode
DIBENG Emulator
DirectDraw Hardware Emulation Layer HEL
DirectVideo
Direct3D Hardware Emulation Layer HEL
Windows Device Driver Interface (DDI)
DirectDraw & Direct3D Hardware Abstraction Layer (HAL) / Device Driver Interface
Graphics Device

**FIGURE 1-4** Video and 2D/3D graphics architecture under Windows 95.

Argonaut, RenderWare by Criterion, and 3DR by Intel. Even though these architectures were not fast enough for realistic 3D, they were fast enough to enable the development of simple 3D applications. (See Figure 1-4.)

To reduce confusion in the marketplace, Microsoft bought Reality Labs from RenderMorphics and introduced Direct3D as the single uniform solution for 3D on the PC. Some 3D games were released using Direct3D, but the general feedback has been that the performance needs to be improved and that the interface needs to be simpler, and more reliable.

By the time this book is published, Microsoft will have introduced, as part of DirectX foundation 5, the next revision of 3D for the PC, called the DrawPrimitive interface. This interface is intended to address the performance deficits and the interface complexity that was identified by previous users.

## 1.6 Audio Architectures on the PC

I remember writing my first program to meddle with the speaker on the PC. It was a police siren program that sent a periodic signal to the speaker and varied the frequency up and down. Boy, that was a long time ago.

Microsoft introduced the WAVE and MIDI interfaces to Windows around the same time that MCI was introduced. Both of these interfaces are still widely used today. To allow for mixing of multiple audio streams, Microsoft introduced DirectSound as part of the DirectX SDK. Since the DirectSound interface is a low-level interface, Intel introduced its own high-level audio interface, Realistic Sound Experience (RSX). RSX allows developers to easily mix multiple audio streams and control the output of these streams. RSX also models the real-world environment and provides support for a realistic 3D sound model.

## WHAT HAVE YOU LEARNED?

After reading this chapter you are more familiar, perhaps, with

- GDI, DirectDraw, and RDX;
- MCI, VFW, QTW, DirectShow, and RealMedia;
- WAVE, DirectSound, and RSX; and
- Direct3D

CHAPTER 2

# Processor Architecture Overview

**WHY READ THIS CHAPTER?**

In this book we're only concerned with Intel Architecture processors running Windows 95 and Windows NT. This chapter provides an introduction to the current multimedia hardware architectures on the PC.

We'll give you an overview of the following technologies:

- the Pentium processor and the architecture of its pipeline,
- the Pentium Pro processor and its internal architecture, and
- MMX technology and the Pentium II processor.

In the early days of multimedia, dedicated hardware was necessary to play back video, audio, and 3D. But with the giant leaps in processor and memory technologies, software-only decoders are now able to decode and render multimedia content on the PC easily. As a result, multimedia authoring and playback have become commonplace on today's PCs.

To attain such performance, developers of these software decoders had to use some of the software architectures discussed in the previous chapter, such as DirectDraw and DirectSound. In addition, they had to optimize their application for the processors that they're targeting the decoder for. In general, multimedia developers dedicate some of their development time for processor-specific optimization so that they can get the best performance out of their application.

With multimedia applications, it's not enough to just optimize for the processor; you have to optimize your application for the system that you're running on—cache, bus, and memory. When you optimize for the processor, you typically assume that the data is in the L1 cache or in a register. But this is not the case with multimedia applications, since you typically deal with a huge amount of data, and usually the data is in either the L2 cache or main memory.

In this chapter, we'll give you an overview of the current breadth of Intel processors and compare their features. We'll also touch on issues related to the system as a whole. You can find a detailed analysis of both topics in Part VI of the book.

## 2.1 Processor Architecture

In the following overview, we'll only be concerned with the Pentium family of processors including the original Pentium, the Pentium Pro, the Pentium II, and the Pentium processor with MMX technology. (See Figure 2-1.)

The Pentium processor is built with two integer execution units (U and V pipes), which allow the processor to execute up to two integer instructions

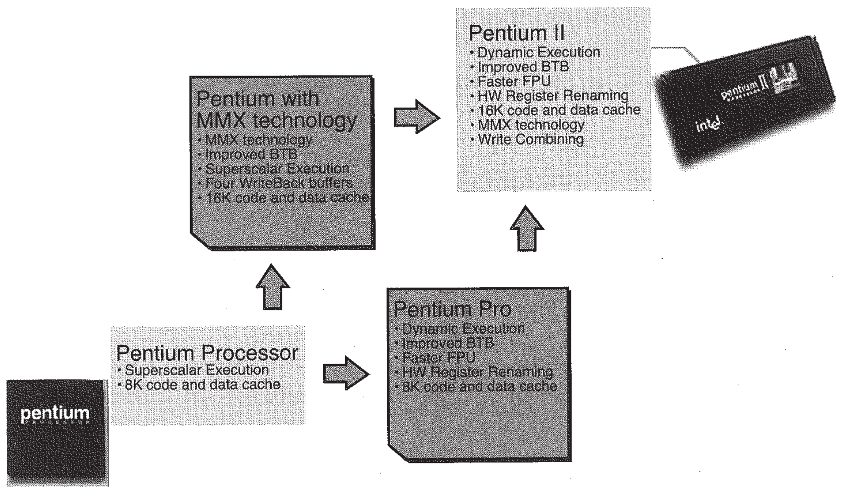


**FIGURE 2-1** The Pentium processor family.

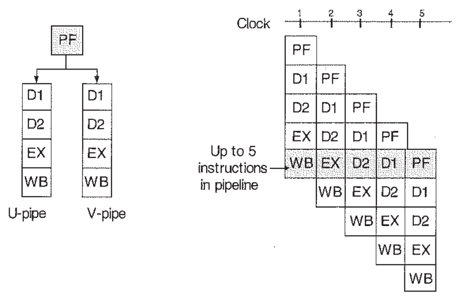


**FIGURE 2-2** Pentium processor pipeline.

every clock cycle. Each execution pipeline has five distinct execution stages: Prefetch, Decode 1, Decode 2, Execute, and Writeback (see Figure 2-2). At any moment, the Pentium processor could be processing up to five instructions in each of the two pipelines. In addition, the Pentium processor includes two separate L1 instruction and data caches of 8K each, which allow the processor to access instructions and data in the same clock cycle.

Typically, applications cannot achieve an optimal instruction rate because of external data/address dependencies or unpaired instructions. Two instructions can execute simultaneously only if they adhere to the Pentium instruction pairing rules; otherwise only one instruction is executed in the U pipe. You can learn how to optimize your application and about instruction pairing rules in Part VI.

The Pentium with MMX technology processor is the first processor that provides Intel's MMX technology. MMX technology is the largest addition to the Intel Architecture since Protected mode was introduced in the Intel 386 processor. Intel added fifty-seven new MMX instructions and eight MMX registers to its Pentium processor. It also doubled the size of the L1 instruction and data caches to 16K each.

In the Pentium Pro processor, Intel moved to a twelve-stage pipeline (compared to a five-stage pipeline in the Pentium processor) with out-of-order execution. The deeper pipeline allows different processor units to operate on multiple instructions at the same time. Such deep pipelining, however, is very expensive in terms of overhead in the case of branch misprediction. To remedy that, the Pentium Pro processor includes a sophisticated branch prediction mechanism to better predict the outcome of branches before they occur.

The Pentium Pro processor has an out-of-order execution unit consisting of five parallel execution ports: two Arithmetic Logic Unit (ALU) ports, an address generation port, a Load port, and a Store port. The out-of-order nature of the execution unit allows the processor to execute future instructions while older instructions are waiting for their data or address to be resolved. You'll learn more about the benefits of out-of-order execution in Part VI.

The core of the Pentium II processor is based on the Pentium Pro processor core, with the addition of MMX technology. The Pentium II processor doubled the size of the L1 code and data caches to 16K each.

## 2.2 System Overview

Typically, it is not enough to just optimize your application for a certain processor. You should also be concerned with the other components in the system that can affect performance—system memory, cache, and video memory.

When you optimize for the processor, you assume that you're dealing with data that exists in the L1 cache. However, with multimedia applications, you typically deal with a huge amount of data that does not fit in the L1 cache—and sometimes not even in the L2 cache. Consequently, you should pay special attention to the access pattern of your data and optimize for a high L1 cache hit rate (see Figure 2-3).

To do that, you can use special techniques in prefetching the data to the L1 and L2 cache. You can also break down your tasks into smaller tasks that can use a smaller amount of data—and probably fit in the L1 cache. See Part VI for more details.

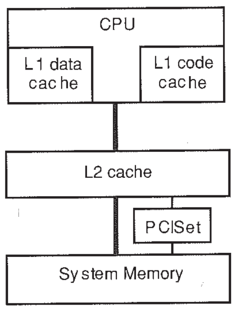


**FIGURE 2-3** Memory architecture of the Pentium processor.

Finally, you should pay special attention when you write the final image to the video screen. Since you're dealing with a huge amount of data, this operation can be very time consuming. You can use DirectDraw to access the video screen directly and write your image to it, bypassing GDI's overhead. You can also off-load some operations to the graphics adapter, such as zooming and color space conversion, and in turn you will be able to do more on the CPU. With the Pentium II processor, you can use the Write Combining memory type to achieve a higher transfer rate when writing to video memory. You can learn more about these topics in Parts II and VI.

## WHAT HAVE YOU LEARNED?

After getting through this chapter, you should know something about

- the Pentium processor and its architecture
- the Pentium Pro processor and its architecture
- MMX technology and the Pentium II processor

PART II

# Animated Graphics, Sprites, and Backgrounds

WE'D LIKE TO EXTEND AN ACKNOWLEDGEMENT TO JACK AND GLENNA RYAN, WENDY AND DAWSON YEE, PAM LUSARDI, DOUG BRUCKS, DEBBIE BURKE, SARAH NAHUM, GREG SCHWENDINGER, TERRI DEGROAT, MICHELLE CAUDILLO, JANET RASH, BLAKE AND NIKKI BENDER, BEN AND JUDY ECHOLS, JERRY ORLECK, LEORA GREGORY, CYNDI YOUNG, ROHIT AND NIDHI AGARWAL, ROGER AND SUSAN TAIT, GARY AND MARY BALDES, JUDI GOLDSTEIN, AND TOM CRONIN.

Part II consists of four chapters that cover rendering 2D graphics images under Windows9x.

The Microsoft Windows Graphics Device Interface (GDI) is a feature-rich library that provides all sorts of primitives to Block Transfer (Blt) graphics images and to draw common 2D objects (such as lines or rectangles). So why bother spending four chapters on 2D graphics? Well, because we are going to focus specifically on the sort of rendering used for composition and animation.

Although the images used for compositions are typically rectangular, the actual contents are irregularly shaped. Some of the data within the rectangle is defined as transparent and is not meant to be seen. In Chapter 3, the first chapter of this part, we will define the animation objects that we use throughout the part—specifically features that can bring potential benefits to performance.

Animation objects are composed using transparent Blt routines. In Chapter 3, we will also work through examples of rendering transparent images using GDI, and then we will measure the performance of rendering with GDI.

In Chapter 4, we examine Microsoft's DirectDraw architecture, which was designed for multimedia developers who want to render animation objects with higher performance than what is offered by GDI. In this chapter, we touch upon the first aspect of higher performance through DirectDraw—bypassing GDI and using custom routines to render directly to the display screen.

Chapter 4 will give you a good starting point for using DirectDraw, but it is by no means a complete guide. In Chapter 5, we study the second aspect of higher performance through DirectDraw—accessing hardware acceleration features. The chapter also examines mechanisms to reduce the sundry, but expensive cost of refreshing the screen.

DirectDraw is a low-level API that enables high performance at the cost of some device dependence. Intel's Realistic Display Mixer (RDX) sits on top of DirectDraw and provides high-performance animation with a higher level device-independent API. Chapter 6 will show you how to get going quickly with RDX.

**Some recommendations:**

- Chapter 3 contains fairly introductory material. If you are familiar with terms like *sprites* and *backgrounds* and are not interested in how to render them with GDI, you need not read this chapter.
- If you don't want to bother with the details of a low-level API like DirectDraw, then RDX in Chapter 6 is a good alternative. Chapter 6 is also a good chapter if you don't want to implement a mixing subsystem or if you want to use RDX's assembly-tuned routines as a complement to your own work.
- If you intend to work with Direct3D, you will need to know DirectDraw, and both Chapters 4 and 5 are important for you.
- If you have your own graphics objects with their own render routines, or if you enjoy high-performance assembly programming, you will *want* to know DirectDraw in enough detail that, again, you should read both Chapters 4 and 5.

CHAPTER 3

# Simple Sprites in GDI

**WHY READ THIS CHAPTER?**

Consider this chapter as a short introduction to animation terms and concepts.

Here we define *sprites* and *backgrounds*. To visibly illustrate the concepts, we walk you through working examples of sprites and backgrounds drawn using Microsoft Windows GDI. Read the code to understand our definitions. Run the demos to visualize these definitions.

Later we use the working examples to measure just how fast we can draw sprites and backgrounds using GDI. With these measurements in hand, we'll be in a better position to assess the performance of alternate options in subsequent chapters of this part.

## 3.1 Graphics Device Interface (GDI) Overview

We expect that most of you (our readers) are very familiar with Microsoft's Graphics Device Interface (GDI). Still, let's not forget GDI's features while on our quest for higher performance options.

Windows GDI handles all graphic output—to the display screen as well as to other graphics output devices such as printers, plotters, and metafiles. In handling graphics output, GDI must handle the various forms of these devices (such as EGA versus VGA and laser printers versus dot-matrix printers). GDI's device drivers shield us, application developers, from many of the complexities of device-dependent issues.

With GDI's device driver model, hardware vendors can provide different levels of hardware acceleration at different price points.

GDI also acts as a sharing agent for graphics output devices. It manages multitasked output to devices through device drivers and device contexts. These management responsibilities include memory ranges, clipping regions, color palettes, and print spoolers.

As a graphics library, GDI provides a variety of objects (brushes, pens, bitmaps, pixels, text); provides attributes for these objects (fills, thickness, font, color); provides commands for manipulating objects and attributes (Create, Load, Select); and offers some other drawing functions (PolyLine, TextOut, Rectangle, BitBlt).

GDI also controls the look and feel of images on graphics devices via the definition of a standard interface for default objects (standard colors, cursors, icons, and base fonts); the definition of sizing attributes (coordinate spaces, text metrics); and the definition of control functions (coordinate mapping, font enumeration, font mapping). In short, GDI does a lot. Bypassing GDI for higher-performance options means bypassing all these capabilities. Choose your path carefully.

## 3.2 Animation Objects

When we mention sprites, you're probably thinking of pixies, and nymphs, and elves, and wood fairies. Toss in a few gnomes, ogres, trolls, and goblins and we'd have quite a fairy tale on our hands. But, it's time to rein in these flights of fancy.

### 3.2.1 Sprites

For this book, let's define *sprite* in a multimedia context. Let's use the term *sprite* to refer to regular bitmap images that are superimposed on top of other graphics images. What's more, the superimposition of sprites is not a simple block copy. Instead, sprites contain both visible and transparent pixels, and the superimposition must only render the visible pixels.

We expect—and may optimize for—

- sprites being drawn repeatedly, so that some of the time spent preparing them can be recovered during drawing;
- sprites being fairly small images so that whatever memory they require can be traded off for performance; and
- partial sprites being rarely drawn, and routines to draw partial sprites may be separate and slower than equivalent routines to draw sprites wholly.

Figure 3-1 shows a sprite that we use in our demo applications.

**FIGURE 3-1** Sprite image.

The sprites we use in our demo applications are of varying sizes. You might also say that they are of odd sizes: that is, they are not square; they are not powers of two; they are not even DWORD or QWORD multiples. We chose these odd sizes deliberately, to provide you with the opportunity to study the performance impact of different sprite sizes.

### 3.2.2 Backgrounds

Let's also define the term *background* in a multimedia context. Let's use *background* to refer to images without transparency. Can a background be drawn on top of another? Sure it can! But we expect that images without transparency are probably going to be behind objects with transparency—hence the term *backgrounds.*

We expect that backgrounds are large images on top of which one or many sprites will be superimposed. They take up a lot of memory, and more memory cannot be used to improve performance. Also a background may be much larger than the displayed image, and moving a source rectangle around within the background is one way of creating an illusion of motion—scrolling backgrounds. Figure 3-2 depicts the background that we use in our demo applications.

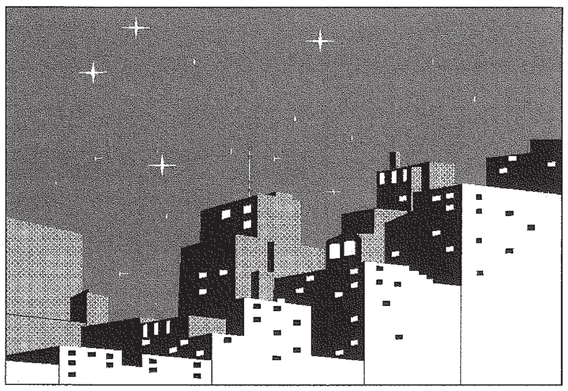


**FIGURE 3-2** Sample background.

Again, the background is of an odd size. We chose this size because we want to point out the special code that needs to be written to handle odd-sized backgrounds.

## 3.3 Transparent Blts with GDI

GDI does not contain any single function to "transparently" Blt images. Therefore our Transparent Blt algorithm uses a combination of RasterOp Bit-Blts. Our approach involves the following steps:

1. Zero out the pixels from the destination that are to be painted with "visible" sprite pixels. (To do so, we create an inverted Mask from the original source at Init-Time. And at run time, we BitBlt the mask with a SRCAND[1] RasterOp onto the destination.)
2. OR-In sprite pixels into the zeroed-out space. (RasterOps operate at the bit level, and a nonzero transparency color in the source could OR-In spurious bits into the "transparent" space. Therefore at Init-Time, we zero out the transparent pixels from the original image.)

## 3.4 Drawing a Sprite Using GDI

OK, now let's take a look at some sample code that implements sprites and scrolling backgrounds using GDI. Here is the base class definition for sprites, CSprite.

```
class CSprite {

public:
  dword     m_dwWidth;                                      ◊ width of sprite
  dword     m_dwHeight;                                     ◊ height of sprite
  byte *    m_pData;                                        ◊ internal sprite data storage
  byte      m_byTransp;                                     ◊ transparency pixel

  CSprite();                                                ◊ constructor -- Cannot err

  bool Init(uint nResID, byte byKey, cdc &pcWnd);           ◊ Init -- Can return errors
  ~CSprite();                                               ◊ destructor
  void Blt(BLTPARAMS *pDst, CPoint &point);                 ◊ blt routine
};
```

Our sample source on the JS97 CD defines and uses a BLTPARAMS structure. This structure is something like a union of various parameters needed by different Blt routines. In explaining the code we will only list the parameters the routine needs.

1. SRCAND is a parameter for the BitBlt function. It performs a logical AND of the source bitmap with the destination bitmap.

The Blt algorithm we use is based on an approach recommended by Microsoft on their Developer Network (MSDN) CDs. The recommended algorithm uses a three-Blt approach, but we improved the algorithm to pre-process the sprite at init-time to allow us to use a two-Blt approach.

Following are the Init and Blt routines. Note that since Windows BitmapInfo-Headers do not allow for transparency colors, we are specifying transparency as a parameter to the sprite *Init()* load function.

```
CSpriteGDI::Init(UINT nResID, BYTE byColorKey, CDC *pcdcWnd)
{
  // load bitmap from resource into a tmp bmp ready for preparation
  CDC cdcTmp;
  cdcTmp.CreateCompatibleDC(pcdcWnd);
  CBitmap cbmTmp;
  cbmTmp.LoadBitmap(nResID);
  cdcTmp.SelectObject(cbmTmp);
  BITMAP bm;
  cbmTmp.GetBitmap(&bm);
  DWORD dwWt = m_dwWidth  = bm.bmWidth;
  DWORD dwHt = m_dwHeight = bm.bmHeight;

  // get transparent color and set DC background (we use system palette)
  PALETTEENTRY peClr;
  GetSystemPaletteEntries(pcdcWnd->m_hDC, (UINT)byColorKey, 1, &peColor);
  cdcTmp.SetBkColor(PALETTERGB(peClr.peRed, peClr.peGreen, peClr.peBlue));

  // create a monochrome mask for run-time clearing of foreground pixels
  CDC cdcMask;
  cdcMask.CreateCompatibleDC(cDcWnd);
  m_pcbmMask = new CBitmap;
  m_pcbmMask->CreateBitmap(m_dwWidth, m_dwHeight, 1, 1, NULL));
  Cbitmap *pcbmOldMask = cdcMask.SelectObject(m_pcbmMask);
  cdcMask.BitBlt(0, 0, dwWt, dwHt, &cdcTmp, 0, 0, SRCCOPY);
```

BitBlt from color-bitmap to mono-bitmap sets pixels with background=1 and foreground=0. We previously *SetBkColor* of cdcTmp to the transparency color. The result here is an inverted mask.

```
  // process src so that transparent pixels are 0
  CDC cdcSrc;
  cdcSrc.CreateCompatibleDC(pcdcWnd);
  m_pcbmSrc = new CBitmap;
  m_pcbmSrc->CreateBitmap(m_dwWidth, m_dwHeight, 1, 8, NULL);
  cdcSrc.SelectObject(m_pcbmSrc);
  cdcSrc.BitBlt(0, 0, dwWt, dwHt, &cdcMask, 0, 0, NOTSRCCOPY);
  cdcSrc.BitBlt(0, 0, dwWt, dwHt, &cdcTmp, 0, 0, SRCAND);
```

Preprocess Source. Zero out pixels of transparent color by
- NOTCOPYing inverted mask to result bitmap
- and then ANDing in the actual source data

```
   return TRUE;
}
```

```
CSpriteGDI::Blt(CDC &cDc, CPoint &pt)
{
  static CDC cdcSrc, cdcMask;
  static CBitmap *pcbmOldSrc, *pcbmOldMask ;

  // setup 2 DCs with bmps prepared during sprite init
  cdcSrc.CreateCompatibleDC(&cDc);
  cdcMask.CreateCompatibleDC(&cDc);
  pcbmOldSrc = cdcSrc.SelectObject(m_pcbmSrc);
  pcbmOldMask = cdcMask.SelectObject(m_pcbmMask;

  // blt: clear away foreground pixels using mono mask (bk=1, fg=0)
  cDc.BitBlt(pt.x, pt.y, m_dwWidth, m_dwHeight, &cdcMask, 0, 0, SRCAND);
  // second blt: or preprocessed src into anded dest
  cDc.BitBlt(pt.x, pt.y, m_dwWidth, m_dwHeight, &cdcSrc,  0, 0, SRCPAINT);

  cdcMask.SelectObject(pcbmOldMask);
  cdcMask.DeleteDC();
  cdcSrc.SelectObject(pcbmOldSrc);
  cdcSrc.DeleteDC();

}
```

Release these DCs, since they are not local to this routine's scope and will not get automatically released until the application terminates.

## 3.5 Backgrounds

The code for backgrounds is similar to that used for sprites.

```
CBackgroundGDI::Init(UINT nResID, CDC *pcdcWnd)
{
  // load bitmap for background from resource file
  CDC cdcSrc;
  cdcSrc.CreateCompatibleDC(pcdcWnd);
  m_pcbmSrc = new CBitmap;
  m_pcbmSrc->LoadBitmap(nResID);
  Cbitmap *pcbmOldSrc = cdcSrc.SelectObject(m_pcbmSrc);
  BITMAP bm;
  m_pcbmSrc->GetBitmap(&bm);
  m_dwWidth  = bm.bmWidth;
  m_dwHeight = bm.bmHeight;
}
```

The Blt routine is straightforward, especially since, in this case, we do not have to worry about transparency. However, note that a sub-rectangle parameter can be specified to draw only a portion of a background.

```
Cbackground::Blt(CDC &cDc, CPoint &cPt, CRect &crView)
{
  // setup DC objects
  static CDC cdcSrc;
  static CBitmap *pcbmOldSrc;
  cdcSrc.CreateCompatibleDC(cDc);
  pcbmOldSrc = cdcSrc.SelectObject(m_pcbmSrc);

  // add code to error check view to within image boundary
  lWt = crView.right - crView.left;
  lHt = crView.bottom - crView.top;

  // straightforward blt
  cDc.BitBlt(cPt.x, cPt.y, lWt, lHt,
  &cdcSrc, crView.left, crView.top,  SRCCOPY);

  // release DC
  cdcSrc.SelectObject(pcbmOldSrc);
  cdcSrc.DeleteDC();

}
```

## 3.6 Demo Time

Run the demo that corresponds to this chapter[2]. You should see a sprite being drawn on the screen. Move the mouse around and the sprite will follow the mouse.

The sprite leaves sprite trails because on startup we have set the application to "not refresh" the background. Turn Background Refresh on and the sprite trails will disappear.

A difficulty of overlaying sprites on backgrounds directly onto the screen is that refreshing the background is followed by the transparent overlay of the sprite, which results in a noticeable flicker. For flicker-free results, the background needs to be refreshed and the sprite overlayed into a nonvisible buffer (memory DC), and then the resulting image in the nonvisible buffer must be transferred to the screen. "Compositing" in nonvisible buffers will be discussed shortly in Chapter 5. For now, treat this as an exercise for you the reader.

2. See the Introduction if you need instructions.

## 3.7 How Fast Does GDI Draw Sprites and Backgrounds?

Table 3-1 measures the speed of drawing sprites and backgrounds with GDI. These measurements were taken on our base platform described in the Introduction and will definitely vary with different configurations. We have included the application for measuring the speed of drawing sprites and its source code on our Internet site that we mentioned in the Introduction; it is called *Timing App*. Run the application on your platform and see what results you get. The source code for the timing application is also included on the CD. We have separated out the timing source code from the source for the chapter demos to simplify reading the base code.

**TABLE 3-1** How Fast Does GDI Draw Sprites and Backgrounds?

| Object | Time (in milliseconds) |
|---|---|
| 16 sprites (width: 84; height: 63) | 14.2 |
| background (width: 734; height: 475) | 8.8 |

### WHAT HAVE YOU LEARNED?

By this time, you know what we mean by sprites and backgrounds, you've seen them work, and you also know how long it takes to mix a sprite on a background. Since this was only an introductory chapter, if you've read this far, you've got to be itching to move on to the next chapters, which introduce you to the meat of this section. Well, what are you waiting for?

nds?
ith
ed in
ns. We
prites
troduc-
nd see
lso
om the

s?

een them
e this was
ove on to
you wait-

# CHAPTER 4

# Sprites with DirectDraw Primary Surfaces

**WHY READ THIS CHAPTER?**

In the previous chapter, you were introduced to drawing a transparent sprite on a background using GDI. But you may not have been satisfied with the performance of sprites under GDI. This chapter will introduce you to faster sprites via the Microsoft DirectDraw interface. Read on and decide if DirectDraw works better for you.

In the previous chapter you were also introduced to graphics rendering objects (and primitives) provided by GDI. But you may have your own graphics rendering objects that are not convenient to render through GDI. In this chapter we will show you how to render your own sprites using DirectDraw. Read on and decide if our sprite example forms an appropriate foundation for rendering your objects.

By reading this chapter, you will

- get an overview of DirectDraw and what it offers,
- learn how to access the display screen and write directly to it,
- use routines to render faster sprites, and
- be exposed to some limitations of writing directly to the display screen.

## 4.1 Introduction to Microsoft's DirectDraw

The Graphics Device Interface (GDI) library within Microsoft Windows provides software developers with image display functions. The library abstracts graphics devices and provides a device-independent interface that developers can write to. Device independence allows developers to use a standard set of functions without having to worry about device specifics or even device capabilities.

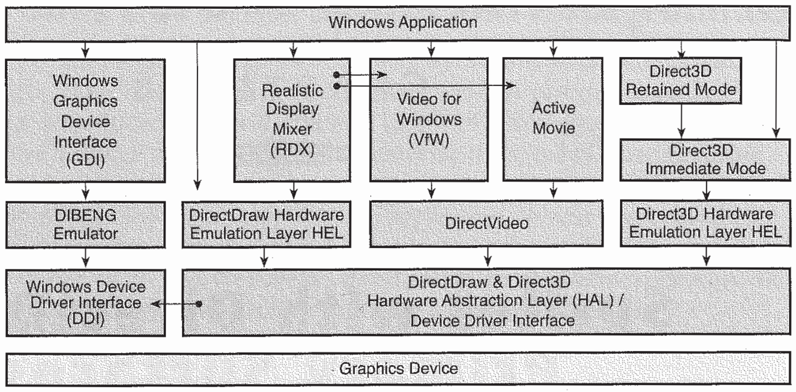

Windows Application
Windows Graphics Device Interface (GDI)
Realistic Display Mixer (RDX)
Video for Windows (VfW)
Active Movie
Direct3D Retained Mode
Direct3D Immediate Mode
DIBENG Emulator
DirectDraw Hardware Emulation Layer HEL
DirectVideo
Direct3D Hardware Emulation Layer HEL
Windows Device Driver Interface (DDI)
DirectDraw & Direct3D Hardware Abstraction Layer (HAL) / Device Driver Interface
Graphics Device

**FIGURE 4-1** Display architecture under Windows 95.

Unfortunately, the overhead of GDI's device independence was too expensive for graphics intensive applications. In 1994 Intel and Microsoft jointly released the Display Control Interface (DCI) as an extension of GDI. DCI allowed direct access to graphics device memory and to device acceleration features under Windows 3.1 and Windows 95.

In 1995 Microsoft released DirectDraw for Windows 95 as a successor to DCI. Similar to DCI, DirectDraw provides direct access to graphics device memory and to device acceleration features. DirectDraw enhances device acceleration by providing access to hardware Blters and hardware palettes.

Figure 4-1 diagrams the current display architecture under Windows 95. Although Figure 4-1 shows the entire display architecture under Windows 95, in this chapter we are primarily concerned with the thick arrow that points directly from the application to the DirectDraw Hardware Emulation Layer (HEL).

Think of DirectDraw as an extension to GDI that allows you to use custom drawing routines or allows you to access custom device-specific acceleration. DirectDraw is part of Microsoft's DirectX Software Development Kit (SDK) and is the lowest level API available for display devices.

## 4.2 Features of DirectDraw

Figure 4-2 shows a graphics card laid out as a block diagram showing typical components.

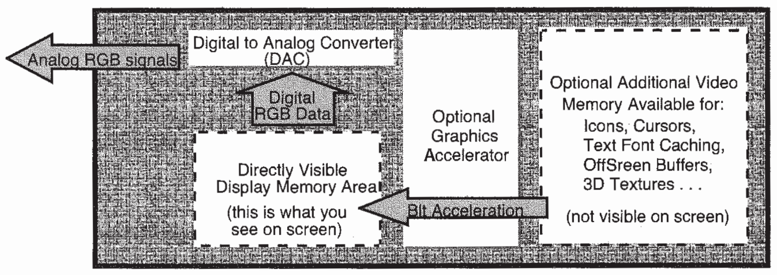

Analog RGB signals
Digital to Analog Converter (DAC)
Digital RGB Data
Directly Visible Display Memory Area
(this is what you see on screen)
Optional Graphics Accelerator
Blt Acceleration
Optional Additional Video Memory Available for:
Icons, Cursors,
Text Font Caching,
OffSreen Buffers,
3D Textures . . .
(not visible on screen)

**FIGURE 4-2** Block diagram of components on a typical graphics card.

All graphics cards have some video memory, so whatever you see on the screen is stored in memory on the graphics card in a place specifically reserved for that purpose. RGB data from this primary screen area is converted via a digital to analog converter (DAC) to analog signals that are sent out to the monitor. DACs support palette lookups during conversion in a palettized graphics mode.

Today's graphics cards typically have an optional graphics accelerator to support standard GDI acceleration, and they are configured with enough memory to support high-resolution (24 or 32 bits per pixel, or bpp) graphics modes. If you're operating in low-resolution graphics modes, this additional memory may be used for other purposes. DirectDraw gives you *direct access* to the graphics card to both its video memory and its acceleration hardware.

DirectDraw gives you access to the video memory through a *surface* object. Device memory exists in many forms and, therefore, there are many types of surfaces to allow you to access the various forms of device memory.

A `PRIMARYSURFACE` gives you direct access to the main display memory area. Anything you write to this memory area is immediately visible on the display screen. In fact you get access to the entire display screen and can write anywhere on the screen. Note that you access the screen in the user's display configuration, which can vary both in screen size (640 × 480 or 1024 × 768) and in color format (8, 16, 24, 32 bpp). Under special circumstances, DirectDraw allows you to reconfigure the display for the duration of your application.

DirectDraw's `OFFSCREENSURFACE` allows you to allocate and access any additional "behind-the-screen" device memory. Why is this useful? Graphics cards have acceleration features like Transparent Blts, non-RGB color formats, fast screen refreshes by Page Flipping, and even 3D graphics primitives. Offscreen surfaces are the mechanism by which you access hardware acceleration features that are not accessible through GDI. There is one caveat, however: the source and destination images for these acceleration features must live in device memory.

Beyond access to device memory and hardware acceleration, DirectDraw also supports additional features such as direct access to the primary palette, support for multiple palettes with DirectDraw *Palettes,* and support for window management with DirectDraw *Clippers.* Given that surface types and device features vary, DirectDraw supports a *capabilities* model, which can be used to query the DirectDraw driver for its capabilities before you use any specific feature.

In this chapter, we will introduce you to initializing and querying the DirectDraw driver and using DirectDraw Primary surfaces. OffScreen surfaces and device acceleration will be discussed in the next chapter.

## 4.3 Before You Get Overly Excited

Writing directly to *device memory* or accessing *device-specific* acceleration defeats the *device-independence* benefits of GDI. Once you access device-specific features using DirectDraw, you must respond gracefully to variations in graphics devices.

For example, memory layouts differ based on the display configuration selected by the user. Variations in configurations include pixel format (such as RGB24, RGB16, or palletized RGB8) and screen size (such as 640 × 480, 800 × 600, 1024 × 768). Memory layouts may also differ based on manufacturers' design choices. There is, for example, more than one format for the size of the color components with RGB16—5:6:5 and 5:5:5 being two popular formats. Similarly RGB24 can be either in a compact 3 bits per pixel

1. The emerging Advanced Graphics Port (AGP) specification will allow graphics cards to provide acceleration using system memory–based source images. DirectDraw Offscreen surfaces will continue to be the mechanism to access AGP-based graphics hardware acceleration.

format or in a DWORD-sized format with the most significant byte ignored. If you write directly to memory, you need to understand what format is currently being used and be able to write your pixels in that format.

Similarly, there is no standard set of acceleration features that all graphics devices must provide. If you use a device-specific feature, you will also need a fallback mechanism to work with devices that do not support that particular feature. DirectVideo, for example, has many code paths to use various color-conversion and stretching features.

Choosing a device-specific development option places a development burden on you. But choosing this option can give you significant performance gain.

Note

Some graphics cards are still banked memory devices. We would need to switch to a new bank before accessing its memory. However, Microsoft now provides a mechanism (VFlatD) to disguise banked access as linear access. VFlatD traps page faults on specific memory ranges and automatically switches banks as needed. The disguise does add a noticeable performance cost.

Use the DDTEST tool that comes with the DirectX SDK to get information about your display device. Some devices, such as some S3 Trio 64 graphics cards, are incorrectly identified as banked devices. Contact the maker of your graphics card for updated DirectDraw drivers.

## 4.4 Instantiating a DirectDraw Object

Let's get dirty. First, let's initialize DirectDraw by instantiating, or creating an instance of, a DirectDraw object:

```
BOOL CSharedHardware::Init(HWND hWnd) {
   LPDIRECTDRAWpDDraw;
   HRESULTerr;

   // create a DirectDraw instance
   DirectDrawCreate(NULL, &pDDraw, NULL);

   // Setup to use as normal windowed app
   err = pDDraw->SetCooperativeLevel(hWnd, DDSCL_NORMAL);
   if (err == DD_OK) {
    pDDraw->Release();
    return FALSE;
   }

   // store into member variable
   m_ppDDraw = pDDraw;
   return TRUE;
}
```

*DirectDrawCreate* is the starting point in using DirectDraw. The DIRECTDRAW structure returned from this function provides access to the entire next level of functionality, such as *CreateSurface*, *EnumDisplayModes*, and so forth.

*SetCooperativeLevel()* sets how we plan to use DirectDraw. The settings can be

| | |
|---|---|
| DDSCL_NORMAL | App will work as a regular Windows app. |
| DDSCL_EXCLUSIVE | App wants exclusive access to display area. |
| DDSCL_FULLSCREEN | App wants responsibility for the entire display area. GDI will be ignored. |
| DDSCL_ALLOWMODEX | App can deal with non-Windows modes |
| DDSCL_ALLOWREBOOT | Allow CTRL_ALT_DEL to work while in fullscreen exclusive mode. |
| DDSCL_NOWINDOWCHANGES | Don't let user change position or minimize application |

## 4.5 Querying and Creating a Primary Surface

Now that DirectDraw has been initialized, let's get access to DirectDraw surfaces. Let's start by examining DDSURFACEDESC, a basic DirectDraw structure that describes *all* forms of surfaces in DirectDraw.

```
typedef struct _DDSURFACEDESC{
  DWORD  dwSize;
  DWORD  dwFlags;
  DWORD  dwHeight;
  DWORD  dwWidth;
  LONG   lPitch;
  union {
    DWORD  dwBackBufferCount;
    DWORD  dwMipMapCount;
    };
    union {
    DWORD           dwZBufferBitDepth;
    DWORD           dwRefreshRate;
  };
  DWORDdwAlphaBitDepth;
  DWORDdwReserved;
  LPVOID  lpSurface;
  DDCOLORKEYddckCKDestOverlay;
  DDCOLORKEYddckCKDestBlt;
  DDCOLORKEYddckCKSrcOverlay;
  DDCOLORKEYddckCKSrcBlt;
  DDPIXELFORMATddpfPixelFormat;
  DDSCAPSddsCaps;
} DDSURFACEDESC, FAR* LPDDSURFACEDESC;
```

Not all fields of a DDSURFACEDESC structure are valid all the time. The DDColorKey fields, for example, are not needed to create a simple Primary surface. Therefore, whenever fields are used, equivalent bits in the dwFlags field indicate that the field is valid.

And now here's how to create a Primary surface:

```
CPrimarySurface::CPrimarySurface(void)
{
  // zero out the memory of the surface descriptor
  memset(&m_SurfDesc, 0, sizeof(m_SurfDesc));
  // init surface descriptor size
  m_SurfDesc.dwSize = sizeof(m_SurfDesc);
}
BOOL CPrimarySurface::Init(LPDIRECTDRAW pdDraw)
{
  // set type of surface within the surface caps structure
  m_SurfDesc.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
```

The dwCaps field is used both to establish the type of surface requested and to set up attributes upon return. Refer to the DirectDraw documentation for more details. Some surface types to note are:

| | |
|---|---|
| _PRIMARYSURFACE | Access to primary display area |
| _OFFSCREENPLAIN | Access to off-screen memory |
| _FLIP | Set up for instantaneous surface swap |
| _SYSTEMMEMORY | Surface is in system memory. |
| _VIDEOMEMORY | Surface is in video memory. Currently, TRUE for Primary surfaces. |
| _VISIBLE | Writes are immediately visible. TRUE for Primary surfaces. |
| _WRITEONLY | Data cannot be read from these surfaces. |
| _MODEX | Setup for 320x200 or 320x240 resolutions. Typically used with SetCooperativeLevel of EXCLUSIVE, FULLSCREEN and MODEX |

Other types that we will experience later include _3D, _TEXTUREMAP, _FRONTBUFFER, and _BACKBUFFER.

```
// tell driver that the caps field is valid
  m_SurfDesc.dwFlags = DDSD_CAPS;

  // call DirectDraw member  function to create primary surface
  HRESULT err;
  err = pdDraw->CreateSurface(&m_SurfDesc, &m_pSurfFns, NULL);
  if (err != DD_OK) {
    handleError(err);
    return FALSE;
  }

  return TRUE;
}
```

*CreateSurface* takes LPDDSURFACEDESC and (LPDIRECTDRAWSURFACE *) as parameters. DDSURFACEDESC is used to describe the surface requested/got. DIRECTDRAWSURFACE holds pointers to the member functions of the created surface.

If *CreateSurface()* was successful, DirectDraw drivers should set relevant fields in the DDSURFACEDESC structure to describe the surface that was created. In particular, dwHeight, dwWidth, and ddpfPixelFormat should be valid. To really make sure that the structure gets filled, use *IDirectDrawSurface::Get-SurfaceDesc()*. However, even *GetSurfaceDesc()* will not return a valid pointer to surface memory (lpSurface). This field will only become valid after you lock the surface. More on *Lock()* shortly.

Note that we have written our libraries for an RGB8 pixel format. What happens if the Primary surface display configuration is not in RGB8 display mode? We could merely "Release" the newly created surface and flag an error. An alternate method is to check the display configuration, before creating a Primary surface:

```
bool CDirectDraw::IsDisplayModeOK(void)
{
  //initialize a surface descriptor
  ddsurfacedesc ddSurf;
  memset(&ddSurf, 0, sizeof(ddSurf));// zero out mem
  ddSurf.dwSize = sizeof(ddSurf);// set size field

  // get the primary display mode
  hresult err = m_pdDraw->GetDisplayMode(&ddSurf);

  // if call was successful, check returned descriptor
  if ((err == dd_ok) &&
     (ddSurf.ddpfPixelFormat.dwFlags == ddpf_paletteindexed8))
  return true
  else
     err = js_baddisplay;
  }

  // was error, flag and return
  handleError(err);
  return false;
}
```

EXTRA CREDIT: Explore changing the display format if it's not one you like. Look at *SetDisplayMode()* and *Set-CooperativeLevel()*.

*SetDisplayMode()* is provided by the IDirectDraw2 interface. Our Primary surface sample code on the JS CD shows the use of the IDirectDraw2 and IDirectSurface2 interfaces. Check it out.

Let's move forward, and *write directly* to the display screen using the Primary surface that we just created.

## 4.6 Implementing a Simple Sprite Class

We use the same class definition for sprites as in the previous chapter. Here is a simple sprite class implementation where we control the drawing.

```
BOOL CSprite::Init(CBitmap &bitmap, BYTE byKeyColor)
{
  // get access to BITMAP to get size; alloc space for data; copy data
  BITMAP bm;
  bitmap.GetBitmap(&bm);
  m_pData = new BYTE[bm.bmWidthInBytes * bm.bmHeight];
  bitmap.GetBitmapBits(bm.bmWidthInBytes * bm.bmHeight, m_pData);
  // init member variables
  m_dwWidth = bm.bmWidth;
  m_dwHeight = bm.bmHeight;
  m_byTransp = byKeyColor;
  return TRUE;
}
```

The sprite Blt function is extremely simple. It takes in a destination pointer and pitch and draws the sprite at the specified point. The function is written in C and relies entirely on the compiler for optimization.

```
void CSprite::Blt(LPVOID lpDst, long lPitch, CPoint &point)
{
  // compute address dst and src pixels.  note pitch can be negative
  PBYTE pDst = (PBYTE)((long)lpDst + point.x + point.y * lPitch);
  PBYTE pSrc = m_pData;

  // blt the sprite on a row by row basis
  for (DWORD row = 0; row < m_dwHeight; row++)  {
    for (DWORD col = 0; col < m_dwWidth;  col++, pSrc++, pDst++)  {
      // test pixel for non-transp and write if so
      if (*pSrc != m_byTransp)
        *pDst = *pSrc;
    }
    // bump dst ptr forward to start of next row
    pDst += lPitch - m_dwWidth;
  }
}
```

**A FASTER WAY OF DOING SPRITES**

Our last sprite-based application needed a lot more performance than what we got with a per-pixel test and branch approach. We researched and developed alternate approaches in assembly. One of these approaches will be analyzed in Part VI where the optimization decisions can be discussed in greater detail. This routine is listed on the CD. We invite you to peruse the routine and adapt/adopt it if it proves suitable. See Chapter 19 for more details.

## 4.7 Drawing a Sprite on the DirectDraw Primary Surface

So far we've initialized DirectDraw, established that our preferred format was supported, and created a Primary surface. Now let's draw a sprite on the Primary surface—*we will be writing directly to the screen.*

```
CPrimarySurface::BltSprite(CSprite &spr, CPoint &point)
{
  // first lock surface, using "wait until lock"
  m_pSurfFns->Lock(NULL, &m_SurfDesc, DDLOCK_WAIT, NULL)
  // invoke sprite blt routin
  spr->Blt(m_SurfDesc.lpSurface, m_SurfDesc.lPitch, poin
  // release the lock
  m_pSurfFns->Unlock(NULL);
}
```

Blt writes directly to the screen.

- Graphics memory is shared by many applications. The surface must be locked to manage access to this common memory. Locking the surface returns a usable pointer in the Surface Descriptor.
- Memory in surfaces is arranged in blocks. lPitch need not equal SurfaceWidth.
- Unlock before you leave. Surface locks can lock out all GDI access.

**DEBUGGING**

Debugging between Lock and Unlock is tricky. Lock shuts out GDI access to the screen, so debuggers are unable to update their windows. *Single-stepping between lock and unlock in most debuggers will hang the system in Windows 95.* Debugging with NuMega's SoftIce for Windows 95 in dual monitor configuration or debugging under Windows NT 4.0 works with most debuggers.

## 4.8 Demo Time

At this point, you should be seeing sprites drawn directly to the display screen via DirectDraw Primary surfaces. Run the demo that corresponds to this chapter. You should see a sprite appear on the screen. Move the mouse around and the sprite will follow the mouse.

How do you know we're writing directly to the display screen? Move the mouse to the white border areas along the right or bottom edges of the clipping window. You will notice that the sprite writes data outside the clipping window. GDI would not let this happen. We have written directly to the screen without GDI.

Why can't we write anywhere on the screen? Our application tracks Mouse Move messages and draws the sprite based on mouse position. Our application stops receiving Mouse Move messages once the mouse cursor has left the main Window area. You can alter this application to write anywhere if you wish.

To be a well-behaved Windows application, your program should respect window overlaps, boundaries, and movements. You will also notice that moving the sprite leaves sprite trails. This is because the application is not set to refresh the background.

## 4.9 Redrawing Backgrounds on a DirectDraw Primary Surface

Here is a quick background Blt routine. Again, this version is extremely simple. It takes in a destination pointer and pitch and draws the background at the specified point. The only subtlety about Bltting backgrounds is that the rows in graphics memory are not necessarily contiguous, and therefore the Blt routine must handle a pitch.

```
void CBackground::Blt(LPBYTE lpDst, long lPitch, CPoint &point)
{
  PBYTEpDst, pSrc;
  DWORDrow, dwLeft, dwWidth, dwTop, dwRows;

  // compute address dst and src pixels.  note pitch can be negative
  pDst = (PBYTE)((long)lpDst + point.x + point.y * lPitch);

  // code removed that clamps ViewRect within background
  // dimensions; generates dwLeft, dwWidth, dwTop, dwRows
  // to define sub-region being bltted

  pSrc = m_pData + dwLeft + dwTop * m_dwWidth;

  // blt the sprite on a row by row basis
  for (row = 0; row < dwRows; row++)  {
    memcpy(pDst, pSrc, dwWidth);// use simple memory copy
    pDst += lPitch;// bump dst ptr forward
    pSrc += m_dwWidth; // bump src ptr forward
  }
}
```

## 4.10 How Fast Can We Draw Sprites and Backgrounds?

Table 4-1 shows the speed at which sprites and backgrounds are drawn to Primary surfaces, and it presents a comparison with the GDI measurements from the previous chapter.

**TABLE 4-1** How Fast Can We Draw Sprites and Backgrounds?

| Method | Object | Time (in milliseconds) |
|---|---|---|
| CSurfacePrimary, CSpriteCCode, CBackground | 16 sprites (width: 84; height: 63) | 11.9 |
| | background (width: 734; height: 475) | 7.7 |
| CSurfacePrimary, CSpriteP5 CBackground | 16 sprites (width: 84; height: 63) | (0.7–1.8) |
| | background (width: 734; height: 475) | 7.7 |
| CSurfaceGDI CSpriteGDI CBackground GDI | 16 sprites (width: 84; height: 63) | 14.2 |
| | background (width: 734; height: 475) | 8.8 |

Some observations on the measurements:

- The sprite routine written in Pentium-optimized assembly language is almost *10 times faster* than the C code version. There are two measurements noted for this routine as it is sensitive to alignment of destination writes. The faster time reflects writing sprites to DWORD-aligned start addresses.
- We did write an assembly routine for background Blts (CBackgroundP5) that maximized DWORD-aligned writes per scan line. But we found that any performance gains detected were negligible, indicating that *memcpy* may already be similarly optimized.

## 4.11 Compositing Objects on a DirectDraw Primary Surface

Aaah, life would be simple if there were just one background and one sprite to worry about. We could be sipping iced teas in some tropical country; or maybe oh-nee-on soup in a Lu-wee-zee-ahna bayou. But . . .

2. Again, these measurements were taken on the base platform described in the Introduction and will definitely vary with different configurations. We have included the application and its source code on the Internet site. Run the application on your platform and see what results you get. We have separated out the timing code from the basic demo applications to simplify reading the base code.

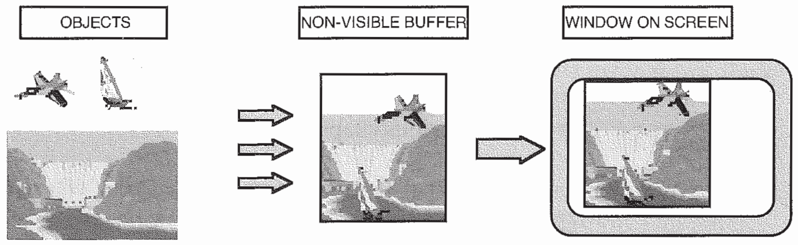


**FIGURE 4-3** Compositing using a nonvisible buffer.

A difficulty of compositing sprites directly onto the Primary surface is that the compositing process is visible, so there is a noticeable flicker on the screen. You can obtain better results when you composite images on a non-visible buffer and then make this buffer visible.

The Timing Application has a menu selection for Compositing. Take a look at both the visible-buffer compositing and nonvisible-buffer compositing options. The nonvisible-buffer compositing is implemented by rendering all the graphics objects in back-to-front order in a system buffer and then Bltting this nonvisible buffer to the Primary surface as shown in Figure 4-3.

With this method we've solved a quality problem, but at the cost of a Blt. Bltting an 800 × 600 image from system memory to video memory costs about 10 milliseconds on the platform we're using (see Table 4-2). In the next chapter we will look at mechanisms to reduce the Blt cost.

CSpriteP5 times are faster when Bltting to system memory. See details in Part VI.

**TABLE 4-2** Composited Drawing to a Primary Surface

| CSpriteP5 84 x 63 (16) | CBackground | Refresh Screen |
|---|---|---|
| 0.6–0.9 | 7.9 | 10.5 |

*Note:* Times in milliseconds.

## WHAT HAVE YOU LEARNED?

By this time you've had an overview of DirectDraw and a taste of device dependence. You know that DirectDraw provides you with a lot more freedom than GDI does, but that there is a development burden associated with this freedom.

If you worked through the code samples, you have handled code and had direct access to the display surface using DirectDraw Primary surfaces. And if you did your extra credit work and perused the CD, you have seen Primary surface sprite demos and fast Sprite Blt code written in Pentium optimized assembly language.

And if you are still reading, you are probably ready and eager to move on to the next chapter and learn about hardware acceleration, and to later chapters where you'll read about processor optimization. Are you ready?

CHAPTER 5

# Hardware Acceleration via DirectDraw

**WHY READ THIS CHAPTER?**

In the previous chapter you were introduced to rendering faster sprites directly to the display screen via Microsoft's DirectDraw interface. You were also introduced to the use of a second buffer to remove flicker with composited images. But making data visible by Blting from the second buffer to the primary screen carries with it an expensive performance penalty. This section demonstrates how hardware acceleration features can reduce the cost of double-buffering.

In the previous chapter you were also introduced to rendering faster sprites directly to the display screen using custom rendering routines. In this chapter you will explore rendering sprites using hardware acceleration features.

In this chapter you will learn how to query for, set up, and use

- hardware Bltters to reduce the cost of double-buffering,
- page flipping hardware to further reduce the double-buffering cost, and
- transparency Blt hardware to reduce the cost of Bltting sprites.

## 5.1 Creating an Offscreen Surface

OK, roll up your sleeves. First, initialize the DirectDraw driver as in Chapter 4.

Once DirectDraw has been initialized, let's get access to an Offscreen surface. Let's create a CSurface object as usual (code follows).

```
CSurfaceOffscreen::CSurfaceOffscreen(void)
{
  // zero out the memory of the surface descriptor
  memset(&m_SurfDesc, 0, sizeof(m_SurfDesc));
  // init surface descriptor size
  m_SurfDesc.dwSize = sizeof(m_SurfDesc);
}
```

Now let's look at making some changes. In creating a Primary surface, we were getting access to the primary display surface. We took what we got—the dwWidth, the dwHeight, and the ddpfPixelFormat were all specified by the DirectDraw driver to match the user's display configuration. With Offscreen surfaces, though, we've got to specify what we want, and the driver will tell us whether or not our request can be satisfied.

DirectDraw drivers can be asked to enumerate the realm of their possibilities. If you can accept a variety of formats, you may want to use *IDirectDraw::EnumSurfaces()* to enumerate the available surfaces, and then you can choose your preference based on your personal criteria (better performance, better picture quality, or some other trade-off).

Our code only accepts RGB8, which is a very basic format and is supported by nearly all DirectDraw drivers. So we will take the easy way out and try to create an RGB8 surface and react to the errors if there are any.

Let's initialize the DDSURFACEDESC structure to our specifications and try to create an Offscreen surface.

```
BOOL CSurfaceOffscreen::Init(LPDIRECTDRAW pdDraw, CWnd *pcWnd)
{
  RECT  rWin;
  pcWnd->GetClientRect(&rWin);
  m_dwWidth = (DWORD)(rWin.right - rWin.left);
  m_dwHeight = (DWORD)(rWin.bottom - rWin.top);

  // set desired fields
  m_SurfDesc.dwHeight = m_dwHeight;
  m_SurfDesc.dwWidth = m_dwWidth;
  m_SurfDesc.ddpfPixelFormat.dwSize = sizeof(DDPIXELFORMAT);
  m_SurfDesc.ddpfPixelFormat.dwRGBBitCount = 8;
  m_SurfDesc.ddpfPixelFormat.dwFlags = DDPF_PALETTEINDEXED8 | DDPF_RGB;
  m_SurfDesc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
  m_SurfDesc.dwFlags =DDSD_WIDTH | DDSD_HEIGHT | DDSD_PIXELFORMAT | DDSD_CAPS;

  // try create surface
  HRESULT err;
  err = pdDraw->CreateSurface(&m_SurfDesc, &m_pSurf, NULL);
```

Need to specify the size of offscreen surface. Get the size of the client area of the application.

Specify size, type, and pixel format of surface. Note that DDPFPIXELFORMAT is a structure and therefore dwSize must be set.

Specify which fields in descriptor were set.

REQUEST SIMPLE OFFSCREEN SURFACE.

```
    if (err != DD_OK) {
      handleError(err);
      return FALSE;
    }
    return TRUE;
  }
```

Exactly how to specify the color format has never been clearly documented, and we have found that techniques that work with previous versions of DirectDraw do not work with current versions. To reach a resolution, we set our primary display format to the color format we wanted, created a primary surface, got the surface descriptor with *GetSurfaceDesc()*, and looked at how the color format was specified there.

## 5.2 Drawing a Sprite on the DirectDraw Offscreen Surface

We created an Offscreen surface based on our preferred format. Now let's draw a sprite on this Offscreen surface. We can use the same sprite and background Blt classes (and routines) we used in the previous chapter.

```
CSurfaceOffscreen::BltSprite(CSprite &spr, CPoint &point)
{
  m_pSurf->Lock(NULL, &m_SurfDesc, DDLOCK_WAIT, NULL);    // first lock surface
  spr->Blt(m_SurfDesc.lpSurface, m_SurfDesc.lPitch, point); // invoke blt
  m_pSurf->Unlock(NULL);                                  // release the lock
}
```

Bltting sprites to an offscreen surface is pretty much the same as Bltting sprites to a Primary surface. Lock to get access to the surface, invoke the *SpriteBlt* routine with the newly obtained surface pointer and surface pitch, and then Unlock the surface.

When you Blt to an Offscreen surface, the results are not immediately visible. We have to Blt data from the Offscreen surface to the Primary surface to see the results.

On occasions when Offscreen surfaces are used for overlays or for texture maps, results may be immediately visible—look up the `DDSCAPS_VISIBLE` flag in the DirectDraw documentation on `DDSCAPS`.

Following is the code to transfer data to the visible surface.

```
CSurfaceOffscreen::Render(LPDIRECTDRAWSURFACE pPrimary, CWnd *pcWnd)
{
  CPOINT   ptTopLeft(0,0);
  pcWnd->ClientToScreen(&ptTopLeft);
  long     lRight  = ptTopLeft.x + m_dwWidth;
  long     lBottom = ptTopLeft.y + m_dwHeight;
  RECT rDst(ptTopLeft.x, ptTopLeft.y, lRight, lBottom);
  RECT rSrc(0, 0, m_dwWidth, m_dwHeight);

  // blt entire offscreen surface to subrect on primary surface
  err = pPrimary->Blt(&rDst, m_pSurf, &rSrc, DDBLT_WAIT, NULL);
}
```

- *IDirectDrawSurface::Blt* uses a BltFrom convention. The Blt function is invoked from the destination object; that is, the object that will get modified, *pPrimary*. This convention is consistent with the convention in MFC's *CDC::Blt*.
- The Blt operation can be further controlled by flags in the fourth parameter. There are over twenty-five controls, which include RasterOps, ColorFills, AlphaBlending, ChromaKeying, Z-Buffering, Rotation, other special effects, and more. We will use `DDBLT_KEYSRC` later in this chapter.
- Blt can be invoked asynchronously with the `DDBLT_ASYNC` flag. In Async mode, a successful return indicates no parameter errors were detected and the operation was successfully posted. See *IDirectDrawSurface::GetBltStatus* to check for completion/errors.
- We use `DDBLT_WAIT` to tell the Bltter to wait in case the Bltter hardware was already in use. The alternate option is for the Bltter to send us a `DDERR_WASSTILLDRAWING` error message if the Bltter was busy.
- Blt permits specifying a sub-rectangle of the source. By moving the sub-rectangle around you can scroll a view window within the source image. Setting Source and Dest rects to be of different sizes invokes a stretch (or shrink).

Blt can return a `DDERR_SURFACELOST` error message. Surfaces can be lost because the display card's mode was changed or because an application used an exclusive access mode. See *IDirectDrawSurface::Restore* to deal with lost surfaces.

## 5.3 Demo Time

At this point, you should be seeing sprites on the Primary surface. These sprites were drawn to an Offscreen surface, and the composited image was transferred to the Primary surface. Select the Primary Surface option from the sample application on the CD. You should see a sprite appear on the screen. Move the mouse around, and the sprite will follow the mouse.

How do you know we're using Offscreen surfaces? Move the mouse to the white border areas along the right or left edges. You may notice that the sprite seems to "wrap around" to the other edge. If this artifact occurs, it is because the rows of the Offscreen surface memory are packed contiguously. When we write beyond an edge, we "happen" to write into the adjacent column. This artifact may not occur if the DirectDraw driver allocated an Offscreen surface with noncontiguous rows (that is, lPitch > dwWidth).

Move the mouse to the bottom edge. You will notice that the sprite disappears *before* the mouse reaches the border. If we were to draw past the bottom border, we would write into unallocated memory and would generate a General

Protection Fault (GPF). Therefore, for this demo, we have deliberately chosen not to draw the sprite if it is going to extend past the bottom edge.

Note

Don't search for CSurface Offscreen on our sample CD. Instead we use two variants, CSurfaceVidMem and CSurfaceSysMem, which you can search for.

## 5.4 How Fast Is OffScreen Surface Drawing?

Table 5-1 measures the speed of drawing sprites and backgrounds to Off-Screen surfaces[1] merely to furnish a preliminary insight. More meaningful measurements, comparisons, and discussions are upcoming in this chapter.

**TABLE 5-1** Preliminary Measurements for Offscreen Surface Drawing

| CSprite 84 × 63 (16) | CBackground 734 × 475 | Refresh Screen 852 × 559 |
|---|---|---|
| 0.6–0.9 | 7.9 | 10.5 |

All times are in milliseconds.

These measurements look the same as the measurements for composited drawing to a Primary surface. So how about some acceleration? Where's the hardware?

## 5.5 Finding Hardware Acceleration

We used *PrimarySurface::Blt* to transfer data from the Offscreen surface to the Primary surface. Was this a hardware accelerated transfer? It's hard to say. DirectDraw has a Hardware Emulation Layer (HEL) that will emulate DirectDraw functionality in software. The purpose of the HEL is to always provide key DirectDraw features, even if the graphics hardware doesn't support them.[2]

1. Again, these measurements were taken on the base platform described in the Introduction and will definitely vary with different configurations.
2. Unfortunately, the DirectDraw HEL does not emulate all the features exposed by the DirectDraw interface. Therefore, you cannot rely on software emulation always being available. *DirectDraw::GetCaps* returns the capabilities of the hardware and the HEL independently.

So between the hardware and the HEL, we can't really say who did the *PrimarySurface::Blt.* Do you really need to know whether it was a hardware Blt? Yes. You may want to know for a couple of reasons.

1. Hardware Bltters are faster than software emulated Blts, and you may want to alter your application's logic to respond to the performance difference.
2. A Hardware Bltter may be available only under constraints, and you may want to constrain the environment to get hardware-accelerated performance. For example, hardware Blitting may only be available if both the source and the destination are located in video memory—therefore you may want to deliberately place objects in video memory. Similarly, the hardware might only be able to stretch in integer multiples, and you may want to disable arbitrary resizing to use hardware stretches. (Look up the DirectDraw documentation on DDCAPS, DDFXCAPS, and look for DDCAPS_CANBLTSYSMEM, DDFXCAPS_BLTSTRETCHXN, and DDFXCAPS_BLTSTRETCHYN flags for more details on these examples.)

## 5.6 Setting Up for Hardware Acceleration

Here's some code we can use to find out some of the hardware's capabilities.

```
CHardware::GetCaps(LPDIRECTDRAW pdDraw)
{
   DDCAPS  hwCaps = {0}, helCaps = {0};
   hwCaps.dwSize = sizeof(DDCAPS);
   helCaps.dwSize = sizeof(DDCAPS) ;
   pDDraw->GetCaps(&hwCaps, &helCaps);
   if (hwCaps.dwCaps & DDCAPS_BLT )              ◊ Can the h/w Blt?
       m_bCanBltVidMem = TRUE;
   if (hwCaps.dwCaps & DDCAPS_CANBLTSYSMEM)      ◊ Can it Blt from/to system memory?
       m_bCanBltSysMem = TRUE;
}
```

- Two DDCAPS structures are passed to *IDirectDraw::GetCaps* in which we get back descriptions of both the hardware device and the Hardware Emulation Layer.
- DDCAPS structures are huge and allow for a wide variety of features to be described. In our code we are mainly interested in DDCAPS_BLT and DDCAPS_CANBLTSYSMEM. Later in this chapter we will look for DDCAPS_COLORKEY.
- Take a look at the documentation for DDCAPS (and its contained structures) to get a feel for the breadth of hardware features that can be exposed via DirectDraw. DDCAPS_GDI, DDCAPS_VBI, and DDCAPS_ PALETTEVSYNC are features that might be useful. DDCAPS_STEREOVIEW and DDCAPS_READSCANLINE at the very least attract attention.
- DDCAPS contains within it a DDSCAPS structure that during *GetCaps* will be filled by the kinds of DirectDraw surfaces that can be created.

Not all features may be available simultaneously. For instance, by using one feature, another may become unavailable.

Of course, a lot more information is returned in the DDSCAPS structure. We've only highlighted the capabilities we're looking for. And now, here's code to situate an Offscreen surface in video memory.

```
BOOL CSurfaceVidMem::Init(LPDIRECTDRAW pdDraw, CWnd *pcWnd)
{
    RECT  rWin;
    pcWnd->GetClientRect(&rWin);
    m_dwWidth = (DWORD)(rWin.right - rWin.left);
    m_dwHeight = (DWORD)(rWin.bottom - rWin.top);
    // check if there's enough memory for vidMem based surface
    DWORD  dwTotal, dwFree;
    DDSCAPS ddsCaps;
    ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
    pDDraw->GetAvailableVidMem(&ddsCaps, &dwTotal, &dwFree);
    DWORD dwSurfSize = m_dwWidth * m_dwHeight;
    if (dwFree < dwSurfSize) {
        handleError(DDERR_OUTOFVIDEOMEMORY);
        return FALSE;
    }

   // set desired fields
   m_SurfDesc.dwHeight = m_dwHeight;
   m_SurfDesc.dwWidth = m_dwWidth;
   m_SurfDesc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN ;
   m_SurfDesc.ddsCaps.dwCaps |= DDSCAPS_VIDEOMEMORY;
   m_SurfDesc.ddpfPixelFormat.dwSize = sizeof(DDPIXELFORMAT);
   m_SurfDesc.ddpfPixelFormat.dwRGBBitCount = 8;
   m_SurfDesc.ddpfPixelFormat.dwFlags = DDPF_PALETTEINDEXED8 | DDPF_RGB;
   m_SurfDesc.dwFlags =DDSD_WIDTH|DDSD_HEIGHT|DDSD_PIXELFORMAT|DDSD_CAPS;

   // try create surface
   HRESULT err;
   err = pdDraw->CreateSurface(&m_SurfDesc, &m_pSurf, NULL);
   if (err != DD_OK) {
      handleError(err);
      return FALSE;
   }
   return TRUE;
}
```

The *IDirectDraw* object has a function to query how much memory is available on the graphics device. Of course, you could just try to create a surface and look at the return code if the attempt failed.

Or in DDCAPS_VIDEOMEMORY flag to force surface to be created with video memory. The HEL will not allocate an emulated offscreen surface if the device failed the request.

Despite a successful negotiation of *GetAvailableVidMem*, the *CreateSurface* call can still return DDERR_OUTOFVIDEOMEMORY. This is because we calculated dwSurfSize as dwWidth*dwHeight. However, the graphics card may use an lPitch (allocated row width) larger than dwWidth and there may not be enough memory for lPitch*dwHeight.

For the times when there isn't enough memory on the video card to create a video memory surface, our code uses a CSurfaceSysMem surface. The code for this option is similar to the code we have used to situate an Offscreen surface in video memory. In this case, the DDSCAPS_SYSTEMMEMORY is used instead of the DDSCAPS_VIDEOMEMORY. Also the check for memory is not needed, since the surface will be allocated in system memory.

Following is the code that uses the hardware Bltter. Looks just like *CSurfaceOffscreen::Render*—doesn't it? Except now we know we're using the hardware Bltter, because we checked that the graphics card did indeed have a hardware Bltter, and when we created the surface we forced it to reside in video memory.

```
CSurfaceVidMem::Render(LPDIRECTDRAWSURFACE pPrimary, CWnd *pcWnd)
{
   CPOINT ptTopLeft(0,0);
   pcWnd->ClientToScreen(&ptTopLeft);
   long lRight  = ptTopLeft.x + m_dwWidth;
   long lBottom = ptTopLeft.y + m_dwHeight;
   RECT  rDst(ptTopLeft.x, ptTopLeft.y, lRight, lBottom);
   RECT rSrc(0, 0, m_dwWidth, m_dwHeight);

  // blt entire offscreen surface to subrect on primary surface
  err = pPrimary->Blt(&rDst, m_pSurf, &rSrc, DDBLT_WAIT, NULL);
}
```

## 5.7 How Fast Is CSurfaceVidMem Drawing?

Table 5-2 measures the speed at which objects can be drawn when using a hardware-accelerated Offscreen surface.

TABLE 5-2 Measurements for Offscreen Surface Drawing

| Surface | CSprite 84 × 63 (16) | Cbackground 734 × 475 | Refresh Screen 852 × 559 | Post Refresh | Total |
|---|---|---|---|---|---|
| CSysMem | 0.6–0.9 | 7.9 | 10.5 | 0.0 | 19.3 |
| CVidMem | 0.7–1.8 | 7.7 | 0.1 | 8.9 | 9.6/18.5 |

All times are in milliseconds.

Wow! Refresh Screen is a minuscule 0.1 millisecond. The total time seems halved. Wow! But what's this new column for Post Refresh? Well, when we invoke *pPrimary->Blt,* the hardware Bltter returns as soon as it has *started* the Blt. We can use the main processor, while the graphics processor does the Blt in the background. This is in effect a form of parallel processing. (Maybe one day there will be many of these little processors working in parallel. Oh wait, isn't that what's in them "soopah computahs"? Never mind.)

Parallel processing is functional as long as we don't want to use the same memory that the graphics processor is using. In other words, we would be denied access to the Primary surface while the graphics processor was still working. In this case we would have to wait until the Blt was complete. Post Refresh is a measurement of the worst-case scenario for wait time—we tried to lock the surface immediately after the Blt, and then we measured how long we had to wait.

The gist of all this is

1. We can, indeed, increase overall application speed as long as the application can work on something else while the graphics processor is Bltting in the background. This seems like a fairly workable situation.
2. We have not gained much benefit in the overall "Composite and Render" time. Time is gained *only* by parallel processing and not by reducing the length of the graphics rendering steps.

## 5.8 Accelerating Offscreen to Primary Transfers by Page Flips

We've used the hardware Bltter to transfer data from Offscreen surfaces to the Primary surface. On timing these data transfers, we find that despite using a hardware Bltter, the actual Blt cost is still about 9 milliseconds. Let's look at *Page Flipping hardware* in graphics devices to tap into an even faster mechanism for making background data visible.

### 5.8.1 What Is Graphics Page Flipping?

Consider that the display screen is being constantly refreshed at the monitor refresh rate (anywhere between 30 and 90 times per second). Pixel data to be displayed on the screen is retrieved from somewhere in graphics card memory.

What if the location of the data was specified by a pointer; that is, what if the monitor refresh hardware used an indirect reference to access pixel data. Change the value of the pointer and an entirely new image is being displayed on the screen. This in effect is Page Flipping.

**TABLE 5-3** Constraints on Using Page Flipping

| | |
|---|---|
| Page Flipping hardware, in general, is designed to get the pointer value and then de-reference the pointer to refresh the entire image. | Therefore, only the entire screen can be Page Flipped (not independent windows). |
| The screen area of the display can be reconfigured to different shapes. But once configured, graphics cards are designed for constant screen areas. | Therefore, all buffers used for Page Flipping must be of the same size. |
| With two buffers, data written into the invisible buffer is not written into the visible buffer. Data will be missing from alternate buffers unless it is written into both, and the result will be an annoying flicker. | Therefore, either the entire scene must be redrawn, or some intelligent logic must be used to make data continue to exist across buffers. |
| Similarly, GDI does not know that we are in Page Flip mode, and the data that GDI drew into one buffer would "vanish" when we Page Flipped. (It would reappear when we Page Flipped again, producing an apparent flicker.) | Therefore Page Flipping can only be used in "Exclusive" mode, and other applications cannot share the display while Page Flipping is in use. |

## 5.8.2 DirectDraw Page Flipping Model

Let's say we set up two buffers. One buffer is the visible buffer and is called the *front* buffer. The second buffer is invisible and is called the *back* buffer. When the graphics card Page Flips, it makes the back buffer visible; that is, the back buffer becomes the front buffer.

Now, let's suppose we wanted to render the next image into what was previously the front buffer. Which surface do we Lock to get back a usable pointer? The code that follows in the next subsection will show that after a Page Flip, DirectDraw makes the front buffer into the back buffer and vice versa, and therefore all we need to do is to *Lock()* what was our back buffer.

## 5.8.3 Does the Hardware Support Page Flipping?

Let's find out whether the hardware supports Page Flipping.

```
CHardware::CanTransparentBlt()
{
   DDCAPS  hwCaps = {0}, helCaps = {0};
   hwCaps.dwSize = sizeof(DDCAPS);
   helCaps.dwSize = sizeof(DDCAPS);
   pDDraw->GetCaps(&hwCaps, &helCaps);

  BOOL bCanPageFlip = FALSE;
  if (hwCaps.ddsCaps.dwCaps & DDSCAPS_FLIP)
    bCanPageFlip = TRUE;
}
```

DirectDraw indicates support for Page Flipping by indicating that Flip surfaces can be created.

## 5.8.4 Setting Up DirectDraw to Use Page Flipping

There are two ways to set up Page Flipping. The first is to have DirectDraw create a *complex* surface that automatically creates and connects multiple buffers. The second approach is to create a Primary surface, create Offscreen Surfaces, and then to *Attach* the Offscreen surfaces to the Primary Surface.

We will demonstrate the second path, because it gives us the opportunity to help you past some tough problems that you would experience if you needed to use this path.

```
BOOL CSurfaceBackBuffer::Init(LPDIRECTDRAW pdDraw, CWnd *pcWnd)
{
    pDDraw->SetCooperativeLevel(pcWnd->hWnd,
                DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN);
```

First we've got to set up DirectDraw to be in FullScreen mode to do Page Flipping.
- FullScreen mode can only be setup, if we have Exclusive access to the screen.
- DirectDraw will return an error if you *SetCooperativeLevel* while you have any surfaces created. Shut down any surfaces prior to using this function.

The documentation states that *IDirectDrawSurface::Flip* can only be invoked on a buffer marked as the DDSCAPS_FRONTBUFFER surface from among a group of buffers that have been marked as DDSCAPS_FLIP. But if you tried to set the DDSCAPS_FRONTBUFFER, DirectDraw will return an error stating that DDSCAPS_FRONTBUFFER is not a settable flag, therefore this code has been commented out. We found that Flips work when you don't set these flags.

```
    // create a primary surface
    memset(&m_PrimDesc, 0, sizeof(DDSURFACEDESC));
    m_PrimDesc.dwSize = sizeof(DDSURFACEDESC);
    m_PrimDesc.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
    m_PrimDesc.dwFlags = DDSD_CAPS;
    // m_PrimDesc.ddsCaps.dwCaps |=  (DDSCAPS_COMPLEX);
    // m_PrimDesc.ddsCaps.dwCaps |=  (DDSCAPS_FRONTBUFFER|DDSCAPS_FLIP);
    pDDraw->CreateSurface(&m_PrimDesc, &m_PrimSurf, NULL);
```

- The back buffer must be pretty much identical to the front buffer. So tell DirectDraw to describe the primary surface and copy it over.
- Create the back buffer as a plain Offscreen surface. Force it to reside in video memory, so that graphics hardware can Flip the surface. Also, see note above on Flip flags.

```
    m_PrimSurf.GetSurfaceDesc(&m_SurfDesc);
    memcpy(&m_SurfDesc, &m_PrimDesc, sizeof(DDSURFACEDESC));
    m_SurfDesc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN|DDSCAPS_VIDEOMEMORY;
    // m_PrimDesc.ddsCaps.dwCaps |=  (DDSCAPS_COMPLEX);
    // m_SurfDesc.ddsCaps.dwCaps |=  (DDSCAPS_BACKBUFFER|DDSCAPS_FLIP);
    m_SurfDesc.dwFlags = DDSD_WIDTH | DDSD_HEIGHT|DDSD_PIXELFORMAT|DDSD_CAPS;
```

Attach the Offscreen surface to the Primary surface. A chain of multiple buffers can be attached.

```
    // create the offscreen surface and attach it to the primary surface
    pdDraw->CreateSurface(&m_SurfDesc, &m_pSurf, NULL);
    m_pPrimSurf->AddAttachedSurface(m_pSurf);

    return TRUE;
}
```

### 5.8.5 "Rendering" Flippable Surfaces

A back buffer is just like any other Offscreen surface, and the code to draw objects to CSurfaceBackBuffer is just like the code to draw objects to CSurfaceVidMem. However, the code to make the data in the back buffer visible is different. We use Flip instead of Blt to "transfer" the data to the Primary surface.

```
CSurfaceBackBuffer::Render()
{
    // Flip with WAIT-UNTIL-READY
    m_pPrimSurf.Flip(NULL, (DWORD)DDFLIP_WAIT);
}
```

## 5.9 How Fast Is CSurfaceBackBuffer Drawing?

Table 5-4 measures the speed of drawing objects with a hardware accelerated Offscreen surface.

**TABLE 5-4** Measurements for Offscreen Surface Drawing

| Surface | CSprite 84 × 63 (16) | CBackground 734 × 475 | Refresh Screen 852 × 559 | Post Refresh | Total |
|---|---|---|---|---|---|
| CSysMem | 0.6–0.9 | 7.9 | 10.5 | 0.0 | 19.3 |
| CVidMem | 0.7–1.8 | 7.7 | 0.1 | 8.9 | 17.4/18.5 |
| CBackBuffer | 0.7–1.8 | 7.7 | 0.1 | 0.5–7.6 | 9.0–17.2 |

All times are in milliseconds.

The variability in the Post Refresh for back buffers is the glaring figure. *IDirectDrawSurface2::Flip()* is always synchronized with the Vertical Blank Interval[3] (VBI) of the monitor. So if you invoked *Flip()* function just before the VBI, then the *Flip()* function would be instantaneous. But if you invoked *Flip()* function just after the VBI, then you would have to wait for VBI for the flip to occur.

3. Vertical Blank Interval is the time interval when no monitor refresh is occurring, because the monitor's beam is returning from the end position (bottom right corner) to the start position (top left corner) after having refreshed the screen.

DirectDraw will not let you lock a back buffer surface that is waiting to be flipped. However, in contrast to the use of the Bltter, a "Wait for VBI" does not consume hardware resources, and DirectDraw will allow you to use other surfaces. So what if you had a second back buffer that you could render to, while the other one was waiting to be flipped? This is known as *triple buffering. Post Refresh cost is negligible with triple buffering,* and the only practical limit to the frame refresh rate is the refresh rate of the monitor.

**EXTRA CREDIT**

Implement triple buffering by extending the double-buffering sample application on the CD. Note: attached buffers are linked in a circular fashion, and the buffer attached last is the next one that will be made visible. Extend your exercise to measure the performance implications of writing into the back buffer in the wrong order. The timing application *TimingApp* has sample code for triple buffering, in case you run into problems.

## 5.10 Hardware Acceleration to Blt Sprites

We used hardware to Blt from Offscreen surface to Primary surface. Why not use hardware to render objects too? Sprites are transparent objects. To accelerate sprite Bltting with hardware, we must find out whether the hardware can handle data with transparency.

```
CHardware::CanTransparentBlt()
{
  DDCAPS  hwCaps = {0}, helCaps = {0};
  hwCaps.dwSize = sizeof(DDCAPS);
  helCaps.dwSize = sizeof(DDCAPS);
  pDDraw->GetCaps(&hwCaps, &helCaps);
```

DDCAPS_COLORKEY says that some form of colorkey is supported. DDCAPS_COLORKEYHWASSIST says that the color-keying is done by hardware. We would have preferred to check only for the second flag, but we found some graphics cards that only set the first flag. Graphics card vendors are not supposed to provide software emulation. You may want to test the performance of color-key implementations.

```
  BOOL bCanKey = FALSE;
  if ((hwCaps.dwCaps & DDCAPS_COLORKEY) ||
      (hwCaps.dwCaps & DDCAPS_COLORKEYHWASSIST))
      bCanKey = TRUE;
  if (bCanKey && (hwCaps.dwCKeyCaps & DDCKEYCAPS_SRCBLT))
      m_bCanTranspBlt = TRUE;
}
```

Color-keying can take many forms—there are about eighteen different flags defined in the DirectDraw documentation. The dwCKeyCaps field in the DDCAPS structure describes supported color-keying forms. Once we know that some form of color key is supported, we've got to check if it's a form we can use. For our definition of sprites, our sample application looks for DDCKEYCAPS_SRCBLT color-keying.

Once we've found that the hardware is indeed capable of Bltting sprites, we can look at code to set up sprites for hardware Bltting and to Blt the sprites.

```
BOOL CSpriteGrfx::Init(LPDIRECTDRAW pdDraw,
CBitmap &bitmap, BYTE byKeyColor)
{
  // load data from bitmap and init member variables
  BITMAP bm;
  bitmap.GetBitmap(&bm);
  pData = new BYTE[bm.bmWidth * bm.bmHeight];
  bitmap.GetBitmapBits(bm.bmWidth * bm.bmHeight, pData);
  m_dwWidth = bm.bmWidth;
  m_dwHeight = bm.bmHeight;
  m_byTransp = byKeyColor;
```

We start out just like we were creating a hardware-accelerated Offscreen surface.

```
  DWORD dwTotal, dwFree;
  DDSCAPS ddsCaps;
  ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
  pDDraw->GetAvailableVidMem(&ddsCaps, &dwTotal, &dwFree);
  DWORD dwSurfSize = m_dwHeight * m_dwWidth;
  if (dwFree < dwSurfSize) {
      handleError(DDERR_OUTOFVIDEOMEMORY);
      return FALSE;
  }
  m_SurfDesc.dwHeight = m_dwHeight;
  m_SurfDesc.dwWidth = m_dwWidth;
  m_SurfDesc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN ;
  m_SurfDesc.ddsCaps.dwCaps |= DDSCAPS_VIDEOMEMORY;
  m_SurfDesc.ddpfPixelFormat.dwSize = sizeof(DDPIXELFORMAT);
  m_SurfDesc.ddpfPixelFormat.dwFlags = DDPF_PALETTEINDEXED8;
  m_SurfDesc.dwFlags =DDSD_WIDTH|DDSD_HEIGHT|DDSD_PIXELFORMAT|DDSD_CAPS;
```

Add in specification of color key and set dwFlags to indicate that this field is valid. The Color key can be a range. Our sample only uses a single color and sets high and low to be the same.

```
  DWORD dwKey = (DWORD)byKeyColor;
  m_SurfDesc.ddckCKSrcBlt.dwColorSpaceLowValue = dwKey;
  m_SurfDesc.ddckCKSrcBlt.dwColorSpaceHighValue = dwKey;
  m_SurfDesc.dwFlags |= DDSD_CKSRCBLT;

  // try create the surface
  HRESULT err = pdDraw->CreateSurface(&m_SurfDesc,&m_pSurf,NULL);
  if (err != DD_OK) {
     handleError(err);
     return FALSE;
  }

}
```

Set up the sprite, by transferring data to the Offscreen surface at Init-Time. Remember to lock surface to get access and unlock surface after use.

```
  m_pSurf->Lock(NULL, &m_SurfDesc, DDLOCK_WAIT, NULL);
  PBYTE pDst = (PBYTE)m_SurfDesc.lpSurface;
  PBYTE pSrc = m_pData;
  for (DWORD dwRow = 0; dwRow < m_dwHeight; dwRow++) {
     memcpy(pDst, pSrc, m_dwWidth);
     pDst += m_SurfDesc.lPitch;
     pSrc += m_dwWidth;
  }
  m_pSurf->Unlock(NULL);
  return TRUE;
}
```

Next we'll check out the code that Blts sprites with the hardware Bltter. It's pretty much like CSurfaceVidMem, except for the DDBLT_KEYSRC added into the Blt control flag.

```
CSpriteGrfx::Blt(LPDIRECTDRAWSURFACE pDstSurf, CPoint &ptDst)
{
  long  lRight  = ptDst.x + m_dwWidth;
  long  lBottom = ptDst.y + m_dwHeight;
  RECTrDst(ptDst.x, ptDst.y, lRight, lBottom);
  RECTrSrc(0, 0, m_dwWidth, m_dwHeight);

  // blt entire sprite surface to subrect on dest surface
  pDstSurf->Blt(&rDst,m_pSurf,&rSrc,DDBLT_WAIT|DDBLT_KEYSRC,NULL);
}
```

Blt sprite to destination surface. Specify that the sprite has transparency and that it is keyed on the source.

## 5.11 How Fast Is CSpriteGrfx (and CBackgroundGrfx) Drawing?

Table 5-5 on the next page records the speed of drawing sprites and backgrounds with and without hardware acceleration. Following the table are some observations on the measurements.

**TABLE 5-5** Measurement of Hardware-Accelerated Object Drawing

| CSpriteP5 84 × 63 (16) | CSpriteGrfx 84 × 63 (16) | CBackgroundP5 734 × 475 | CBackgroundGrfx 734 × 475 |
|---|---|---|---|
| 0.7–1.8 | 2.4–2.7 | 7.9 | |

All times are in milliseconds.

- CSpriteGrfx is not as sensitive to unaligned writes as is CSpriteP5. But on the platform we're using the CPU-based sprite drawing routine is faster than the graphics hardware–based routine. Note that CSpriteP5's time will vary with CPU speed, while CSpriteGrfx's time probably won't.
- Non-transparency Blts run faster when you use hardware acceleration. The benefit is especially noticeable for large backgrounds. The CPU in general is faster at Transparent Blts, with the performance difference increasing with faster CPU speeds.

Note

CPU-based rendering needs surfaces to be locked, but hardware-accelerated sprites do not—so you would need to do constant Locks and Unlocks. Lock/Unlock costs may be comparatively large when you are rendering small sprites. Our surface-rendering code adds logic to avoid unnecessary locks and unlocks. Based on your sprite sizes, it may be worth making this effort to minimize locks and unlocks.

## WHAT HAVE YOU LEARNED?

By this time, you've got a taste of accessing device features to accelerate multimedia performance under Windows. If you worked through the code samples, you would have handled code to create and accelerate offscreen buffers, set the display into full-screen mode, and performed Page Flipping. You would also have worked through code to draw sprites and backgrounds using hardware acceleration.

You should also have gotten a taste for the performance costs of various options. As a result you have some experience of how important it is to apportion device memory resources thoughtfully. Our acceleration strategy for the multimedia objects in this section would be to give triple buffering the highest priority, to give background Blting the second highest priority, and to try to use the CPU for Transparent Blts. We hope we have sparked some ideas on what memory allocation strategy would serve your application best.

Device dependence can be burdensome. There's a lot of code to be written, debugged, profiled for performance, and optimized. In the next chapter you will learn of software libraries with higher-level APIs that work to provide high-performance multimedia without the burden of device dependence—sort of like a "GDI for Multimedia."

CHAPTER 6

# RDX: High-Performance Mixing with a High-Level API

**WHY READ THIS CHAPTER?**

In the previous chapters, you were introduced to several device-dependent paths for achieving higher-performance sprites using Microsoft DirectDraw. But for routine multimedia, wouldn't it be nice to have the multimedia equivalent of GDI—to be able to program with multimedia objects like sprites, backgrounds, and video streams without having to worry about the idiosyncrasies of device implementations?

This chapter will introduce you to Intel's Realistic Display Mixer (RDX), which offers a device-independent interface for multimedia objects. RDX has hand tuned assembly code for accomplishing high performance on unaccelerated platforms, and it accesses hardware features whenever available for further acceleration. Read on and decide whether the ease of programming makes up for having to learn a new interface. Decide whether the features and performance make up for any reduced flexibility. As you work through this chapter, you will

- get an overview of what RDX is and what it offers,
- learn how to use RDX to render fast sprites and backgrounds, and
- learn how to direct RDX's use of DirectDraw.

## 6.1 Introduction to Intel's RDX Animation Library

Direct access to device features, through interfaces like DCI and DirectDraw, is one way of addressing performance problems for multimedia under Windows. The downside of direct access is device dependence. Direct access also forces the programmer to learn how a variety of graphics devices work.

Intel developed the Realistic Display Mixer (RDX) system to provide developers with a high-level interface to manage multimedia objects and multimedia devices. Since multimedia applications are performance sensitive, Intel's RDX system has been engineered to provide its high-level abstraction without sacrificing performance.

Figure 6-1 shows RDX within the context of the Windows 95 display architecture. The system can be considered as "middle-ware," providing abstractions above Microsoft's DirectDraw and Direct Video, and interacting with Video for Windows (VFW). RDX can also interface with Microsoft's Direct3D and ActiveMovie components.

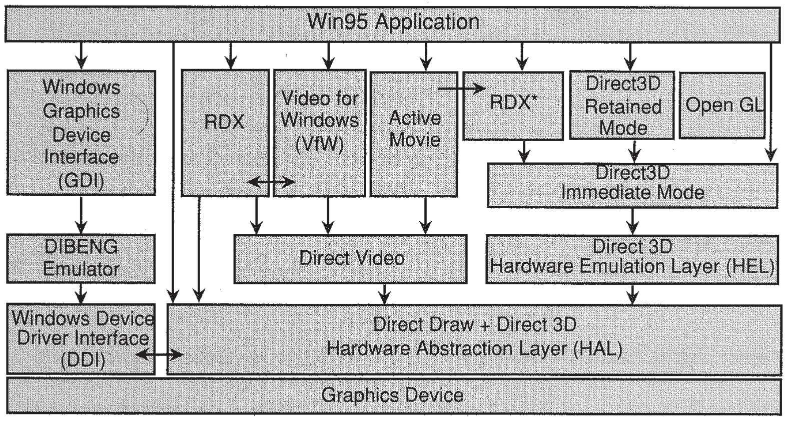

Win95 Application
Windows Graphics Device Interface (GDI)
RDX
Video for Windows (VfW)
Active Movie
RDX*
Direct3D Retained Mode
Open GL
Direct3D Immediate Mode
DIBENG Emulator
Direct Video
Direct 3D Hardware Emulation Layer (HEL)
Windows Device Driver Interface (DDI)
Direct Draw + Direct 3D Hardware Abstraction Layer (HAL)
Graphics Device

**FIGURE 6-1** RDX within the Windows 95 display architecture.

## 6.1.1 Features of RDX

RDX is a high-performance multimedia object management system that allows developers to program at a higher level without a performance penalty. The RDX system makes extensive use of hand-tuned assembly code to obtain high performance even on unaccelerated platforms. RDX can use hardware acceleration when available and can also be upgraded with assembly code modules tuned to future processors.

Figure 6-2 shows the RDX object management system architecture. The architecture allows multimedia objects like sprites, backgrounds, video, or 3D to be mixed with one another. The system can handle even complex mixing scenarios such as video on video with differing frame rates.

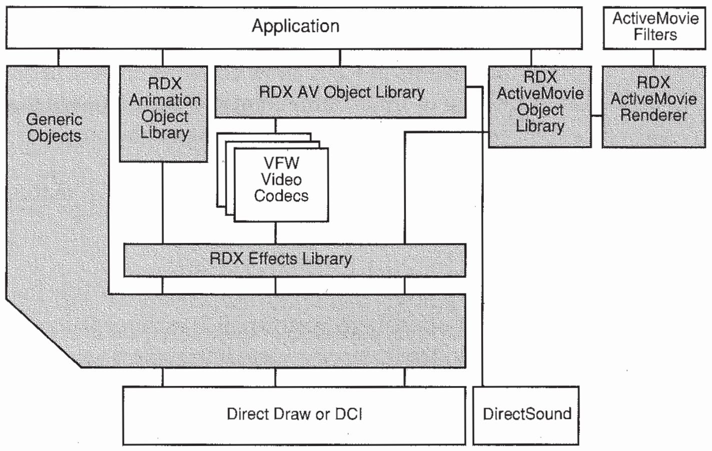


**FIGURE 6-2** RDX object management system architecture.

The RDX system consists of a mixer module and some other object modules:

- The Mixer module defines generic mixable objects, object attributes, and attribute management functions. Objects that follow the rules of the mixer module can be mixed with one another without knowing about one another. The Mixer module interfaces with DirectDraw and accesses suitable hardware acceleration.
- The RDX Audio-Video (AV) module supports video tracks, audio tracks, and AV streams as mixable objects. It interfaces with the Video for Windows Audio-Video Interleaved (AVI) file format and VFW-based codecs. The AV module supports sources of transparent video such as Indeo Video Interactive.
- The RDX Animation module contains hand-tuned assembly code to support sprites, backgrounds, and tiled grids as mixable objects. The RDX Animation module also exports effects that can be applied on objects from both the Animation module and the AV module. Examples of effects include shearing and horizontal or vertical flipping.

In addition to the mixing and the predefined objects, RDX offers a fairly straightforward interface based on all objects having generic attributes. RDX uses simple function calls for various DirectDraw display modes to hide underlying device dependence.

RDX offers grouping to easily manipulate the attributes of many objects. It offers effects to transform the image data of an object at draw-time. Timers and events in RDX allow activities to be scheduled and allow the synchronizing of time-stamped material such as video streams. Collision and hot spots detect interactions between objects.

This chapter deals primarily with the Animation module and the Mixer module.

### 6.1.2 Before You Get Overly Excited

RDX offers high performance without the programming burden or device dependence, but the ease of use may come at the expense of reduced flexibility. For example, for practical reasons RDX will only use a subset of the features that are offered by DirectDraw and Video for Windows. Similarly, RDX will only tap into a subset of the acceleration features offered by hardware devices.

Just as when we were evaluating GDI, we must measure the strengths of RDX's device independence against the possibility of reduced flexibility. RDX does not prevent simultaneous use of DirectX, but in that case, the benefits of programming ease are defeated. And! Having to learn more than one set of APIs and debugging more than one system component are added burdens.

## 6.2 Using RDX

RDX is an object-based system. The object hierarchy in RDX contains generic objects, render objects, and source data objects.

- All displayable RDX objects are derived from a *generic objects* base class. The generic objects base class defines a set of generic attributes and a set of attribute management functions.
- Displayable objects derive from the base class and can be called *render objects*. Sprites, backgrounds, grids/tiles, and videotracks are examples of render objects. In addition to the inherited functions, render objects also define object-specific functions, which work only on objects of that particular kind.
- Render objects inherently do not contain image data; rather they maintain links to data objects called *source data objects* (for example, bitmaps, avFiles). A single source data object can be shared by several render objects.

The RDX system also provides support objects and support functions, which operate on groups, effects, timers, events, and so forth.

## 6.2.1 Generic Objects with RDX

All displayable RDX objects are derived from a *base class* and inherit a set of *generic attributes* from the base class. The system provides a set of *attribute functions*, which an application can use to set, get, and change attribute values.

Table 6-1 lists the generic attributes that are inherited by all RDX objects. Functions to manipulate these generic attributes all begin with the *obj* mnemonic. For example, the functions to modify destination are *objSetDestination*, *objGetDestination*, and *objAdjDestination*.

**TABLE 6-1** Generic Attributes Inherited by RDX Objects

| | |
|---|---|
| View | Application-defined subset of an object's source image area |
| Visibility | Toggle on/off whether an object should be rendered |
| Draw Order | Priority order of objects drawn on top of each other |
| Current Image | Index to image to be rendered from within a sequence |
| Destination | Surface to which the object is mapped; surface destinations can themselves be mapped to windows, memory, or other surfaces |
| Destination Rectangle | Application-defined subset of destination area |
| Position | Location of object on its destination; modifying the position also modifies the Destination Rectangle and vice versa |

**RDX INTERFACE CONVENTION**

The object-oriented nature of RDX allows objects to be treated in similar ways, giving the system a significant degree of uniformity and consistency. Though RDX is object oriented, its interfaces are based in C.

RDX uses a noun/verb naming convention for its function calls. Every function call is prefixed by a mnemonic for the object to be operated on. The prefix is followed by a verb, and further descriptors then follow.

In keeping with the noun/verb naming convention, all RDX function calls (except *xxxCreate*) take in the object as the first parameter. All RDX function calls return a `DINORVAL` return value. Return values are returned using pointer indirection. The return pointers are always placed last in the parameter list.

The nature of some objects forces minor variations in the generic attribute semantics. For example, an audio track has the visibility attribute, but its meaning is undefined. Similarly, the current image attribute is ambiguous for nonsequenced objects. The exceptions are fairly minor.

### 6.2.2 The Programming Model

The general model for building an RDX render object can be broken into three parts:

1. Create a Source Data Object (SDO) and load its data, or reference a created SDO.
2. Create the render object itself and associate it to the SDO.
3. Set any appropriate object attributes to nondefault values.

The general model for preparing render objects for display has two parts:

1. Create a surface, or reference an already created surface. Map the surface to a window or a memory buffer. This defines where the surface will be drawn at draw time.
2. Map render objects to the surface. Each time you map an object to a surface, the system uses the object's draw order to place it in an ordered display list associated with the surface.

Now draw the surface. When you do this, the system traverses the display list, drawing each object in the list into a buffer, and then the system "transmits" the buffer to the final destination. (The system will use the appropriate DirectDraw Primary or Offscreen buffers if the final destination is a window.)

## 6.3 Working with RDX

### 6.3.1 Creating an RDX Surface

Enough reading! Time to work. Here's the code for creating an RDX surface.

: attribute
, but its
ıbiguous

ken into

ence a

vo parts:

:he surface
ıce will be

ect to a
ın ordered

e display
tem "trans-
appropri-
ion is a

ƚDX

```
CSurfaceRdx::Init(CWnd *pcWnd)
{
    RECT rWin;
    pcWnd->GetClientRect(&rWin);
    DWORD dwWidth = (DWORD)(rWin.right - rWin.left);
    DWORD dwHeight = (DWORD)(rWin.bottom - rWin.top);
    srfCreate(dwWidth, dwHeight, RGB_CLUT8, &m_hSurf);
```

1. Create a surface with same size as the client area. The size does not have to match since RDX will clip to destination if needed. Create the smallest surface needed—reducing size increases performance by reducing the area to be redrawn and also makes it easier to fit the surface onto graphics memory.
2. Specify the color format of the surface. Objects will be mixed in this color format. We could insert color converters, but our example is simple and is designed for everything to be set up in RGB 8-bit mode.
3. RDX returns a handle to the surface in the space we pointed to in the last parameter, M_HSURF.

```
    srfSetDestWindow(m_hSurf, pcWnd->m_hWnd);
    return TRUE;
}
```

Map the surface to the destination window. RDX will watch for window movements or size changes and will clip the image if needed.

## 6.3.2 An RDX Sprite Class

And here's the code for initializing an RDX-based CSprite.

```
BOOL CSpriteRDX::Init(HOBJ hSurf, UINT nResID, BYTE byKeyColor)
{
    BITMAP bm;
    bitmap.GetBitmap(&bm);
    m_dwWidth = bm.bmWidth;
    m_dwHeight = bm.bmHeight;
    m_byTransp = byKeyColor;
```

Create and set up an *hbmp* (SDO).

```
    HBMPHEADER bmpHeader;
    hbmpCreate(m_dwWidth,m_dwHeight,RGB_CLUT8,&m_hBmp);  ◊ Create RDX hbmp object.
    BYTE *pData;
    hbmpGetLockedBuffer(m_hBmp, &pData, &bmpHeader);      ◊ Get access to RDX
                                                            space.
    bitmap.GetBitmapBits(m_dwWidth*m_dwHeight, pData);    ◊ Load data into RDX
                                                            space.
    hbmpReleaseBuffer(m_hBmp);                            ◊ Release access.
    hbmpSetTransparencyColor(m_hBmp, (DWORD)byKeyColor);◊ Specify that bitmap is

                                                            transparant.
```

Create sprite; associate data to it; associate sprite to surface.

```
    sprCreate(&m_hSpr);
    sprSetData(m_hSpr, m_hBmp);
    objSetDestination(m_hSpr, m_hSurf);

    return TRUE;
}
```

Pass sprite handle to *Obj* function call. The generic object (*Obj*) will be manipulated from the actual object.

### 6.3.3 Drawing the RDX Sprite

With RDX, we don't have to actually draw the sprite. We merely adjust any relevant attributes (such as position and/or draw order) and "draw" the surface to which the sprite was connected.

```
CSurfaceRdx::BltSprite(CSprite &spr, CPoint &point)
{
   objSetPosition(m_hSpr, point);       // set location in surface
   objSetDrawOrder(m_hSpr, 1);          // smaller number means in front
}

CSurfaceRdx::Render()
{
   srfDraw(m_hSurf);
}
```

*SrfDraw* invokes, in back-to-front order, render routines of all objects mapped to the surface. This call then transfers the resulting composited image to the destination window using the appropriate *IDirectDraw* method.

## 6.4 Demo Time

At this point, select the RDX option in the sample application on the CD. You should be seeing sprites on the application window. These sprites were drawn using an RDX surface and an RDX sprite.

How do you know we're using RDX? Move the mouse to the white border areas along the right or left edges. The sprite is clipped to the boundaries of the clipping window. Move the window to a different position, and now move the mouse over the clipping window. The mouse is drawn at the window's new position. The Clipper code within the RDX library is automatically handling window moves.

### 6.4.1 How Fast Does CSurfaceRdx Draw?

Table 6-2 shows the speed at which objects are drawn with RDX in comparison to the methods used in the previous chapter. Some observations on the measurements:

- RDX's drawing time is as good or better than our optimized routines from the previous chapter. RDX automatically senses the MMX technology capabilities of the platform we are using. The improved performance can be attributed to more finely optimized Pentium code or to benefits from MMX technology enhancements.

**TABLE 6-2** Measurements for CSurfaceRdx Drawing

| Surface | CSprite 84 × 63 (16) | CBackground 734 × 475 | Refresh Screen 852 × 559 | Post Refresh | Total |
|---|---|---|---|---|---|
| CSysMem | 0.6–0.9 | 7.9 | 10.5 | 0.0 | 19.3 |
| CVidMem | 0.7–1.8 | 7.7 | 0.1 | 8.9 | 16.7/18.5 |
| CRdx | 1.3 | 4.3 | 11.7 | 0.0 | 17.3 |

All times are in milliseconds.

- Independently separating sprite, background, and screen refresh time when using RDX is not straightforward, since RDX always draws the entire surface. Sprite draw times were obtained by the following equation:

$$t_{spr} = t_{surf+sprites} - t_{surf}$$

and backgrounds were measured by

$$t_{bkg} = t_{surf+bkg} - t_{surf}$$

Refresh Screen still takes the most time. So how about asking RDX to use hardware acceleration if it's available?

## 6.5 Hardware Acceleration with RDX

With RDX we can apply effects on objects. An effect modifies the way that data is rendered, but it does not modify the original data. For example, we could render a sprite upside down by applying a vertical flip effect on it. RDX provides a variety of effects. Some of these effects can be applied on all objects; others can be applied only to specific objects. Refer to the RDX documentation for more detail on effects.

Let's start with asking RDX to set up the application in Full Screen mode and use Page Flip hardware if it's available.

### 6.5.1 Full Screen Mode with RDX

Full Screen mode is an effect that can be applied to surfaces. Here is some code to apply the Full Screen effect on our CSurfaceRdx.

```
class CSurfaceRdx : public CSurface
{
   // add these two member variables into class structure
   FULL_SCREEN_PARAMS  m_fxParams;
   HFX                 m_hFx;
}

CSurfaceRdx::MakeFullScreen()
{
   fxParams.dwWidth = 640;
   fxParams.dwHeight = 480;
   fxParams.iColorType = RGB_CLUT8;

   err = objApplyEffect(m_hObj, FX_FULL_SCREEN, &fxParams, &hFx);
}
```

Effect parameters are not copied but are used by reference. Therefore, effect parameters must not be declared in local (temporary) scope; they must be declared with lasting scope.

Specify the size and ColorType of the full screen window desired. If these are different from the current display mode, RDX will change the display mode to suit the parameters.

Apply the effect on the surface using the surface handle returned during *srfCreate()*. *ObjApplyEffect()* returns a handle to the newly "created" effect. Use this handle to manage the effect or to modify its parameters.

EXTRA CREDIT: Explore inserting color conversion effects on our RGB8 surface to handle non-RGB8 display modes.

## 6.5.2 How Fast Does CSurfaceRdx Draw in Full Screen Mode?

Table 6-3 on the next page compares CSurfaceRDX drawing in Full Screen mode with our previous measurements.

- Notice how the Refresh Screen and the Post Refresh Screen times are negligible for CRdxFullScreen. The CRdxFullScreen surface times are as good as the CBackBuffer surface times. Remember in Section 5.8.5 that CBackBuffer was set up to use Page Flipping. RDX automatically sets the system up to use Page Flipping, as soon as we request Full Screen mode. *We don't have to do anything special to turn on Page Flipping.*
- Also, notice how the variance is low for the Post Refresh Screen times for CRdxFullScreen. The variance for CRdxFullScreen is lower than the variance for CBackBuffer. *RDX automatically sets up for triple buffering, if the graphics card can support it*

The results are in! This is a simple interface with high performance.

## 6.5.3 Accelerating Objects with RDX

But what about accelerating objects like sprites and backgrounds?

**TABLE 6-3** CSurfaceRdx Drawing in Full Screen Mode

| Surface | CSprite 84 × 63 (16) | CBackground 734 × 475 | Refresh Screen 852 × 559 | Post Refresh | Total |
|---|---|---|---|---|---|
| CSysMem | 0.6–0.9 | 7.9 | 10.5 | 0.0 | 19.3 |
| CVidMem | 0.7–1.8 | 7.7 | 0.1 | 8.9 | 8.5/18.5 |
| CBackBuffer | 0.7–1.8 | 7.7 | 0.1 | 0.5–7.6 | 9.0–17.2 |
| CRdx | 1.3 | 4.3 | 11.7 | 0.0 | 17.3 |
| CRdxFullScreen | 1.7 | 7.3 | 0.0 | 0.0–4.0 | 9.0–13.0 |

All times are in milliseconds.

Well, it's really quite simple. RDX supports *sprSetFlags()* and *bkgSetFlags()* calls that turn on special features of these objects. Looking at the documentation for these calls, we find that both of these objects support a HWBLIT special feature (currently this is the only special feature supported).

Here's the code that turns on and off sprite acceleration.

```
// To HW-Blt a sprite
sprSetFlags (m_hObj, SPR_FLAG_HWBLIT) ;
```

You must describe source and destination (that is, *objSetData, objSetDestination*) before you use the `HWBLIT` special flag. Add this line after you have completely initialized the sprite.

```
// To turn off HW-Bltting
sprClearFlags (m_hObj, SPR_FLAG_HWBLIT);
```

Table 6-4 measures the speed of hardware-accelerated objects and shows them in comparison with other RDX objects.

**TABLE 6-4** Measuring Hardware-Accelerated RDX Objects

| Objects Drawn with RDX | Time for 16 sprites (84 × 63) | Time for Background 734 × 475 |
|---|---|---|
| Software objects in system memory | 0.9 | 4.3 |
| Software objects in video memory | 1.7 | 7.3 |
| Hardware objects in video memory | 2.3 | Out of Video Memory |

All times are in milliseconds.

Some observations based on the results:

- RDX's software-based spriting is actually faster than hardware-accelerated sprites. So with this configuration there is no real benefit to using HWBLIT.
- To use hardware acceleration you (or RDX) must place the source objects into video memory. Video memory is a scarce resource. After RDX set up the system for triple buffering, we did not have any memory left for our background.
- As we mentioned in the Introduction, timings are configuration dependent, and you may see different results on different configurations. With faster CPUs the software may be even faster. With faster hardware, the graphics card could be faster.

Video memory is a scarce resource, and in general you will get the best results by setting up for triple buffering before you accelerate individual objects. In the future, AGP-based graphics cards may offer Bltting from system memory, and then the scarcity of video memory will not be an issue. Although you still might not see any performance boosts with Transparent Blts, you will most probably see performance boosts with non-Transparent Blts—that is, your backgrounds will run faster. And when that time comes, you will be armed with the knowledge of how to accelerate your backgrounds with RDX.

## WHAT HAVE YOU LEARNED?

By this time, you've had an overview of RDX and gotten a taste of what it is like to use a high-level interface to manage multimedia objects. You've also gotten a feel for RDX's performance capabilities. You should have an idea of how the interface provides device independence and how to control RDX's usage of DirectDraw. In short, you should have a good starting point for using RDX and for deciding whether it will work for you.

We've come to the end of this part. Hope you had a pleasant trip.

WE'D LIKE
GRIFFIN FR
MICHAEL C
COMPCORE

PART III

# Making the Media Mix

WE'D LIKE TO EXTEND AN ACKNOWLEDGEMENT TO ROGER HURWITZ, TODD SCHWARTZ, TIMOTHY STRELCHUN, MARK LEAVY, BYRON GRIFFIN FROM INTEL; PAT BOYLE FROM REAL NETWORKS; DANNY MILLER, JASON WHITE AND RALPH LIPE FROM MICROSOFT CORP.; DR. MICHAEL CHWIALKOWSKI FROM THE UNIVERSITY OF TEXAS AT ARLINGTON; CHRIS EDDIE FROM XING TECHNOLOGIES, GEORGE HABER FROM COMPCORE MULTIMEDIA.

**Chapter 11** **Streaming Down the Superhighway with RealMedia**

- Real-time Internet streaming
- Data flows and data management interfaces
- File-Format plug-in and rendering plug-in
- RealMedia Audio Services

In the past few years, the PC has become powerful enough to handle both the capture and the playback of motion video under Windows. In the process, Microsoft has defined a few multimedia architectures on the Windows platform, including the Multimedia Command Interface (MCI), Video for Windows (VFW), and, lately, DirectShow (a.k.a. ActiveMovie). Apple, on the other hand, defined a multimedia architecture for both the Macintosh OS and Windows "QuickTime." Recently, with the explosion of the Internet, RealNetworks defined RealAudio, RealVideo, and, later, RealMedia.

In this part of the book we'll address a few of these multimedia architectures; namely, DirectShow from Microsoft, RealMedia from RealNetworks, and RDX from Intel. DirectShow is a streaming media architecture that supports multi-stream synchronization and MPEG-style video. The first release of DirectShow, known as ActiveMovie 1.0, lacked support for capture and compression. DirectShow, however, includes both capture and compression interfaces.

To understand the DirectShow architecture, it's best to first understand the filter graph model. To do that, you should first launch the graph editor application that comes with the DirectShow SDK and construct a filter graph. After doing so, you should be ready to delve into the details of the internals of filters—Chapter 8. You'll learn how to create a filter and pins and how to connect filters together. You'll also learn about how to add property pages to a filter, as well as custom interfaces.

You can then jump to Chapter 9, where you'll learn how to build filter graphs from an application using the DirectShow ActiveX control, the COM interface, or the GRF file. The ActiveX control is the easiest way to render a media file using the DirectShow filters. The ActiveX control provides all the necessary GUI interfaces to play, stop, and pause a media file. To have more control over the creation of a filter, you can use the COM interface or a GRF file to create and manipulate the filter. In this case, you have to provide the GUI interface and manage the events of the filters. Finally, in this chapter you'll learn how to expose a filter's property page and how to hook into a filter's custom interface.

Now, if you don't necessarily want to understand the internal architecture of DirectShow and its filters, or if you want to mix multiple video, audio, or animation objects together, you can use Intel's Realistic Display Mixer (RDX) to do that. RDX is a high-level interface that uses DirectShow to play and mix multiple video and audio objects.

Finally, since the Internet has been exerting a huge force on the computing environment, we thought it only appropriate to discuss one of the major architecture advancements for multimedia delivery on the Internet—the RealMedia Architecture (RMA). RMA is a modular extendible version of the RealAudio architecture. It uses a combination of a RealMedia server and client to deliver real-time multimedia content (audio, video, stock quotes, and so forth) over the Internet. With RMA you can stream any media type by adding a custom plug-in on both the server and the client sides.

To help you understand the RealMedia architecture, we will first focus on the topology of the architecture and then delve into the details of the plug-ins. To deliver custom data using RMA, you must first learn how to build File-Format and Rendering plug-ins. To play audio data on the client, you should use the RMA Audio Services, since it supports multiple platforms and performs the mixing of multiple audio streams. It also allows for pre- and post-processing of the audio streams.

the capture
t has defined
timedia Com-
a.k.a. Active-
th the
he Internet,

s; namely, Di-
l. DirectShow
n and MPEG-
d support for
compression

filter graph
comes with
be ready to
create a filter
add property

ıs from an ap-
RF file. The
w filters. The
ause a media
1 interface or
ide the GUI
learn how to
ace.

f DirectShow
cts together,
evel interface

environment,
ncements for
is a modular
RealMedia
quotes, and

# CHAPTER 7

# Video under Windows

**WHY READ THIS CHAPTER?**

This chapter gives a brief introduction to motion video and discusses the supporting architectures under Windows. It is meant to give background on the topics that are discussed in the rest of Part III.

If you feel comfortable with

- motion video on the personal computer,
- multimedia architectures under Windows (MCI, VFW, QTW, and ActiveMovie), and
- the principles of video compression and decompression,

you may wish to skip this chapter.

## 7.1 Concepts of Motion Video

I am sure you've watched a few cartoons in your life; my all-time favorite is *Bugs Bunny.* As you know, these cartoons, as well as real movies, are made up of a series of pictures displayed at a rate fast enough that it looks like motion video to the human eye. Throughout the world there are three dominant standards for television: NTSC, PAL, and SECAM. NTSC is primarily used in North America and specifies an interlaced refresh rate of 59.94 fields per second[1] (approximately 30 frames per second, fps). Both of the European standards, PAL and SECAM, specify an interlaced refresh rate of 50 fields per second (25 fps).

1. Interlaced display rate specifies the rate of displaying both the odd and even fields in a frame.

**MOTION VIDEO TERMS**

The NTSC television standard supports an *interlaced* video format of 30 frames per second. An *interlaced video frame* is composed of two separate *field* pictures, each of which has the same width as the full picture but is half the height of the entire picture: an even and an odd field. Both fields are displayed at 60 fields per second. The even field is displayed first starting at the top line and skipping every other line on the screen. The odd field is then displayed in the locations where the even field skipped.

In addition to the interlaced display, computer monitors support a *non-interlaced* video format, which basically displays the picture one line at a time on the screen (see Figure 7-1).

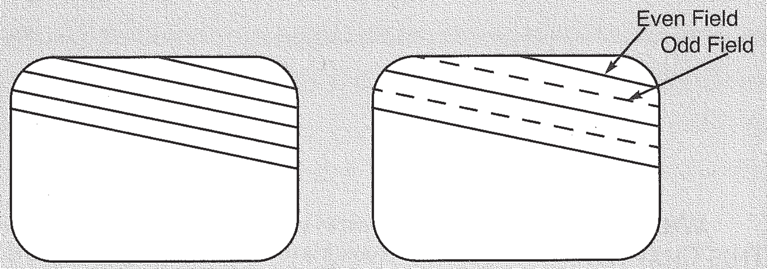


**FIGURE 7-1** Non-interlaced and interlaced video formats.

## 7.2 Capturing and Compressing Video

In the past few years, personal computers have become powerful enough to be able to play back motion video at the specified frame rates, and even faster. To play back a visual sequence on the PC, you must first digitize it with a video capture adapter and store the digitized clip on your hard disk or a CD-ROM. Typical video capture adapters can digitize an NTSC clip up to a 640 × 480 in size. If our memory serves us right, this results in a huge file if the video captured spans a few seconds or minutes. Let's calculate the amount of space required to store a video clip. To achieve the best quality, each pixel should contain a 24-bit RGB color quantity. (*RGB* stands for the red, green, and blue color format used in computers.)

$$\text{Size of 1 Frame} = 640\ (\text{width}) \times 480\ (\text{height}) \times 3\ \text{bytes/pixel} = 900\ \text{K}$$

To store 1 full second (30 frames) requires

$$\text{Size of a 1-second clip} = 900\ \text{K/frame} \times 30\ \text{frames/second} = 27\ \text{MB}$$

*27 MB/second!* Even the fastest CD-ROM today cannot sustain such a high data rate. For example, a 10× CD-ROM can only sustain, at best, a transfer rate of 1.5 MB/second. So that the video file will be usable, we must reduce the size of the video clip before storing it on a CD-ROM.

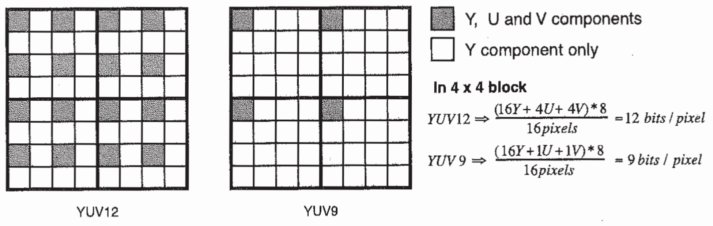
To enable motion video on such media, the digitized video clip must go through a few compression steps:

1. You could sacrifice some of the image quality by capturing a smaller size of the image (for example, 320 × 240 or 352 × 288). As a result, the data transfer rate is reduced by 75 percent to about 6.5 MB/sec for a 24-bit color clip.
2. Depending on the type of application, you could capture the video clip in either RGB or YUV color format. Typically, the YUV color space is more suitable for applications with motion video. The YUV color space contains one luminance (black-and-white) component and two chrominance (color) components (U and V). Naturally, there is a direct relationship between the RGB color space and the YUV color space.

**YUV AND RGB**

The RGB color space is used to represent colors on computer graphics adapters and terminals. In true color, with 8 bits for each color, you can generate up to $2^{24}$ distinct colors. The YUV color space is used to represent color in motion video. The YUV format represents the same format as that of the analog television signal, where there are one luminance component (black-and-white) and two chrominance (color) components. In this case, the Y is the luminance and the U and V are the chrominance. The YUV color scheme is used for motion video simply because it is easier to separate the black-and-white component of the picture from the color components, which makes it easier to compress the video clip.

The YUV color space is a good choice for motion video because it separates the black-and-white contents of the picture from its color components. This is very useful since the human eye cannot easily detect degradation in the color of an image, but it is extremely sensitive to any loss in luminance. Hence, the color information can be easily reduced without any noticeable degradation in video quality.



$$YUV12 \Rightarrow \frac{(16Y+4U+4V)*8}{16pixels} = 12\ bits\ /\ pixel$$

$$YUV9 \Rightarrow \frac{(16Y+1U+1V)*8}{16pixels} = 9\ bits\ /\ pixel$$

**FIGURE 7-2** UV color subsampling for motion video.

Figure 7-2 shows two UV subsampling formats, YUV12 and YUV9. On average, each pixel requires 12 bits to represent in the YUV12 format, and 9 bits in the YUV9 format. To reconstruct the color information for the "white" pixels, two or more neighboring U and V components are linearly interpolated to generate the appropriate color information.

Notice that the size of the final YUV12 image (Figure 7-2) is reduced by 50 percent since it only requires 12 bits to represent each pixel rather than 24. Similarly, the size of the final YUV9 image is reduced even further by 62.5 percent because it only requires 9 bits to represent each pixel.

3. The final YUV12 or YUV9 image is compressed even further using some of the well-known compression algorithms such as MPEG or Intel's Indeo. Even though these algorithms are lossy in nature, they can reduce the size of the image dramatically while maintaining superb image quality. Such algorithms can produce motion video clips suitable for a 1X CD-ROM at 150Kps.

As you can see, using these compression techniques allows you to smoothly integrate motion video with the PC. In 1997 a new breed of CD-ROMs and processors will allow for even better multimedia experience on the PC. DVD-ROM is a new CD-ROM media that can hold up to 17 GB of data on a single platter and can sustain up to 1.5MB per second. The Pentium II processor will be capable of playing back MPEG2 video clips as large as 720 × 576 at 25 to 30 fps.

## 7.3 Windows Multimedia Architectures

Microsoft's first attempt at multimedia came through the Multimedia Command Interface (MCI). MCI is a simple VCR-like interface with useful commands such as Play, Pause, Stop, Rewind, Seek, and so forth. In fact, MCI is an integral part of Windows and is still used by a certain class of applications such as audio CD players. But as a compromise to simplicity, MCI lacks many of the basic features required for multimedia recording and editing.

In 1993 Microsoft introduced Video for Windows (VFW) as an answer to the missing features in MCI. VFW defines interfaces for recording and editing both audio and video clips. As part of the standard, VFW also defined the Audio Video Interleaved (AVI) file format, which allows for interleaving multiple video, audio, or text streams in the same file. VFW also defined an interface for installable codecs to enable installation of custom compression and conversion algorithms. (Codecs are compression/decompression drivers.)

Even though Video for Windows was a great step for multimedia under Windows, it lacks some essential features. For example, even though VFW allows for multiple streams in an AVI file, it does not provide any means of synchronizing these streams together. In addition, VFW lacks the necessary features to support certain classes of algorithms such as MPEG video[2].

Around the same time, Apple moved its QuickTime architecture from the Macintosh environment to Windows and called it QuickTime for Windows (QTW). QTW only allowed for video playback in the Windows environment and did not allow for video capture or editing. All the video production remained on the Macintosh.

Back to Microsoft. To resolve some of the deficiencies in VFW, Microsoft introduced the first release of its latest multimedia architecture, ActiveMovie, at the end of 1996. ActiveMovie is targeted specifically for Windows 95 and Windows NT. The first release supports video and audio streaming and provides synchronization mechanisms between multiple streams. The first release, however, lacks capture and compression support.

As a follow-up to their commitment, Microsoft is releasing a follow-up technology, DirectShow, which is basically ActiveMovie with a name change and added support for capture and compression. We've dedicated the next three chapters to showing you how to use DirectShow.

Intel, on the other hand, released a graphics and video mixing architecture called Realistic Sound Experience (RDX). We've dedicated a chapter to showing you how easy it is to use RDX to mix multiple video and graphics objects.

Finally, RealNetworks is releasing their RealMedia Architecture, which allows for real-time streaming of video, audio, or any other media type over

2. VFW lacks support for future frame prediction techniques required by MPEG. Refer to the section "Overview of Video Codes" later in this chapter.

the Internet. We've dedicated one chapter to discussing the RealMedia plug-in architecture.

## 7.4 Overview of Video Codecs

Regardless of the multimedia architecture used, most video codecs apply similar methods to compress and decompress video. Let's have the ten-thousand-foot view of what a video codec does.

As we've mentioned earlier, video capture hardware produces an image composed of three color planes: Y, U, and V. Typically, the codec uses the same algorithm to compress each of the planes separately.

Typically, each plane is subdivided into 8 × 8 blocks, and each block is processed separately. The blocks are then transformed into a frequency domain using one of the well-known transformation processes (DCT, HAAR, SLANT, and so forth). The frequency domain block represents the amount of change in color from one pixel to the entire 8 × 8 pixel grid. Typically, video images don't change that drastically within an 8 × 8 block, and therefore, the high-frequency components of the frequency domain end up being mostly zeros. In fact, this is why the frequency domain is most suitable for video compression since consecutive zeros are easily represented with a small number of bits using the run length encoding (RLE) algorithm. Finally, the frequency domain block is quantized and encoded using the Huffman coding algorithm.

To decompress a frame, the exact opposite process is used. First the inverse Huffman algorithm is applied on the input bit stream generating 8 × 8 quantized blocks. These blocks are then dequantized using the same quantization matrix used when the frame was compressed. The Inverse frequency transformation is then applied on the dequantized block in order to produce the corresponding 8 × 8 Y, U, or V block. Finally, the Y, U, and V blocks are converted to RGB either by the application software or by specialized color conversion hardware on the graphics adapter (see Chapter 5, "Hardware Acceleration via DirectDraw").

The method that we've described so far is called *intra-frame* compression, and the frame is called the *I-frame* or *Key frame.* Intra-frames are compressed and decompressed independently from any other frames in the video sequence.

*Inter-frames*, on the other hand, can only be compressed or decompressed using data from other frames in the video sequence. Typically, at 25 or 30 fps, changes from one frame to the next are small enough that you can use the information from previous or future frames to predict the contents of the current frame. In fact, this type of a frame is called the *Predicted frame* or *P-frame*.

When compressing inter-frames, the difference between this frame and the reference frame is found first, then the difference information is transformed into the frequency domain. This technique is very useful in compressing motion video where some of the blocks end up being zero because the change between the two frames is insignificant.

The *Bi-directional frame* type is an extension of the P-frame. Here a previous and a future frame are used for reference at the same time. Typically, *B-frames*, as they're also called, produce higher compression than P- or I-frames, but they require more computational bandwidth and more memory to hold the reference frames.

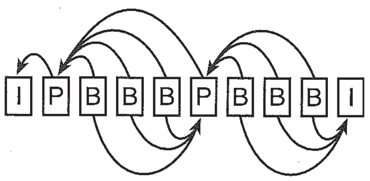


**FIGURE 7-3** MPEG frame types.

## WHAT HAVE YOU LEARNED?

After making your way through this chapter you probably have a good sense of

- motion video on the PC,
- multimedia architectures under Windows,
- video compression and decompression.

CHAPTER 8

# DirectShow Filters

**WHY READ THIS CHAPTER?**

You must have heard of or even tried the latest release of Microsoft's DirectShow (formerly known as ActiveMovie), but you're not sure what it has to offer. You're ready to move your current drivers from VFW or MCI to DirectShow, but you don't know where to start.

This chapter helps you

- understand the architecture of DirectShow and filter graphs,
- build source, transform, and rendering filters,
- understand the connection mechanism between filters,
- know how to use a registry file to add a filter to the registry or do filter self-registration,
- add custom interfaces to your filter, and
- add property pages to a filter.

To help you along the way, you can use the following articles on the CD:

- debugging hints for filters,
- adding a custom file type,
- how to build and run the sample files.

## 8.1 DirectShow Components

Figure 8-1 shows a high-level block diagram of the current multimedia architectures under Windows 95/NT. The DirectShow components are shown inside the dotted line.

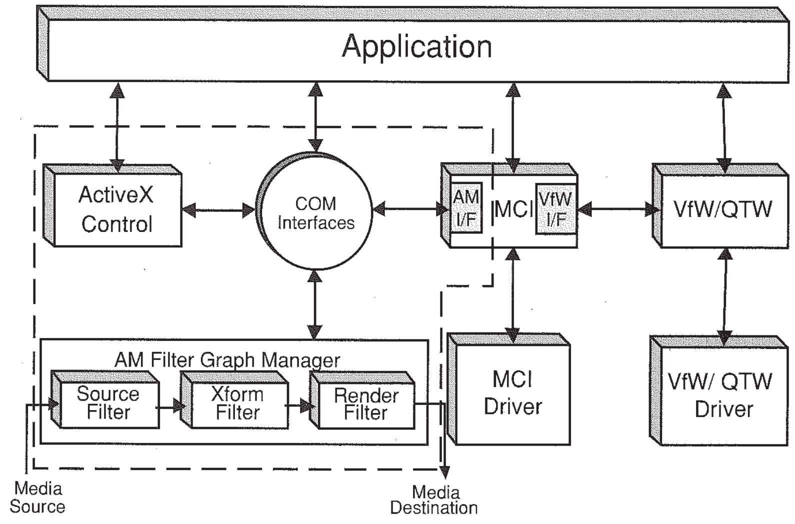


**FIGURE 8-1** Multimedia architectures under Windows (DirectShow components fall within dotted line).

As you can see, you can access the DirectShow components in one of three ways:

- directly through the COM interface and the Filter Graph Manager (FGM),
- using the MCI command set where the MCI layer has been updated to communicate with the DirectShow FGM, or
- through the ActiveX control interface, which is part of the DirectShow SDK. The ActiveX interface provides a high-level interface that gives applications a simple method for controlling DirectShow and its components. It also acts as an easy plug-in for the Internet Explorer.

We'll show you how to use the COM interface and the ActiveX control to access the DirectShow filter in the next chapter.

The Filter Graph Manager and the associated filters are the crux of DirectShow. The Filter Graph Manager provides applications with interfaces through the COM layer. (Applications cannot access the filters directly. They have to go through the FGM.) The FGM orchestrates the connection of the filters with the applications and the allocation of the shared buffers between them. It also controls the streaming of data and provides synchronization services (clock) so that filters can synchronize the delivery of multiple time-stamped data samples at the right time.

## 8.2 What's a Filter Graph?

Before we delve into the details of filters and filter graphs, it might be a good idea to go ahead and play with the Filter Graph Editor (FGE) applet that comes with the DirectShow SDK. The Filter Graph Editor is a tool that comes with the DirectShow SDK. Typically, applications will interact with the filter graph directly using the COM interface or the ActiveX control, discussed later.

Note

You can use the Filter Graph Editor to build a custom filter graph and save it in a *.*grf* file. The *.*grf* file can then be used to construct the exact same filter graph—without using DirectShow's automatic filter graph construction methods.

We're assuming that you've installed the FGE on your PC by now—NO? What are you waiting for? Once you've installed the SDK, launch the FGE applet and select the File->RenderMediaFile option from the menu. At the prompt, select the name of the sample MPEG file that comes with the DirectShow SDK (*Blastoff.Mpg*) and press OK. You should see something similar to what is shown in Figure 8-2.

Each individual rectangle in the figure represents a *filter* that performs a specific function. The arrows between the filters indicate that the output pin of one filter is connected to the input pin on the filter to its right. Notice that the media flows in the direction indicated by the arrows. The entire

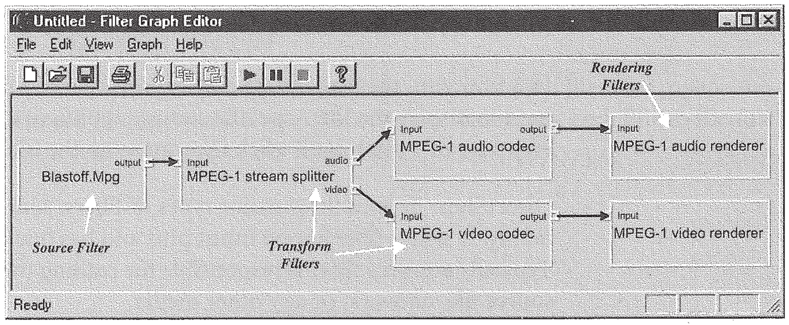


FIGURE 8-2 Filter graph for an MPEG-1 file.

mesh of connected filters is called the *filter graph*. You'll learn more about filters, pins, and the connection between them in the remainder of this chapter.

What's nice about the filter graph model is that you can easily replace one of the filters without even touching the remainder of the graph. For example, you can easily replace the source filter that reads the MPEG file from a hard disk or a CD-ROM with another source filter that reads it off the Internet or from a digital satellite link. This is a big win for developers since they only need to implement and distribute one filter rather than the entire filter graph or an entire VFW driver.

You can also insert other filters between any two filters and change the behavior of the filter graph—again, without touching any of the other filters. For example, you can insert a contrast filter in between the MPEG-1 video codec and the MPEG-1 video renderer. The contrast filter allows you to change the contrast of the video data on its way to the renderer. *Try it!*

## 8.3 Understanding the Mighty Filter

The filter is the basic building block of a DirectShow filter graph. A filter is basically a COM object with its own Global Unique Identifier (GUID). Typically, each filter comes with one or more input/output pins, which are used to move the data from one filter to the next. In order to connect two filters, the pins have to go through a simple process of negotiation.

At connection time, under the direction of the filter graph manager (FGM), the two pins negotiate on a media type that is common between them. Once the two pins agree on a media type, they negotiate on the allocation of the shared memory buffer used to transport the data between the two filters. Once the two pins settle their differences, they are joined in holy matrimony till death do them part.

In addition to the pins, filters may expose a set of *property pages,* which are used to display the filter-specific status or configuration. To see the property page of a filter, right-click the mouse on the filter and select Properties.

DirectShow defines three major types of filters: *source, transform,* and *rendering.* A *source filter* has no input pins and has one or more output pins. Typically a source filter is responsible for reading the raw data from a source file, network, or any other media.

8.4 A

8.5 C

A *transform filter* has one or more input pins and one or more output pins. Typically, a transform filter accepts data from an input pin or pins and converts it to another format before sending it out to the downstream filter.

A *rendering filter* has one input pin and no output pins. A rendering filter accepts data on the input pin and delivers it to its final destination (screen, audio card, file, and so forth).

## 8.4 An Overview of the Samples

Let's have an overview of the samples and explain what they do before we jump into the code. In this chapter, we'll show you how to create the three types of filters: source, transform, and rendering filters. To make it simple, we've chosen "simple text" as the media type to transport (see Figure 8-3).

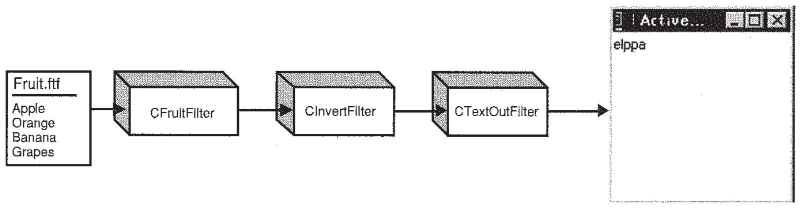


**FIGURE 8-3** Overview of the sample filters covered in this chapter.

The source filter CFruitFilter[1] reads one line at a time from the text file *Fruit.ftf* and passes it to the next filter. The CInvertFilter is a transform filter that accepts a string on the input pin and delivers an inverted string to its output pin. Finally, the CTextOutFilter is a rendering filter that displays the string presented at the input pin to a text window. This is a good time to run the sample application for this chapter on the companion CD.

## 8.5 Creating a Source Filter

Our source filter, CFruitFilter, prompts the user for a filename, opens the selected file, and reads it one line at a time. It delivers each line to the filter connected to its output pin for further processing.

1. We used the name *CFruitFilter* merely because each line in the input file is a name of a piece of fruit.

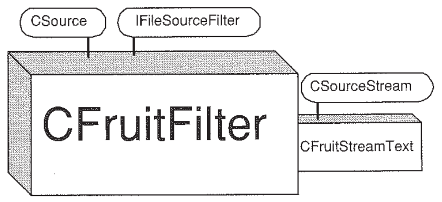


**FIGURE 8-4** Source filter.

For a better understanding of the code below, it might be useful to install the filter and see how it works before you go on. You can find detailed instructions on how to install and run this filter on the companion CD.

Notice that, before you can use any of these filters, you must first add them to the system registry. We'll show you how to do this at the end of the chapter (see "Section 8.10 Adding a Filter to the Registry").

It's actually pretty simple to create a source filter. DirectShow provides a couple of built-in classes that you can use to derive your source filter: The CSource and CSourceStream. CSource is the base class for all source filters. CSourceStream represents the output pin of a source filter. It handles moving the data from the file to the downstream filter. Notice in Figure 8-4 that our CFruitFilter also derives from the IFileSourceFilter, which is necessary to manage the filename of the input file. You'll see how this works later.

In our discussion, we'll first step through the CFruitFilter class where we'll show you how to create an instance of the filter, how to attach the source stream to it, and how to handle the IFileSourceFilter interface. We'll then step through the CFruitStreamText class, which handles the connection of the output pin, opening the input file, and transporting the data from the source file to the next filter down the stream.

### 8.5.1 The Source Filter Class

To create your own source filter, you need only derive a filter from the CSource base class, then override and implement a few of the base class member functions. As we've mentioned earlier, you can also derive a filter from the IFileSourceFilter to manage the input filename.

```
class CFruitFilter:
    public Csource,                    // Base source filter
    public IFileSourceFilter           // This is for accepting input file
{
```

Must be **static** since it is called before the class is created.

```
public :
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN lpunk, HRESULT *phr);

private:
```

The following lines are required for iFIleSourceFilter support.

```
    DECLARE_IUNKNOWN
    STDMETHODIMP GetCurFile(LPOLESTR * ppszFileName,AM_MEDIA_TYPE *pmt);
    STDMETHODIMP Load(LPCOLESTR pszFileName, const AM_MEDIA_TYPE *pmt);
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
```

Notice that the constructor is in the *private* section; therefore you can only create this object from within the only static function, *CreateInstance()*.

```
    CFruitFilter(LPUNKNOWN lpunk, HRESULT *phr);
    OLECHAR m_szFileName[_MAX_PATH];
};
```

## 8.5.2 Create an Instance of the Source Filter

Looking closely at the class declaration, you can see that the *CreateInstance()* function is the only public member of the class—even the constructor is private. As a result, you can only create an instance of the filter from within the *CreateInstance()* member function. In addition, notice that *CreateInstance()* must be declared as a *static* function so that it can be called even before the filter is created.

When the filter graph manager (FGM) loads a filter, it looks for the variables g_Templates[] and g_cTemplates in the executable file of the filter. The FGM uses these variables to figure out which objects exist and how to create them. For example, FGM uses the third element of the g_Templates to retrieve a pointer to the *CreateInstance()* function, which is called to create an instance of the filter. The function returns the address of the newly created instance.

```
CFactoryTemplate g_Templates[] = {
  { L"Fruit Source Filter"
  , &CLSID_FruitFilter
  , CFruitFilter::CreateInstance
  , NULL
  , NULL}
};
```

```
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);

CUnknown * WINAPI
CFruitFilter::CreateInstance(LPUNKNOWN lpunk, HRESULT *phr)
{
    // Create and return an instance of the filter
    CUnknown *punk = new CFruitFilter(lpunk, phr);
    if (punk == NULL) {
        *phr = E_OUTOFMEMORY;
    }
    return punk;
}
```

When you create a new instance of the filter, the CFruitFilter constructor is called. The constructor creates the streams supported by the filter and adds these streams' output pins to the m_paStreams member variable. DirectShow uses this list to keep track of the streams attached to the source filter. In our case, we only create the CFruitStreamText stream and add it to the m_paStreams list.

The sample source filters in the DirectShow SDK show the following code to create each of the source streams (pins) in the source filter constructor:

```
    m_paStreams    = (CSourceStream **) new CSourceStream[1];
    if (m_paStreams == NULL) {
        *phr = E_OUTOFMEMORY;
               return;
    }

    m_paStreams[0] = new CFruitStreamText(phr, this, L"Text!");
    if (m_paStreams[0] == NULL) {
       *phr = E_OUTOFMEMORY;
       return;
    }
```

However, we found that the source filter was leaking memory. After tracing through the CSource and CSourceStream classes, we found that both filters are properly handling the m_paStreams array. Therefore, we don't have to assign anything to the m_paStreams variable. So the above "erroneous" code should be replaced as shown in our example below.

```
CFruitFilter::CFruitFilter(LPUNKNOWN lpunk, HRESULT *phr) :
    CSource(NAME("Fruit Source Filter"),
    lpunk, CLSID_FruitFilter)
{
    CAutoLock cAutoLock(&m_cStateLock);
```

*NAME()* is used in debug builds for object tracing. See the DirectShow SDK for more details.

```
    // The CSourceStream constructor handles the allocation and assignment
    // of the m_paStreams[] array.  On return from the constructor, the
    // m_paStreams[0] would have the right value.
    //
    new CFruitStreamText(phr, this, L"Text!");
    if (m_paStreams[0] == NULL) {
        *phr = E_OUTOFMEMORY;
        return;
    }
}
```

This is the name of the output pin.

Once you've created the filter and its pins, the FGE interrogates the filter for the interfaces that it wants to use. As shown in the code below, the filter responds to the *IUnKnown::NonDelegatingQueryInterface()* member function to expose its own interfaces. In our case, the source filter supports all the interfaces of the base CSource class plus the IFileSourceFilter interface.

```
STDMETHODIMP
CFruitFilter::NonDelegatingQueryInterface(REFIID riid, void ** ppv)
{
    CheckPointer(ppv,E_POINTER);

    // We support the IFileSourceFilter interface and whatever
    // the base CSource supports..
    if (riid == IID_IFileSourceFilter)
        return GetInterface((IFileSourceFilter *) this, ppv);

    return CSource::NonDelegatingQueryInterface(riid, ppv);
}
```

Now that we've indicated that we support the IFileSourceFilter interface, the FGE prompts the user for the input filename and then calls the *Load()* function using that filename and the media type associated with that file. For example, an MPEG file is of MEDIATYPE_MPEGVideo type. Typically, the *Load()* function saves the filename and media type so that the filter can supply them when the *GetCurFile()* function is called. DirectShow or another application could request the active filename and media type anytime throughout the life of the filter.

```
STDMETHODIMP
CFruitFilter::Load(LPCOLESTR pszFileName,const AM_MEDIA_TYPE *pmt)
{
    lstrcpyW(m_szFileName, pszFileName);       // This is a UNICODE name
    return NOERROR;
}
```

```
STDMETHODIMP
CFruitFilter::GetCurFile(LPOLESTR * ppszFileName,AM_MEDIA_TYPE *pmt)
{
    CheckPointer(ppszFileName, E_POINTER);

    // Allocate an instance specific buffer to hold the filename.
    *ppszFileName = (LPOLESTR)
        CoTaskMemAlloc(sizeof(WCHAR) * (1+lstrlenW(m_szFileName)));
    if (*ppszFileName != NULL) {
        lstrcpyW(*ppszFileName, m_szFileName);
    }

    // we didn't save the media type, since we always return a NULL type.
    if(pmt) {
        ZeroMemory(pmt, sizeof(*pmt));
        pmt->majortype = MEDIATYPE_NULL;
        pmt->subtype = MEDIASUBTYPE_NULL;
    }
    return S_OK;
}
```

### 8.5.3 The Source Stream Class

As mentioned earlier, the CFruitStreamText class represents the output pin of the source filter (see Figure 8-4 on page 84). As a descendant of CSourceStream, the CFruitStreamText handles the connection process with the downstream filter, buffer allocation, and the movement of data from the input file to the downstream filter. In addition, CFruitStreamText is responsible for processing the Start, Stop, Pause, and other commands coming from the application through the filter graph manager.

```
class CFruitStreamText: public CSourceStream
{
public:
    CFruitStreamText(HRESULT *phr, CFruitFilter *pParent, LPCWSTR pPinName);
    ~CFruitStreamText();

    HRESULT FillBuffer(IMediaSample *pms) ;           ◊ Called to fill the buffer with data.

    HRESULT GetMediaType(int iPos, CMediaType *pmt)   ◊ Returns all media types supported.
    HRESULT CheckMediaType(const CMediaType *pmt);    ◊ Verifies if media type is acceptable.
    HRESULT SetMediaType(const CMediaType *pmt);      ◊ Accepts media type.
    HRESULT DecideBufferSize(IMemAllocator *pima,     ◊ Decides how big the buffer needs
            ALLOCATOR_PROPERTIES *pProperties);         to be for data movement.

    // Called when the stream is started and stopped
    HRESULT OnThreadCreate(void);
    HRESULT OnThreadDestroy();
};
```

### 8.5.4 The Connection Process

The filter graph manager starts the connection process by retrieving the output pin of one filter and trying to connect it to an input pin of another filter. In order to do that, the FGM calls upon the output pin of the source filter to connect to the input pin of the downstream filter. This is where the negotiation begins.

The output pin queries the input pin for a list of the media types that it supports—it repeatedly calls the input pin's *GetMediaType()* function to get the media type list. For each of these media types, the output pin calls its own *CheckMediaType()* function to see if it supports this media type. If the output pin can handle one of the media types, it returns `S_OK`, and the negotiation continues for the shared buffer; otherwise, *CheckMediaType()* returns an error.

```
HRESULT
CFruitStreamText::CheckMediaType(const CMediaType *pMediaType)
{
    CAutoLock cAutoLock(m_pFilter->pStateLock());

    if (*(pMediaType->Type()) != MEDIATYPE_Text)
        return E_INVALIDARG;
    return S_OK;
}
```

Of course, it is possible that the output pin could reject all the input pin media types. In that case, the output pin tries its preferred list of media types on the input pin. To do so, the output pin first calls its own *GetMediaType()* function to retrieve its own list of media types. Again, for each media type, the output pin calls the *CheckMediaType()* function of the input pin, of the downstream filter, to qualify that media type. In the case where the input pin rejects all the media types suggested by the output pin, the connection process is aborted and an error is returned to the application; otherwise, the negotiation continues for the shared memory buffer.

Notice that the default connection process tries the media types in the same order as *GetMediaType()* returns them. Therefore, the first media type returned by the function has the highest priority over any consequent media types. For example, if your filter supports RGB 8-, 15-, and 24-bit video but prefers the RGB24 format, then you should return the RGB24 format first.

```
HRESULT CFruitStreamText::GetMediaType(int iPosition, CMediaType *pmt)
{
    CAutoLock cAutoLock(m_pFilter->pStateLock());
    if (iPosition < 0)                                  ◊ Index must start with 0.
        return E_INVALIDARG ;

    if (iPosition > 0)                                  ◊ Only support 1 media type.
        return VFW_S_NO_MORE_ITEMS;

    pmt->SetType(&MEDIATYPE_JS97Text);                  ◊ Here it is, "Simple Text."
    return NOERROR;
}
```

Once the two pins agree on media types, the *SetMediaType()* is called to confirm that selection. Typically, the output pin saves the media type in order to use it later to calculate the size of the shared buffer. This is simply done by calling the corresponding function in the base class.

```
HRESULT CFruitStreamText::SetMediaType(const CMediaType *pMediaType)
{
    CAutoLock cAutoLock(m_pFilter->pStateLock());

    return CSourceStream::SetMediaType(pMediaType);
}
```

To allocate the shared buffer, the output pin determines the size of the shared buffer by calling its member function *DecideBufferSize()*. This function calculates the amount of memory required based on the media type and the header information of the input file (for example, the picture width and height). After determining what size the buffer should be, *DecideBufferSize()* calls the allocator function, *SetProperties()*, to verify that there is enough memory to allocate this buffer—the actual buffer is allocated later.

Typically, you don't have to worry about who allocates the buffer or when it gets allocated; you only have to assure that you calculate the size of the buffer correctly.

```
HRESULT
CFruitStreamText::DecideBufferSize(
    IMemAllocator *pAlloc,                      // Allocator object
    ALLOCATOR_PROPERTIES *pProperties           // Allocator properties
    )
```

```
{
    HRESULT hr = NOERROR;
    ALLOCATOR_PROPERTIES Actual;
    CAutoLock cAutoLock(m_pFilter->pStateLock());

    // Request the allocation of one buffer of size 1024 bytes
    pProperties->cBuffers = 1;
    pProperties->cbBuffer = 1024;

    hr = pAlloc->SetProperties(pProperties,&Actual);
    if (FAILED(hr)) {
        return hr;
    }

    // Verify that the allocator is able to allocate what we requested.
    if (Actual.cbBuffer < pProperties->cbBuffer) {
        return E_FAIL;
    }

    return NOERROR;
}
```

## 8.5.5 Starting and Stopping

All set and ready to roll—well, at least these two filters are. To complete the filter graph, the remaining filters down the stream follow the same negotiation process to connect their output pins with the appropriate input pins. Once the remaining filters are connected, the filter graph is ready to rumba.

At this stage, an application can *start* the filter graph, causing the source stream to read data from the file and send it downstream for further processing. When the filter graph is started, DirectShow creates a new thread for each CSourceStream in the filter graph, that is, a new thread for each output pin in the source filter. For example, if you have two output pins on the source filter, the FGM creates two additional threads to handle each of these pins. This allows the two pins to pump their data independent from one another.

When the thread is first created, the FGM calls the *OnThreadCreate()* function on the output pin, to initialize the state of the source stream—in our case, we open the source file. When you *stop* the filter graph, the FGM calls the *OnThreadDestroy()* function before it destroys the thread—in our case, we close the input file.

Notice that these functions are called every time the filter is started and stopped.

```
HRESULT CFruitStreamText::OnThreadCreate()
{
    CAutoLock cAutoLockShared(&m_cSharedState);

    // Convert file name from UNICODE to single byte char..
    char szTmp[256];
    WideCharToMultiByte(CP_ACP, 0,
            ((CFruitFilter*)m_pFilter)->m_szFileName, -1,
            szTmp, sizeof(szTmp), NULL, NULL);
    m_inFile.open(szTmp);

    return NOERROR;
}

HRESULT CFruitStreamText::OnThreadDestroy()
{
    m_inFile.close();
    return NOERROR;
}
```

### 8.5.6 Moving the Data

As long as the filter graph is running, each of the threads repeatedly calls the *FillBuffer()* function to fill the shared buffer with raw data from the input file. *FillBuffer()* then calls the *SetActualDataLength()* function in order to set the size of valid bytes in the shared buffer. The data buffer (media sample object) is automatically delivered to the downstream filter when the *FillBuffer()* function returns successfully.

```
HRESULT CFruitStreamText::FillBuffer(IMediaSample *pms)
{
    BYTE *pData;
    long lDataLen;

    pms->GetPointer(&pData);    // Retrieve a pointer to the buffer
    lDataLen = pms->GetSize();  // How big is the buffer - should be 1024

    // Read one line at a time till end of file..
    if (m_inFile.getline(pData, lDataLen))
        pms->SetActualDataLength(strlen((char*)pData)+1);
    else {
        return S_FALSE;
    }

    return S_OK;
}
```

When the input file runs out of data, *FillBuffer()* returns an error, `S_FALSE`, which marks the end of the stream, and, as a result, the filter graph stops, and, finally, the threads are terminated.

## 8.6 Creating a Transform Filter

Now, let's see how you can create a transform filter. As you recall, a *transform filter* accepts data from its input pin, applies some transformation on the data, and then sends it out to the filter connected to its output pin (see Figure 8-5). In our sample, the transform filter, CInvertFilter, inverts a text string before sending it out to the next downstream filter.

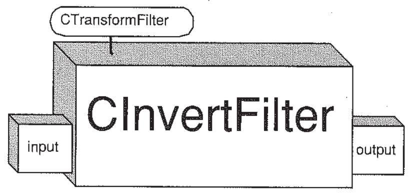


**FIGURE 8-5** Transform filter.

As with the source filter, you can easily create a transform filter by deriving it from the base CTransformFilter. As shown in Figure 8-5, the CTransformFilter defines one input pin and one output pin connected to an upstream and a downstream filter respectively. Notice that you can add additional pins to the transform filter; for example, the MPEG-1 stream splitter in Figure 8-2 has one input pin and two output pins.

```
class CInvertFilter : public CTransformFilter
{
public:
    // Input Pin override functions...
    HRESULT CheckInputType(const CMediaType* mtIn);
    HRESULT CheckTransform(CMediaType* pmtIn,CMediaType* pmtOut);
    HRESULT Receive(IMediaSample *pSample);

    CInvertFilter(TCHAR *pName, LPUNKNOWN pUnk, HRESULT *pHr);
    ~CInvertFilter();

    // Necessary COM functions...
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN, HRESULT *);
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
    DECLARE_IUNKNOWN;
    // Output pin datatype and buffer size functions
    HRESULT GetMediaType(int iPos, CMediaType *pmt);
    HRESULT CheckMediaType(const CMediaType *pmt);
    HRESULT SetMediaType(const CMediaType *pmt);
    HRESULT DecideBufferSize(IMemAllocator *pima, ALLOCATOR_PROPERTIES
            *pProperties);
};
```

Notice that the output pin of a transform filter overrides the same functions as the output pin of the source filter discussed in the previous section, namely *GetMediaType()*, *CheckMediaType()*, *SetMediaType()*, and *DecideBufferSize()*. Therefore, we're going to skip these functions and only discuss the new ones: *CheckInputType()*, *CheckTransform()*, and *Receive()*. Now, when the output pin of an upstream filter tries to connect to the input pin of a transform filter, the upstream filter verifies its media types against the transform filter's by calling the *CheckInputType()* function[2] of the transform filter. If the transform filter supports the media type, it returns `S_OK`; otherwise, it returns an error.

```
HRESULT CInvertFilter::CheckInputType(const CMediaType* pmtIn)
{
    DbgLog((LOG_TRACE, 2, TEXT("CInvertFilter::CheckInputType")));
    if ( (*pmtIn->Type() != MEDIATYPE_Text ) )
        return E_INVALIDARG;
    return S_OK;
}
```

*DbgLog()* is a useful debug macro. *See article on CD.*

Typically, the input pin of the transform filter is connected before its output pins. But it is possible for an input pin of a transform filter to connect to an upstream filter *after* one of the output pins has already established a connection with a downstream filter. In such a case the transform filter calls the *CheckTransform()* function to assure that the transform filter can convert the input media type to the output media type.

```
HRESULT CInvertFilter::CheckTransform(CMediaType* pmtIn,CMediaType* pmtOut)
{
    return S_OK;
}
```

2. Actually, the output pin of the upstream filter calls the *CheckMediaType()* of the input pin of the transform filter, which in turn calls our transform filter, which is a member of the CTransformFilter.

The transform filter is ready to run once both input and output pins are connected to their respective filters. When the filter graph is started, the upstream filter calls the *Receive()* function to deliver data to the transform filter. In our case, the transform filter inverts the string before delivering it to the downstream filter. The *Receive()* function accepts an IMediaSample as an input, which is the interface used to transport the data.

The *Receive()* function calls the *IMediaSample::GetPointer()* function to retrieve a pointer to the input buffer. It then calls the output pin's *GetDeliveryBuffer()* function in order to retrieve a pointer to the shared output buffer. The *Receive()* function inverts the input string and inserts it in the output buffer. Finally, it calls the output pin's *Deliver()* function in order to deliver the data to the downstream filter.

```
HRESULT CInvertFilter::Receive(IMediaSample *pSample)
{
    LPBYTE pData;
    HRESULT   hr;
    CAutoLock lck(&m_csReceive);

    // Get pointer to input buffer and size of valid data
    hr = pSample->GetPointer(&pData);
    int lDataLen = pSample->GetActualDataLength();

    if (FAILED(hr))
        return hr;

    // Get the output pin sample buffer
    IMediaSample *pOutSample;
    if ( FAILED(m_pOutput->GetDeliveryBuffer(&pOutSample, NULL, NULL, 0)) )
        return E_POINTER;

    LPBYTE pDst;
    if ( FAILED(pOutSample->GetPointer(&pDst)) )
        return E_POINTER;

    // Copy inverted string to output buffer
    CopyMemory(pDst, pData, lDataLen);
    strrev((LPTSTR)pDst);

    // deliver data to downstream filter
    pOutSample->SetActualDataLength(lDataLen);
    m_pOutput->Deliver(pOutSample);
    pOutSample->Release();
    return S_OK;
}
```

## 8.7 Creating a Rendering Filter

Finally, let's create a rendering filter. As you recall, a *rendering filter* supports one input pin and no output pins. It accepts data from an upstream filter and renders it to a dump file, the screen, an audio device, or the Internet. The rendering filter is the last stop for the data in the filter graph. In our sample renderer, the input pin accepts a text string and displays it to a text window on the screen (see Figure 8-3 on page 83).

DirectShow implements a base renderer, CBaseRenderer, which makes it easy to derive our text-rendering filter. Again, we'll discuss only the new functions that are relevant to the rendering filter: *CompleteConnect()*, *BreakConnect()*, *OnReceiveFirstSample()*, and *DoRenderSample()*.

```
class CTextOutFilter : public CBaseRenderer
{
    CTextOutWindow m_TextWindow;

public:
    HRESULT CompleteConnect(IPin *pReceivePin);
    HRESULT BreakConnect();
    void OnReceiveFirstSample(IMediaSample *pMediaSample);
    HRESULT DoRenderSample(IMediaSample *pMediaSample);

    CTextOutFilter(LPUNKNOWN pUnk,HRESULT *phr);
    ~CTextOutFilter();
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN pUnk, HRESULT *phr);
    STDMETHODIMP NonDelegatingQueryInterface(REFIID, void **);
    DECLARE_IUNKNOWN
    HRESULT CheckMediaType(const CMediaType *pmt);
};
```

Notice that in the code we've also defined the member variable m_TextWindow, which handles the output window. CTextOutWindow is based on the CBaseControlWindow class, which is part of the DirectShow class library. CBaseControlWindow simplifies the creating and handling of output to the client window. We're not going to discuss the CBaseControlWindow interface in detail here; we'll just initialize the m_TextWindow variable in the constructor of the filter and respond to the query of the interface as follows:

```
CTextOutFilter::CTextOutFilter(LPUNKNOWN pUnk,HRESULT *phr) :
    CBaseRenderer(CLSID_TextRender, NAME("TextOut Filter"), pUnk, phr),
    m_TextWindow(NAME("TextOut"), GetOwner(),phr, &m_InterfaceLock, this)
{
}

STDMETHODIMP
CTextOutFilter::NonDelegatingQueryInterface(REFIID riid,void **ppv)
```

```
{
    CheckPointer(ppv,E_POINTER);
    if (riid == IID_IVideoWindow) {
        return m_TextWindow.NonDelegatingQueryInterface(riid,ppv);
    }
    return CBaseRenderer::NonDelegatingQueryInterface(riid,ppv);
}
```

As with the transform filter, the input pin of the rendering filter connects to an upstream filter when both filters agree on the media type and the shared buffer size. Consequently, the *CompleteConnect()* function of the rendering filter is called to affirm the connection between the two pins. This is the last chance for the rendering filter to reject the connection between the two pins.

When the input pin is disconnected from the upstream filter, the *BreakConnect()* function is called, which typically hides and destroys the output window.

```
HRESULT CTextOutFilter::CompleteConnect(IPin *pReceivePin)
{
    // It's a good time to create the window
    return S_OK;
}

HRESULT CTextOutFilter::BreakConnect()
{
    m_TextWindow.InactivateWindow();
    m_TextWindow.DoShowWindow(SW_HIDE);
    return S_OK;
}
```

At this stage, the rendering filter is ready to run. The filter exposes two functions to handle the rendering of the data: *OnReceiveFirstSample()* and *DoRenderSample()*. The *OnReceiveFirstSample()* function is always called to render the first sample of data. Typically, this function handles the first sample of data that arrives after the Pause or Start commands are issued to the filter graph. In motion video, it is necessary to display the last video frame when the video clip is paused.

```
void CTextOutFilter::OnReceiveFirstSample(IMediaSample *pMediaSample)
{
    if(IsStreaming() == FALSE) {
        ASSERT(pMediaSample);
        DrawText(pMediaSample);
    }
}
```

PART III

The *DoRenderSample()* function is repeatedly called when the upstream filter delivers the samples to the rendering filter. This function handles rendering the data to the screen, file, audio device, or the Internet. Typically, you only need to implement this function; you don't have to worry about the *OnReceiveFirstSample()* function.

```
HRESULT CTextOutFilter::DoRenderSample(IMediaSample *pMediaSample)
{
    ASSERT(pMediaSample);
    DrawText(pMediaSample);
    return NOERROR;
}
```

## 8.8 Adding Your Own Interface

Now you know how to create a source filter, a transform filter, and a rendering filter. As you can see, DirectShow defines the necessary interfaces to build a filter graph and control the state of this graph. It allows you to start, stop, and pause the filter graph. All fine and dandy, but what if you have a cool feature that's not supported by one of the DirectShow interfaces? This is when you have to add your own interface. As it turns out, adding a new interface is easily supported by the COM paradigm.

Suppose you'd like to retrieve the statistics of the stream received by CTextOutFilter. For example, you'd like to figure out how many characters and how many words the renderer handles from start to stop. To do this, you must add your own interface to CTextOutFilter.

To create a custom interface, you must first create an interface template, declare its name and methods, and assign a unique GUID for it—in this case, we'll call it the ITextStat interface. A template only *declares* the member functions of the interface; it does not *define* or *implement* the body of these functions. The function bodies must be defined in the class that derives from this interface.

Use *GuidGen.Exe* to generate unique GUIDs.

```
DEFINE_GUID(IID_ITextStat,
0x48025244, 0x2d3a, 0x11ce, 0x87, 0x5d, 0x0, 0x60, 0x8c, 0xb7, 0x80, 0x66);

DECLARE_INTERFACE_(ITextStat, IUnknown )
{
    STDMETHOD(get_NumberOfChars) (THIS_  int *pNumChar) PURE;
    STDMETHOD(get_NumberOfWords) (THIS_  int *pNumWords) PURE;
};
```

The deriving class must implement all pure interfaces.

Next you need to include the ITextStat interface as one of the base classes of the CTextOutFilter class.

```
class CTextOutFilter :
    public CBaseRenderer,
    public ITextStat
{
    CTextOutWindow m_TextWindow;

public:
    CTextOutFilter(LPUNKNOWN pUnk,HRESULT *phr);
    ~CTextOutFilter();
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN pUnk, HRESULT *phr);
    STDMETHODIMP NonDelegatingQueryInterface(REFIID, void **);
    DECLARE_IUNKNOWN

    // These are the custom functions
    STDMETHODIMP get_NumberOfChars(int *pNumChar);
    STDMETHODIMP get_NumberOfWords(int *pNumWords);
    int m_nChars;
    int m_nWords;
};
```

In addition, you must implement all the functions of the ITextStat interface. Notice on the CD that the m_nChars and m_nWords fields are incremented in the *DoRenderSample()* function of the rendering filter (not shown here).

```
STDMETHODIMP CTextOutFilter::get_NumberOfChars(int *pChars)
{
    *pChars = m_nChars; // number of chars received so far.
    return NOERROR;
}

STDMETHODIMP CTextOutFilter::get_NumberOfWords(int *pWords)
{
    *pWords = m_nWords; // number of words received so far..
    return NOERROR;
}
```

Finally, you need to respond to the *NonDelagatingQueryInterface()* function of the filter in order to satisfy queries for the ITextStat interface.

```
STDMETHODIMP CTextOutFilter::NonDelegatingQueryInterface(REFIID riid,void **ppv)
{
     if (riid == IID_ITextStat) {
        return GetInterface((ITextStat *)this, ppv);
     else if (riid == IID_IVideoWindow) {
       return m_TextWindow.NonDelegatingQueryInterface(riid,ppv);
     }
     return CBaseRenderer::NonDelegatingQueryInterface(riid,ppv);
}
```

Well, now that we've arrived at this point, you're ready to use your custom interface in your application. You can access your custom interface by first calling the *QueryInterface()* function of the CTextOutFilter—specifying your custom GUID, IID_ITextStat, as the first parameter. *QueryInterface()* returns a pointer to the custom interface in the second parameter of the function. You can use that pointer to call the appropriate member function in the custom interface, for example, *Get_NumberOfChars(), Get_NumberOfWords()*.

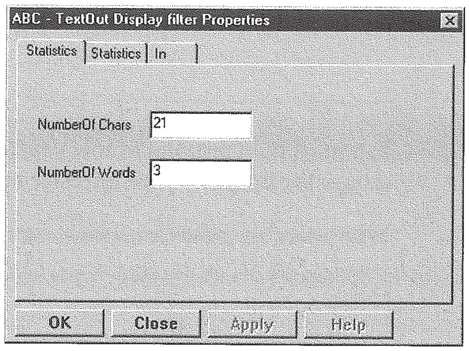
```
    ITextStat *pTextStat;
    hr = pUnknown->QueryInterface(IID_ITextStat,(void **)&pTextStat);
    if (FAILED(hr))
        return E_NOINTERFACE;

    m_pTextStat->get_NumberOfChars(&m_Chars);
    m_pTextStat->get_NumberOfWords(&m_Words);
```

## 8.9 Adding Property Pages to Filters

As we've mentioned earlier, a filter can expose one or more property pages that are specific to that filter. Typically, you would use a property page to display the status or configuration of your filter. You can access property pages either from the graph editor or from your application. We'll show you how to access property pages from an application in the following chapter.

To view the property pages for CTextOutFilter in the graph editor, right-click the mouse on the filter and select Properties. You should see something similar to what is shown in Figure 8-6.



NumberOf Chars 21

NumberOf Words 3

FIGURE 8-6 Property pages for CTextOutFilter.

Oooh, your fingers must be tingling at the thought of adding a property page to your own filter. It's actually pretty simple. To add one or more property pages to your filter you need to take the following steps:

1. Add the property page interface to the filter.
2. Implement the property page interface.

We break down these two steps in more detail in the following subsections.

## 8.9.1 Adding the Property Interface to the Filter

Actually, adding a property page is very similar to adding a custom interface. First, you must add the ISpecifyPropertyPage property interface as a base class to the CTextOutFilter declaration.

```
class CTextOutFilter :
    public CBaseRenderer,
    public ITextStat,
    public ISpecifyPropertyPages
{
   CTextOutWindow m_TextWindow;

public:
    CTextOutFilter(LPUNKNOWN pUnk,HRESULT *phr);
    ~CTextOutFilter();
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN pUnk, HRESULT *phr);
    STDMETHODIMP NonDelegatingQueryInterface(REFIID, void **);
    DECLARE_IUNKNOWN

    STDMETHODIMP get_NumberOfChars(int *pNumChar);
   STDMETHODIMP get_NumberOfWords(int *pNumWords);
    int m_nChars;
    int m_nWords;
    // required for ISpecifyPropertyPages
    STDMETHODIMP GetPages(CAUUID *pPages);
};
```

You must then respond to the *NonDelegatingQueryInterface()* in order to expose the property page interface to the application as follows:

```
STDMETHODIMP CTextOutFilter::NonDelegatingQueryInterface(REFIID riid,void
**ppv)
{
    if (riid == IID_ISpecifyPropertyPages) {
        return GetInterface((ISpecifyPropertyPages *) this, ppv);
    } else if (riid == IID_ITextStat) {
      return GetInterface((ITextStat *)this, ppv);
    } else if (riid == IID_IVideoWindow) {
         return m_TextWindow.NonDelegatingQueryInterface(riid,ppv);
    }
    return CBaseRenderer::NonDelegatingQueryInterface(riid,ppv);
}
```

Since the property page is actually a separate COM object, you must inform DirectShow of how to create an instance of that object. To do that, you must add the property page template to the factory template list, g_Templates[].

```
DEFINE_GUID(CLSID_TextOutPropertyPage,
0x48025243, 0x2d39, 0x11ce, 0x87, 0x5d, 0x0, 0x60, 0x8c, 0xb7, 0x80, 0x66);

CFactoryTemplate g_Templates[] = {
    { L"ABC - TextOut Display filter"
    , &CLSID_TextRender
    , CTextOutFilter::CreateInstance
    , NULL
    , &sudTextoutAx}
  ,
      { L"ABC - TextOut Property Page"
      , &CLSID_TextOutPropertyPage
      , CTextOutProperties::CreateInstance }
};
```

Finally, you must implement the *GetPages()* function of the filter. When an application displays the property page, DirectShow calls this function to retrieve the GUIDs of the property pages exposed by this filter. *GetPages()* returns a list of the property pages supported by this filter. In CTextOutFilter we're exposing the same property page twice just to show you how to support more than one property page in a filter.

```
STDMETHODIMP CTextOutFilter::GetPages(CAUUID *pPages)
{
    pPages->cElems = 2;

    // allocate enough memory to hold their GUIDs
    pPages->pElems = (GUID *) CoTaskMemAlloc(sizeof(GUID));

    if (pPages->pElems == NULL) {
        return E_OUTOFMEMORY;
    }
    pPages->pElems[0] = CLSID_TextOutPropertyPage; // 1st property page
    pPages->pElems[1] = CLSID_TextOutPropertyPage; // 2nd property page

    return NOERROR;
}
```

## 8.9.2 Implementing the Property Page Interface

To implement the property page, you must first derive an interface from the base CBasePropertyPage class and implement a couple of its member functions.

```
class CTextOutProperties : public CBasePropertyPage
{
public:
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN lpunk, HRESULT *phr);

private:
    CTextOutProperties(LPUNKNOWN lpunk, HRESULT *phr);
    HRESULT OnConnect(IUnknown *pUnknown);
    HRESULT OnDisconnect();
    HRESULT OnActivate();

    ITextStat *m_pTextStat;
    int m_Chars;
    int m_Words;
};
```

DirectShow then calls the *CreateInstance()* function, which creates an instance of the specified property page. Notice that the property page resource ID and name is specified when the base constructor is called.

```
CUnknown * WINAPI
CTextOutProperties::CreateInstance(LPUNKNOWN lpUnk, HRESULT *phr)
{
    return new CTextOutProperties(lpUnk, phr);
}
```

```
CTextOutProperties::CTextOutProperties(LPUNKNOWN pUnk,HRESULT *phr) :
CBasePropertyPage(NAME("TextOut Prop Page"), pUnk, IDD_PROPPAGE, IDS_NAME)
{
    ASSERT(phr);
}
```

IDD_PRPPAGE Property page resource ID.

Before the property page is displayed, DirectShow calls the *OnConnect()* function of the property page interface, using the address of the filter as a parameter. At this stage, you must retrieve any information that your property page needs from the filter. The property page retrieves the custom interface that we defined earlier, ITextStat, from CTextOutFilter in order to figure out the number of characters and words that the filter has processed already.

```
HRESULT CTextOutProperties::OnConnect(IUnknown *pUnknown)
{
    HRESULT hr;

    // get a pointer to the ITestStat interface..
    hr = pUnknown->QueryInterface(IID_ITextStat,(void **)&m_pTextStat);
    if (FAILED(hr))
        return E_NOINTERFACE;

    // get the statistics of #chars & #words from filter
    m_pTextStat->get_NumberOfChars(&m_Chars);
    m_pTextStat->get_NumberOfWords(&m_Words);
    return NOERROR;
}
```

When the property page is displayed, the *OnActivate()* function is called to update the fields of the property page.

```
HRESULT CTextOutProperties::OnActivate()
{
    TCHAR buf[50];

    wsprintf(buf,"%d", m_Chars);
    SendDlgItemMessage(m_Dlg, IDC_NumberChars, WM_SETTEXT,0, (DWORD) buf);
    wsprintf(buf,"%d", m_Words);
    SendDlgItemMessage(m_Dlg, IDC_NumberWords, WM_SETTEXT, 0, (DWORD) buf);

    return NOERROR;
}
```

Finally, when the property page is dismissed, the FGM calls the *OnDisconnect()* function in order to release any interfaces or memory.

```
HRESULT CTextOutProperties::OnDisconnect()
{
    if (m_pTextStat == NULL)
        return E_UNEXPECTED;
    m_pTextStat->Release();
    m_pTextStat = NULL;
    return NOERROR;
}
```

## 8.10 Adding a Filter to the Registry

DirectShow uses the system registry to hold its configuration information, filter list, and media types. When you create your own filter, you must add it to the appropriate part of the registry so that DirectShow can recognize and load your filter. Here is a list of the registry keys used by DirectShow with a brief description of each key:

| | |
|---|---|
| \\ Hkey_Class_Root **\Filter** | DirectShow looks here for a list of filter IDs (GUIDs). |
| \\ Hkey_Class_Root **\CLSID** | This is where all COM objects live. Holds the settings for each GUID (for example, the filename of executable). DirectShow looks up filter GUIDs here to get information about the filter. |
| \\Hkey_Class_Root **\Media Type** | List of media types (for example, MPEG1Stream) and the associated source filter that can handle this media type. This is used for automatic rendering of source files. You can find more information on the CD under "Adding Custom File Types." |
| \\Hkey_Local_Machine **\Software\Debug** | This area holds useful debug configuration for each filter. You can find more information on the CD under "DirectShow Debugging Hints." |

You can easily add the necessary entries for your filter in the registry in one of two ways. You can build a registry file with all the necessary entries and add it to the registry with the RegEdit.Exe. Or you can embed the registry information in the filter and use the RegSvr32.Exe command to add the information to the registry.

### 8.10.1 Using a Registry File Is Not Recommended

Windows registry editor, RegEdit.Exe, supports a command line option, *-s*, which allows you to specify a registry file in order to add information to the registry. Here is the registry file for CTextOutFilter:

```
; FileName: TextOut.Reg
[HKEY_CLASSES_ROOT\Filter\{CC01B761-A537-11d0-9C71-00AA0058A735}]

@="ABC - Text Display Filter"

[HKEY_CLASSES_ROOT\Clsid\{CC01B761-A537-11d0-9C71-00AA0058A735}]
@="Text Display Filter"
"Merit"=dword:00800000

[HKEY_CLASSES_ROOT\Clsid\{CC01B761-A537-11d0-9C71-00AA0058A735}\InprocServer32]
@="c:\\filter\\textout.ax"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT\Clsid\{CC01B761-A537-11d0-9C71-00AA0058A735}\Pins]
[HKEY_CLASSES_ROOT\Clsid\{CC01B761-A537-11d0-9C71-00AA0058A735}\Pins\TextOut]
"AllowedMany"=dword:00000000
"AllowedZero"=dword:00000000
"Direction"=dword:00000000
"IsRendered"=dword:00000001

[HKEY_CLASSES_ROOT\Clsid\{CC01B761-A537-11d0-9C71-00AA0058A735}\Pins\TextOut\Types]
[HKEY_CLASSES_ROOT\Clsid\{CC01B761-A537-11d0-9C71-00AA0058A735}\Pins\TextOut\Types\
{73747874-0000-0010-8000-00AA00389B71}]
[HKEY_CLASSES_ROOT\Clsid\{CC01B761-A537-11d0-9C71-00AA0058A735}\Pins\TextOut\Types\
{73747874-0000-0010-8000-00AA00389B71}\{00000000-0000-0000-0000-000000000000}]
```

◊ Register filter GUID and name with DirectShow.

◊ Add filter GUID to CLSID section.

◊ Path and filename of filter.

◊ Supported pins and their properties.

◊ Major and minor media types that the output pin supports.

To add the information in the file to the registry, run:

```
RegEdit.Exe -s TextOut.Reg
```

The problem with this method is that the information in the registry file is static and may not reflect the current state of the filter. For example, the path to the filter is hard coded and must be manually updated if the path changes.

### 8.10.2 Using Filter Self-Registration Is Recommended

DirectShow supports the COM self-registration procedure, which allows a filter to automatically add its information to the registry. To use self-registration, you must embed the information in the filter and then run the RegSvr32.Exe command. This command retrieves the embedded information from the filter and adds it to the system registry.

First you must add the DirectShow setup information, *sudTextOutAx,* in the factory template *m_gTemplates[].* The DirectShow setup information allows you to specify the filter name, the number of pins, and the supported media types for each pin.

```
const AMOVIESETUP_MEDIATYPE sudIpPinTypes ={
    &MEDIATYPE_Text,                        ◊ Major and minor media types supported
    &MEDIASUBTYPE_NULL                        by this pin.
};
                                              Pin information array. If there is more
const AMOVIESETUP_PIN sudIpPins [] = {    ◊ than one pin, just add its information here.
  { L"TextOut",               ◊ Pin name.
    FALSE,                    ◊ Does the pin render the data it receives?
    FALSE,                    ◊ Is it an output pin?
    FALSE,                    ◊ Is filter allowed to have zero pins of this type?
    FALSE,                    ◊ Does the filter have more than one instance of this pin?
    &CLSID_NULL,              ◊ The pin connects to the pin with this CLSID.
    NULL,                     ◊ The pin connects to the pin with this name.
    1,                        ◊ Number of supported media types.
    &sudIpPinTypes }          ◊ Address of media type list.
};

                                              Filter information that is inserted in the
const AMOVIESETUP_FILTER sudTextoutAx = { ◊ registry.
    &CLSID_PlainText,                 ◊ GUID of this filter.
    L"ABC Text Display Filter",       ◊ Filter name.
    MERIT_NORMAL,                     ◊ Filter merit. This is used for automatic connection
    1,                                ◊ Number of pins.
    &sudIpPins                        ◊ Address of list of pins.
};

CFactoryTemplate g_Templates[] = {
  {  L"ABC - TextOut Display filter"
    , &CLSID_PlainText
    , CTextOutFilter::CreateInstance
    , NULL
    , &sudTextoutAx }         ◊ Pointer to filter self-registration information (optional).
};
```

Finally, you must implement and export two functions: *DllRegisterServer()* and *DllUnregisterServer()*. *DllRegisterServer()* is called to add the filter information to the registry, and *DllUnregisterServer()* is called to remove that information from the registry. Both functions call the appropriate DirectShow function to do the actual registration and de-registration. Of course, you can add your own code here to add information to or remove information from the registry.

```
STDAPI DllRegisterServer()
{
    return AMovieDllRegisterServer2( TRUE );
}

STDAPI DllUnregisterServer()
{
    return AMovieDllRegisterServer2( FALSE );
}
```

You can export the functions in the Definition (DEF) file as follows:

```
EXPORTS
        DllRegisterServer
        DllUnregisterServer
```

Once you've built the filter successfully, you can register the filter by running:

```
RegSvr32.Exe TextOut.Ax
```

(*TextOut.Ax* is the filter filename), and you can unregister the filter by running:

```
RegSvr32.Exe -u TextOut.Ax
```

## What Have You Learned?

By the end of this chapter, you should

- have an understanding of the filter graph model of DirectShow;
- understand the connection between input and output pins;
- know how to create your own source, transform, and rendering filters and know the difference between them;
- be able to add custom interfaces to your filter;
- know how to add property pages to a filter; and
- understand how to either create a registry file or embed the registry information in the filter.

In the following chapter, we'll show you how to access these filters from the application point of view. After all, you wrote these filters for a reason.

# CHAPTER 9

# DirectShow Applications

**WHY READ THIS CHAPTER?**

Now that you've created your own DirectShow filters, you probably want to use them within your application. What a coincidence! This is exactly what we're covering in this chapter.

We'll show you how to access DirectShow filters using your application by one of two methods: direct low-level access using the COM interfaces, and high-level access using the DirectShow ActiveX control.

For the COM interface, we'll show you how to

- use the automatic method for building filter graphs,
- use a preprepared filter graph file (**.grf*) for building filter graphs,
- use a manual method for building filter graphs,
- control the state of the filter graph (Start/Stop/Pause),
- access custom interfaces within a specific filter,
- display filter-specific property pages from within your application, and
- handle events posted by the filters and the filter graph manager.

As for the ActiveX interface, we'll show you how to

- add the DirectShow ActiveX control to your application and then access it,
- control the state of the filter graph (Start/Stop/Pause), and
- handle events posted by ActiveX control.

## 9.1 DirectShow Mechanisms for Working on Filter Graphs

In the previous chapter, you learned how to build your own DirectShow filters—source, transform, and rendering. You also learned how to add property pages and custom interfaces to your filters. To test your filter, you used the *Graph Editor* graphical applet, which allows you to add individual filters and connect their pins together. The applet also allows you to run the filter graph, save the filter graph into a **.grf* file for later use, and display that slick property page that you embedded in your filter.

In this chapter, you'll learn how to manipulate filters in the same manner as you did using the graph editor. As you recall from the previous chapter, DirectShow provides three ways to access the filter graph (see Figure 9-1):

- directly through the Filter Graph Manager (FGM) COM interfaces;
- through the DirectShow ActiveX control, which is part of the DirectShow SDK; and
- indirectly through the MCI interface.

In this chapter, we'll only discuss the first two methods: the COM interface and the ActiveX control.

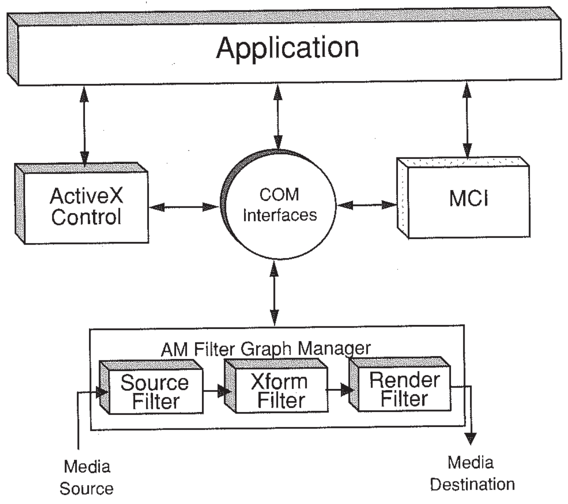


**FIGURE 9-1** Interfaces that provide applications access to the DirectShow filter graph.

DirectShow provides various levels of application support. At the high level, ActiveX, you only need to embed the ActiveX control in your application, and ActiveX control will handle the rest. It opens the media file, builds the filter graph, and even provides the user interface for controlling the filter graph (Run, Stop, Pause, and so forth).

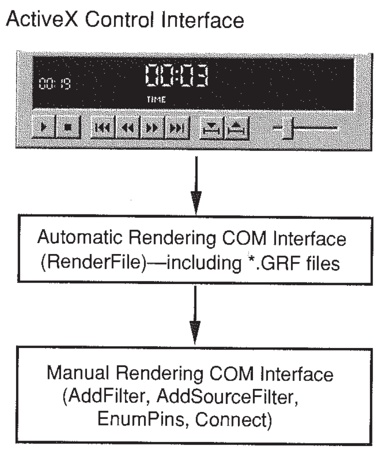

ActiveX Control Interface
00:03
TIME
Automatic Rendering COM Interface (RenderFile)—including *.GRF files
Manual Rendering COM Interface (AddFilter, AddSourceFilter, EnumPins, Connect)

**FIGURE 9-2** Various levels of DirectShow application interfaces.

For mid-level control, DirectShow offers an automatic rendering method using its COM interfaces. At this level, you can construct the entire filter graph with a simple function call, *IFilterGraph::RenderFile()*, and DirectShow determines the appropriate filters to load and how to connect their pins. You can also use the *RenderFile()* function to re-create a preconfigured filter graph from a GRF file—GRF files can be created with the Filter Graph Editor (FGE). When you use the mid-level method, you have direct access to your custom interfaces and property pages. In addition, you can receive and handle filter graph notification events.

At the lowest level, DirectShow offers a wide range of manual rendering COM interfaces that allow you to have total control over your filters. You can manually load each filter and connect its pins together. You can use this low-level method when you want to bypass DirectShow automatic rendering techniques and assure that you are loading the correct filters.

## 9.2 COM: Automatic Construction of Filter Graphs

Let's start at this level, in the middle, because it offers the best of both worlds. At this level, you can construct a filter graph easily and still maintain full access to the filters and their events.

IGraphBuilder functions: *AddFilter(), AddSourceFilter(), RemoveFilter(), EnumFilters(), FindFilterByName(), Connect(), Render(), RenderFile(), SetLogFile(), and more*

DirectShow defines the graph builder interface, IGraphBuilder, which allows you to build filter graphs from within your application. To do this, the graph builder interface uses the settings in the system registry to determine the appropriate filters to load and connect in order to construct the filter graph.[1]

Before you create the graph builder object, you must first initialize the COM libraries with the *CoInitialize()* function. You can then call the *CoCreateInstance()* function to create an instance of the IGraphBuilder interface. The function looks up the CLSID_FilterGraph in the system registry and loads the appropriate Dynamic Link Library (DLL) associated with this class ID. It creates an instance of the IGraphBuilder object and returns it in the `m_pGraph` parameter.

```
if(FAILED(CoInitialize())                  // initialize the COM intreface
    return FALSE;

// Create a Graph Builder object
IGraphBuilder * m_pGraph;
HRESULT hr = CoCreateInstance(
    &CLSID_FilterGraph,                    ◊ CLSID for DirectShow graph builder (Quartz.Dll)
    NULL ,                                 ◊ Created standalone—not aggregated
    CLSCTX_INPROC_SERVER,                  ◊ Created within the same process space of the app
    &IID_IGraphBuilder,                    ◊ GUID of requested interface
    &m_pGraph);                            ◊ Holds a pointer to IGraphBuilder interface on
                                             return
```

Once you've obtained an instance of the filter graph builder, you can call its *RenderFile()* function to create the entire filter graph. This function accepts the input filename as a parameter.

```
if (FAILED(hr) || FAILED( m_pGraph->RenderFile(wPath, NULL) ))
     return FALSE;
```

1. You can find out more about the system registry setting on the CD under "*Adding Your Own Custom File Format.*"

The *RenderFile()* function first determines which source filter to load based on the contents of the input file. Notice that DirectShow does not use the file extension (**.mpg*, **.avi*, and so forth) to determine the type of media in the file; rather, it searches the input file for certain byte values residing at certain byte locations, and, based on what it finds, it determines what type of media the file represents. The source filter that handles that type of file typically knows those byte locations and their values. For more information, refer to "Adding Your Own Custom File Format" on the CD.

**BUILDING A LIST OF FIGURES**

The following code segment illustrates how to retrieve the list of filters in the system registry. You can use the IAMCollection interface and the *IMediaControl::get_RegFilterCollection()* to enumerate the filters in the system registry as follows:

```
BOOL CRegFiltersDlg::OnInitDialog()
{
     IAMCollection *pMCollection;
     char szTmp[256];
     LONG lCount;

     // Get the collection of filters from the registry and find
     // how many filters is there..
     m_pMC->get_RegFilterCollection((IDispatch**)&pMCollection);
     pMCollection->get_Count(&lCount);

     IRegFilterInfo *pInfo;
     BSTR pTmp;

     // For each filter, add its name the list box…
     for (int i=0; i<lCount; i++) {
               pMCollection->Item(i, (IUnknown**)(&pInfo));
               pInfo->get_Name(&pTmp);
               WideCharToMultiByte(CP_ACP, 0, pTmp, -1, szTmp,
                sizeof(szTmp), NULL, NULL);
               m_FilterList.AddString(szTmp);
               pInfo->Release();
     }
     pMCollection->Release();
     return TRUE;
}
```

You can get a similar list of loaded filters in the filter graph. All you have to do is replace the *get_RegFilterCollection()* with the *get_FilterCollection()* and the IRegFilterInfo with the IFilterInfo.

Once *RenderFile()* determines and loads the source filter, it enumerates the filter pins and determines the media types that they support. For each of the media types, the graph builder searches the system registry for a filter that accepts that media type as an input. If such a filter is found, *Render-File()* loads that filter and connects its pin to the output pin of the source filter. From there on, the same process is repeated for each output pin until the entire filter graph is assembled.

IMediaControl functions: *Run(), Pause(), Stop(), StopWhenReady(), GetState(), RenderFile(), AddSourceFilter(), get_FilterCollection(), and more*

Now that the filter graph is built, you can start, pause, or stop the filter graph at will. To do that, you must first obtain a pointer to the media control interface, IMediaControl, of the filter graph. You can simply call the *IGraph-Builder::QueryInterface()* function to retrieve a pointer to the media control interface using `IID_IMEDIACONTROL` as a parameter. The *QueryInterface()* function returns the media control interface in the second parameter, `&m_pMC`, which you can use to call the appropriate function to run, stop, or pause the filter graph.

```
// Obtain the interface to our filter graph
    if(FAILED(m_pGraph->QueryInterface(IID_IMediaControl,(void**) &m_pMC)))
    return FALSE;

    hr = m_pMC->Run();
    hr = m_pMC->Stop();
    hr = m_pMC->Pause();

    m_pMC->Release();
```

Make sure to Release the interface when you're done with it.

Notice that the Stop command exposed by the media control interface stops the video clip at the last key frame, or intra-frame.[2] However, you could choose to modify the behavior of the Stop command so that after the video stopped the video clip would always rewind to the beginning of the sequence.

IMediaPosition functions: *get_Duration(), get_CurrentPosition(), put_CurrentPosition(), get_StopTime(), put_StopTime(), get_PrerollTime(), set_PreRollTime(), get_Rate(), put_Rate(), and more*

To do that, you must first acquire a pointer to the media position interface, IMediaPosition, which manages the position of the stream. As with the media control, you must first call the *IGraphBuilder::QueryInterface()* to retrieve a pointer to the media position interface. You can then use that pointer to call the *IMediaPosition::put_CurrentPosition()* function and set the position of the stream to zero.

2. A key or intra-frame is a frame that can be decoded independent of any other frame in the sequence.

```
hr = m_pMC->Pause();        // Ask the filter graph to pause

// Rewind the stream to the beginning when the user issues the STOP
// command or when the EC_COMPLETE event is received.
IMediaPosition * pMP;
hr = m_pGraph->QueryInterface(IID_IMediaPosition, (void**) &pMP);

if (SUCCEEDED(hr)) {
    pMP->put_CurrentPosition(0);
    pMP->Release();
}

// wait for pause to complete
OAFilterState state;
m_pMC->GetState(INFINITE, &state);

// now really do the stop
m_pMC->Stop();
```

*IMediaControl::Pause()* is asynchronous and returns immediately before it completely pauses the clip. The *GetState()* function blocks until the filter is completely paused.

## 9.3 COM: Manual Construction of Filter Graphs

That was surprisingly easy! So why would you want to create a filter graph manually anyway? We're sure you have your own reason for doing this, but it usually boils down to this: you don't want to rely on the setting of the registry, and you want to guarantee that your filter is always loaded. After all, you spent your heart and soul writing it.

### 9.3.1 Adding Filters to the Filter Graph

DirectShow defines two methods for adding filters to the filter graph: one for adding source filters and the other for adding transform and rendering filters. To add a source filter, you can call either the *IMediaControl::AddSourceFilter()* function or the *IGraphBuilder::AddSourceFilter()* function. To add a transform or rendering filter, you can call either the *IGraphBuilder::AddFilter()* function or the *IRegFilterInfor::Filter()* function.

For simplicity's sake, we'll use CFruitFilter and the CTextOut filter from the example of Chapter 8 to demonstrate how to construct a filter graph manually. First we'll show you how to add the CFruitFilter source filter and then how to add the CTextOut rendering filter. We'll then enumerate their pins and connect the two filters together.

## 9.3.2 Adding Source Filters

The DirectShow SDK only describes how to use the *AddSourceFilter()* function to add a source filter to a filter graph. But this function behaves in a similar manner to the *RenderFile()* function in that it uses the system registry to determine the source filter associated with an input file. Unlike the *RenderFile()* function, the *AddSourceFilter()* only loads the source filter; it does not build the entire graph. As a result, you have full control when it comes to loading the remaining filters in the graph. (Pssst . . . Do you really want to have full control over loading your source filter? Read on below.)

OK, here is how you would use the *IFilterGraph::AddSourceFilter()* function to add a source filter to the filter graph. It's pretty simple!

```
// Load the source file associated with the Fruit.FTF file
WCHAR wFileName[] = L"Fruit.FTF";
m_pGraph->AddSourceFilter(wFileName, wFileName, &pSrcFilter);
```

The *Fruit.ftf* file has the string FRUIT TEXT at the beginning of the file (the *ftf* file extension stands for "fruit text file"). The registry associates any file starting with that string with CFruitFilter. See "Adding Your Custom File Types" on the CD.

What actually happens when you call the *AddSourceFilter()* function? DirectShow associates the specified filename with a source filter through the mapping specified in the registry. In this case, any file that starts with "FRUIT TEXT" is associated with CFruitFilter. In turn, the *AddSource-Filter()* function loads CFruitFilter into the filter graph. DirectShow then queries the filter for the IFileSourceFilter interface. The IFileSourceFilter interface is used to pass the filename and media type information to/from the filter. If the interface exists, DirectShow calls the *IFileSourceFilter::Load()* function, using the input filename and media type as parameters.

You could ask the question, "Well, can I mimic this behavior if I don't want to rely on the registry mapping to load my source filter?" The answer is, "Yes you can."

To do this, you must first find your source filter in the registry and then add it to the filter graph (see our *LoadFilter()* function below). To find the source filter in the registry, you must call the *IMediaControl::get_RegFilter-Collection()* function to enumerate all the DirectShow filters in the registry. The function returns an IAMCollection interface, which you can use to retrieve information about each of the filters, IRegFilterInfo. You can then use the *IRegFilterInfo::get_Name()* function to retrieve the filter name and match it against the name of your filter.

Once you have found your source filter, you can call the *IRegFilterInfo::Filter()* function to load the filter into the filter graph. Notice that the *Filter()* function returns a pointer to an IFilterInfo interface rather than the filter itself; you should call the *IFilterInfo::get_Filter()* function to retrieve a pointer to the newly added filter.

```
HRESULT
CCustomFilterGraph::LoadFilter (
    WCHAR *pszName,            // Filter Name to match against registry
    IBaseFilter **pFilter)     // on return holds a pointer to added filter
{
    BSTR pTmp;
    LONG lCount;
    HRESULT hr;
    IAMCollection *pMCollection=NULL;
    IRegFilterInfo *pRegInfo=NULL;
    IFilterInfo *pFilterInfo=NULL;

    // Get a list of all registered DirectShow filters
    m_pMC->get_RegFilterCollection((IDispatch**)&pMCollection);
    pMCollection->get_Count(&lCount);

    // For each filter, find out if its name matches the name of our filter
    for (int i=0; i<lCount; i++) {
        RetOnErr( pMCollection->Item(i, (IUnknown**)(&pRegInfo)) );
        pRegInfo->get_Name(&pTmp);

        // Once found, add filter to the graph and get a pointer to it.
        if (lstrcmpW(pszName, pTmp) == 0) {
            pRegInfo->Filter((IDispatch**)&pFilterInfo);
            pFilterInfo->get_Filter((IUnknown**)pFilter);
            break;
        }

        pRegInfo->Release();
        pRegInfo=NULL;
    }

    pMCollection->Release();
    pRegInfo->Release();
    return hr;
}
```

So far we've loaded the source filter without using the automatic mapping of the registry. To mimic the exact behavior of the *AddSourceFilter()* function, you need to query the source filter for its IFileSourceFilter interface. If you find it, you must call the *Load()* function of that interface, using the input filename as a parameter. That's it!

```
LoadFilter (L"ABC - Fruit Source Filter", &pFilter);

hr = pFilter->QueryInterface(IID_IFileSourceFilter,(void**)&pFSFilter);
if (SUCCEEDED(hr)  {
    pFSFilter->Load(wPath, NULL);
    pFSFilter->Release();
}
```

## 9.3.3 Adding Transform and Rendering Filters

That wasn't too complex, was it? Let's keep going. To add a transform or a rendering filter, you only have to call the same *LoadFilter()* function with the filter name as a parameter. And, Voilà!

```
// Load the Rendering Filter.
LoadFilter (L"ABC - Text Display Filter", &pRenderFilter);
```

## 9.3.4 Connecting the Two Pins

Once the two filters are loaded, you need to connect the output pin of CFruitFilter with the input pin of the CTextOut filter. Well, first you need to find the pins before you can connect them together.

Once you have a pointer to a filter, you can enumerate a list of its pins by calling the member function *IBaseFilter::EnumPins()*, which returns an IEnumPins object. You can then call the *IEnumPins::Next()* function to retrieve a pointer to the pin interface, IPin. Once you have a pointer to the pin, you can get the pin name by calling the *IPin::QueryPinInfo()* function, which returns the pin name in a `PIN_INFO` structure.

```
HRESULT CCustomFilterGraph::FindPin(
    IBaseFilter *pFilter,       // Filter to search for pins
    WCHAR *pszName,             // Name of pin
    IPin **ppPin)               // returns the IPin interface
{
    IEnumPins *pEnumPins=NULL;
    IPin *pPin=NULL;
    PIN_INFO PinInfo;
    HRESULT hr;
    ULONG nFetched;

    // Get an enumerated list of the pins in this filter
    pFilter->EnumPins(&pEnumPins);
```

Notice that the *IBaseFilter::FindPin()* method exposed by the filter does not find the pin based on the actual pin name. This function works in conjunction with the *QueryId()* function to implement graph persistency (Save/Restore). Refer to the SDK for more information about persistence.

```
    while (SUCCEEDED(hr = pEnumPins->Next(1, &pPin, &nFetched)) ) {
        pPin->QueryPinInfo(&PinInfo);

        // Silly, but we have to release this since QueryPinInfo() adds
        // a reference to the filter..
        if (PinInfo.pFilter)
            PinInfo.pFilter->Release();

        // Return the IPin interface when you find a match..
        if (lstrcmpW(PinInfo.achName, pszName) == 0) {
            *ppPin = pPin;
            break;
        }
        pPin->Release();
    }

    pEnumPins->Release();
    return hr;
}
```

You can call our *FindPin()* function to find the output pin of the source filter, Text!, and the input pin of the rendering filter, In, as follows:

```
FindPin(pSrcFilter, L"Text!", &pTextPin);
FindPin(pRenderFilter, L"In", &pRenderPin);
```

Finally, you can call the *IGraphBuilder::Connect()* function to connect the two pins together. Then the media type negotiation starts. Once that phase is done, you will have the complete filter graph.

```
m_pGraph->Connect(pTextPin, pDisplayPin);
```

## 9.4 COM: Accessing Custom Interfaces

In the previous chapter you learned how to add the custom interface ITextStat to CTextOutFilter. As you recall, the ITextStat interface exposes two functions that return the number of characters and the number of words processed by the CTextOutFilter. You also learned how to access this custom interface and retrieve this data from the property page of the filter.

Now, suppose that you want to find out the same information from within your application so you could log it to a file or display it in some other format, in a chart, for example. Actually, the process is a bit similar to what we did in the previous chapter except that you must first obtain a pointer to the CTextOutFilter and then get a pointer to the custom interface.

Assuming that you've already loaded the filter in the filter graph, you can call the *IFilterGraph::FindFilterByName()* function to get a pointer to the filter. The function accepts the filter name as a parameter and returns a pointer to the filter.

```
#include <initguid.h>                          ◊ Must have this here for proper COM
                                                 initialization.
#include "..\\Filters\\Frender\ITextStat.h"  ◊ ITextStat custom interface definition.
void CActiveFilterDlg::OnItextstat()
{
    HRESULT hr;
    IBaseFilter *pFilter;
    IFilterGraph *pFGraph;

    // First, find the filter in the filter graph..
    m_pMC->QueryInterface(IID_IFilterGraph, (void**)&pFGraph);
    hr = pFGraph->FindFilterByName(L"ABC - Text Display Filter", &pFilter);
    pFGraph->Release();

    if (FAILED(hr))
        return;
```

Once you have a pointer to the filter, you can call the *IBaseFilter::QueryInterface()* function to get a pointer to the custom interface. Now you can call the appropriate functions exposed by the custom interface, *get_NumberOfChars()* and *get_NumberOfWords()*, to retrieve the number of characters and words processed by the CTextOutFilter.

```
    // Get a reference to the custom ITextStat interface..
    ITextStat *pTextStat;
    int nChars, nWords;
    hr = pFilter->QueryInterface(IID_ITextStat, (void**)&pTextStat);

    if (SUCCEEDED(hr)) {
        // Call the declared interface methods..
        pTextStat->get_NumberOfChars(&nChars);
        pTextStat->get_NumberOfWords(&nWords);
        wsprintf (szTmp, "NumWords: %d, NumChars: %d", nWords, nChars);
        AfxMessageBox(szTmp);
        pTextStat->Release();
    }

    pFilter->Release();
}
```

## 9.5 COM: Showing Filter Property Pages

Typically, filters display their status and configuration information in their own property pages. In the previous chapter, you learned how to add a property page to the CTextOutFilter and used the graph editor to display it. Let's see how you would display property pages from within your own application.

As when you worked with the custom interface, you must first retrieve a pointer to the TextOut filter. You must then call the *IBaseFilter::QueryInterface()* function to retrieve a reference to the property page interface, ISpecifyPropertyPages. If the filter exposes a property page, you can then call the *ISpecifyPropertyPages::GetPages()* function to obtain a list of the property pages exposed by the filter. This function actually returns the CLSIDs of the property pages. Finally, you can call the standard COM function *OleCreatePropertyFrame()* to display the property pages.

```
void CActiveFilterDlg::OnPropertypage()
{
    HRESULT hr;
    IBaseFilter *pFilter;

    // Find the TextOut filter
    IFilterGraph *pFGraph;
    m_pMC->QueryInterface(IID_IFilterGraph, (void**)&pFGraph);
    hr = pFGraph->FindFilterByName(L"ABC - Text Display Filter", &pFilter);
    pFGraph->Release();

    // Get a reference to the ISpecifyPropertyPage interface from the
    // filter.  It will fail if the filter does not contain any prop pages
    ISpecifyPropertyPages *pPropertyPage;
    CAUUID Pages;

    hr = pFilter->QueryInterface(IID_ISpecifyPropertyPages,
             (void**)&pPropertyPage);

    // Now get the property page information and
    // call the OLE function to display the prop page…
    if (SUCCEEDED(hr) ) {
        pPropertyPage->GetPages(&Pages);
        OleCreatePropertyFrame(
            m_hWnd,                       ← Handle of parent window
            0 ,                           ← X position of Window
            0,                            ← Y position of Window
            L"Hello",                     ← Title of property sheet dialog box
            1,                            ← Number of objects in next parameter
            (LPUNKNOWN *)&pFilter,        ← Object that holds the prop sheet
            Pages.cElems,                 ← Number of property pages
            Pages.pElems,                 ← Pointer to their CLSIDs (or GUIDs)
            NULL,                         ← local identifier (ignore)
            0,                            ← reserved
            NULL                          ← reserved
            );
        CoTaskMemFree(Pages.pElems);
        pPropertyPage->Release();
    }
    pFilter->Release();
}
```

Free this pointer to avoid memory leaks.

Notice that the *GetPages()* function allocates a task specific memory to hold the list of property pages. Make sure to free that memory to avoid memory leaks.

## 9.6 Creating Events under DirectShow

Filters and the Filter Graph Manager (FGM) send messages, also known as events, to alert the application of special conditions. For example, there are events that inform the application when an error occurs, when the end of a stream is reached, or when the video size has changed.

DirectShow uses the Win32 *CreateEvent()* function to create a manual-reset event, which is used for signaling the application. On one side, the filter sets the Win32 event when it needs to post a message, and it inserts the message in an internal queue. On the other side, the application waits for the event to be signaled, retrieves the message from the internal queue, and resets the Win32 event.

From an application, you can call the *IMediaEvent::GetEventHandle()* function to retrieve a handle to the Win32 event. You can then call the *WaitForSingleObject()* or *MsgWaitForMultipleObjects()* function to wait for that event to be signaled. The latter function returns when a Windows message is posted to the application message queue or when the Win32 event is signaled. Refer to the sample applications on the DirectShow SDK to learn how to use this function for event handling.

In our example, we choose to use the *WaitForSingleObject()* function running in a separate thread. We choose to do so because we are using a dialog box as the main window and could not hook into the message pump. Also this implementation turned out to be a bit simpler.

```
// This is called when the thread is running
int CFilterGraph::Run()
{
    // get hold of the event notification handle so we can wait for
    // completion
    IMediaEvent *pME;
    HANDLE hGraphNotifyEvent;
    HRESULT hr = m_pGraph->QueryInterface(IID_IMediaEvent, (void **) &pME);
    if (FAILED(hr))
        return FALSE;

    hr = pME->GetEventHandle((OAEVENT*) &hGraphNotifyEvent);
    pME->Release();

    if (FAILED(hr))
        return FALSE;

    DWORD result;

    while (TRUE) {
        // Block until an event arrives from the filter graph
        result = WaitForSingleObject (hGraphNotifyEvent, INFINITE);
        OnGraphNotify();
    }

    return FALSE;
}
```

The *WaitForSingleObject()* function returns only when the Win32 event is set, so there is an infinite timeout. At that point, you can call *IMediaEvent::GetEvent()* to retrieve the pending DirectShow event from the internal queue. This function returns the event code sent by the filter or the filter graph manager and resets the Win32 event. Notice that you must call the *IMediaEvent::FreeEventParams()* function in order to free any memory allocated when the *GetEvent()* was called.

```
void CFilterGraph::OnGraphNotify()
{
    IMediaEvent *pME;
    long lEventCode, lParml, lParm2;

    // Get a reference to the IMediaEvent interface
    HRESULT hr = m_pGraph->QueryInterface(IID_IMediaEvent, (void **)&pME);

    // Now, get the event and handle it accordingly.
    if( SUCCEEDED(hr)){
        if( SUCCEEDED(pME->GetEvent(&lEventCode, &lParml, &lParm2, 0)) ) {
            switch (lEventCode) {
                case EC_COMPLETE:
                    Stop();
                    break;
                case EC_USERABORT:
                case EC_ERRORABORT:
                    Stop();
            }
        }
        // frees memory used for GetEvent()
        pME->FreeEventParams(lEventCode, lParml, lParm2);
        pME->Release();
    }
}
```

## 9.7 ActiveX: A Simple Way to Control DirectShow

The DirectShow SDK includes an ActiveX control, which communicates directly with the DirectShow COM interface discussed above. It is a high-level interface that includes its own user interface for controlling the media stream. We'll show you how to disable this user interface if you so desire.

Notice that with the ActiveX control you have less control over the filter graph. For example, you cannot access any of your custom filter interfaces or display the filter property pages as you can with the COM interface.

### 9.7.1 Playing a File Using the ActiveX Interface

Let's see how you can load and play a file with the ActiveX control interface using the Microsoft Visual C++ development environment. We're assuming that you're using Microsoft Foundation Classes (MFC) to implement a dialog-based application.

Similar to any OLE/COM component, you can insert the DirectShow Control Object into your project from the Microsoft Visual C++ compiler. The compiler creates a new MFC class, which serves as a wrapper to the ActiveX control. The default class name is CActiveMovie.

At this stage you can either use the *Create()* member function of the class to create an instance of the DirectShow control, or you can embed it in a dialog box. In the interests of simplicity, let's see how we can do this within a dialog box. You should be able to figure out the *Create()* function easily.

Notice that when you add an ActiveX control to your project, you automatically add a new icon for that control in the control toolbar (Figure 9-3). Now you can add the DirectShow ActiveX control in the same fashion as you would add a push button or an edit box.

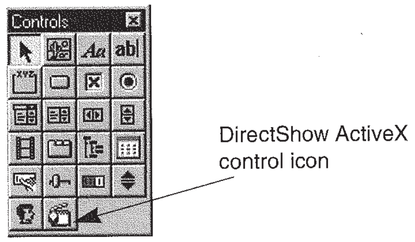
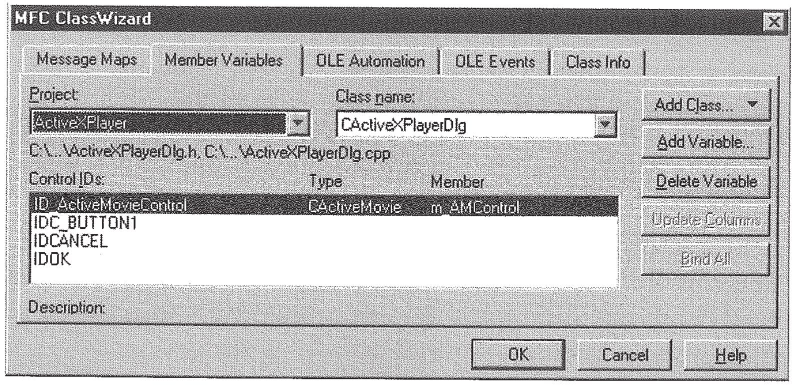


**FIGURE 9-3** Visual C++ dialog editor template.



Project: ActiveXPlayer
Class name: CActiveXPlayerDlg
C:\...\ActiveXPlayerDlg.h, C:\...\ActiveXPlayerDlg.cpp

| Control IDs: | Type | Member |
|---|---|---|
| ID_ActiveMovieControl | CActiveMovie | m_AMControl |
| IDC_BUTTON1 | | |
| IDCANCEL | | |
| IDOK | | |

Description:

**FIGURE 9-4** Associate the DirectShow control with a member variable.

To access this control from your application, you must first associate the newly added control with a member variable in the dialog box class. To do so, launch the class wizard (available on the Microsoft compiler) and add a variable for the ActiveX control ID as shown in Figure 9-4. Note that we used the name *m_AMControl.*

Now you're ready to play the file. To play a file, you need only call the *CDirectShow::SetFileName()* and pass the filename as a parameter. In turn, the ActiveX control communicates directly with the DirectShow COM interface and uses the automatic method to build the entire filter graph.

You can add your own file extensions here; *.ftf* is the Fruit text file extension.

```
void CActiveXPlayerDlg::OnOpenFile()
{
    // Prompt the user for a file name..
    static char cszFilter[] =
    "Multimedia Files|*.mpg; *.avi; *.mov; *.FTF | All Files (*.*)|*.*||";

    CFileDialog cFile(TRUE, NULL, NULL,
            OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT, cszFilter);

    if (cFile.DoModal() == IDOK) {
        CString str = cFile.GetPathName();
        m_AMControl.SetFileName(str);    // Set ActiveX control filename...
    }
}
```

Whenever the ActiveX control is used, it displays the standard user interface shown in Figure 9-5. You can use this interface to Start/Stop/Pause or navigate through the movie.

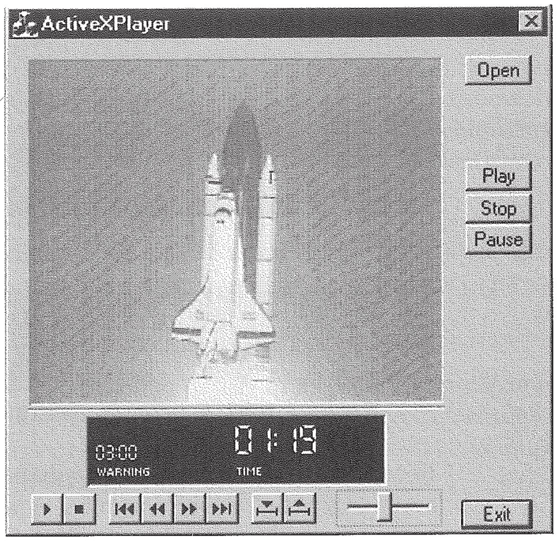


FIGURE 9-5 ActiveMovie ActiveX control interface.

## 9.7.2 Controlling the ActiveX Control from Your Application

Suppose that the supplied user interface does not match your application look and feel, or you just don't like it. Luckily, with the ActiveX control you can disable the user interface and instead control the movie from within your own application.

Figure 9-6 shows the default user interface of the ActiveX control with the controls grouped according to their functionality. You can enable/disable or show/hide each of these groups separately.

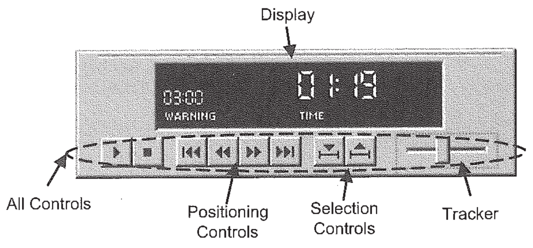


FIGURE 9-6 ActiveX control default user interface components.

You can use the *SetShowControls()* function to either show or hide all the controls. The setting of the master control takes precedence over the individual group settings described below. For example, if you call this function with FALSE as a parameter, none of the individual controls will be shown regardless of their individual settings.

```
m_AMControl.SetShowControls(TRUE);     ◊ Controls are shown per group setting.
m_AMControl.SetShowControls(FALSE);    ◊ All controls are hidden regardless of group setting.
```

Assuming that the master control is enabled, you can use the *SetShowPositionControls()* function to show/hide the positioning buttons and the *SetEnablePositionControls()* function to enable/disable the same buttons.

```
m_AMControl.SetShowPositionControls(TRUE);
m_AMControl.SetShowPositionControls(FALSE);
m_AMControl.SetEnablePositionControls(TRUE);
m_AMControl.SetEnablePositionControls(FALSE);
```

Similar functions are used to show/hide and enable/disable the selection controls, the tracker bar, display, and the context menu.

```
// Show/Hide functions
m_AMControl.SetShowSelectionControls(TRUE or FALSE);
m_AMControl.SetShowTracker (TRUE or FALSE);
m_AMControl.SetShowDisplay(TRUE or FALSE);

// Enable/Disable functions
m_AMControl.SetEnableSelectionControls(TRUE or FALSE);
m_AMControl.SetEnableTracker(TRUE or FALSE);
m_AMControl.SetEnableContextMenu(TRUE or FALSE);
```

In addition to controlling which part of the user interface to show, you can also control the running state of the movie. For example, you can call the *Run()*, *Stop()*, and *Pause()* functions to perform the specified operation.

```
m_AMControl.Run();
m_AMControl.Stop();
m_AMControl.Pause();
```

PART III