Programming Cameras and Pan-Tilts with DirectX and Java

Ioannis Pavlidis *Vassilios Morellas* *Pete Roeber*

Programming Cameras and Pan-Tilts with DirectX and Java

Programming Cameras and Pan-Tilts with DirectX and Java

Ioannis Pavlidis, University of Houston ■ *Vassilios Morellas,* Honeywell, Inc. ■ *Pete Roeber,* Vital Images

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER SCIENCE

AMSTERDAM BOSTON LONDON NEW YORK
OXFORD PARIS SAN DIEGO SAN FRANCISCO
SINGAPORE SYDNEY TOKYO

Senior Editor	Tim Cox
Publishing Services Manager	Edward Wade
Senior Production Editor	Cheri Palmer
Editorial Coordinator	Stacie Pierce
Project Management	Dusty Friedman, The Book Company
Cover Design	Yvo Riezebos Design
Cover Image	© National Geographic/gettyimages
Text Design	Lisa Devenish
Illustration/Composition	Interactive Composition Corporation
Copyeditor	Jane Loftus
Proofreader	Martha Ghent
Indexer	Micki Taylor
Printer	The Maple-Vail Book Manufacturing Group

Morgan Kaufmann Publishers
An imprint of Elsevier Science
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205
www.mkp.com

Printed in the United States of America

07 06 05 04 03 5 4 3 2 1

Library of Congress Control Number: 2002112509
ISBN: 1-55860-756-0

This book is printed on acid-free paper.

To our families

Praise for *Programming Cameras and Pan-Tilts with DirectX and Java*

This book will be an invaluable supplemental text for any image processing/computer vision course both at the undergraduate and graduate level. I would also consider it in lower level undergraduate programming courses! Many faculty members, including myself, with expertise in computer vision have started integrating computer vision research results into core courses like Data Structures and Algorithms. There will soon be strong industry demand for computer vision scientists and engineers—for people who understand vision technology and know how to apply it.

—Dr. George Bebis, Director, Computer Vision Lab, University of Nevada

This book is an excellent source for the amateur programmer and the professional developer alike. It demonstrates the fundamentals for developing useful applications using both C++ with DirectX and Java with JMF, and it collects all of the critical information in one place.

—Jonathan Dowdall, University of Nevada, Reno

I commend the authors for the effort of providing tools and techniques for many users who are struggling to acquire an in-depth understanding of video programming. There is a growing need for this topic, but there exists limited useful information for the average user or developer who may not have enough knowledge in this particular field. Most documents are either not suitable for novice users or they are difficult to follow.

—Dr. Bir Bhanu, Professor of EECS and Director Center for Research in Intelligent Systems, University of California at Riverside

FOREWORD

This very timely book appears just as low-power, small, inexpensive silicon-based video cameras are becoming ubiquitous in our daily lives. Vast networks of distributed cameras across the Internet, as well as in intranets, enable high-bandwidth instantaneous communication either across the country or across different offices in a building. But how does all this data become integrated using the most modern computer technology? In most computer vision classes taught, there is a gap between the basic understanding of image-processing algorithm issues and how cameras are programmed at the system level to actually implement these algorithms. This new book by Pavlidis et al. bridges this gap with an excellent introduction and resource for how video cameras interface with computers with the most up-to-date protocols such as serial IEEE 1394 FireWire and USB. Furthermore, the most modern object-oriented programming paradigms, including Java, are used to show specific practical implementations including pan-tilt manipulation and object-tracking applications.

This book is recommended for motivated students or individuals who want to go beyond a typical laboratory computer vision undergraduate curriculum and get their hands dirty solving real-world problems by building working camera systems using PC programming for a host of important applications. A number of concrete examples are clearly presented, which can be immediately implemented and then built upon. This is an excellent supplement for most any experimental computer vision course, as well as for

computer engineers in commercial industry who are designing systems products requiring the serious use of camera technology.

Larry Wolff
Equinox Corporation
New York, New York

Video cameras, especially digital video cameras, that easily interface with PCs proliferate at a record pace. Cameras interface with PCs through devices that are widely known as frame grabbers. Frame grabbers are usually special Peripheral Component Interconnect (PCI) cards that are added on to the basic PC configuration. In recent years, with the advent of Universal Serial Bus (USB) and the IEEE 1394 standard, the future of traditional frame grabbers seems doubtful. Both USB and IEEE 1394 are universal communication protocols over serial expansion buses. They provide enhanced PC connectivity, not only with digital cameras, but also with a number of other peripheral devices. In this book, when we mention frame grabbers we refer to both the traditional cards and the new serial expansion buses. Irrespectively of the interfacing hardware, program control over the video frames usually depends on the Windows Software Development Kit (SDK) DirectShow. DirectShow function calls can be made from within a Visual C++ program. Alternatively, program control can be exercised by Java. The relative advantage of Java is easier Web connectivity. The proliferation of PC-based cameras has started to popularize an array of applications where computer video programming plays a crucial role. Such applications include smart-user interfaces (eye-controlled), videoconferencing, automated security and surveillance, industrial inspection, and video games.

One of the most intriguing possibilities when you connect a camera to a PC is the ability to program the device to view the video feed across the Internet and through a Web browser. This gave rise to a

new generation of small and cheap digital cameras for PCs that are aimed particularly for videoconferencing and remote monitoring. Web cameras usually conform to the USB connectivity standard. The programming methods explained in this book apply equally well to any camera that connects to a PC through a digital interface or a DirectShow-compliant frame grabber.

In this book we address the issue of computer video programming with Visual C++ and Java in the Windows XP environment. Our book describes step-by-step how to develop Visual Studio .NET and Java Media Framework programs that interface with frame grabbers. The emphasis of the book is on the programming aspect of video acquisition, display, and processing. We provide brief explanations, however, of hardware and software configuration issues where appropriate for completeness. We also provide clear explanations of relevant computer vision concepts when we introduce video-processing coding examples. Special attention is given to the development of Web-based video-processing applications through the use of Java.

This book comes at a very critical time. PC cameras and cameras in general are becoming ubiquitous and communicate with computers one way or another. This trend is fueling the demand for developers who are well versed in computer video-processing applications. Our book aspires to cover the present gap in the relevant literature. It targets primarily professional developers who work or are about to work in the area of computer video programming. The book can also be used as a complementary programming book in upper-level undergraduate or graduate level courses in image processing, computer vision, and multimedia. We assume that the prospective reader has a basic knowledge of the Visual C++ and Java environments.

Our book has a number of outstanding features:

- It emphasizes computer video programming and not simple image processing. The readers will be able to program real devices (cameras) for a number of emerging applications.
- It emphasizes practical working examples. We believe that learning by example is the best way for someone to learn programming and, in particular, to learn computer video programming. For each example we provide a clear and down-to-earth explanation of the theoretical aspects and a fully documented recital of the complete code.

- It includes such unique topics as programming of pan-tilt devices for computer control of camera motion.
- It is accompanied by a CD that contains all the code described in the book.

Acknowledgments

We would like to thank our families for their patience throughout the book-writing process. We would also like to thank our acquisitions editor Tim Cox and his staffer Stacie Pierce for being incredibly forgiving. We are equally thankful to Cheri Palmer and Dusty Friedman from the production team for a job well done. Finally, we are indebted to our reviewers for their valuable comments.

CHAPTER 1

Introduction

We preface the subject of this book by describing the technological wonders that the combination of cameras and computation have introduced in a number of applications from interpersonal communications to manufacturing inspections. We also provide a brief description of the hardware architecture of a computer-camera system. Although the focus of this book is on software, this general hardware introduction will help us maintain a perspective of the bigger picture. We close the chapter by explaining the Windows driver model and the connecting role it plays between the hardware level and the high-level software. Strictly speaking, this chapter is not a prerequisite for any other chapter. Nevertheless, it is an important chapter, especially for the uninitiated, and it provides a holistic picture of the computer-camera technology, which can be easily lost if we concentrate only on the remaining, more software–specific chapters. ■

1.1 Ubiquitous Cameras and Computers—A Technological Revolution

It took almost 30 years for the Intel processor to break the barrier of 1-GHz clock speed. Then, miraculously, it took only a year and a half to break the 2-GHz barrier. This exponential growth of processor speed paved the way for a number of exotic technologies to arrive at our desktop. One of the technologies that seems to benefit the most is digital video processing. Computation-intensive video-processing algorithms can now run in real time even on a low-end PC. Naturally, this creates a trend for increased connectivity between cameras (sources of live video) and computers. On their end, cameras are becoming digital and therefore lend themselves to easier connectivity with computers, too. Moreover, cameras are becoming smaller and cheaper. We are now at a point where cameras are almost considered computer peripherals. Certain PC manufacturers, such as SONY, offer PC models with an integrated digital camera (see Figure 1.1).

FIGURE 1.1
The C1 PictureBook laptop model by SONY, featuring an integrated PC camera.

Another recent development that blessed the "marriage" of cameras and computers is the advent of broadband connections. Through *Digital Subscriber Line* (DSL) or cable subscriptions you can send live video from your local PC camera to a remote PC. Therefore, video can not only be displayed and processed locally, but can be transmitted globally through an increasingly faster Internet.

But what is the significance of all these developments? Why should we be so excited if video can be processed at the PC level in real time and transmitted at high speeds through the Internet? The answer is simple: The value is in the range of applications now becoming possible. We will touch upon the full spectrum of computer-camera applications by category. All of these up-and-coming applications are based primarily on computer-camera programming, the topic of this textbook.

1.1.1 *Interpersonal Communication*

Small cameras can be attached to desktop or laptop PCs and cater videoconferencing needs for corporate as well as home use. The awkward conventional videoconferencing setups may soon be a thing of the past. PC videoconferencing benefits tremendously from the introduction of two technological advancements:

1. standard communication protocols over fast serial expansion buses, like IEEE 1394 (Firewire) and *Universal Serial Bus* (USB) 2.0
2. fast Ethernet at corporate networks (> 100 Mbits/s) and broadband connectivity at home (> 256 Kbits/s)

On one hand, fast serial expansion buses allow digital cameras to communicate at full-frame speed with the PC. On the other hand, fast computer networks allow PCs to communicate high-bandwidth data such as live video over the Internet. Reliable, cheap videoconferencing associated with such a ubiquitous device as a desktop or laptop PC is bound to make huge inroads in the immediate future. At the business level, it will increase the amount of cooperation between global virtual corporate divisions. In the last few years, all the major U.S. corporations have restructured themselves to align across global business lines and not along national border lines. For example, an avionics plant of U.S. Honeywell belongs to the same global division as an avionics plant of Honeywell in France. This

requires a substantial amount of cooperation between groups of people that are located in geographically different places across the globe. Corporations have responded to this need by essentially doubling the amount of business travel. Widespread videoconferencing will eliminate some of this travel and enhance the day-to-day unimodal communication (voice or text) to a multimodal one (video, voice, and text). At the home consumer level, PC videoconferencing will bring a qualitatively superior way of staying in touch with friends and relatives. This is very important for a society characterized by a high degree of relocation activity.

1.1.2 *Entertainment*

At the moment, entertainment is the most visible application area of computer-camera programming. Electronics stores are filled with IEEE 1394 camcorders that are supported by video-editing software packages. People can shoot a video clip, input it on their hard disk, manipulate it to their satisfaction, and burn it on a DVD for archiving and distribution. This brings a taste of Hollywood to their home, and it is a market bound to increase as more and more young couples want to be their own film directors in the important moments of their lives (e.g., wedding, childbirth, and birthdays).

1.1.3 *Security and Surveillance*

Cameras have been associated with security and surveillance systems for some time now. Until recently, however, the state of camera technology in security applications could be characterized as *analog*. Cameras in conventional security systems act in isolation and not in concert with computers. They send their time-lapsed video feeds into analog recorders to be recorded onto VHS tapes. Later, if the user wants to access a particular piece of video, it must be accessed sequentially—a major inconvenience. Since there is no software processing done on the live video, there is no capability for automated intrusion alert. Therefore, security is based on the vigilance of the security guard who must pay attention on multiple screens for long hours. By connecting security cameras to computers, video can be recorded digitally on computer-storage media and accessed nonsequentially. Moreover, the video can be processed online, and certain

inferences can be made automatically that enhance the functionality of the video-based security system.

The most well-known video-processing operation is the one related to motion detection. Such an operation can give instant alerts when an object such as a vehicle or a human moves through the camera's field of view. The same software also caters to smart recording. In other words, when the scene is stationary, and therefore uninteresting from the security point of view, no digital recording is taking place. Only video frames that contain moving objects are recorded. The space-saving gains from such a mode of operation are obvious.

Another up-and-coming video-processing operation for security applications is face recognition. This is a biometric application that so far has been used successfully in access control applications, for example, to enforce controlled access to secure buildings or through border control stations. As the algorithms and systems grow more sophisticated, face recognition can be used in a surveillance mode. For example, it can be employed to search for the faces of wanted people in public locations such as subway stations or airports. Because of the clear and present danger of terrorism, these computer camera applications acquire a particular significance and are expected to drive the security market for years to come.

1.1.4 *Transportation*

Many of us are familiar with radio stations that update us on the status of metro freeways during morning and evening rush hours. These radio stations report directly from the so-called *transportation management centers,* facilities with hundreds of monitors that display live video feeds from cameras dispersed across the transportation network. For now, it is mostly human operators that interpret the video scenes. It requires a lot of personnel and a lot of attention. But emerging products and prototypes promise to change this very soon. For example, Autoscope, a smart camera product by Image Sensing Systems, Inc., features software that is capable of visually measuring traffic density and controlling the relevant traffic lights. Other computer-camera prototypes, still at corporate and academic labs, are capable of performing automatic incident detection by understanding particular traffic patterns associated with traffic accidents. As the technology unfolds, it will result in full computerization of the

existing transportation camera infrastructure. The live video feeds at the transportation management centers will not end at analog monitors but at the frame grabbers of PC computers for automated processing and analysis. Local and federal government is committed to this end as part of the general *intelligent transportation initiative* that aims to make traffic on U.S. freeways safer and more efficient.

1.1.5 *Manufacturing Inspection*

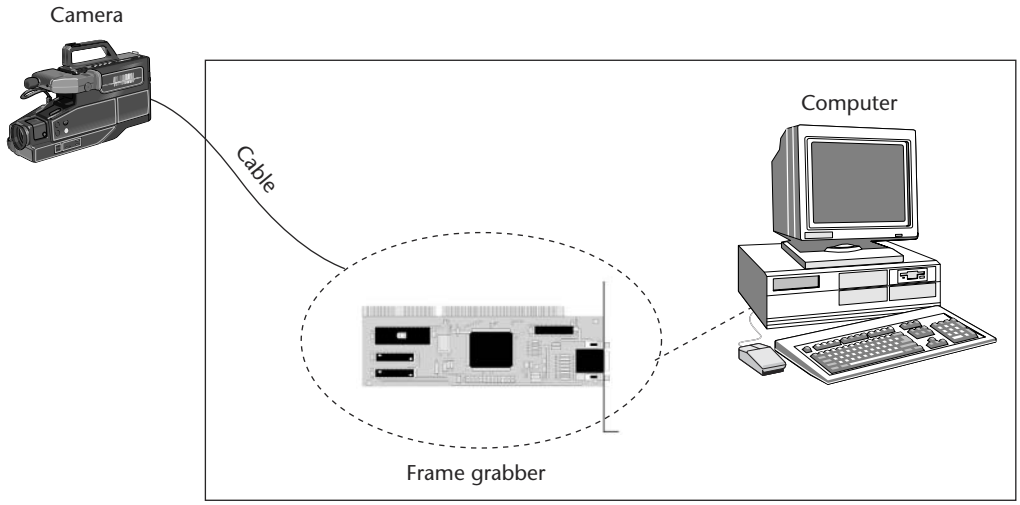
One of the most tedious and labor-intensive tasks in manufacturing is assembly inspection. For some time now, machine vision systems offer automated inspection solutions at the factory shop floor. Such machine vision systems consist of cameras connected to special computers that run computer vision software. These special computers are usually outfitted with some archaic real-time operating system, which is very poorly documented and has inadequate API support. As a result, development and maintenance of application software in this type of system is difficult and expensive. But this has started to change. PC computers outfitted with video acquisition cards have started replacing old, specialized machines. More sophisticated machine vision software allows for the automation of a greater range of inspection tasks. Therefore, manufacturing inspection is another burgeoning application for computer-camera programming. The introduction of fast PC processors, the adoption of standard, well-documented, and well-tooled Windows-based operating systems, as well as the increased sophistication of machine vision software, are bound to bring value in the manufacturing inspection market.

1.2 Hardware Architecture

A computer-camera system capable of video acquisition, processing, and display consists of the following components (see Figure 1.2):

- Computer
- Camera
- Frame Grabber
- Expansion Bus
- Software

FIGURE 1.2 Diagram of a computer camera system.



Live video frames are sent from the camera to the computer. If the camera is analog, the video signal is converted to digital in the frame grabber, which acts as a mediator in the camera and the computer. The frame grabber also functions as a temporary storage for the incoming video frames. This was more important in the old days when bus, processor, and memory access speeds were slow, and in the absence of a frame buffer, video frames were occasionally missed. Nowadays, the existence of a buffer in the frame grabber is not necessary. This reduces the role of the frame grabber to the *analog-to-digital* (A / D) signal conversion. Therefore, if the camera is digital, the need for a frame grabber may be obviated.

The frame grabber (or the digital camera) communicates the digitized frames to the *computer processor* (CPU) and memory through an expansion bus. It is important for this bus to be fast to maintain real-time video transmission. In this section we will expand upon the hardware components of a computer-camera system (computer, camera, frame grabber, and bus). We will describe their roles, the various choices that exist, and the relative advantages and disadvantages of each choice. In the next section we will elaborate upon the software components of a computer-camera system.

1.2.1 *Computer*

The computer is the most important component of a computer-camera system. It enables a conventional video-capturing device (camera) to expand into a powerful live video-processing system. Nowadays, PC-type computers are used almost exclusively in computer-camera configurations. We need to be mindful of two specifications in our choice for a suitable PC: processor speed and memory size. Any PC processor equivalent to Pentium II or better is capable of handling video acquisition and processing to one degree or another. As far as memory size is concerned it is recommended that the computer camera system features at least 128 MB. The display of video information also benefits from the existence of a high-definition computer monitor with true color capability.

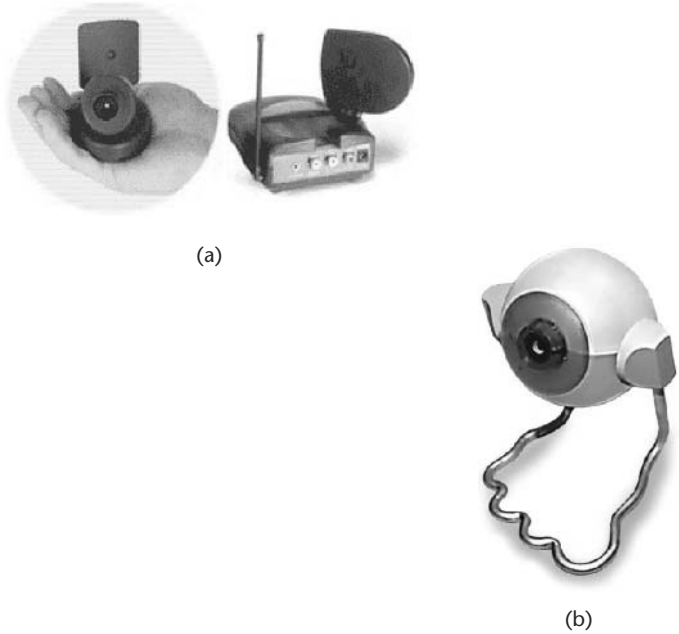
1.2.2 *Camera*

The camera is the defining component of a computer camera system. Cameras can be analog or digital, depending on the type of the video signal they output. Cameras can also be characterized with regard to their intended use. In this respect, we have entertainment cameras for the home market and security and industrial cameras for the corporate market. Other specialized types of cameras also exist for scientific and medical applications. In this book we will focus primarily on home and security/industrial cameras only.

In general, home cameras are quite different from security/industrial cameras. The majority of home cameras are digital. All newer models support the IEEE 1394 digital interface. They also feature a recording capability and for that are known as *camcorders*. The recording takes place on a digital videocassette that can be loaded on the camcorder itself. Almost all camcorders produce color video signals.

Security/industrial cameras are usually analog. Only the high end of the product line is digital at this time. Most of the security/industrial cameras produce a monochrome and not a color video signal. Also, these cameras do not feature a recording function.

Lately, a new category of cameras is gaining popularity at a lightning pace. These are small digital color cameras that are intended for PC use only. They support either the USB or IEEE 1394 interfaces, and



(a)

(b)

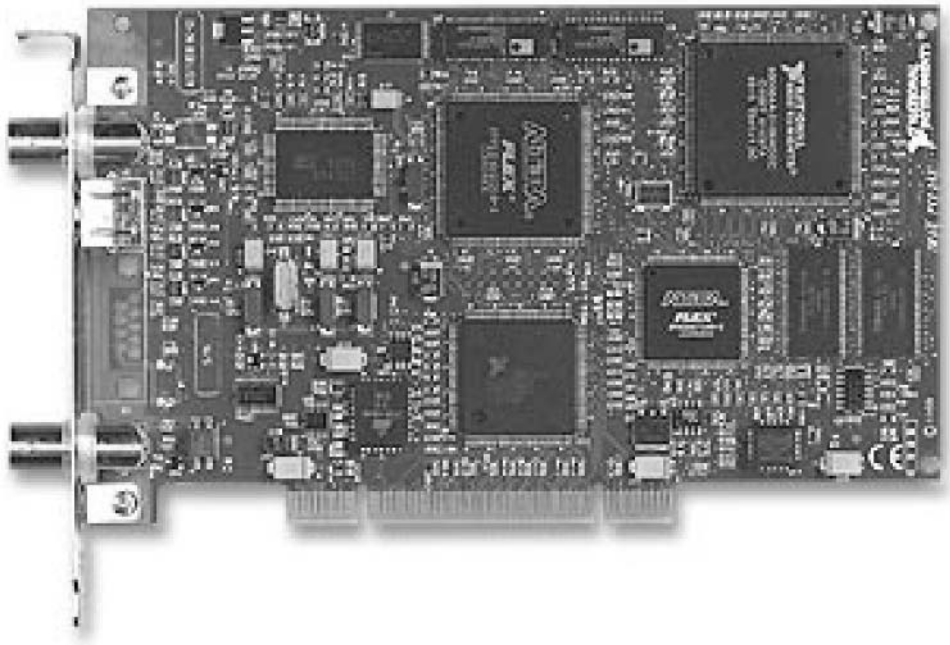
FIGURE 1.3
Two popular PC cameras. (a) The USB XCam2 wireless camera by X10. (b) The IEEE 1394 iBOT camera by Orange Micro.

they are meant to be used for videoconferencing, computer games, face recognition, and video-based security systems for the home. PC cameras exemplify a major trend and contribute to the increased demand for computer-camera programming. Figure 1.3 shows two popular PC camera models. We have used both of these models in testing the code cited in this book.

1.2.3 *Frame Grabber*

Frame grabbers are add-on PC cards (see Figure 1.4) that facilitate the connection between a camera and a PC. They mainly cater to analog cameras. Their primary function is to convert the analog video signal to digital video signal, so that it can be input to the computer. Most frame grabbers are compatible with RS170, *National Television Standards Committee* (NTSC), CCIR, and *Phase Alternating Line* (PAL) video-format standards developed for television applications in North America and Europe. The distinct advantage here is compatibility with widely available and relatively inexpensive cameras made popular by mainstream applications, which keeps the

FIGURE 1.4 The PCI-1411 analog frame grabber by National Instruments.



overall system cost down. The disadvantage is that these standards impose restrictions on image-processing systems, mainly because of limited display resolution, speed (frames per second), and the fact that the video is interlaced (each frame split into odd and even fields read out separately). Several vendors offer frame grabbers that can interface with nonstandard, noninterlaced cameras (with higher resolution or higher frame rates), line-scan cameras, scanning electron microscopes, and medical imaging equipment. This enables images to be captured from devices that have capabilities well beyond that of standard video, a necessity in certain applications.

1.2.4 *Expansion Bus*

The digital video signal from the output of either a frame grabber or digital camera is a piece of digital information provided by a PC peripheral. Digital information from the various PC peripherals is communicated to the CPU and the main memory through expansion buses. A PC bus is a piece of electronic hardware and is associated with

a communication protocol that is implemented in midlevel software (driver). Then, there is the higher-level software, that is, the *Software Development Kit* (SDK). The SDK software provides the basic building blocks for programming a computer camera application.

There are three prevalent bus architectures in the PC market today:

1. PCI bus
2. IEEE 1394 bus
3. USB bus

The *Peripheral Component Interface* (PCI) bus is the computer industry's leading bus architecture. It greatly benefits video applications that require high throughput coupled with fast CPU and peripheral access. An added bonus is that its platform and processor independence has allowed it to become the bus standard for both Intel and Macintosh machines. The functional benefits of the PCI bus are

- real-time display on the system display controller,
- real-time acquisition of images to system memory (negating the need for expensive frame grabber memory), and
- high-speed access to today's powerful CPUs for high-speed processing.

An up-and-coming expansion bus standard is the so called IEEE 1394 or Firewire. It is capable of transporting data at 100, 200, or 400 *Megabits per second* (Mbps). The functional benefits of the IEEE 1394 bus are

- a digital interface—there is no need to convert analog data into digital and tolerate a loss of data integrity;
- easy to use—there is no need for terminators, device IDs, or elaborate setup;
- hot pluggable—users can add or remove 1394 devices with the bus active;
- scalable architecture—may mix 100, 200, and 400 Mbps devices on a bus;
- flexible topology—support of daisy chaining and branching for true peer-to-peer communication;

- inexpensive—guaranteed delivery of time critical data reduces costly buffer requirements;
- nonproprietary—there is no licensing problem to use for products; and
- overall configuration control—serial bus management provides control of the serial bus in the form of optimizing arbitration timing, guarantee of adequate electrical power for all devices on the bus (cycle master assignment for a particular IEEE 1394 device), assignment of isochronous channel ID, and notification of errors.

It is apparent that the IEEE 1394 bus standard comes with an impressive set of advantages. It can communicate with IEEE 1394–enabled cameras directly, thus eliminating the need for an A/D converter. It also features a unique isochronous data channel interface. Isochronous data channels provide guaranteed data transport at a predetermined rate. This is especially important for time-critical video data where just-in-time delivery eliminates the need for costly buffering. It is not a surprise that IEEE 1394 has become the digital interface of choice for camcorders. Many home PCs feature an IEEE 1394 bus incorporated in the motherboard to facilitate easy camcorder connectivity. Also, an increasing number of IEEE 1394 security/industrial and PC cameras are breaking out in the market. For legacy PCs that do not have a true IEEE 1394 bus, the only way to connect an IEEE 1394 camera is through a PCI adapter. This adapter translates the IEEE 1394 interface to the PCI interface (the standard bus interface on every PC). This is obviously a suboptimal solution because the overall performance of the fused interface between the camera and the PC corresponds to the lesser of the two constituent interfaces, which in this case is the PCI.

A very popular serial bus standard is the *Universal Serial Bus* (USB). Full-speed USB-enabled devices can signal at 12 Mbps/s, while low-speed devices use a 1.5 Mbps/s subchannel. This is a much less impressive speed compared with that afforded by IEEE 1394–enabled devices. The IEEE 1394 bus can move more data in a given amount of time, but is considerably more expensive than USB due to its more complex protocol and signaling rate. For that, the USB bus supports devices with lower bandwidth requirements, like mice, keyboards, and audio. USB is also very popular with low-end PC cameras. In the latter case, USB can get away with its low speed by transmitting

smaller frames and compromising on image quality by performing lossy compression.

1.3 Software

Cameras are actual hardware devices that communicate with the CPU through the frame grabber–bus conduit. This communication happens at the hardware level. There is a gap between this mode of communication and the higher level software. This gap is bridged by a special piece of midlevel software called a driver. High-level software cannot deal directly with hardware devices such as cameras and hard drives. It can deal very effectively, however, with files. In the Windows operating system, a driver makes a device look like a file. A handle to the device can be opened. An application program can then issue read and write requests to the driver before the device handle is finally closed.

A driver becomes part of the operating system kernel when it is loaded. In Windows 2000 and Windows XP, device drivers must be designed according to the *Windows Driver Model* (WDM). The WDM has two separate but equally important aspects.

1. The core model describes the standard structure for device drivers. In particular, it describes how device drivers are installed and started and how they should service user requests and interact with hardware.
2. A series of system drivers are provided by Microsoft. These drivers have all the basic functionality needed to service many standard types of devices. Among others, they support the two types of expansion buses we are most interested in: USB and IEEE 1394.

1.4 Summary

In this chapter we introduced the subject matter of this textbook. We described the close connection between cameras and computers in today's digital world. We referred to a number of emerging applications that are based on live video processing. We finished the chapter by providing a brief description of the various hardware and software components of a typical computer-camera system.

CHAPTER 2

The DirectShow Software Development Kit and the Component Object Model

We introduce the Microsoft DirectShow *Software Development Kit* (SDK) and the *Component Object Model* (COM) upon which it is based. DirectShow is a member of Microsoft's DirectX suite. The main objective of DirectShow is to allow for the development of powerful and dynamic multimedia applications. DirectShow exposes a number of COM objects and interfaces that empower the developer to access, process, and display multimedia streams. Our objective is to use these objects to build powerful video-processing applications. Where appropriate, we expand the functionality of DirectShow by building new objects from scratch. First, this chapter elaborates on the definition of COM objects and interfaces, concepts that are absolutely necessary in building our own DirectShow objects. Then it introduces the secrets of building simple DirectShow applications. Most of the rest of this book is devoted to constructing specialized DirectShow objects for video processing and their exemplary use within applications. Therefore, understanding the fundamentals of COM theory and application building in DirectShow is a prerequisite for following the subsequent chapters up to Chapter 9, which switches to computer camera programming with Java. ■

2.1 Overview

The Microsoft DirectShow SDK [6] is a media-streaming software architecture for the Microsoft Windows platform. It supports capture using WDM devices or older *Video For Windows* (VFW) devices.

DirectShow is based on COM technology. To write a DirectShow application, you must understand COM client programming. For some applications we do not need to implement our own COM objects. DirectShow provides the components we need. But, if we want to extend DirectShow by writing our own components, then we must implement them as COM objects [9].

The basic building block of DirectShow is a software component called a *filter*. A filter is implemented as a COM object. It generally performs a single operation on a multimedia stream. For example, there are DirectShow filters that

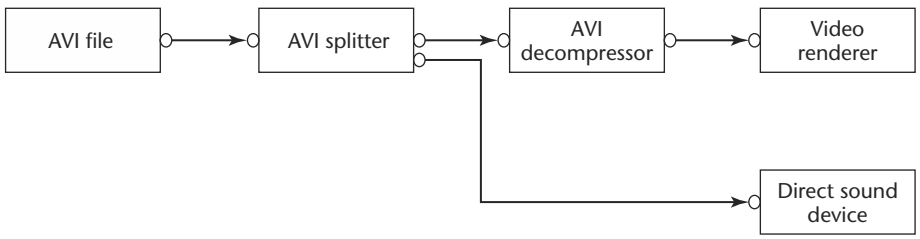
- get video from a video capture device, and
- decode a particular stream format, such as *Audio Video Interleave* (AVI) video.

Filters receive input and produce output. For example, if a filter decodes AVI video, the input is the AVI-encoded stream and the output is an uncompressed *red-green-blue* (RGB) video stream.

To perform a given task, an application connects several filters so that the output from one filter becomes the input for another. A set of connected filters is called a *filter graph*. As an illustration of this concept, Figure 2.1 shows a filter graph for playing an AVI file.

A DirectShow application does not have to manage the individual filters in a filter graph. Instead, DirectShow provides a high-level component called the Filter Graph Manager. The Filter Graph Manager controls the flow of data through the graph. The application makes

FIGURE 2.1 Diagram of a filter graph for playing an AVI file.



high-level SDK calls such as Run (to move data through the graph) or Stop (to stop the flow of data). If we require more direct control of stream operations, we can access the filters directly through COM interfaces. The Filter Graph Manager also passes event notifications to the application so that your application can respond to events, such as the end of a stream.

The DirectShow software filter structure mimics the structure of an electronic media device. A typical DirectShow application comprises three kind of filters: *source filters*, *transform filters*, and *rendering filters*. Source filters support multimedia stream acquisition from capture devices such as digital cameras. Transform filters act on the acquired multimedia stream and alter it in some way. Alteration may involve an operation as simple as noise suppression or as complicated as scene understanding. Most of the supplied transform filters perform standard image/audio processing operations such as compression and decompression. One of the purposes of this book is to teach you how to develop custom transform filters for more elaborate computer vision applications. Finally, rendering filters render either the original or the transformed media streams on the respective PC peripherals. This includes the display of the video stream on the PC screen and audio play on the computer's speakers. Figure 2.2 depicts the

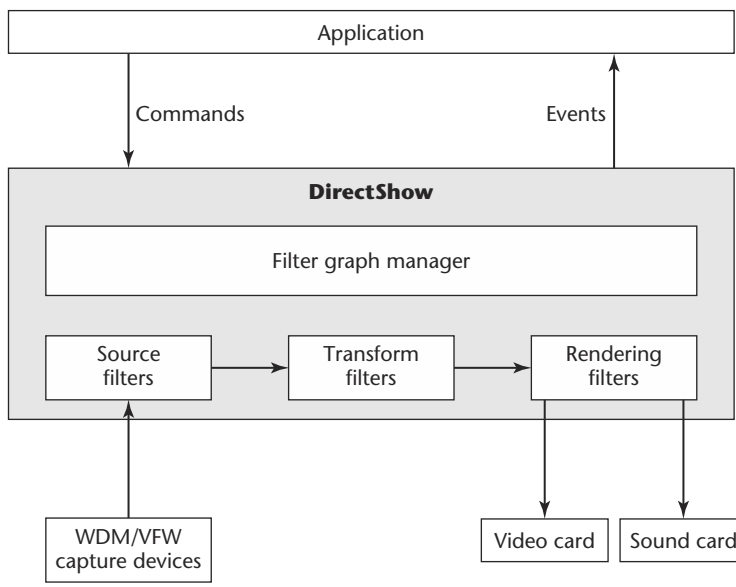


FIGURE 2.2
Relationship diagram between an application, the DirectShow components, and some relevant inputs and outputs.

relationship between an application, the DirectShow filters, and some of the input/ouput components with which DirectShow interacts.

Like their electronic counterparts, DirectShow filters have (virtual) *pins*. The pins are software objects that handle the transfer of a multimedia stream between filters. Pins are depicted diagrammatically as small circles attached to the filter graphic (see Figure 2.1). We have two types of pins: input and output. The input pin of one filter connects to the output pin of the filter up the stream. Most transform filters have only one input and one output pin. In contrast, source filters have no input pins and renderer filters have no output pins.

2.2 The COM

In our introduction to the DirectShow SDK we used terms such as *components* and *interfaces*. This is the language of COM programming, and a basic knowledge of COM is absolutely essential to the DirectShow programmer. But what is COM? COM is a way for software components to communicate with each other. It is a binary and network standard that allows any two components to communicate regardless of what machine they're running on (as long as the machines are connected), what operating systems the machines are running (as long as it supports COM), and what language the components are written in.

COM technology was born out of the necessity for software reuse. But, some may exclaim that reuse is the hallmark of C++ and object-oriented programming. Why then do we need COM? Well, the truth is that C++ does not easily support the creation of reusable binary components. Rather, C++ makes it relatively easy to reuse source code. Most major C++ libraries are shipped in source form, not compiled form. In this context, programmers are required to look at the source code to inherit correctly from an object. Also, programmers rely on the implementation details of the original library. Finally, programmers are tempted to modify the original source code and privately build the modified library. Therefore, "reusability" in the context of C++ is a very constrained term.

COM objects and C++ objects share some common traits. They both expose methods that our application can call to perform any of

the supported tasks. Nevertheless, there are some distinct differences that truly set COM objects apart.

1. COM objects enforce stricter encapsulation than C++ objects. You cannot simply create the object and call any public method. A COM object's public methods are grouped into one or more interfaces. To use a method, you must create the object and obtain the appropriate interface from the object. An interface typically contains a related set of methods that provide access to a particular feature of the object.
2. COM objects are not created in the same way as C++ objects. There are several ways to create a COM object, but all involve COM-specific techniques. The Microsoft DirectShow SDK includes a variety of helper functions and methods that simplify creating most of the DirectShow objects.
3. We must use COM-specific techniques to control the lifetime of the object.
4. COM objects do not need to be explicitly loaded. COM objects are typically contained in a *Dynamic Link Library* (DLL). However, you do not need to explicitly load the DLL or link to a static library to use a COM object. Each COM object has a unique registered identifier that is used to create the object. COM automatically loads the correct DLL.
5. COM is a binary specification. COM objects can be written and accessed from a variety of languages. You don't need to know anything about the object's source code. For example, Microsoft Visual Basic applications routinely use COM objects that were written in C++.

It is important to understand the distinction between objects and interfaces.

- An object may expose any number of interfaces. For example, while all objects must expose the `IUnknown` interface, they normally expose at least one additional interface, and they might expose many. In order to use a particular method, you must not only create the object, you must also obtain the correct interface.

- More than one object might expose the same interface. An interface is a group of methods that perform a specified set of operations. The interface definition specifies only the syntax of the methods and their general functionality. Any COM object that needs to support a particular set of operations can do so by exposing a suitable interface. Some interfaces are highly specialized and are exposed only by a single object. Others are useful in a variety of circumstances and are exposed by many objects. The extreme case is the `IUnknown` interface, which must be exposed by all COM objects.

2.3 COM Interfaces

A COM interface is a C++ base class (actually, a C++ struct type object) that declares a group of virtual functions. These functions completely control some aspect of derived class behavior. Pure virtual functions declared in the base class of a hierarchy are not intended to be invoked. For this reason they lack any implementation. Their implementation is taking place at each COM object that exposes the respective interface. That does not mean, however, that the interface is void of a semantic definition. The semantic definition (or interface contract) describes what the virtual functions do when they are called by a COM object. The interface contract is defined in terms of the object's behavior and is not tied to a particular implementation. In fact, the object is free to implement the interface contract in any way it chooses (as long as it honors the contract). If a COM object exposes an interface, it must support every method (function) in the interface definition. However, the details of how a particular method is implemented may vary from COM object to COM object. For example, different COM objects may use different algorithms to arrive at the final result.

Things may become clearer by following a concrete example. One very useful DirectShow interface is `IGraphBuilder` (see Listing 2.3.0.1). This interface provides methods that enable an application to build a filter graph. Interestingly, some of these methods construct graphs from partial information. For example, the `RenderFile` method builds a graph for file playback, given the name of the file. The `Render` method renders data from an output pin by connecting

LISTING 2.3.0.1 Base class definition of the `IGraphBuilder` interface.

```
 1: struct IGraphBuilder {
 2:    // Connects two pins. If they will not connect directly,
 3:    // this method connects them with intervening transforms.
 4:    virtual HRESULT Connect(
 5:        IPin *ppinOut,
 6:        IPin *ppinIn
 7:        ) = 0;
 8:
 9:    // Adds a chain of filters to a specified output pin to
10:    // render it.
11:    virtual HRESULT Render(
12:        IPin *ppinOut
13:        ) = 0;
14:
15:    // Builds a filter graph that renders the specified file.
16:    virtual HRESULT RenderFile(
17:        LPCWSTR lpwstrFile,
18:        LPCWSTR lpwstrPlayList
19:        ) = 0;
20:
21:    // Adds a source filter to the filter graph for a specific file.
22:    virtual HRESULT AddSourceFilter(
23:        LPCWSTR lpwstrFileName,
24:        LPCWSTR lpwstrFilterName,
25:        IBaseFilter **ppFilter
26:        ) = 0;
27:
28:    // Sets the file for logging actions taken when
29:    // attempting to perform an operation.
30:    virtual HRESULT SetLogFile(
31:        HANDLE hFile
32:        ) = 0;
33:
34:    // Requests that the graph builder return as soon as
35:    // possible from its current task.
36:    virtual HRESULT Abort(void) = 0;
37:
38:    // Queries whether the current operation should continue.
39:    HRESULT ShouldOperationContinue(void) = 0;
40}:
```

new filters to the pin. An application that uses either the `RenderFile` or the `Render` method does not need to specify every filter and every pin connection in the graph. Instead, the Filter Graph Manager selects filters that are registered on the user's system, introduces them into the graph, and connects them.

In addition to the base class declaration of the pure virtual functions, an interface is accompanied by a semantic definition (or interface contract). The interface contract describes in detail the virtual functions (parameters/return values) and what they do when they are called by a COM object. We will provide the semantic description for every member function (method) of the `IGraphBuilder` interface. We start with the `Connect` method. As can be seen in Table 2.1, the interface contract for each method encompasses three

TABLE 2.1 Interface contract for the `Connect` member function.

`IGraphBuilder::Connect`	
Parameters	
`ppinOut`	Pointer to the output pin.
`ppinIn`	Pointer to the input pin.
Return Value	
`S_OK`	Success.
`VFW_S_PARTIAL_RENDER`	Unsupported stream format.
`E_ABORT`	Operation aborted.
`E_POINTER`	NULL pointer argument.
`VFW_E_CANNOT_CONNECT`	No intermediate filters to make the connection.
`VFW_E_NOT_IN_GRAPH`	Cannot perform the requested function.
Remarks	
This method connects pins from two filters directly or indirectly, using transform filters if necessary. The method either succeeds or leaves the filter graph unchanged. First, the *Filter Graph Manager* attempts a direct connection. If that fails, it attempts to use any available transforms provided by filters that are already in the filter graph. (It enumerates these in an arbitrary order.) If that fails, it attempts to find filters from the registry to provide a transform. These will be tried in order of merit.	

pieces of information:

1. Details about the method's parameters (Parameters section in Table 2.1).
2. Details about the method's return values (Return Value section in Table 2.1).
3. Details about the method's overall functionality (Remarks section in Table 2.1).

The `Connect` method connects an output pin of a filter to an input pin of another filter. The method follows a certain protocol to achieve the filter connection. If a direct connection is not feasible, then the method may introduce appropriate intermediate filters in the filter graph to achieve its objective. For example, if the method is assigned to connect the output pin of the Windows Media Source Filter with the input pin of the Video Renderer Filter, then it may invoke the AVI Decompressor Filter to complete the connection. The invocation of the AVI Decompressor Filter will be necessary if the `Connect` method establishes that the AVI stream provided through the Windows Media Source Filter is in compressed form. Unfortunately, the Video Renderer Filter accepts only uncompressed AVI streams in RGB format. The apparent miscommunication problem can be solved only with the intervention of a third filter that transforms compressed AVI to raw RGB streams. This filter is the AVI Decompressor Filter.

Given the mission of the `Connect` method it is not surprising that its input parameters are pointers to the input and output pins that are trying to connect. The return value of the method is a coded value that allows us to infer success or failure. This coded value is of type `HRESULT`. All DirectShow methods return error and success notifications in the form of `HRESULT` values.

The `Render` method adds a chain of filters to a specified output pin to render it. Therefore, this is another method that bears the hallmark of the `IGraphBuilder` interface: self-initiated search and discovery of the filters necessary to complete a particular task (in this case, rendering). Moreover, assembly of these filters into a coherent graph. The method has only one parameter: the pointer to the output pin of the filter that is to be rendered. The method returns

TABLE 2.2 Interface contract for the `Render` member function.

`IGraphBuilder::Render`	
Parameters	
`ppinOut`	Pointer to the output pin.
Return Value	
`S_OK`	Success.
`VFW_S_AUDIO_NOT_RENDERED`	Cannot play back the audio stream.
`VFW_S_DUPLICATE_NAME`	Duplicate filter added with a modified name.
`VFW_S_PARTIAL_RENDER`	Stream formats in the movie are not supported.
`VFW_S_VIDEO_NOT_RENDERED`	Cannot play back the video stream.
`E_ABORT`	Operation aborted.
`E_POINTER`	NULL pointer argument.
`VFW_E_CANNOT_CONNECT`	No intermediate filters to make the connection.
`VFW_E_CANNOT_RENDER`	No filters to render the stream.
`VFW_E_NO_ACCEPTABLE_TYPES`	There is no common media type between the pins.
`VFW_E_NOT_IN_GRAPH`	Cannot perform the requested function.
Remarks	
This method connects the output pin of a filter directly or indirectly to other filter or filters that will render it. The method is using transform filters as intermediary filters if necessary. Filters are tried in the same order as for the `IGraphBuilder::Connect` method.	

an `HRESULT` value that signifies success or failure. The pertinent `HRESULT` values for the `Render` method are listed in Table 2.2.

The `RenderFile` method performs exactly as the `Render` method. The only difference is that the `RenderFile` method has a more specialized functionality; it renders only files (e.g., video files). In contrast, the `Render` method can render output from different kinds of sources, including live sources such as cameras. The interface contract of the `RenderFile` method is shown in Table 2.3. The only useful parameter of the method is a pointer to the name of the file. A second parameter serves as a placeholder for now.

The `AddSourceFilter` (see Table 2.4) method adds a source filter to the filter graph for a specific file. By source filter we mean a filter that takes data from some source such as the hard drive or the Internet and introduces it into the filter graph. The `lpwstrFileName`

TABLE 2.3 Interface contract for the `RenderFile` member function.

`IGraphBuilder::RenderFile`	
Parameters	
`lpwstrFile`	Pointer to the name of the file.
`lpwstrPlayList`	Reserved. Must be NULL.
Return Value	
`S_OK`	Success.
`VFW_S_AUDIO_NOT_RENDERED`	Cannot play back the audio stream.
`VFW_S_DUPLICATE_NAME`	Duplicate filter added—name modified.
`VFW_S_PARTIAL_RENDER`	Stream format is not supported.
`VFW_S_VIDEO_NOT_RENDERED`	Cannot play back the video stream.
`E_ABORT`	Operation aborted.
`E_FAIL`	Failure.
`E_INVALIDARG`	Argument is invalid.
`E_OUTOFMEMORY`	Insufficient memory.
`E_POINTER`	NULL pointer argument.
`VFW_E_CANNOT_CONNECT`	No intermediate filters to connect.
`VFW_E_CANNOT_LOAD_SOURCE_FILTER`	Source filter could not be loaded.
`VFW_E_CANNOT_RENDER`	No filters to render the stream.
`VFW_E_INVALID_FILE_FORMAT`	The file format is invalid.
`VFW_E_NOT_FOUND`	An object or name was not found.
`VFW_E_NOT_IN_GRAPH`	Cannot perform the requested function.
`VFW_E_UNKNOWN_FILE_TYPE`	File media type not recognized.
`VFW_E_UNSUPPORTED_STREAM`	Cannot play back the file.
Remarks	
This method adds a source filter that can handle the specified file. Then it renders the output pins on the source filter, adding intermediate filters if necessary. Filters are tried in the same order as for the `IGraphBuilder::Connect` method. To determine the media type and compression scheme of the specified file, the *Filter Graph Manager* reads the first few bytes of the file, looking for file signatures.	

parameter provides the pointer to the file for which a source filter is sought. The `lpwstrFilterName` parameter is used to allow the filter to be identified by this name in the filter graph. The `ppFilter` parameter returns the `IBaseFilter` interface exposed by the source filter.

TABLE 2.4 Interface contract for the `AddSourceFilter` member function.

`IGraphBuilder::AddSourceFilter`	
Parameters	
`lpwstrFileName`	Pointer to the file.
`lpwstrFilterName`	Name to give to the added source filter.
`ppFilter`	Address of a pointer to `IBaseFilter`.
Return Value	
`S_OK`	Success.
`E_NOINTERFACE`	No support for `IFileSourceFilter`.
`E_OUTOFMEMORY`	Insufficient memory.
`E_POINTER`	NULL pointer argument.
`VFW_E_CANNOT_LOAD_SOURCE_FILTER`	The source filter could not be loaded.
`VFW_E_NOT_FOUND`	An object or name was not found.
`VFW_E_NOT_IN_GRAPH`	Cannot perform the requested function.
`VFW_E_UNKNOWN_FILE_TYPE`	File media type not recognized.
Remarks	
This method enables you to obtain and retain more control over building the rest of the graph.	

The `AddSourceFilter` method plays a complementary role to the `Render` method. An application that aims to render the contents of a file first calls the `AddSourceFilter` method. This method adds the source filter that will read off the specific file. Then, it calls the `Render` method upon the output pin of the source filter to build the rest of the filter graph automatically.

The `SetLogFile` method (see Table 2.5) sets the file for logging actions taken when attempting to perform an operation. While all the other methods of the `IGraphBuilder` interface so far help us to automatically construct filter graphs, the present method helps us to determine the cause of any failure. The `SetLogFile` method has only one parameter, which is a handle to an open file.

The `Abort` method (see Table 2.6) requests that the graph builder return as soon as possible from its current task. The current task may or may not fail to complete. It is possible that the fastest way for

TABLE 2.5 Interface contract for the `SetLogFile` member function.

`IGraphBuilder::SetLogFile`	
Parameters	
`hFile`	Handle to the log file.
Return Value	
`S_OK`	Success.
Remarks	
This method is for use in debugging; it is intended to help you determine the cause of any failure to automatically build a filter graph. Your application is responsible for opening the file and for closing it when you are done logging. Before closing the file handle, call `SetLogFile` with a NULL file handle. This will ensure that the component does not attempt to use the file handle after you have closed it.	

TABLE 2.6 Interface contract for the `Abort` member function.

`IGraphBuilder::Abort`	
Parameters	
Return Value	
`S_OK`	Success.
Remarks	
It requests that the graph builder return as soon as possible from its current task.	

the graph builder to return from its current task is to complete it. The method has no parameters. It provides the capability to stop the graph that other methods of the interface helped create.

The `ShouldOperationContinue` method (see Table 2.7) queries whether the current operation should continue. A filter that is performing some operation at the request of the graph can call this method to determine whether it should continue. Applications do not normally call this method. The method has no arguments.

TABLE 2.7 Interface contract for the `ShouldOperationContinue` member function.

`IGraphBuilder::ShouldOperationContinue`	
Parameters	
Return Value	
`S_OK`	Success.
`S_FALSE`	The current operation should not continue.
`S_UNEXPECTED`	Unexpected error.
Remarks	
It requests that the graph builder return as soon as possible from its current task.	

2.4 COM Objects

By now we know that interfaces are base classes of pure virtual functions. These classes define particular behaviors that are implemented by COM objects [8]. For example, the `IGraphBuilder` interface that we dissected in Section 2.3 contains the methods for building filter graphs. Interfaces are being implemented by COM objects. Every COM object has at least one interface, the `IUnknown`.

The central COM object in DirectShow is the Filter Graph Manager. The Filter Graph Manager builds and controls filter graphs. It also handles synchronization, event notification, and other aspects of the filter graph. To perform its duties the Filter Graph Manager implements a number of interfaces. Table 2.8 shows all the interfaces supported by the Filter Graph Manager object as well as the object's globally unique identifier (ID) `CLSID_FilterGraph`. The interfaces supported by a COM object and its globally unique ID constitute a full description for the object.

Among the interfaces implemented by the Filter Graph Manager is the `IGraphBuilder` interface. To build a filter graph we always create an instance of the Filter Graph Manager object and obtain a pointer to its `IGraphBuilder` interface. The generic code for

TABLE 2.8 Description of the Filter Graph Manager object.

Filter Graph Manager
Globally Unique ID `CLSID_FilterGraph`
Interfaces `IAMGraphStreams, IBasicAudio, IBasicVideo, IBasicVideo2, IFilterChain, IFilterGraph, IFilterGraph2,IFilterMapper2, IGraphBuilder, IGraphConfig, IGraphVersion, IMediaControl, IMediaEvent, IMediaEventEx, IMediaEventSink, IMediaFilter, IMediaPosition, IMediaSeeking, IQueueCommand, IResourceManager, IVideoFrameStep, IVideoWindow`

LISTING 2.4.0.1 Generic code for initiating graph building.

```
1:  IGraphBuilder* pIGB;
2:  HRESULT hr;
3:  hr = CoCreateInstance(CLSID_FilterGraph,
4:                NULL,
5:                CLSCTX_INPROC_SERVER,
6:                IID_IGraphBuilder,
7:                (void **)&pIGB);
```

initiating the graph-building operation is shown in Listing 2.4.0.1. In the next section, we will use our newly acquired knowledge of the Filter Graph Manager object and its primary interface (i.e., `IGraphBuilder`) to develop an application for playing an AVI file. This will be our first hands-on experience in manipulating video information. And, although this time we will be accessing "canned" video information, in subsequent chapters we will have the opportunity to manipulate live video information from PC cameras.

2.5 Our First DirectShow Application—Playing an AVI File

A typical DirectShow application (see Figure 2.2) performs three basic steps:

1. It creates an instance of the Filter Graph Manager using the `CoCreateInstance` function.
2. It uses the Filter Graph Manager to build a filter graph.
3. It controls the filter graph and responds to events.

We will follow these steps in creating a simple DirectShow application that plays a media file.

Step 1 We first initialize the COM library by calling the `CoInitialize` function (line 2 in Listing 2.5.0.1). Then, we call the `CoCreateInstance` function (line 3 in Listing 2.5.0.1) to create the Filter Graph Manager. The `CoCreateInstance` function returns a pointer to the Filter Graph Manager's `IGraphBuilder` interface. We use this interface pointer to query for the other two interfaces that are needed, `IMediaControl` and `IMediaEvent` (lines 6–7 in Listing 2.5.0.1).

LISTING 2.5.0.1 The initialization methods for creating the graph manager.

```
1:  IGraphBuilder *pGraph;
2:  CoInitialize(NULL);
3:  CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
                     IID_IGraphBuilder, (void **)&pGraph);
4:  IMediaControl *pMediaControl;
5:  IMediaEvent   *pEvent;
6:  pGraph->QueryInterface(IID_IMediaControl, (void**)&pMediaControl);
7:  pGraph->QueryInterface(IID_IMediaEvent, (void**)&pEvent);
```

Step 2 Next, the program constructs a filter graph by invoking the method `RenderFile` (line 1 in Listing 2.5.0.2). This method plays the video file specified in its first parameter.

LISTING 2.5.0.2 The graph building method.

```
1:  pGraph->RenderFile(L"skiing.avi", NULL);
```

LISTING 2.5.0.3 The methods for controlling the graph.

```
1:  pMediaControl->Run();
2:  pEvent->WaitForCompletion(INFINITE, &evCode);
3:  pMediaControl->Release();
4:  pEvent->Release();
5:  pGraph->Release();
6:  CoUninitialize();
```

Step 3 After the Filter Graph Manager has constructed the filter graph, it is ready to begin playback. The `IMediaControl::Run` method (line 1 in Listing 2.5.0.3) switches the graph into running mode. When the application invokes this method, media data begins to move through the filter graph and is rendered as video, audio, or both.

The `IMediaEvent::WaitForCompletion` method (line 2 in Listing 2.5.0.3) blocks until the file is done playing. Playback continues on a separate thread. Finally, the application cleans up by releasing all the interface pointers and the COM library (lines 3–6 in Listing 2.5.0.3).

2.6 How to Write a DirectShow Filter

In this book we mostly focus on the writing of transform filters. The source and rendering filters that come with the DirectShow SDK are satisfactory for most application needs. We can break the writing of a transform filter into the following discrete steps:

1. Determine if the filter must copy media samples or handle them in place. The fewer copies in the media stream, the better. However, some filters require a copy operation; this influences the choice of base classes.

2. Determine which base classes to use and derive the filter class (and pin classes, if necessary) from the base classes. In this step, create the header or headers for your filter. In many cases, you can use the transform base classes, derive your class from the correct transform filter class, and override a few member functions. In other cases, you can use the more generic base classes. These classes implement most of the connection and negotiation mechanisms; but these classes also allow more flexibility at the cost of overriding more member functions.
3. Add the code necessary to instantiate the filter. This step requires adding a static `CreateInstance` member function to your derived class and also a global array that contains the name of the filter, a CLSID (object type describing the class ID), and a pointer to that member function.
4. Add a `NonDelegatingQueryInterface` member function to pass out any unique interfaces in your filter. This step addresses the COM aspects of implementing interfaces, other than those in the base classes.
5. Override the appropriate base-class member functions. This includes writing the transform function that is unique to your filter and overriding a few member functions that are necessary for the connection process, such as setting the allocator size or providing media types.

The description of the filter building steps is rather generic at this point. In Chapter 3 this generic description will acquire a clear meaning as we will work our way through a specific example.

2.7 Summary

In this chapter we introduced the DirectShow SDK, which is the primary software package we will use to develop live and canned video-processing applications. The introduction was based on the development of a simple video play application. The DirectShow SDK is part of the DirectX Microsoft suite and is based on the COM architecture. To enhance the understanding of the inner workings of DirectShow we have provided a brief but comprehensive introduction to the COM architecture and its associated methodology.

CHAPTER 3

A Video Preview Application—Source and Render Filters

In this chapter we introduce programming methods specific to the DirectShow SDK. Surprisingly, basic DirectShow programming does not involve writing a single line of C++ code. DirectShow comes complete with an assortment of COM objects that we can readily use for our programming needs. Connecting and running these components is made easy thanks to a visual programming tool called `GraphEdit`. Therefore, simple programming applications such as acquiring and displaying a live video stream are only a few mouse clicks away. This chapter is very important because it gives us an idea of how many things we can do by simply using the standard objects of DirectShow. At the same time, the chapter also points out the limitations of this set of premanufactured objects, providing the motivation for studying the subsequent, more advanced programming chapters. ■

3.1 The GraphEdit Utility

The beauty of DirectShow is that it is based on COM technology. As we know by now the hallmark of COM technology is reusability. Components can be readily reused within other components or within applications. There is no better manifestation of COM reusability than the GraphEdit tool. GraphEdit is a visual tool for building filter graphs. Every DirectShow component or filter is mapped into a graphical entity. Then, the programmer can invoke filters and connect them together by means of pure visual programming without ever writing a single line of code. Moreover, the programmer can run the graph at a click of a button.

This visual environment is useful not only for quickly building applications out of available components but also for testing new custom-made filters. For example, we can build an application that previews live video feed from a camera by connecting the graphic that corresponds to the source filter for our camera with the graphic that corresponds to the rendering filter. We can also test a custom transform filter that we have developed by interjecting it between a source and a rendering filter in the graphical user interface of GraphEdit.

We can access GraphEdit by clicking on the Start button and then choosing from the cascading menu options the following sequence:

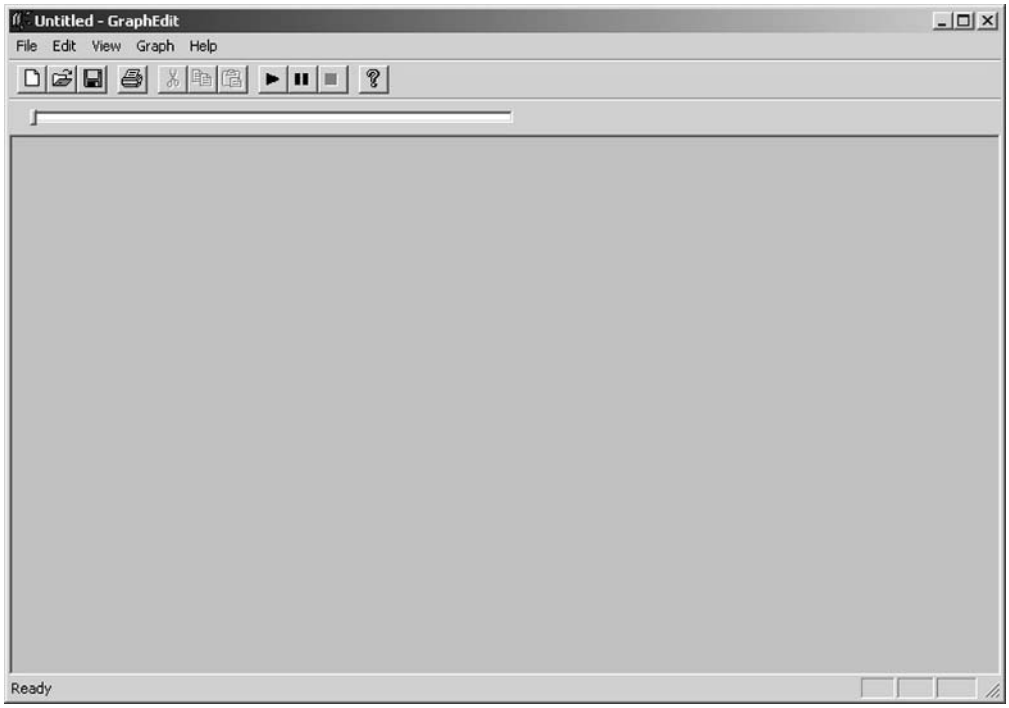
```
Programs—> Microsoft DirectX 8 SDK—> DirectX
Utilities—> Graph Edit.
```

Figure 3.1 shows the application window for the GraphEdit utility.

3.2 Building the Video Preview Filter Graph

In this book we are dealing with acquisition, manipulation, and display of live video information. Within the context of DirectShow we acquire video streams using ready source filters. Video acquisition by itself is not very useful to the application user. At a minimum, the acquired video stream must be displayed on the computer screen to have some usefulness. Display of raw video information from a camera is what we call a video preview application. It is a digital

FIGURE 3.1 The `GraphEdit` window.

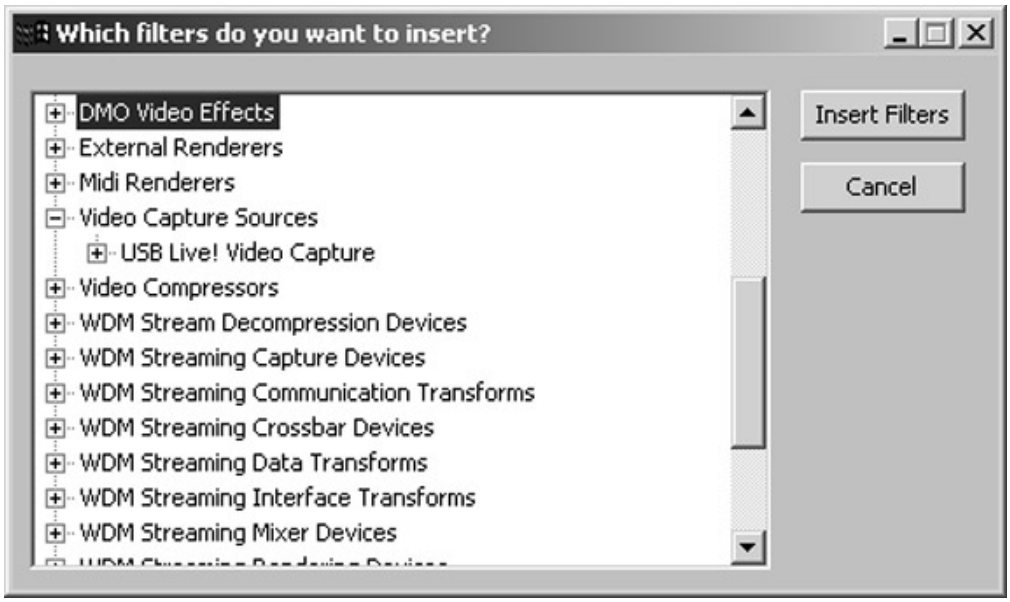


replacement to the old camera–CRT monitor pair. In contrast to video display on CRT monitors, video preview does not take up the entire computer screen. Instead, it takes up only a window within the computer screen. Therefore, more than one video feed can be displayed simultaneously.

In this chapter we will develop step-by-step a video preview application. What is truly exciting is that we will develop this application by point-and-click operations. In other words, we will not write a single line of code. This is because all the filters required for a preview application are provided by the basic DirectShow SDK. In subsequent chapters, we will write our own filters to perform custom transformations on the raw video streams. Only at that time we will resort to good old C++ programming [1] [8].

Following the flow of information, the first filter we need to introduce in our filter graph is a source filter. This filter will read off the video stream from our PC camera. Provided that we use the XCam2

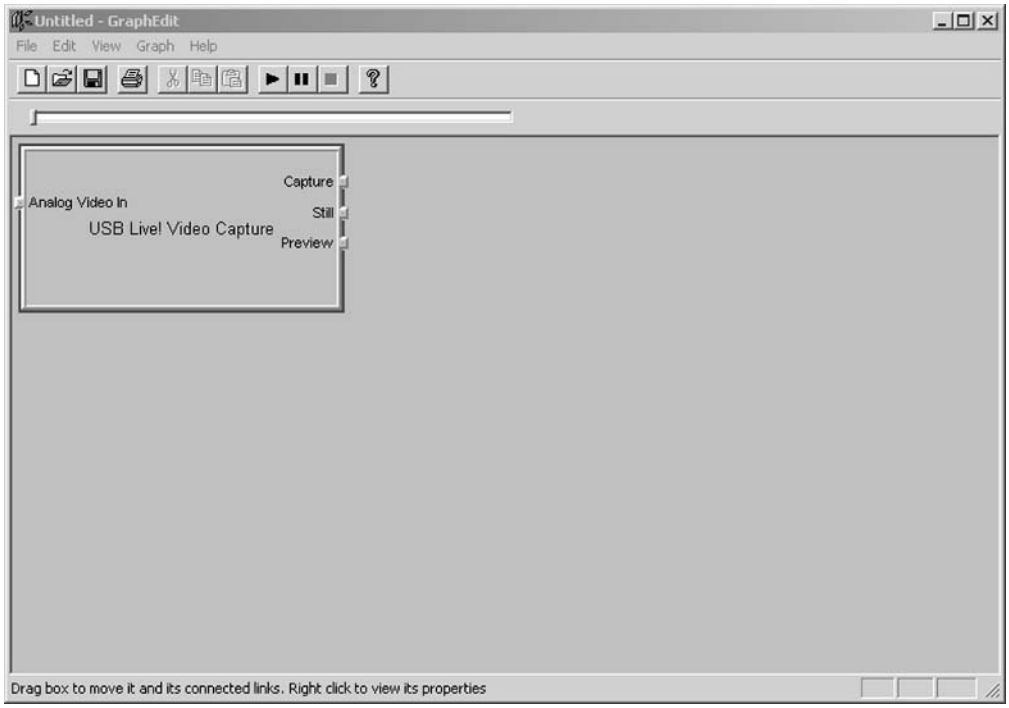
FIGURE 3.2 The list of registered filters in the local computer system.



camera by X10 and have installed the relevant driver software, a specific source filter should have been added to the COM object registry. By clicking on the `Graph` menu in the `GraphEdit` window and choosing the `Insert Filters` item, we are presented with the window shown in Figure 3.2. This window shows all the filters known to the local COM object registry. The filters are organized in categories. If we scroll down the window we will discover the category `Video Capture Sources`. We can expand this category by clicking the "+" button. Then we can see under this category that there is at least one filter named `USB Live! Video Capture`. This is the filter that performs video capture from the XCam2 USB camera. If we use a different camera, the corresponding driver software would install a different filter under the `Video Capture Sources` category. Its fundamental function though, would be the same: capture the live video stream from the specific camera.

If we highlight the `USB Live! Video Capture` filter entry and click the `Insert Filters` button, then the graphic of the XCam2 source filter will appear on the `GraphEdit` window (see Figure 3.3).

FIGURE 3.3 The graphic of the XCam2 source filter on the `GraphEdit` window.

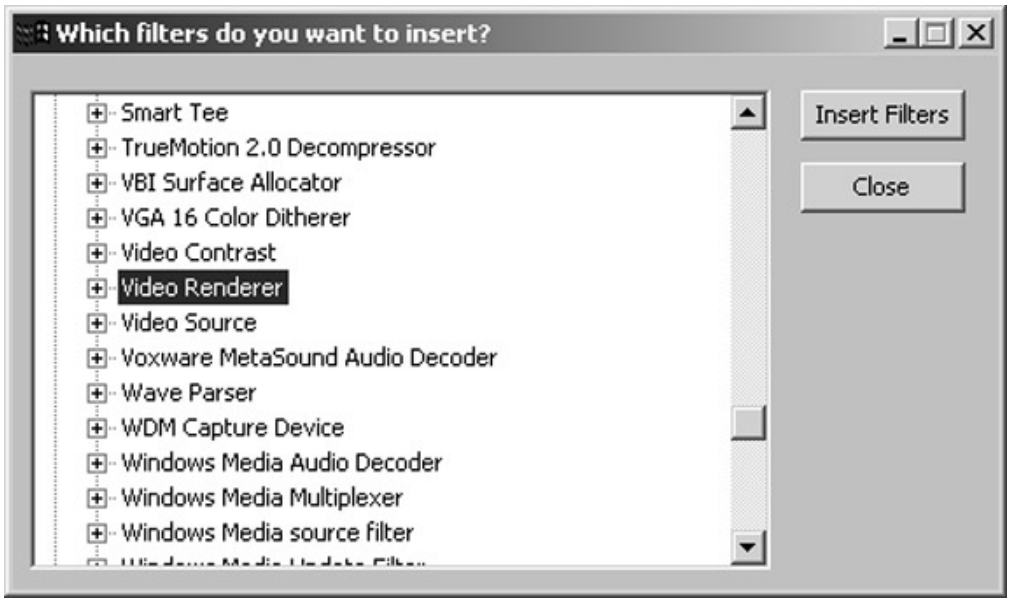


We notice immediately the three output pins of the XCam2 source filter:

1. `Capture`.
2. `Still`.
3. `Preview`.

We are primarily interested in the `Preview` pin of the source filter. This pin passes along the raw video stream to whatever input pin is connected. We need to connect a rendering filter to the `Preview` pin to have the live video stream displayed. For that we resort again to the list of filters registered in the local computer. Under the category `DirectShow Filters` we highlight the `Video Renderer` filter and we click the `Insert` button (see Figure 3.4). The

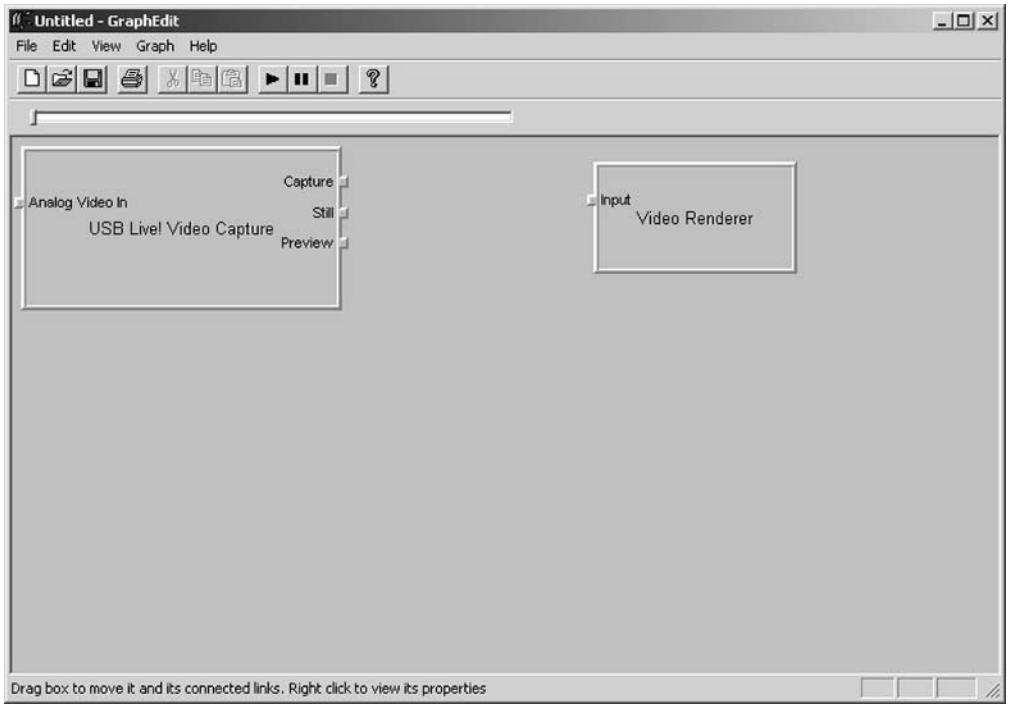
FIGURE 3.4 The video-rendering filter listed in the registered filters window.



rendering filter graphic appears on the `GraphEdit` window next to the Xcam2 source filter (see Figure 3.5). The video renderer has only one input pin, which should be connected to the output pin of the source filter.

The next step is to connect the source filter to the rendering filter, thus establishing a filter graph that will allow the preview of live video feed on the computer screen. We click with the mouse on the `Preview` pin of the source filter and drag the pointer until we meet the `Input` pin of the video-rendering filter. Then, a filter graph is formed on the `GraphEdit` window as it is shown in Figure 3.6 on page 40. The arrow lines indicate successful connections between the filters. Surprisingly, a new filter has been automatically placed between the source and rendering filter. This filter is an `AVI Decompressor` and was placed by the initiative of the `GraphEdit` manager. In Section 3.4 we will investigate why this was necessary. For now, however, let's click the play button in the toolbar of the `GraphEdit` window and enjoy the fruit of our effort. A window displaying live video from our camera is popping up, much like the one shown in Figure 3.7 on page 41.

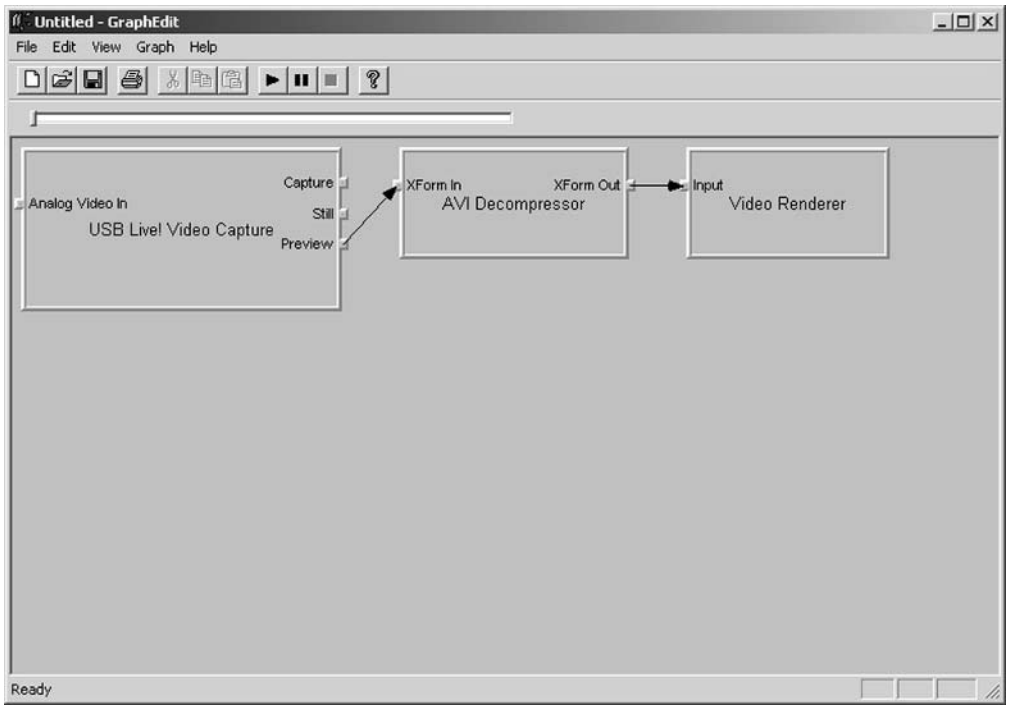
FIGURE 3.5 The source and rendering filter graphics on the `GraphEdit` window.



3.3 The Property Pages of Filters

In the previous section we have experienced just how easy it is to set up a video preview application using the `GraphEdit` utility. The question now is how flexible such an application could be. For example, can we change the brightness or the contrast levels of the incoming frames? Or, are we stuck with no choice and have to live with the default settings? To better understand the issue we can draw an analogy from the still-camera domain. A cheap still camera has nothing but a shutter button. On the other end of the spectrum, a sophisticated still camera features various controls to set up the speed, the exposure, the zoom, and a host of other properties. The DirectShow SDK as a true object-oriented paradigm mirrors the functionality that exists in the physical world to the software domain. For that, it is using a COM object called *property page*.

FIGURE 3.6 The video preview graph.



A property page is one way for a filter to support properties that the user can set. The property page also communicates valuable information regarding the type of data processed by the filter to the user. A properly written filter features the relevant property pages. Provided that the filter is registered, then it can be accessed through `GraphEdit`. In this case, the property pages of the filter can be displayed and altered through the standard Windows *Graphical User Interface* (GUI).

In the video preview graph of Figure 3.6, if we right click on the `USB Live! Video Capture` graphic and choose the `Properties...` entry from the drop-down list, we will be presented with the property page window shown in Figure 3.8. A property page can be composed out of many *property sheets*. In the `GraphEdit` framework, property sheets show as tabs. If we click the `Video Proc Amp` tab of the video capture property page, then we expose the

FIGURE 3.7 A window displaying live video from an XCam2 camera.

sheet shown in Figure 3.9. In this property sheet we can set various parameters that are related to the appearance of the incoming frames. In particular, we can interactively set values for `Brightness`, `Contrast`, `Hue`, and `Saturation` levels. For example, we can alter the `Brightness` value by moving the slider to the left or to the right. If the video preview window is active, we can watch the result of our actions live. By shifting the slider to the left (lower values), the incoming frames become darker. By shifting the slider to the right, (higher values) the incoming frames become brighter.

3.4 The AVI Decompressor

It is now time to investigate closer this mysterious object that the Filter Graph Manager interjected automatically in our filter graph.

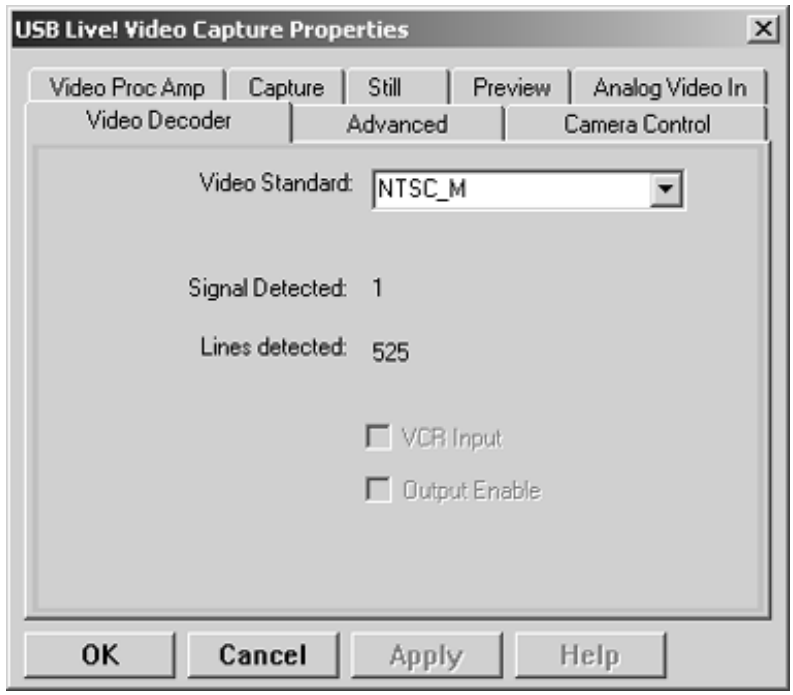


FIGURE 3.8
The property page of the XCam2 USB camera.

The best way to learn more about an object present in the `GraphEdit` window is to invoke its property page. In our case, we can do that by right clicking at the `AVI Decompressor` box. A property page appears that features two tabs:

1. The `XForm In` tab (see Figure 3.10).
2. The `XForm Out` tab (see Figure 3.11).

Each tab describes the type and format of video handled at the input and output pin of the filter correspondingly. The input pin of the AVI Decompressor receives video encoded in YUY2 at 16 bits. This is a fairly compressed format that is outputted by the X10 video camera. The problem is that the rendering filter understands only the fully uncompressed RGB format (32 bits). Therefore, the necessity of a decompressor is obvious. The AVI Decompressor decodes the YUY2-16 bit format to the RGB-32 bit format understandable

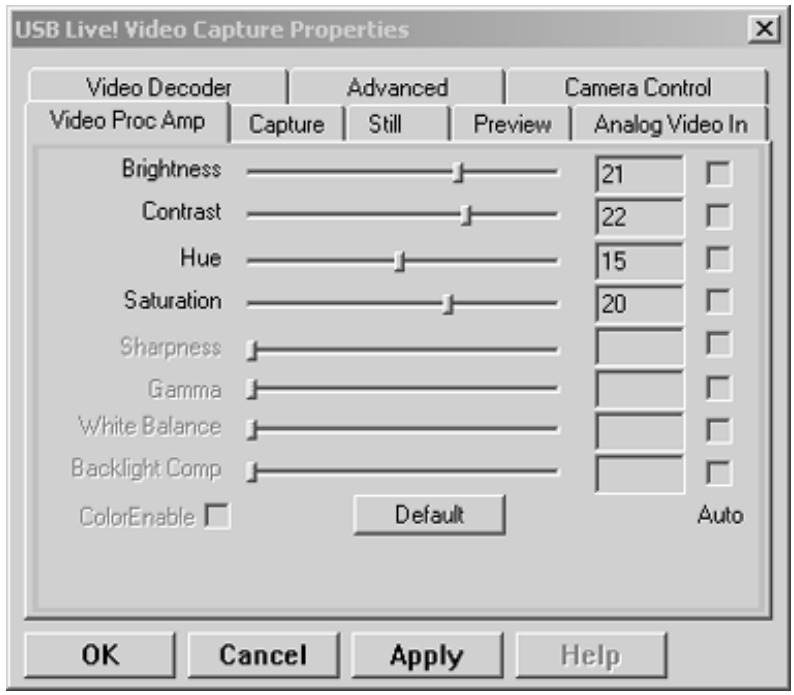


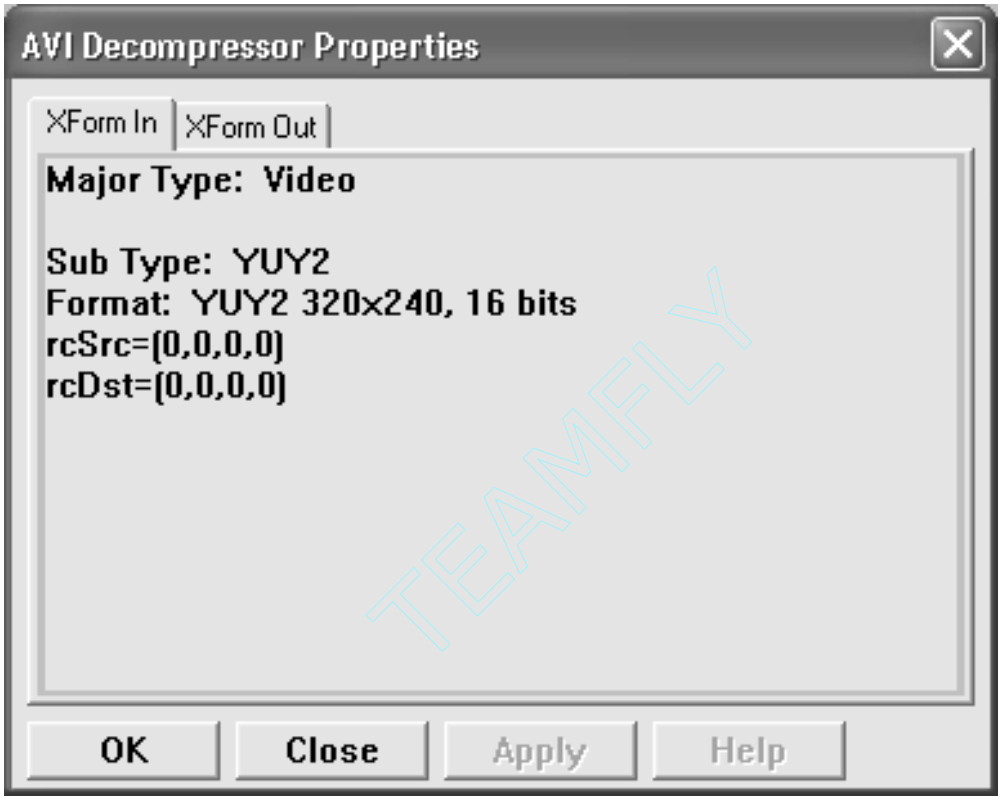
FIGURE 3.9
The property sheet corresponding to the visual properties of the incoming frames.

by the destination filter. The `XForm Out` tab of the property page (see Figure 3.11) describes the uncompressed output of the decoding filter.

The uncompressed RGB video format used by the rendering filter is associated with the color space most people understand (Red-Green-Blue). However, video capture usually works in the YUV space. The reason is that YUV takes less data to represent a similar RGB frame (compressed format)—as much as half the data. This highly efficient compression scheme facilitates real-time video transmission and capturing. YUY2 is a type of YUV encoding and is the one outputted by the X10 camera. The fact that we need to decompress the captured video frames before we display them on the computer screen is a major performance bottleneck.

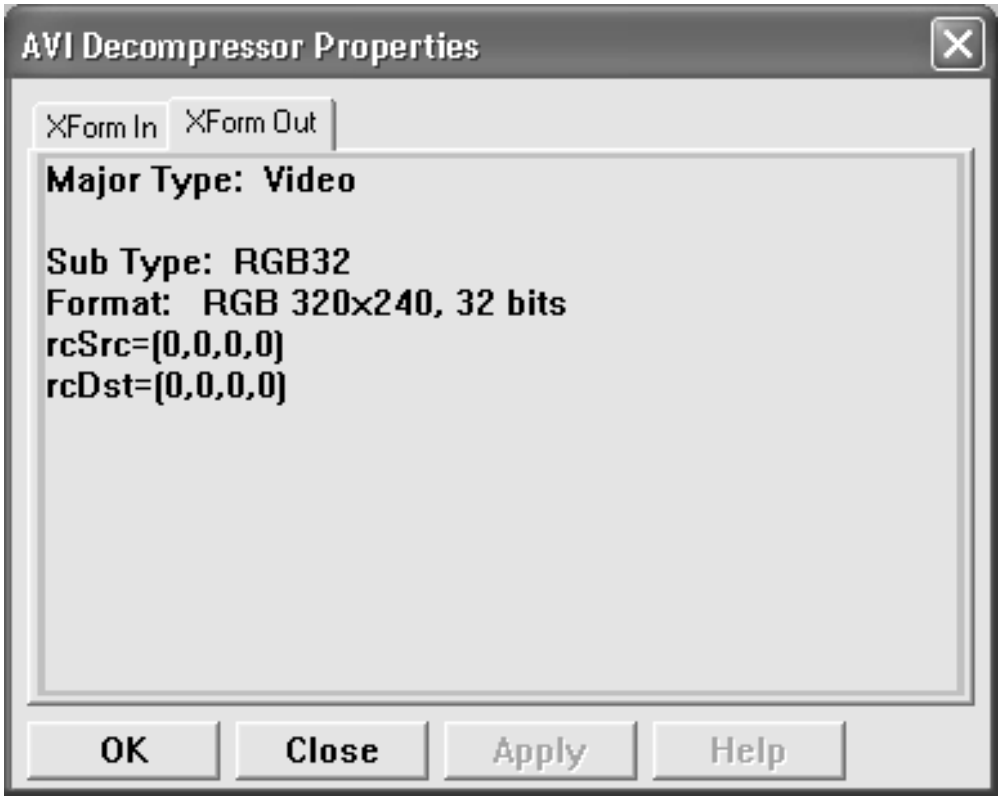
While in the RGB color space, R is linked to the Red, G to the Green, and B to the Blue channel in YUV, Y is linked to the

FIGURE 3.10 The `AVI Decompressor` property page tab describing the compressed video format outputted from the source filter.



component of luminancy, and U and V to the components of chrominancy. One of the advantages of YUV is that the chroma channels (U and V) can have a lower sampling rate than the luminance channel (Y) without a dramatic degradation of the perceptual quality. This is the property that makes compression possible in the YUV space. A notation called the A:B:C notation is used to describe how often U and V are sampled relative to Y. For example, 4:2:2 means 2:1 horizontal downsampling, with no vertical downsampling. Every scan line contains four Y samples for every two U or V samples.

FIGURE 3.11 The `AVI Decompressor` property page tab describing the uncompressed video format outputted to the rendering filter.



Actually, the YUY2 format used by X10 follows the 4 : 2 : 2 sampling scheme.

3.5 Summary

In this chapter we have built a simple but complete DirectShow application. The application acquired live video frames from a digital camera and displayed them on a window. The application was built without writing a single line of C++ code. All the filters we used were

pre-manufactured COM objects that were available through a graphical programming environment called `GraphEdit`.

There was discrepancy in the video format used by the video source filter versus the video format used by the rendering filter. The Filter Graph Manager reconciled this discrepancy with the interjection of the AVI Decompressor filter that served as the bridge between the two different video formats. In subsequent chapters we will delve into how to build our own custom filters by writing actual C++ code. We will be able to use these filters within the `GraphEdit` framework or within a more sophisticated application.

CHAPTER 4

A Custom Filter for Video Processing

Microsoft DirectShow is based on the Component Object Model (COM) technology. In this chapter we will develop step-by-step our own DirectShow object. As we mentioned in Chapter 2, COM objects in DirectShow nomenclature are called *filters*. Our filter will perform a simple video-processing operation. Specifically, it will compare incoming frames from a live video source with a reference frame. Whenever a discrepancy beyond a certain threshold is ascertained, the filter will issue a warning (change detection).

In the next chapter we will learn how to call our custom-made filter from within an application. Such an application could serve as the backbone of a video-based security system for a home or office environment. Therefore, this chapter and the next are very important, because they will teach us how to expand the functionality of DirectShow and the way to use it within an application framework. ■

4.1 A Simple Change Detection Filter

The filter we are about to build belongs to the category of *transform filters*. A transform filter takes a media input and alters it in some way. The media input in our case is video. In the broader context of DirectShow, however, the media input may be audio or something else. Our custom transform filter may derive from one of three transform base classes:

`CTransformFilter` This class is designed for implementing a transform filter with one input pin and one output pin. It uses separate allocators for the input pin and the output pin.

`CTransInPlaceFilter` This class is designed for filters that transform data in place, rather than copying the data across buffers.

`CVideoTransformFilter` This class is designed primarily as a base class for AVI decompressor filters. It is based on a "copying" transform class and assumes that the output buffer will likely be a video buffer. The main feature of this class is that it enables quality-control management in a transform filter. Quality control in this case refers to dropping of frames under certain conditions.

The more specific the transform filter from which we choose to inherit (see Figure 4.1) the less function overriding work we have to do. In the case of the *change detection* (CD) filter we don't have any reason to keep the original data intact. We are only interested in the result of the processing. The initial video frame pixel values are of no importance as soon as they are transformed. Therefore, we can transform data in place, and the natural choice for a parent class is `CTransInPlaceFilter`.

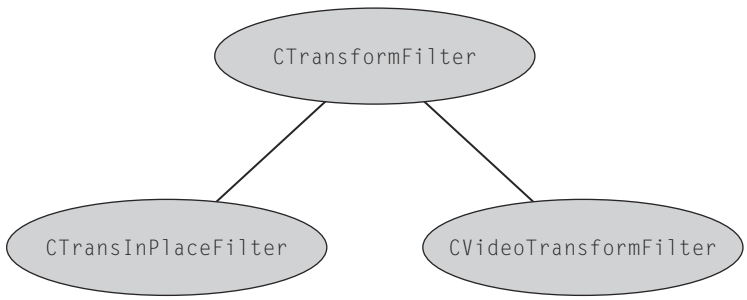


FIGURE 4.1
The inheritance tree for the transform filter family.

We will follow a disciplined approach in building the CD filter by dividing the development work into four sequential steps. These same steps will apply to the development of any custom-made transform filter. In particular, we will describe how to

1. define and instantiate our filter's class,
2. override the appropriate base class member functions,
3. access additional interfaces,
4. create the property page, and
5. create registry information.

Before we elaborate on each of the above major steps, we create a new project in the Visual Studio Developer by clicking the `New->Project` entry under the `File` menu. In the wizard window that appears we choose the `Win32 Project` template under the `Visual C++ Projects` type. We name our project `CDServer` (see Figure 4.2). Then we click the `OK` button. In the next wizard window

FIGURE 4.2 The wizard window for project definition.

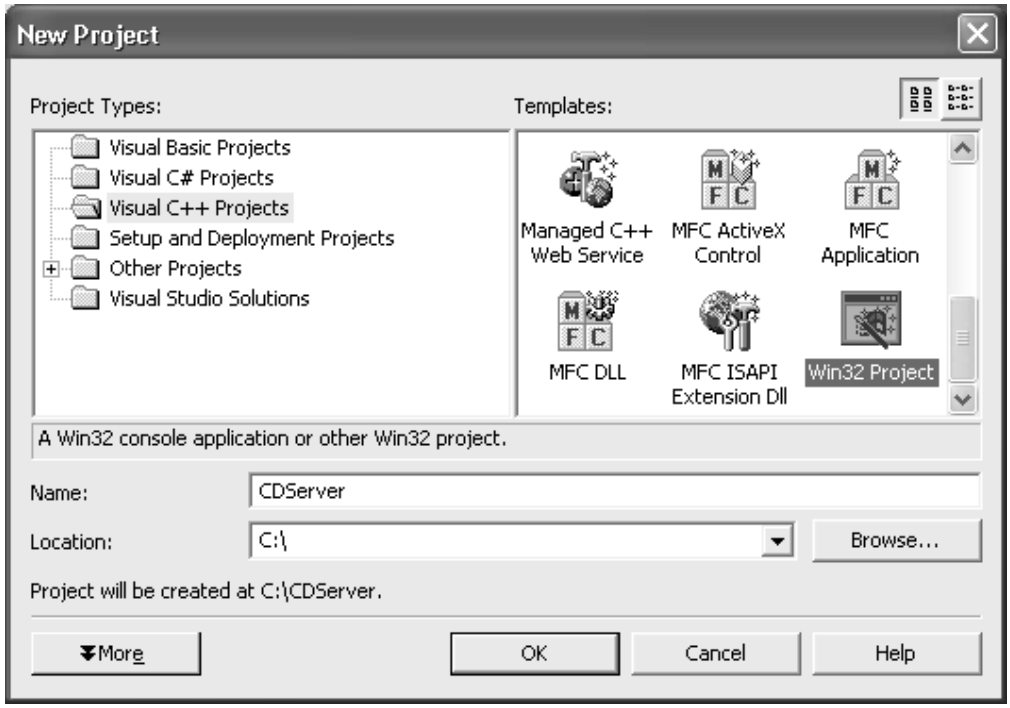
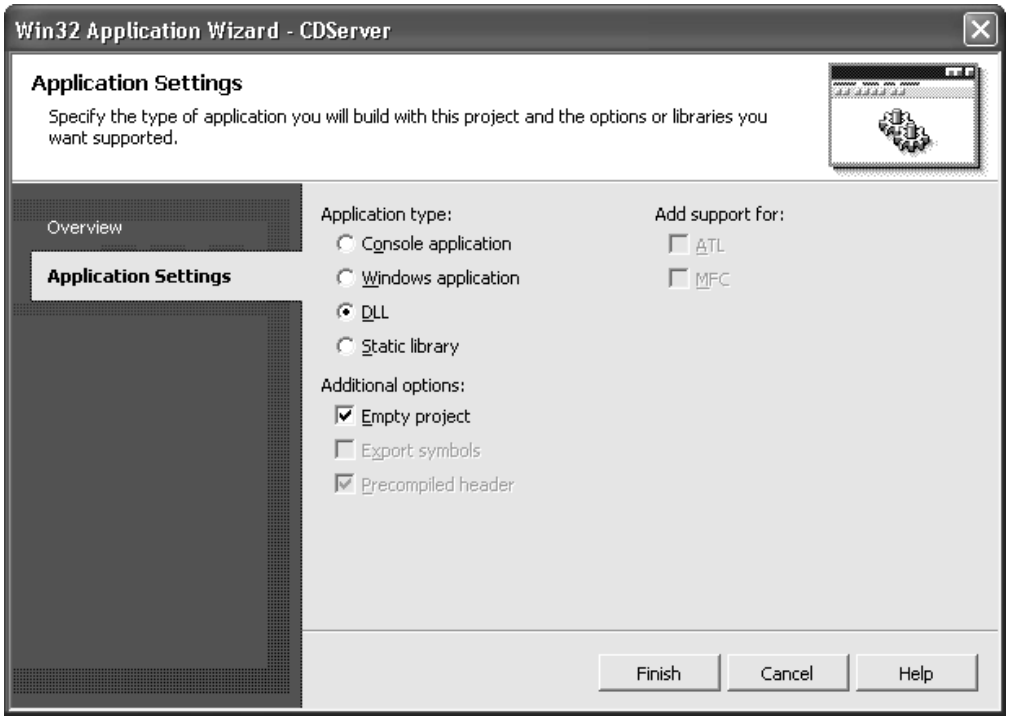


FIGURE 4.3 The wizard window for application definition.



we click on the `Applications Settings` tab and choose `DLL` under the `Application type` group and `Empty project` under the `Additional options` group (see Figure 4.3). Finally, we click the `Finish` button to create the project.

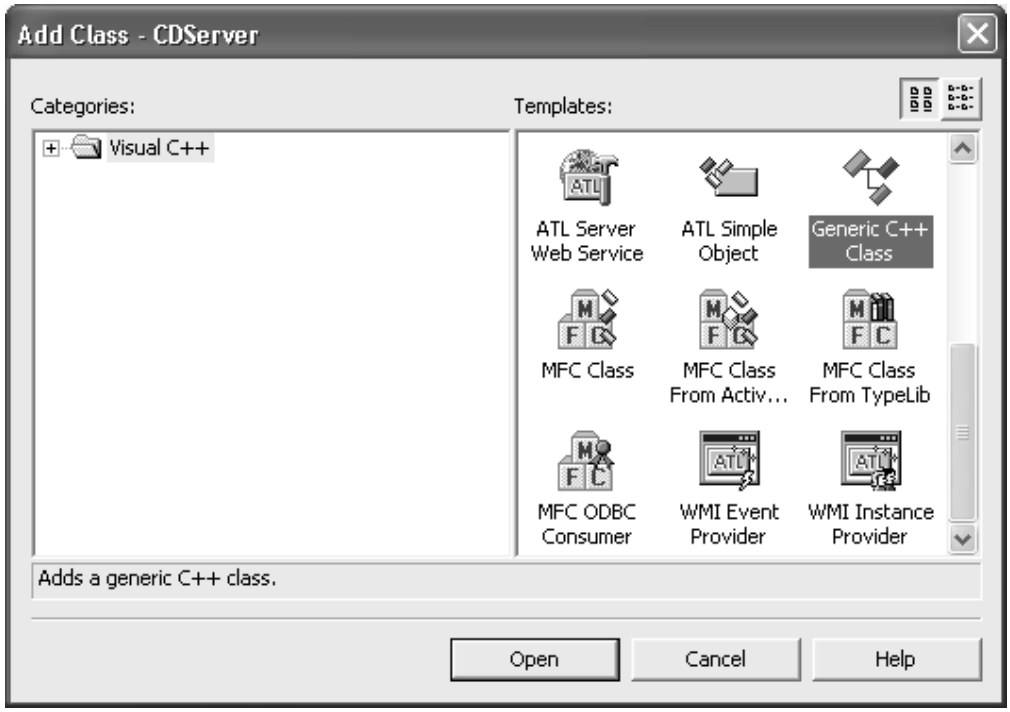
4.2 Define and Instantiate Our Filter's Class

The step of filter definition and instantiation could be further divided into smaller steps. It instills a certain discipline in our filter development effort if we follow these substeps in a specific order.

4.2.1 *Class Declaration*

Having decided from which base class we derive our CD filter, we make the appropriate class declaration. As we explained in the beginning of this chapter our CD filter should be derived from the

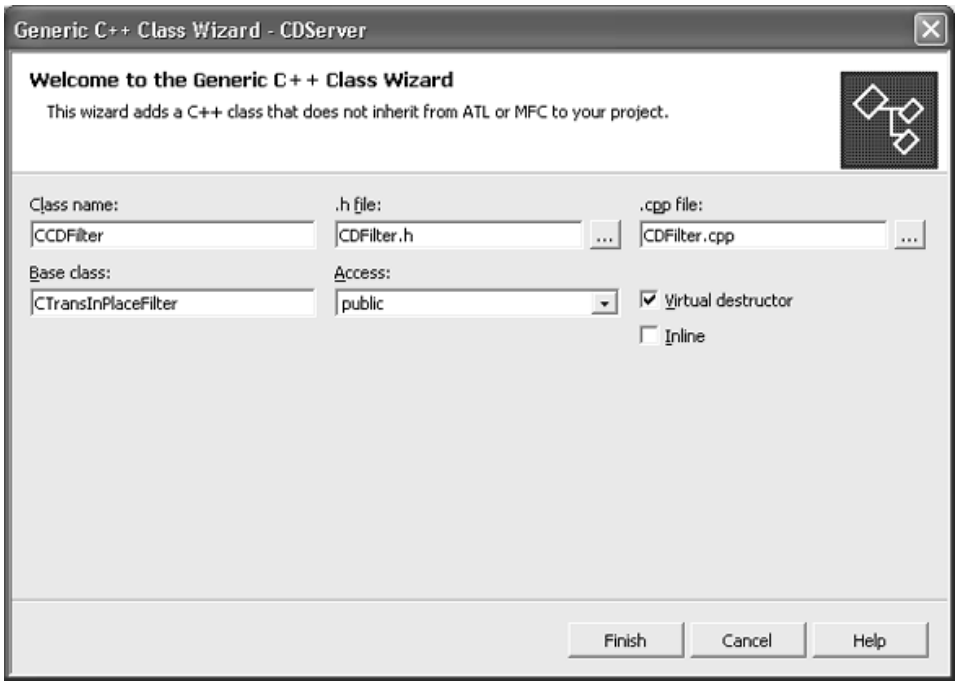
FIGURE 4.4 The class wizard window.



`CTransInPlaceFilter` base class. We right click at the `CDServer` project icon in the `Class View` pane and choose to add a new class by selecting `Add->Add Class ...` from the cascading menu. A class dialog window appears. We choose `Generic C++ Class` in the `Templates` pane (see Figure 4.4), and then we click on the `Open` button. A second class dialog window appears. Then we enter the class name and the name of the base class as shown in Figure 4.5. We also check the `Virtual destructor` option. By clicking the `Finish` button, the Developer Studio generates the class declaration and definition. We ignore an intermediate warning message about the base class `CTransInPlaceFilter`. Then we access the declaration of the constructor by right clicking at its icon in the `Class View` pane and choosing the `Go to Declaration` menu entry. We modify the declaration of the class constructor as follows:

```
CCDFilter(TCHAR *tszName, LPUNKNOWN lpUnk, HRESULT
*phr);
```

FIGURE 4.5 Creation of the `CCDFilter` class.

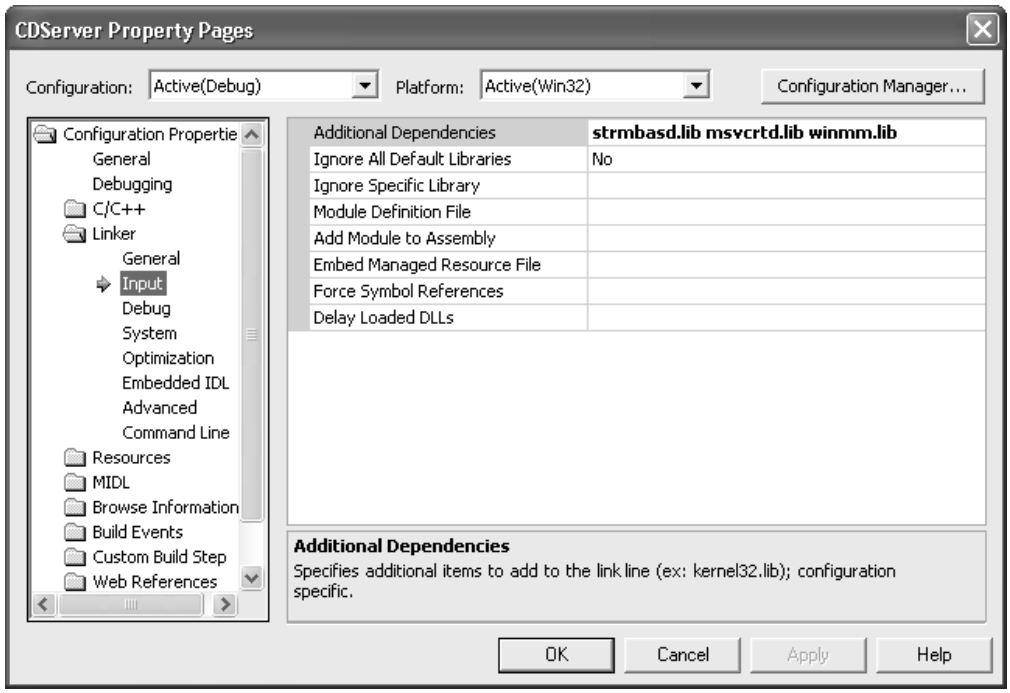


We switch to the `Solution Explorer` pane and open the *CDFilter.cpp* file. There we modify the definition of the class constructor as it appears in Listing 4.2.1.1. The initialization code will be added piece by piece as we enter the various member variables. For each variable type the Visual C++ compiler assigns an appropriate default initial value. For example, for Boolean variables the default initial value is FALSE. Whenever we need to modify the default initialization, we will comment accordingly.

LISTING 4.2.1.1 The `CCDFilter` class constructor.

```
1: // Filter constructor.
2: CCDFilter(TCHAR *tszName, LPUNKNOWN lpUnk, HRESULT *phr):
3:     CTransInPlaceFilter(tszName, lpUnk, CLSID_CDFilter, phr, TRUE)
4: {
5: }
```

FIGURE 4.6 The `Property Pages` dialog window.



In order to use base classes such as `CTransInPlaceFilter` we must include at the beginning of the *CDFilter.cpp* file the header file:

```
#include ⟨streams.h⟩
```

This header file contains all of the base class and interface definitions of the DirectShow SDK. We also need to provide the necessary linking support. In the `Class View` pane we right click at the `CDServer` icon and choose `Properties` from the drop-down menu. The `Property Pages` dialog window appears (see Figure 4.6). Under the `Linker->Input` tab we add as `Additional Dependencies` the following three libraries:

```
strmbasd.lib msvcrtd.lib winmm.lib
```

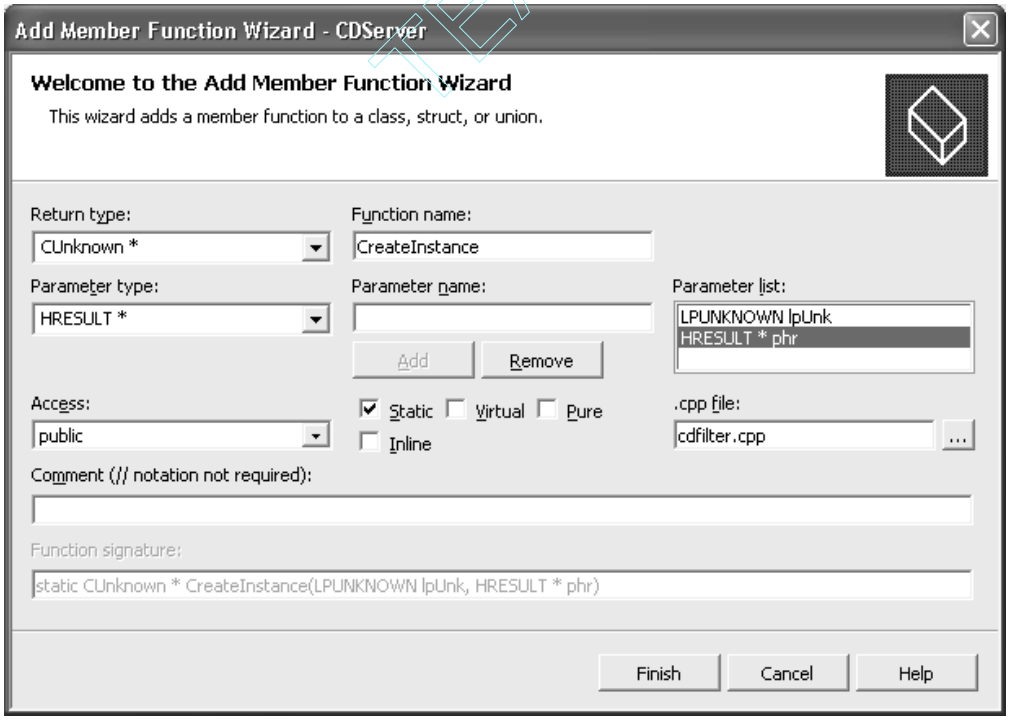
The *strmbasd.lib* library supports the DirectShow base classes. The *msvcrtd.lib* is the import library for dynamically linking the

debug version of the *msvcrt40.dll* library. The library supports both single-threaded and multithreaded applications. The *winmm.lib* library supports the multimedia services.

4.2.2 Filter Instantiation

In COM technology we cannot create objects directly. We should rather use a *class factory* to instantiate our object (filter). For that we must declare and define the `CreateInstance` member function and a means of informing the class factory as to how to access this function. In the `Class View` pane we right click the icon of the `CCDFilter` class and choose `Add->Add Function ...`. A dialog window appears (see Figure 4.7). In its `Return type` field we enter: `CUnknown *`. In the `Function name` field we enter: `CreateInstance`. Next, we add the input variables. First, we enter `LPUNKNOWN` in the `Parameter type` field and `lpUnk` in the

FIGURE 4.7 The member function dialog box for `CreateInstance`.



Parameter name field. Then we press the Add button. We repeat the same procedure for the parameter HRESULT *phr. We also check the Static check box. We conclude by clicking the Finish button. The CreateInstance member function icon is added under the CCDFilter class tree in the Class View pane. By right clicking this function icon and choosing Go to Declaration, we are placed at the function declaration. There we enter in a separate line the DECLARE_IUNKNOWN; macro. Then, by right clicking again the function icon and choosing Go to Definition, we are placed at the function definition where we add the code shown in Listing 4.2.2.1.

LISTING 4.2.2.1 The filter's CreateInstance function.

```
1:  CUnknown * CCDFilter::CreateInstance(LPUNKNOWN lpUnk,
                                         HRESULT *phr)
2:  {
3:    CCDFilter *pNewObject =
      new CCDFilter(NAME("CD Filter"),
                            lpUnk, phr);
4:     if (pNewObject == NULL)
5:     {
6:         *phr = E_OUTOFMEMORY;
7:     }
8:     return pNewObject;
9:  }
```

The CreateInstance member function calls the constructor of the CCDFilter class. In turn, the CreateInstance is called by the class factory. To communicate with the class factory, we declare a global array g_Templates of CFactoryTemplate objects (see Listing 4.2.2.2). The g_cTemplates variable (line 13, Listing 4.2.2.2) defines the number of class factory templates for the filter. In our case, we have two templates. The first template (lines 3–7, Listing 4.2.2.2) furnishes the link between COM and our filter. It provides the filter's name (line 3, Listing 4.2.2.2), its class identifier (CLSID) (line 4, Listing 4.2.2.2), and a pointer to the static CreateInstance

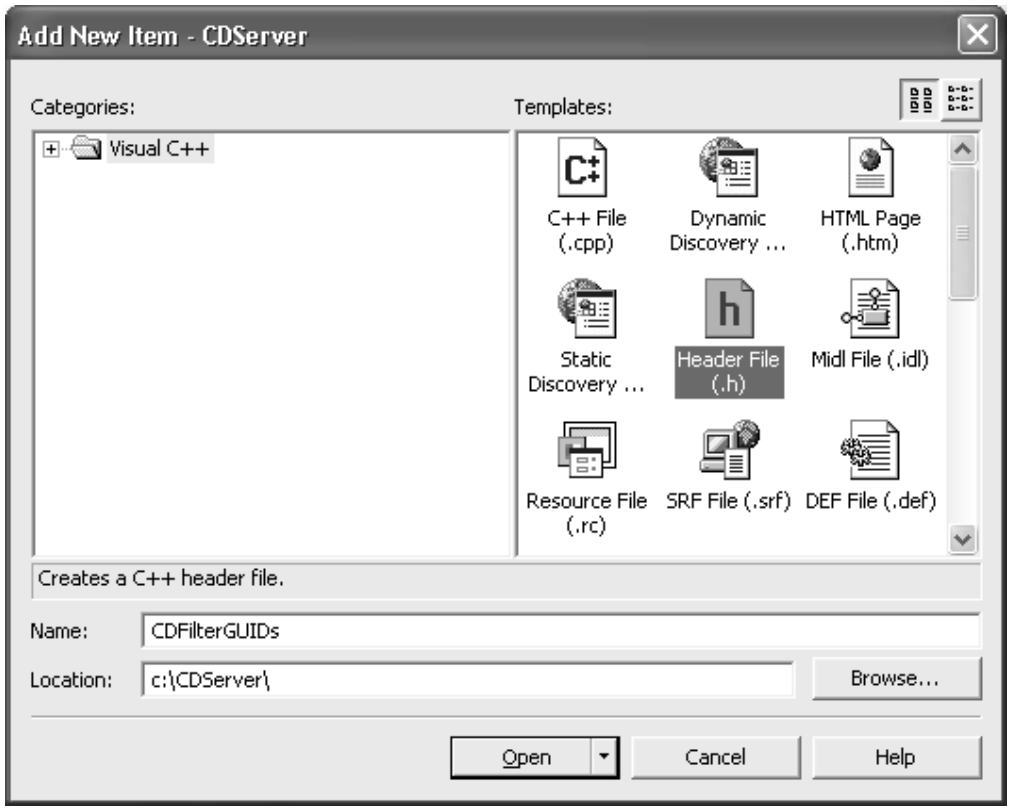
LISTING 4.2.2.2 A global array of objects to be communicated to the class factory.

```
 1:  CFactoryTemplate g_Templates[] =
 2:  {
 3:    { L"CD Filter"                    // name
 4:    , &CLSID_CDFilter                 // CLSID
 5:    , CCDFilter::CreateInstance       // creation function
 6:    , NULL
 7:    , &sudCDFilter }                  // pointer to filter information
 8:    ,
 9:    { L"CD Property Page"
10:    , &CLSID_CDPropertyPage
11:    , CCDPropertyPage::CreateInstance }
12:  };
13:  int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
```

member function that creates the filter (line 5, Listing 4.2.2.2). The second template (lines 9–11, Listing 4.2.2.2) furnishes the link to the property page for our filter.

In lines 4 and 10 of Listing 4.2.2.2 we make use of the class identifiers for the filter and its property page, respectively. We define these identifiers as follows: In Developer Studio we click the `Project->Add New Item ...` menu item. In the dialog window that appears (see Figure 4.8) we choose the `Visual C++` category and select the `Header File` icon. Then we enter *CDFilterGUIDs* in the `Name` text field, and we press the `Open` button. At this time a blank file named *CDFilterGUIDs.h* is opened. In this header file we will copy the unique class identifiers for our filter and its property page. We use the utility `GUIDGEN.EXE` to produce these identifiers. The utility can be invoked by selecting the `Tools->Create GUID` menu item. In the utility's dialog window we choose to produce unique identifiers in the `DEFINE_GUID` format by checking the second radio button (see Figure 4.9). Then we click at the `Copy` button to place the unique identifier in the clipboard. We paste the contents of the clipboard at the beginning of the *CDFilterGUIDs.h* header file. The code will look like the following example, except that it will have its own unique identifier.

FIGURE 4.8 The creation of the *CDFilterGUIDs.h* header file.




```
// F3CBD19C-5AA9-4c79-B207-8ACF0A559CFE
DEFINE_GUID(<<name>>,
0xf3cbd19c, 0x5aa9, 0x4c79, 0xb2, 0x7, 0x8a, 0xcf,
0xa, 0x55, 0x9c, 0xfe);
```

There is a generic tag `<<name>>` for the filter's CLSID. We replace this by the tag `CLSID_CDFilter`.

To generate the unique identifier for the filter's property page, we click at the `New GUID` button in the dialog window of `GUIDGEN.EXE`. Then we need to follow a similar process for declaring the GUID of the property page. However, we will talk in more detail about this in Section 4.5. Finally, we include first the standard ⟨initguid.h⟩ and then the specific *CDFilterGUIDs.h* header files at the beginning of the *CDFilter.cpp* file.

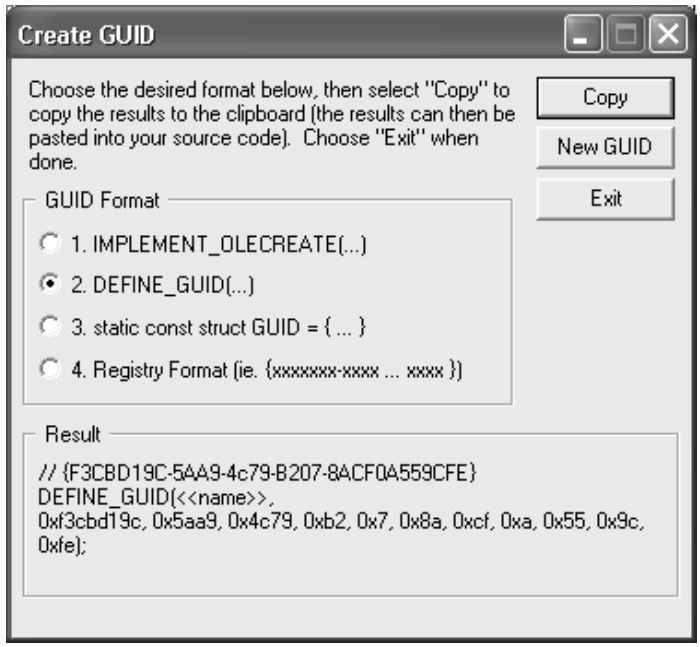


FIGURE 4.9
The dialog window of the `GUIDGEN.EXE` utility.

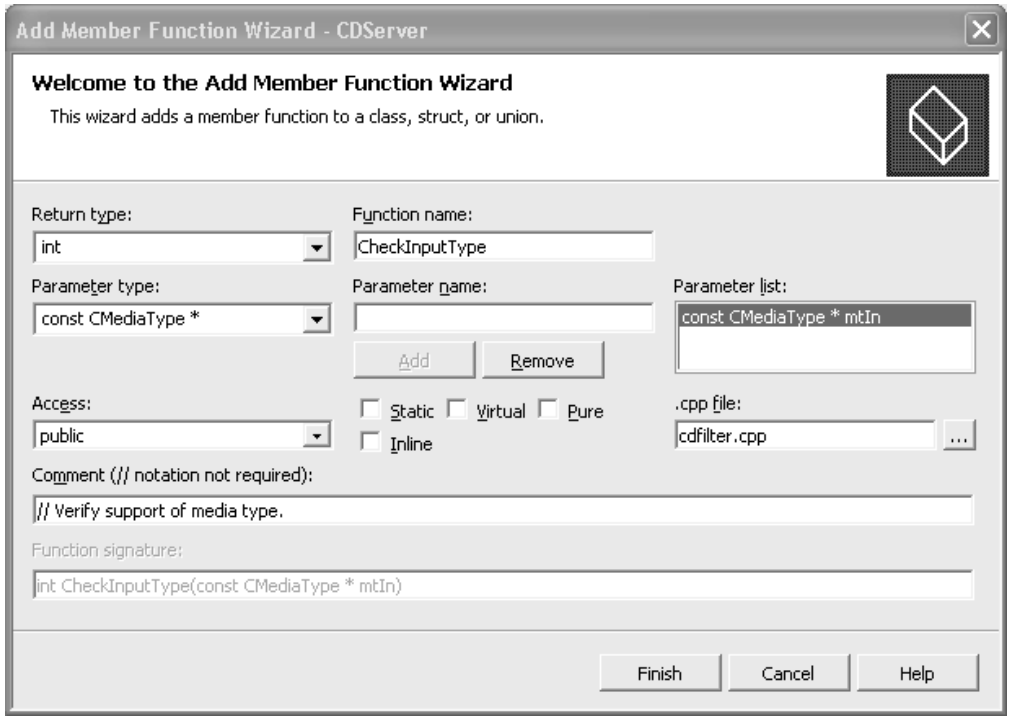
4.3 Override the Appropriate Base Class Member Functions

Since our CD filter inherits from the `CTransInPlaceFilter` base class, we need to override only two base member functions: `CheckInputType` and `Transform`.

4.3.1 *The `CheckInputType` Member Function*

We must override the `CheckInputType` member function to determine if the data arriving at the input of the CD filter is valid. We design our filter to accommodate only video media type and, in particular, the RGB 24-bit format. The allowed input media types for the filter are designated within the `AMOVIESETUP_MEDIATYPE` structure (see Section 4.6). We add the `CheckInputType` member function to our filter class by right clicking the `CCDFilter` class icon in the `Class View` pane and picking the `Add->Add Function ...` menu

FIGURE 4.10 The member function dialog box for `CheckInputType`.



entry. We fill out the relevant dialog window as shown in Figure 4.10. Then we click the `Finish` button to create the declaration and core definition of the `CheckInputType` function. We go to the definition and declaration of the function and change the return type from `int` to `HRESULT` to be compatible with the base class. Due to some incompatibility bug, .NET does not accept the `HRESULT` type as the return type of a function defined through the relevant wizard window. Finally, at the function's definition we add the code in Listing 4.3.1.1. In lines 5–7 of Listing 4.3.1.1 we make sure that the input media type is video. In lines 9–11 of Listing 4.3.1.1 we make sure that the video type is RGB, 24 bit. The latter is accomplished by using a helper function: `CheckInputSubType`. The code for the helper function is shown in Listing 4.3.1.2. We add this helper function to our filter class using the usual procedure.

LISTING 4.3.1.1 The `CheckInputType` function definition.

```
 1:  // Verify support of media type.
 2:  HRESULT CCDFilter::CheckInputType(const CMediaType *mtIn)
 3:  {
 4:    // make sure this is a video media type
 5:    if (*mtIn->FormatType() != FORMAT_VideoInfo) {
 6:      return E_INVALIDARG;
 7:    }
 8:    // can we transform this type?
 9:    if (CheckInputSubType(mtIn)) {
10:      return NOERROR;
11:    }
12:    return E_FAIL;
13:  }
```

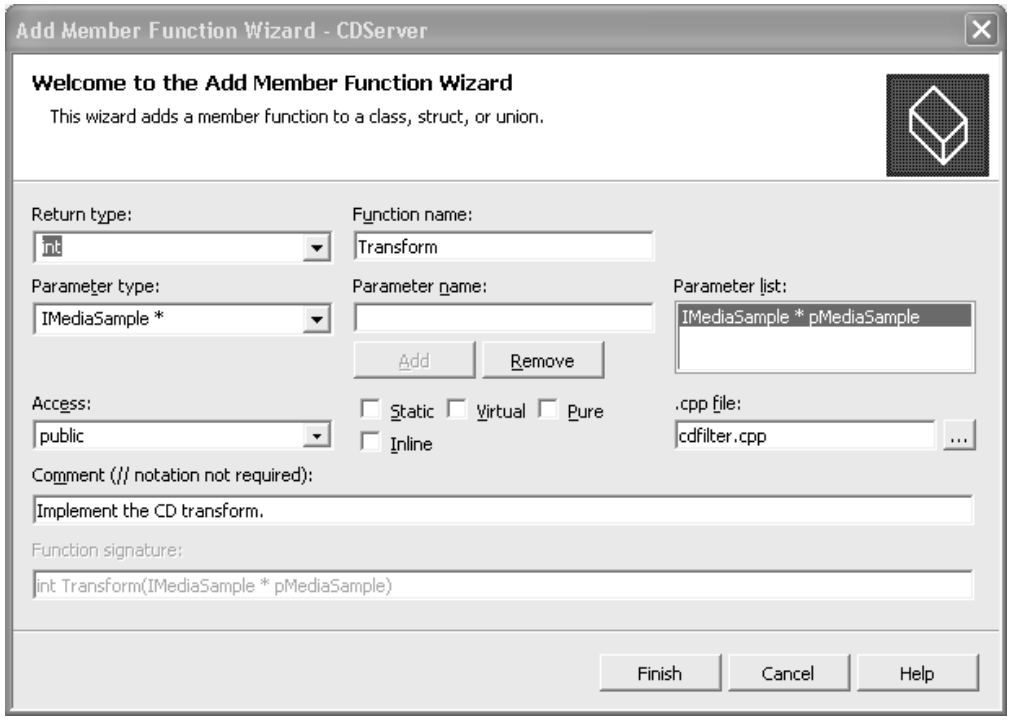
LISTING 4.3.1.2 The `CheckInputSubType` helper function definition.

```
 1:  // Verify support of media subtype.
 2:  BOOL CCDFilter::CheckInputSubType(const CMediaType *pMediaType) const
 3:  {
 4:    if (IsEqualGUID(*pMediaType->Type(), MEDIATYPE_Video)) {
 5:      if (IsEqualGUID(*pMediaType->Subtype(),
                MEDIASUBTYPE_RGB24)) {
 6:        VIDEOINFOHEADER *pvi =
                (VIDEOINFOHEADER *) pMediaType->Format();
 7:        return (pvi->bmiHeader.biBitCount == 24);
 8:      }
 9:    }
10:    return FALSE;
11: }
```

4.3.2 *The `Transform` Member Function*

Since we are building our own transform filter, we must override by definition the `Transform` member function. In our case, the overridden `Transform` function will provide the core change detection

FIGURE 4.11 The member function dialog box for `Transform`.



capability. We add the `Transform` member function to the `CCDFilter` filter class by right clicking at the class icon and picking the `Add->Add Function ...` menu entry. We fill out the relevant dialog window as shown in Figure 4.11. Then we click the `OK` button to create the declaration and core initial definition of the `Transform` function. We go to the definition and declaration of the function and change the return type from `int` to `HRESULT` to be compatible with the base class. Finally, we go to the function's definition and add the code in Listing 4.3.2.1.

To construct the transform function we need several member variables. We define these variables by right clicking the `CCDFilter` class icon and choosing the `Add->Add Variable ...` menu entry. First, we need a variable to hold the data of the designated reference frame. This is the frame from which every incoming frame is subtracted. We define the variable `m_pReferenceImage` of type `BYTE *` to play

LISTING 4.3.2.1 The `Transform` function definition.

```
 1:  HRESULT CCDFilter::Transform(IMediaSample *pMediaSample)
 2:  {
 3:    AM_MEDIA_TYPE* pType = &m_pInput->CurrentMediaType();
 4:    VIDEOINFOHEADER *pvi = (VIDEOINFOHEADER *) pType->pbFormat;
 5:    BYTE *pData;        // pointer to the data from the input pin
 6:    // get the input frame data and assign it to the data pointer
 7:    pMediaSample->GetPointer(&pData);
 8:
 9:    // first time?
10:   if(m_bInitializeFlag)
11:   {
12:       // get the image properties from the BITMAPINFOHEADER
13:       m_width  = pvi->bmiHeader.biWidth;
14:       m_height = pvi->bmiHeader.biHeight;
15:       m_colors = 3;
16:       AllocateFilterMembers();
17:       m_bInitializeFlag = FALSE;
18:   }
19:
20:   // copy the current frame into the reference frame
21:   if(m_bReferenceFlg)
22:   {
23:      for(int i=0; i<m_height; i++)
24:         for(int j=0; j<m_width; j++)
25:         {
26:             // Red
27:             *(m_pReferenceImage + 0 + 3*(j*m_height + i)) =
                 *(pData + 0 + 3*(j*m_height + i));
28:             // Green
29:             *(m_pReferenceImage + 1 + 3*(j*m_height + i)) =
                 *(pData + 1 + 3*(j*m_height + i));
30:             // Blue
31:             *(m_pReferenceImage + 2 + 3*(j*m_height + i)) =
                 *(pData + 2 + 3*(j*m_height + i));
32:         }
33:      m_bReferenceFlg = FALSE;
34:      m_bReferenceFrameSelected = TRUE;
35:   }
36:
37:   // perform change detection if a reference frame has been selected
```

```
38:   if(m_bReferenceFrameSelected && m_bRunCDFlg)
39:   {
40:       if(DifferencingThresholding(pData))
41:           m_bIntruderDetected = TRUE;
42:       ...
43:   }
44:   return NOERROR;
45: }
```

this role. We also need similar variables to hold the results of the frame differencing and thresholding operations. These variables are respectively `m_pDifferenceImage` and `m_pThresholdImage`. The frame-differencing operation captures the disparity of the current scene from the original (reference) scene. Due to the noisy video acquisition process and small light variations there is disparity even if the current scene is exactly the same as the reference scene. To avoid frequent false alarms, we apply upon the difference image a thresholding operation to eliminate small variations.

Other variables that are useful to the `Transform` function are Boolean variables to communicate the user's inputs through the Graphical User Interface (GUI). First, we define the Boolean variable `m_bInitializeFlg` that signals the very first time the `Transform` function is invoked. Then, we define the Boolean variable `m_bReferenceFlg` to signal the acquisition of a new reference frame per the user's request. We also define the Boolean variable `m_bReferenceFrameSelected` to ascertain the existence of a valid reference frame in the filter's memory. Finally, the Boolean variable `m_bRunCDFlg` denotes if the property page is open or not. We will see how all these variables play out as we describe the specifics of the `Transform` function.

In order to process information in a transform filter we first have to have a handle on this information. In the case of the CD filter the incoming media information is standard video. We get a handle on the incoming media data by invoking the `GetPointer` method from the `IMediaSample` interface (line 7, Listing 4.3.2.1). Then, we can access the byte data (`pData`) that corresponds to tricolor pixel values for each incoming frame. One problem that still hampers us, though,

is that we don't know when to stop accessing pData. In other words, we don't know the dimensions of the incoming video frames.

We can get access to the dimensions of the video frames by following a top-down approach. Our custom CD filter inherits from the CTransformFilter class. One of the protected member variables of the CTransformFilter is the m_pInput pointer to the input pin. The input pin is a class itself, and among its methods features the CurrentMediaType. Through the invocation of the CurrentMediaType method we can access the media type of the CD filter's input connection (line 3, Listing 4.3.2.1). The media type that is passed from the source to the transform filter is expressed as a structure. One of the members of the AM_MEDIA_TYPE structure is pbFormat, which is a pointer to the format structure of the media type. In our case, since we are receiving standard video frames from the source filter, the format structure of the media type is VIDEOINFOHEADER (line 4, Listing 4.3.2.1). The VIDEOINFOHEADER structure contains information for standard video. In particular, the bmiHeader member of the VIDEOINFOHEADER structure contains color and dimension information for the individual video frames. In standard video, the frames are bitmap images. In fact, bmiHeader is a BITMAPINFOHEADER structure, and among its members are biWidth and biHeight, which specify the width and height of the bitmap frames (lines 13–14, Listing 4.3.2.1). We also set the m_colors variable to 3 (line 15, Listing 4.3.2.1). This represents the number of planes for the red, green, and blue components in a standard bitmap image. In other words, each frame consists of three planes with exactly the same dimensions (biWidth x biHeight).

We use our newly acquired knowledge of the video frame's dimensions and the number of color channels to allocate space for all the variables that will hold image data (i.e., m_pReferenceImage, m_pDifferenceImage, and m_pThresholdImage). The allocation is performed by the helper function AllocateFilterMembers in line 16 of Listing 4.3.2.1.

The Transform function is invoked every time an incoming frame is fetched from the capture device. Both the determination of the frame's dimensions/color channels and the dynamic space allocation need to take place only once, when the very first frame is fetched from the video source. To assure this we flag the lines 12–16 of Listing 4.3.2.1 with the Boolean variable m_bInitializeFlag. We

also change the default initialization of `m_bInitializeFlag` from FALSE to TRUE in the class constructor.

We copy the pixel values from `pData` to `m_ReferenceImage` (lines 23–32, Listing 4.3.2.1), where `m_ReferenceImage` is a variable we defined to hold the reference image data. The reference image is the image from which every subsequent incoming image is subtracted. For a stationary camera, it represents a safe scene with no humans in the picture. The user orders the acquisition of a new reference image at a time of his choosing by clicking the `Select Reference Frame` button on the filter's property page (see Figure 4.12). The press of this button sets the `m_bReferenceFlg` flag. This means that the `if` statement in line 21 of Listing 4.3.2.1 checks true. Therefore, we can see how the orders of the user translate into a pixel-copying operation through appropriate flagging. Before we exit the reference-copying block of statements we set the flag `m_bReferenceFrameSelected` to true. This allows the subtraction and thresholding operation to occur (lines 38–43, Listing 4.3.2.1) for every subsequent frame until the user indicates to the system that he no longer wishes to perform change detection. This latter wish is communicated by closing the filter's property page.

The actual frame differencing and thresholding algorithm performed upon the image data is invoked in line 40 of Listing 4.3.2.1.

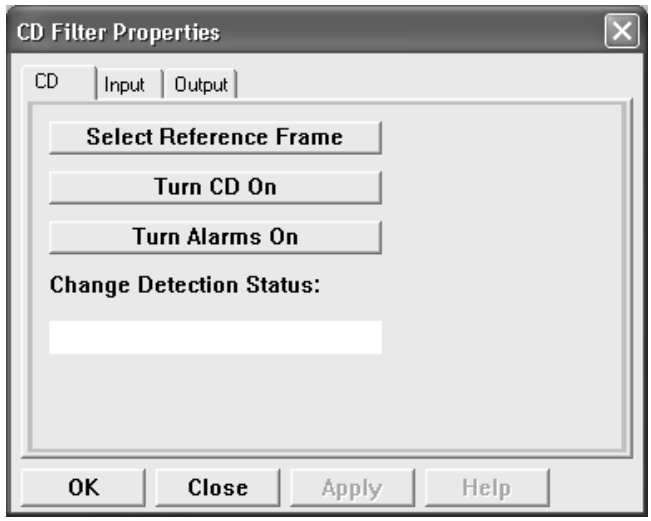


FIGURE 4.12
The property page of the CD filter.

If the threshold operation yields a sufficiently high value, then the `DifferencingThresholding` function returns as true and the following events happen simultaneously:

- The message `Intruder Detected` is displayed.
- The icon of a red light is displayed.
- An alarm sound is played.

4.3.3 *The Differencing and Thresholding Operations*

The `DifferencingThresholding` function performs the differencing and thresholding operation upon the incoming image. The most important pieces of code for the function are shown in Listing 4.3.3.1. The function carries as an input parameter the pointer `pData` to the incoming frame's pixel values. The pixel values of the reference frame have already been stored in the member variable `m_pReferenceImage`. Therefore, we are ready to perform the subtraction operation of the incoming frame from the reference frame (lines 5–13, Listing 4.3.3.1). We subtract pixel by pixel per color plane; this is the reason for the triple `for` loop in lines 5–7 of Listing 4.3.3.1. Figure 4.13 shows the way the pixel data are organized in the bitmap frame.

LISTING 4.3.3.1 The `DifferencingThresholding` function definition.

```
 1: bool CChangeFilter::DifferencingThresholding(BYTE * pData)
 2: {
 3:   ...
 4:   // compute the difference between incoming and reference images
 5:   for (i=0; i<m_height; i++)
 6:     for (j=0; j<m_width; j++)
 7:       for (k=0; k<m_colors; k++)
 8:       {
 9:           if ((*(pData + k + m_colors*(j*m_height + i))) >=
                    *(m_pReferenceImage + k + m_colors*(j*m_height + i)))
10:           {
11:              *(m_pDifferenceImage + k + m_colors*(j*m_height + i)) =
                    (BYTE)( (*(pData + k + m_colors*(j*m_height + i))) -
                    (*(m_pReferenceImage + k + m_colors*(j*m_height + i))));
12:           }
```

```
13:           ...
14:  // apply the adaptive thresholding algorithm
15:  float redThr   = GetImageThreshold(RED);
16:  float greenThr = GetImageThreshold(GREEN);
17:  float blueThr  = GetImageThreshold(BLUE);
18:
19:  // based on the computed thresholds binarize the pixel values
20:  int tally = 0;
21:  for (i=0; i<m_height; i++)
22:  {
23:    for (j=0; j<m_width; j++)
24:    {
25:        if( (*(m_pDifferenceImage + 0 + m_colors*(j*m_height + i)) >
                   ((int) redThr )   + 25)  ||
               (*(m_pDifferenceImage + 1 + m_colors*(j*m_height + i)) >
                   ((int) greenThr ) + 25)  ||
               (*(m_pDifferenceImage + 2 + m_colors*(j*m_height + i)) >
                   ((int) blueThr )  + 25))
26:        {
27:          // index back to the original image pixels
28:          *(m_pThresholdImage + 0 + m_colors*(j*m_height + i))  =
29:                 (*(pData + 0 + m_colors*(j*m_height + i)));
30:          *(m_pThresholdImage + 1 + m_colors*(j*m_height + i))  =
31:                 (*(pData + 1 + m_colors*(j*m_height + i)));
32:          *(m_pThresholdImage + 2 + m_colors*(j*m_height + i))  =
33:                 (*(pData + 2 + m_colors*(j*m_height + i)));
34:          tally = tally + 1;
35:        }
36:        else
37:        {
38:          *(m_pThresholdImage + 0 + m_colors*(j*m_height + i)) =
39:          *(m_pThresholdImage + 1 + m_colors*(j*m_height + i)) =
40:          *(m_pThresholdImage + 2 + m_colors*(j*m_height + i)) =
               (BYTE) 0;
41:        }
42:    }
43:  }
44:  ...
45:  // is intrusion detected (a "large enough" difference was found)?
46:  if ( ( 100.0 * ((float)tally) / ((float)(m_width*m_height*m_colors)))
          > m_ThresholdValue)
47:  ...
```

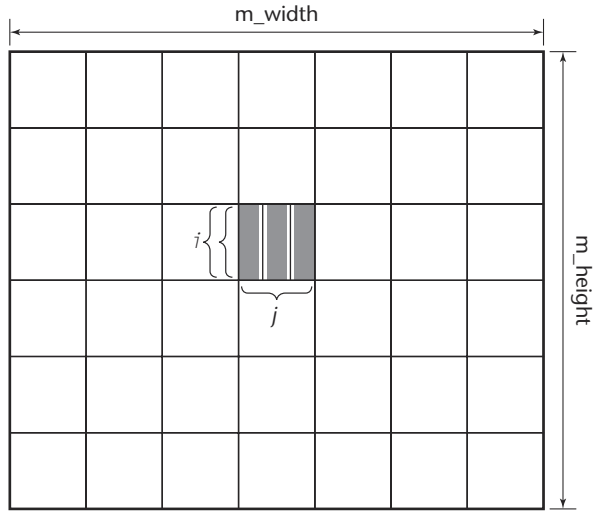


FIGURE 4.13
Organization of the bitmap frame. Each pixel (*i*, *j*) packs three byte numbers that represent the constituent primary colors for the pixel (red, green, and blue).

After having obtained the difference image we apply the thresholding algorithm upon it (lines 15–17, Listing 4.3.3.1). Actually, we apply the thresholding operation per each color plane (red, green, and blue). Three different threshold values are produced, one for the red, one for the green, and one for the blue components of the pixel values. Then, in line 25 of Listing 4.3.3.1, we weigh each color to see if any of the color values of a pixel is above or below the corresponding threshold value. If it is above the threshold value, then we maintain the original pixel value in the threshold image (lines 28–33, Listing 4.3.3.1). If it is below the threshold value, then we zero the pixel value in the threshold image—black pixel (lines 38–40, Listing 4.3.3.1). This weighing process repeats for every pixel in the image (lines 21–24, Listing 4.3.3.1).

In line 34 of Listing 4.3.3.1 we keep count of the number of pixels that have a color component above the corresponding threshold. Then, in line 46 we check if the percentage of the pixels found to be sufficiently different exceeds a certain overall percentage threshold. The overall percentage threshold (`m_Threshold`) is set by us. In the default version of the code it has been set to 0.5, which means that if more than 50% of the image's pixels have changed sufficiently from the reference image, an alarm is set off. The reader may set this variable higher or lower depending on how sensitive he/she prefers the change detection system to be.

One point of great interest that we have left unanswered so far is how, exactly, the threshold values `redThr`, `greenThr`, and `blueThr` are computed for the three color planes of the image. We are about to give an answer to this question by dissecting the function `GetImageThreshold` in the following section.

4.3.4 *The Thresholding Algorithm*

Thresholding offers a method of segmentation in image processing. We are trying to delineate foreground from background pixels in tricolor-difference images. We are less interested in background pixels, and for this reason we depict them as black. For the foreground pixels, however, we maintain the exact color values included in the incoming image. Thus, if, for example, a human or other object has moved in the original scene, then all the scene appears black in the thresholded image except the region where the human or object exists. This human or other silhouette represents the change that was introduced to the original reference scene.

But how does thresholding determine if a pixel belongs to the foreground or background? Or, equivalently, if it should be painted black or maintain its original color? In color images such as the ones we are dealing with in our case, three separate thresholds can be established for the corresponding color channels. Each color channel has a range of values between [0 – 255], where 0 represents the absence of color and 255 represents full color. In an ideal world, with no light changes and without sensor noise, the difference image would not need thresholding at all. The difference pixels in the regions of the scene that haven't changed would cancel out completely. In the real world, however, the difference pixels corresponding to unchanged scene points may not cancel out completely and present nonzero values. Still, the difference pixels that correspond to scene points that have drastically changed due to the presence of a foreign object usually present higher residual values.

Therefore, we have two distributions of color pixels on each color plane of the difference image two distributions of color pixel values. One distribution is clustered toward the lower portion of the intensity range [0 – 255] and represents color pixel values that correspond to background points. The other distribution is clustered toward the higher portion of the intensity range [0 – 255] and represents color pixel values that correspond to foreground points. There

is often some overlapping between the background and foreground distributions. A good thresholding algorithm locates the demarcation (thresholding) point in a way that minimizes the area of one distribution that lies on the other side's region of the threshold [7].

It is very important to realize that we know only the parameters of the total pixel distribution per color channel. We assume that the background and foreground distributions exist within it. We don't know exactly what they are. We are trying to guess by computing a value that separates them (threshold). As we adjust the threshold, we increase the spread of one distribution and decrease the other. Our goal is to select the threshold that minimizes the combined spread.

We can define the *within-class* variance $\sigma_w^2(t)$ as the weighted sum of variances of each distribution.

$$\sigma_w^2(t) = w_1(t)\sigma_1^2(t) + w_2(t)\sigma_2^2(t) \tag{4.3.1}$$

where

$$w_1(t) = \sum_{i=1}^{t} P(i) \tag{4.3.2}$$

$$w_2(t) = \sum_{i=t+1}^{N} P(i) \tag{4.3.3}$$

$$\sigma_1^2(t) = \text{the variance of the pixels in the background distribution (below threshold)} \tag{4.3.4}$$

$$\sigma_2^2(t) = \text{the variance of the pixels in the foreground distribution (above threshold)} \tag{4.3.5}$$

The weights $w_1(t)$ and $w_2(t)$ represent the probabilities of the background and foreground distributions, respectively. These probabilities are computed as the sum of the probabilities of the respective intensity levels. In turn, the individual intensity probabilities $P(i)$ are computed as the ratio of the number of pixels bearing the specific intensity to the total number of pixels in the scene. We symbolize the number of intensity levels by N. Since the range of intensity values per color channel is $[0 - 255]$, the total number of intensity values is $N = 256$.

In Listing 4.3.4.1 we compute the number of pixels for each specific intensity in the range [0 – 255]. This is the histogram of the color channel. Based on the histogram, we compute the individual intensity probabilities in lines 17–18 of Listing 4.3.4.1.

LISTING 4.3.4.1 The `GetImageThreshold` function definition.

```
 1:  // Adaptive thresholding algorithm.
 2:  int CChangeFilter::GetImageThreshold(short int color) {
 3:  ...
 4:  switch (color)
 5:  {
 6:    case RED:
 7:      for (i=0; i<m_height; i++)
 8:        for (j=0; j<m_width; j++)
 9:        {
10:          colorj = *(m_DifferenceImage + 0 + m_colors*(j*m_height + i));
11:          hgram[colorj] += 1;
12:        }
13:        ...
14:  }
15:
16:  // compute the probability P for each pixel intensity value
17:  for (i=0; i<256; i++)
18:    P[i] =  (hgram[i]) / ((float) (m_width*m_height));
19:
20:  // total mean value
21:  float mu = 0.0;
22:  for (i=0; i<256; i++)
23:    mu += ((float) (i+1)) * P[i];
24:  ...
25:  for (k=i+1; k<256; k++)
26:  {
27:      w1 += P[k];
28:      MU1 += (k+1) * P[k];
29:      if ( (w1 <= 0) || (w1 >= 1))
30:      {
31:      }
32:      else
33:      {
```

```
34:         ftemp = mu * w1 - MU1;
35:         sigma_B_sq = (ftemp * ftemp) / (float) (w1 * (1.0 - w1));
36:         if ( sigma_B_sq > sigma_B_sqr_max )
37:         {
38:           sigma_B_sqr_max = sigma_B_sq;
39:           k_thresh = k;
40:         }
41:       }
42:   }
43:
44:   return k_thresh;
45:   }
```

If we subtract the within-class variance $\sigma_w^2(t)$ from the total variance σ^2 of the pixel population, we get the between-class variance $\sigma_b^2(t)$:

$$\begin{aligned}\sigma_b^2(t) &= \sigma^2(t) - \sigma_w^2(t) \\ &= w_1(\mu_1 - \mu)^2 + w_2(\mu_2 - \mu)^2\end{aligned} \tag{4.3.6}$$

where μ_1 is the mean of the background pixel distribution, μ_2 is the mean of the foreground pixel distribution, and μ is the total mean. The means can be computed by the following equations:

$$\mu_1(t) = \mathcal{M}_1(t)/w_1(t) \tag{4.3.7}$$

$$\mathcal{M}_1(t) = \sum_{i=1}^{t} iP(i) \tag{4.3.8}$$

$$\mu_2(t) = \mathcal{M}_2(t)/w_2(t) \tag{4.3.9}$$

$$\mathcal{M}_2(t) = \sum_{i=t+1}^{N} iP(i) \tag{4.3.10}$$

$$\mu(t) = \sum_{i=1}^{N} iP(i) \tag{4.3.11}$$

We use Equation (4.3.11) to compute the total mean in lines 22–23 of Listing 4.3.4.1. By observing carefully Equation (4.3.6), we

notice that the between-class variance is simply the weighted variance of the distribution means themselves around the overall mean. Our initial optimization problem of minimizing the within-class variance $\sigma_w(t)$ can now be cast as maximizing the between-class variance $\sigma_b(t)$. We substitute Equations (4.3.7)–(4.3.11) in Equation (4.3.6) to obtain

$$\sigma_b^2(t) = \frac{(w_1(t)\mu(t) - \mathcal{M}_1(t))^2}{w_1(t)(1 - w_1(t))}. \tag{4.3.12}$$

For each potential threshold value t ($t \in [0 - 255]$) we compute the weight (probability) of the background distribution w_1 and the mean enumerator $\mathcal{M}_1$. We use these values to compute the between-class variance for every pixel intensity t. Then, we pick as the optimal threshold value t_{opt} the value that yields the maximum σ_b^2. This sounds like a lot of work, but fortunately it can be formulated as a recursive process. We can start from $t = 0$ and compute incrementally w_1 and $\mathcal{M}_1$ up to $t = 255$ by using the following recursive equations:

$$w_1(+1) = w_1(t) + P(t + 1), \tag{4.3.13}$$

$$\mathcal{M}(t + 1) = \mathcal{M}(t) + (t + 1)P(t). \tag{4.3.14}$$

We employ Equations (4.3.13) and (4.3.14) in lines 27–28 of Listing 4.3.4.1 to compute w_1 and $\mathcal{M}_1$ incrementally at each step. Based on these values and the value of the total mean μ computed once in lines 21–23 of Listing 4.3.4.1, we calculate the between-class variance in lines 34–35 by straight application of Equation (4.3.6). We compare the current step value of the between-class variance with the maximum value found up to the previous step in line 36 of Listing 4.3.4.1. As a result of this comparison, we always store away the maximum between-class variance value along with the intensity value (potential threshold value) at which it occurred (lines 38–39, Listing 4.3.4.1). When we exhaust the full range of the intensity values (`for` loop—line 25 in Listing 4.3.4.1) the `GetImageThreshold` function returns the intensity value that produced the maximum between-class variance (line 44 in Listing 4.3.4.1). This is the threshold value that separates foreground from background pixels for the particular color channel (RED, GREEN, or BLUE).

4.4 Access Additional Interfaces

We create a new header file named *iCDFilter.h* through the `Project-> Add New Item ...` menu selection. This is the file where we will declare our own interface, the `ICDFilter`. This interface will be of value to the `CCDFilter` class and possibly other similar classes that we may design in the future. Its role is to cover more specific functionality that is not covered by the base DirectShow classes.

First, we generate the GUID for the `ICDFilter` interface by using the `Tools->CreateGUID` menu option. Then, we go ahead and write the interface declaration as in Listing 4.4.0.1.

LISTING 4.4.0.1 The declaration of the `ICDFilter` custom interface.

```
 1:  DECLARE_INTERFACE_(ICDFilterInterface, IUnknown)
 2:  {
 3:     STDMETHOD(IDisplayCDStatus) (THIS_
 4:                 HWND *Whdlg
 5:                 ) PURE;
 6:
 7:     STDMETHOD(IManageCD) (THIS_
 8:                 BOOL flgValue
 9:                 ) PURE;
10:
11:     STDMETHOD(IGetReferenceFrame) () PURE;
12:
13:     STDMETHOD(IManageAudioAlarm) () PURE;
14:  };
```

The `IDisplayCDStatus` method displays the status of the CD algorithm. The interface description is given in Table 4.1. The description is quite general and leaves significant latitude to the COM object that will be implementing the interface method. The mandate is for the method to display if the CD algorithm has detected a foreign object in the scene or not. The display can take a number of forms (textual, graphical, and audible) but the interface contract

TABLE 4.1 Interface contract for the `IDisplayCDStatus` member function.

`ICDFilter::IDisplayCDStatus`	
Parameters	
`Whdlg`	Handle on the property dialog window.
Return Value	
Remarks	
This method displays the status of the CD filter at every point in time. The status could indicate either an alert or a safe situation. The alert corresponds to the detection of a foreign object in the original scene. The display of the status may include a textual, graphical, and audible sign.	

does not specify if some or all should be used. It also does not specify on which window the display should take place.

Listing 4.4.0.2 shows the implementation of the `IDisplayCDStatus` pure virtual method in the `CCDFilter` class. In this particular implementation we have opted to convey the status of the CD algorithm by employing all three modes: textual, graphical, and audible. The method determines the status of the detection algorithm by checking the Boolean variable `m_bIntruderDetected`. This variable is set in the `Transform` method. When it is `TRUE`, the algorithm has detected a foreign object in the scene and the statements 15–26 of Listing 4.4.0.2 are executed. In line 16 we print the textual message "Intruder Detected" in the designated window. In line 22 we draw a red circular region that indicates potential danger. We complement the previous two alerts with an audible alert in line 26. This is an annoying ringing sound that is stored in the *Warning.wav* file. For this reason we have another Boolean variable, the `m_bAlarmsOnFlg` that controls if the sound file will be played or not (line 25). In case no foreign object is detected the Boolean variable `m_bIntruderDetected` is `FALSE`, and the statements 30–37 of Listing 4.4.0.2 are executed. This time we print the textual message `All Clear` in line 31. We also draw a green circular region in line 37 to indicate a safe scene.

LISTING 4.4.0.2 The definition of the `IDisplayCDStatus` function of the `ICDFilter` interface.

```
 1: // Display the status of the CD filter.
 2: STDMETHODIMP CCDFilter::IDisplayCDStatus(HWND* hdlg)
 3: {
 4:    CAutoLock cAutolock(&m_ICDFilterInterfaceLock);
 5:
 6:    // set filter pointers
 7:    if(m_bFirstWarningCall)
 8:    {
 9:        m_Whdlg = hdlg;
10:        m_bFirstWarningCall = FALSE;
11:     }
12:
13:     if(m_bIntruderDetected)
14:     {
15:         // display a text message
16:         Edit_SetText(GetDlgItem(*hdlg, ID_CD_STATUS_EDIT),
                         "Intruder Detected");
17:
18:         // set the status light to red
19:         HDC hdc = GetDC(*hdlg);
20:         HBRUSH brush = CreateSolidBrush(0X000000FF);
21:         SelectObject(hdc, brush);
22:         Ellipse(hdc,212,100,232,120);
23:
24:        // play an audible alert
25:        if(m_bAlarmsOnFlg == TRUE)
26:            PlaySound("..\\Warning.wav", NULL, SND_FILENAME);
27:     }
28:     else
29:     {
30:        // display a text message of the status
31:        Edit_SetText(GetDlgItem(*hdlg, ID_CD_STATUS_EDIT),
                        "All Clear");
32:
33:        // set the status light to green
34:        HDC hdc = GetDC(*hdlg);
35:        HBRUSH brush = CreateSolidBrush(0X0000FF00);
36:        SelectObject(hdc, brush);
37:        Ellipse(hdc,212,100,232,120);
```

```
38:     }
39:
40:     return NOERROR;
41: }
```

LISTING 4.4.0.3 The definition of the `IManageCD` function of the `ICDFilter` interface.

```
1: // Manage the operation of the CD algorithm.
2: STDMETHODIMP CCDFilter::IManageCD(BOOL flgValue)
3: {
4:     CAutoLock cAutolock(&m_ICDFilterInterfaceLock);
5:
6:     m_bRunCDFlg = flgValue;
7:
8:     return NOERROR;
9: }
```

The `IManageCD` method manages the operation of the CD algorithm. In other words, the filter may be active (CD algorithm running) or inactive (CD algorithm stopped). The interface contract does not specify the management scheme (see Table 4.2). In the implementation of the `CCDFilter` class, the method sets the

TABLE 4.2 Interface contract for the `IManageCD` member function.

`ICDFilter::IManageCD`	
Parameters	
`flgValue`	Boolean flag indicating the CD availability.
Return Value	
Remarks	
This method manages the operation of the CD algorithm.	

LISTING 4.4.0.4 The definition of the `IGetReferenceFrame` function of the `ICDFilter` interface.

```
1: // Manage the acquisition of the reference frame.
2: STDMETHODIMP CCDFilter::IGetReferenceFrame(void)
3: {
4:    CAutoLock cAutolock(&m_ICDFilterInterfaceLock);
5:
6:    m_bReferenceFlg = TRUE;
7:
8:    return NOERROR;
9: }
```

Boolean member variable `m_bRunCDFlg` to the value of the incoming `flgValue`. If the member variable is set to `TRUE`, then the condition in line 38 of Listing 4.3.2.1 holds and the statements associated with the CD algorithm are executed (lines 40–42). If the member variable is set to `FALSE`, then the condition in line 38 of Listing 4.3.2.1 does not hold and we have only a trivial execution of the `Transform` function (without the CD part).

The `IGetReferenceFrame` method manages the acquisition of the reference frame. The interface contract does not specify the management scheme (see Table 4.3). In the implementation of the `CCDFilter` class, the method sets to `TRUE` the member Boolean variable `m_bReferenceFlg`. This flag is checked within the `Transform` function (lines 21–35, Listing 4.3.2.1) and controls the storage of the current frame bytes into the `m_pReferenceImage` variable.

TABLE 4.3 Interface contract for the `IGetReferenceFrame` member function.

`ICDFilter::IGetReferenceFrame`
Parameters
Return Value
Remarks
This method manages the acquisition of the reference frame.

TABLE 4.4 Interface contract for the `IManageAudioAlarm` member function.

`ICDFilter::IManageAudioAlarm`
Parameters
`flgValue` Boolean flag indicating the audio alarm availability.
Return Value
Remarks
This method manages the operation of the audio alarm.

The `IManageAudioAlarm` method manages the operation of the audio alarm. There are varying degrees to which people are annoyed by audible alarms. Therefore, the interface provides control of the on/off function of the audio alarm to fit individual taste. In the case no audible alarm mode is used in the filter, there is no reason to implement this pure virtual function. But, we have chosen to employ an audible alarm mode when we implemented the interface function `IDisplayCDStatus`. Consequently, we follow up with an implementation of the `IManageAudioAlarm` method to give the choice of either turning on or shutting off the audio alarm operation at will. Again, the interface contract (see Table 4.4) for the function does not specify the management mechanism. But, as we did in the implementation of the `IManageCD` function, we choose to control the audio alarm through the setting and unsetting of a member Boolean variable. In the implementation of the `IManageAudioAlarm` function we treat the `m_bAudioAlarmFlg` as a toggle. If it is `TRUE`, we switch it to `FALSE` (lines 6–7, Listing 4.4.0.5) and vice versa.

LISTING 4.4.0.5 The definition of the `IManageAudioAlarm` function of the `ICDFilter` interface.

```
1: // Manage the operation of the audio alarm.
2: STDMETHODIMP CCDFilter::IManageAudioAlarm(void)
3: {
4:    CAutoLock cAutolock(&m_ICDFilterInterfaceLock);
5:
```

```
 6:    if(m_bAudioAlarmFlg == TRUE)
 7:        m_bAudioAlarmFlg = FALSE;
 8:    else
 9:        m_bAudioAlarmFlg = TRUE;
10:
11:    return NOERROR;
12: }
```

4.5 Create the Property Page

So far in this chapter we have described the `CCDFilter` class, which is the class corresponding to the CD filter we are building. We have also described the `ICDFilter` interface, which is the custom interface implemented by the `CDFilter` COM object. What we are missing is a way for the filter to communicate with the user. This is exactly what a property page can provide. The property page of a filter is a dialog window that allows access to the custom properties of the filter. In our case, the property page contains GUI objects (e.g., buttons) that serve as invocation devices for the filter's custom interface methods. We take the property page functionality a step further and use part of the dialog window to display the status of the CD algorithm.

We declare the property page class `CDPropertyPage` for the CD filter as in Figure 4.14. We derive this class from the `CBaseProperty-Page` class. We access the declaration of the class constructor and modify it as follows:

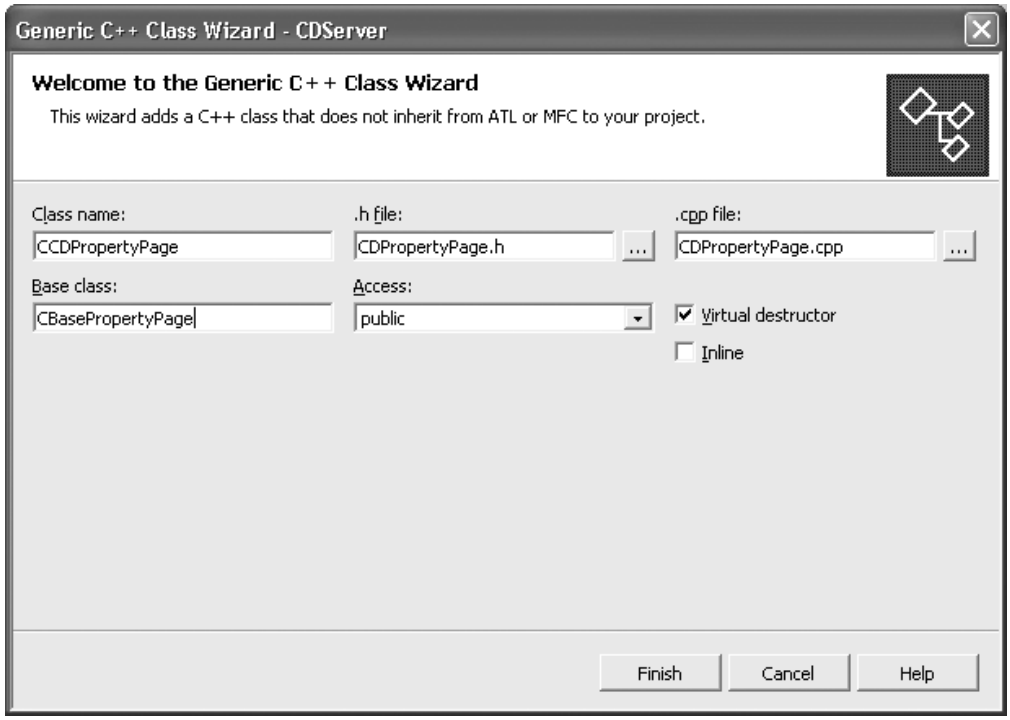
```
CCDPropertyPage(LPUNKNOWN lpUnk, HRESULT *phr);
```

Then, we access and modify the definition of the class constructor as it appears in Listing 4.5.0.1.

LISTING 4.5.0.1 The `CCDPropertyPage` class constructor.

```
1: // Property page constructor.
2: CCDPropertyPage::CCDPropertyPage(LPUNKNOWN lpUnk, HRESULT *phr) :
3:    CBasePropertyPage(NAME("CD Filter Property Page"),
                        lpUnk,IDD_CDPROPERTYPAGE,IDS_TITLE)
4: {
5: }
```

FIGURE 4.14 Creation of the `CCDPropertyPage` class.



The `IDD_CDPROPERTYPAGE` is the ID of the dialog window we add to our project to serve as its property page. The dialog window can be added by right clicking the `CDServer` project icon in the `Class view` pane and selecting `Add->Add Resource...`. The `IDS_TITLE` is the ID of the string table resource we add to our project. We add this resource by following a course of action similar to the case of dialog window.

The property page, like the `CDFilter` itself, are COM objects. Therefore, we must declare its GUID. In the *CDFilterGUIDs.h* file we add a declaration similar to the one below by using the `Tools->CreateGUID` utility.

```
// A744CF3A-C3BC-46fe-8BC9-3735F1B67A6F
DEFINE_GUID(CLSID_CDPropertyPage,
0xa744cf3a, 0xc3bc, 0x46fe, 0x8b, 0xc9, 0x37, 0x35,
0xf1, 0xb6, 0x7a, 0x6f);
```

Since our filter's property page is a COM object we cannot create it directly. Therefore, we should use a class factory to instantiate our property page much the same way we did for the CD filter in Section 4.2. We create a `CreateInstance` member function in class `CDPropertyPage`. We have already created the means of informing the class factory as to how to access this function through the `g_Templates` global array (see Listing 4.2.2.2). The code for the `CCDPropertyPage::CreateInstance` member function is shown in Listing 4.5.0.2. It is very similar to the code of the `CCDFilter::CreateInstance` in Listing 4.2.2.1.

LISTING 4.5.0.2 The filter's `CreateInstance` function.

```
 1: CUnknown * WINAPI CCDPropertyPage::CreateInstance(LPUNKNOWN lpUnk,
                                                      HRESULT * phr)
 2: {
 3:     CUnknown *pNewObject =
 4:     new CCDPropertyPage(lpUnk, phr);
 5:     if (pNewObject == NULL)
 6:     {
 7:         *phr = E_OUTOFMEMORY;
 8:     }
 9:     return pNewObject;
10: }
```

The `CreateInstance` member function calls the constructor of the `CCDPropertyPage` class. In turn, the `CreateInstance` is called by the class factory. The second template of the `g_Templates` global array in lines 9–11 of Listing 4.2.2.2 links the class factory to the property page of our filter.

One of the first things that we need to create on the property page is a way for the user to select a reference frame. As we have already explained in Section 4.3.2, the reference frame depicts a static scene without human presence (safe environment). Every subsequent frame is subtracted from the incoming frame, and if a substantial change is ascertained, an alarm is issued. Listing 4.5.0.3 cites the code for the `CreateReferenceButton` member function that

LISTING 4.5.0.3 The `CreateReferenceButton` member function.

```
 1: // Create the button for selecting a new reference frame.
 2: HWND CCDPropertyPage::CreateReferenceButton(HWND hwndParent)
 3: {
 4:    HWND ReferenceButton;
 5:
 6:    // styles for the button
 7:    ULONG Styles = WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON;
 8:
 9:    // create the button
10:    ReferenceButton = CreateWindow("BUTTON",
11:                 "Select Reference Frame",
12:                 Styles,
13:                 10, 10,
14:                 200, 20,
15:                 hwndParent,
16:                 (HMENU)ID_REFERENCE_BUTTON,
17:                 g_hInst,
18:                 NULL);
19:
20:    // return the button that we have created
21:    return ReferenceButton;
22: }
```

relates to the reference button in the property page. The button is a window itself. Therefore, we create it with the `CreateWindow` function in line 10. The `CreateWindow` function specifies the window class (line 10), window label (line 11), window style (line 12), and the initial position and size of the window (lines 13–14). The function also specifies the window's parent or owner (line 15). Another important parameter that is specified is the child window identifier in line 16. In our case, this is represented by the symbolic constant `ID_REFERENCE_BUTTON`. The reference button, along with all the other GUI items that we are adding, are child windows to the property page window, which acts as the parent. Each child window identifier should be unique in the context of its family. We define the symbolic constants that represent the child windows of the property page at

the *CPropertyPage.h* header file. A sample definition is as follows:

```
#define ID_REFERENCE_BUTTON 10000
#define ID_ALARM_BUTTON 10001
#define ID_CD_STATUS_EDIT 10002
#define ID_CD_ON_OFF_BUTTON 10003
```

As we will see, the child window identifiers are useful in pinpointing the GUI devices that were chosen by the user. The label we choose for the reference button (line 11, Listing 4.5.0.3) is `Select Reference Frame`.

A very important GUI item that we need to create on the property page is a button to activate/deactivate the CD algorithm. By default, the CD algorithm is off and the filter simply passes along the video stream. Once we turn the CD algorithm on, if a reference frame has been selected, the algorithm performs the change detection operation and alters the original incoming frames. The code for creating the activation button for the CD algorithm is shown in Listing 4.5.0.4 and is very similar to the code in Listing 4.5.0.3. We label this button `Turn CD On` in line 11. We also assign as its identifier the symbolic constant `ID_CD_ON_OFF_BUTTON` in line 16.

LISTING 4.5.0.4 The `CreateCDOnOffButton` member function.

```
 1: // Create the CD On/Off button.
 2: HWND CCDPropertyPage::CreateCDOnOffButton(HWND hwndParent)
 3: {
 4:   HWND CDButton;
 5:
 6:   // styles for the button
 7:   ULONG Styles = WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON;
 8:
 9:   // create the button
10:   CDButton = CreateWindow("BUTTON",
11:                   "Turn CD On",
12:                   Styles,
13:                   10, 40,
14:                   200, 20,
15:                   hwndParent,
16:                   (HMENU)ID_CD_ON_OFF_BUTTON,
```

```
17:                     g_hInst,
18:                     NULL);
19:
20:    // return the button that we have created
21:    return CDButton;
22: }
```

We create one more push button that allows us to control the operation of an audio alarm. When the CD algorithm is on and there is human presence, the audio alarm will sound. The audio is off by default. However, we may elect to turn it on by pressing the audio alarm button. The code for creating the audio alarm button is similar to the code of the reference and CD buttons and is given in Listing 4.5.0.5. In line 11 we assign to the button the label `Turn Alarms On`. In line 16 we assign as the identifier for the button the symbolic constant `ID_ALARM_BUTTON`.

LISTING 4.5.0.5 The `CreateAudioAlarmButton` member function.

```
 1: // Create the button for turning on/off the audio alarm.
 2: HWND CCDPropertyPage::CreateAudioAlarmButton(HWND hwndParent)
 3: {
 4:   HWND AudioAlarmButton;
 5:
 6:   // styles for the button
 7:   ULONG Styles = WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON;
 8:
 9:   // create the button
10:   AudioAlarmButton = CreateWindow("BUTTON",
11:                     "Turn Alarms On",
12:                     Styles,
13:                     10, 70,
14:                     200, 20,
15:                     hwndParent,
16:                     (HMENU)ID_ALARM_BUTTON,
17:                     g_hInst,
18:                     NULL);
19:
20:    // return the button that we have created
21:    return AudioAlarmButton;
22: }
```

LISTING 4.5.0.6 The `CreateCDStatusEdit` member function.

```
 1: // Create the CD status edit box.
 2: HWND CCDPropertyPage::CreateCDStatusEdit(HWND hwndParent)
 3: {
 4:   HWND StatusEdit;
 5:
 6:   // styles for the edit box
 7:   ULONG Styles = WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON;
 8:
 9:   // create the edit box
10:   StatusEdit = CreateWindow("EDIT",
11:                    "",
12:                    Styles,
13:                    10, 130,
14:                    200, 20,
15:                    hwndParent,
16:                    (HMENU)ID_CD_STATUS_EDIT,
17:                    g_hInst,
18:                    NULL);
19:
20:    // return the edit box that we have created
21:    return StatusEdit;
22: }
```

We also need to create an edit box to communicate the status of change detection as it is reported by the CD algorithm (Listing 4.5.0.6). We set the class atom parameter of the `CreateWindow` function to `EDIT` instead of `BUTTON` (line 10 of Listing 4.5.0.6). This is the most noticeable change from the code pattern that we used for the reference, CD, and alarm buttons. The symbolic constant that identifies the edit box is `ID_CD_STATUS_EDIT` (line 16). We use the empty string to label the edit box in line 11. This is consistent with the fact that the CD algorithm is off by default, and, therefore, there is no state to report initially. If we select a reference frame and turn the CD algorithm on by pressing the corresponding buttons, then we flash on the edit box either `Intruder Detected` or `All Clear`, depending on the scene status. We will explain the mechanism by which these messages are flashed on the edit box later in this section.

LISTING 4.5.0.7 The `CreateLabel` member function.

```
 1: // Create label for the CD edit box.
 2: HWND CCDPropertyPage::CreateLabel(HWND hwndParent)
 3: {
 4:   HWND Label;
 5:
 6:   // styles for the label
 7:   ULONG Styles = WS_CHILD | WS_VISIBLE;
 8:
 9:   // create the label
10:   Label = CreateWindow("STATIC",
11:                "Change Detection Status:",
12:                Styles,
13:                10, 100,
14:                200, 20,
15:                hwndParent,
16:                NULL,
17:                g_hInst,
18:                NULL);
19:
20:    // return the button that we have created
21:    return Label;
22: }
```

To improve user friendliness, we create an accompanying label for the edit box to highlight its role (Listing 4.5.0.7). The class atom parameter we use for the `CreateWindow` function this time is `STATIC` to indicate that this is a static text box and not an edit box. The static text message we choose to assign to the box is `Change Detection Status:` (line 11, Listing 4.5.0.7). We place the static text box strategically above the edit box to serve as its label (lines 13–14 in Listing 4.5.0.7 versus lines 13–14 in Listing 4.5.0.6). It is worth noting that the symbolic constant identifier in line 16 of Listing 4.5.0.7 is this time `NULL`. Since this is a static text box we cannot alter it in any way, and after creating it we do not expect to call upon it.

Regarding the property page, the sequence of events starts when it is connected to the filter. Upon connection, the member function `OnConnect` is called. Actually, `OnConnect` is a pure virtual function

LISTING 4.5.0.8 The `OnConnect` member function.

```
 1: // It is called when the property page is connected to the filter.
 2: HRESULT CCDPropertyPage::OnConnect(IUnknown * pUnknown)
 3: {
 4:   ASSERT(m_pICDFilter == NULL);
 5:
 6:   HRESULT hr = pUnknown->QueryInterface(IID_ICDFilter,
                      (void **) &m_pICDFilter);
 7:   if (FAILED(hr))
 8:   {
 9:       return E_NOINTERFACE;
10:   }
11:
12:   ASSERT(m_pICDFilter);
13:
14:   // get the initial properties
15:   m_pICDFilter->IDisplayCDStatus(&m_Dlg);
16:
17:   return NOERROR;
18: }
```

that is a member of the `CBasePropertyPage` class. We override this function as shown in Listing 4.5.0.8. The first action we take is to request a pointer to the interface of the filter with which the property page is associated. In our case, this is the `ICDFilter` interface, and we are obtaining its pointer through the `QueryInterface` method in line 6 of Listing 4.5.0.8. The interface pointer opens our way to accessing the `IDisplayCDStatus` method (line 15, Listing 4.5.0.8). We pass as an input parameter to the `IDisplayCDStatus` method a pointer to the member variable `m_Dlg`, which is the handle to the property page window (line 15, Listing 4.5.0.8). Please note that `m_Dlg` is an inherited member variable from the base class `CBasePropertyPage`. This initial call of the `IDisplayCDStatus` interface function initializes certain objects associated with the property page window (e.g., alarm audio and icon) but activates nothing. Activation is made possible only after we open the property page and the member function `OnActivate` is called.

Initially, the property page is connected to the filter, but is still inactive. This is the case when the graph to which the filter belongs is operational, but we have not opened the filter's property page yet. The moment we open up the property page window, the member function `OnActivate` is called. This is another pure virtual function associated with the `CBasePropertyPage` class, which we override as shown in Listing 4.5.0.9. Our main coding action here is to set the member variable `m_bIsInitialized` to `TRUE`, thus signaling that the property page has been activated.

LISTING 4.5.0.9 The `OnActivate` member function.

```
1: // It is called when the property page is activated.
2: HRESULT CCDPropertyPage::OnActivate(void)
3: {
4:   m_bIsInitialized = TRUE;
5:
6:   return NOERROR;
7: }
```

Once the property page is connected to the filter and activated, it starts receiving messages that are intercepted by the `OnReceiveMessage` member function. At the very moment of creation of the property page, the system emits the message `WM_INITDIALOG`. This matches the first case in the code of the `OnReceiveMessage` function (Listing 4.5.0.10). The respective action is the creation of all the buttons, edit boxes, and labels invoked by the functions we have described in Listings 4.5.0.3–4.5.0.7.

LISTING 4.5.0.10 The first part of the `OnReceiveMessage` member function.

```
1: // It is called when a message is sent to the property page dialog box.
2: BOOL CCDPropertyPage::OnReceiveMessage(HWND hwnd, UINT uMsg,
                          WPARAM wParam, LPARAM lParam)
3: {
4:    switch (uMsg)
```

```
 5:    {
 6:       // creation of the property page
 7:       case WM_INITDIALOG:
 8:       {
 9:          // create the label
10:          CreateLabel(hwnd);
11:
12:          // create a button for selecting the reference frame
13:          m_ReferenceButton = CreateReferenceButton(hwnd);
14:          ASSERT(m_ReferenceButton);
15:
16:          // create a button for turning the CD algorithm on/off
17:          m_CDOnOffButton = CreateCDOnOffButton(hwnd);
18:          ASSERT(m_CDOnOffButton);
19:
20:          // create a button for turning the audio alarm on/off
21:          m_AudioAlarmButton = CreateAudioAlarmButton(hwnd);
22:          ASSERT(m_AudioAlarmButton);
23:
24:          // create an edit box for displaying the CD status
25:          m_CDStatusEdit = CreateCDStatusEdit(hwnd);
26:          ASSERT(m_CDStatusEdit);
27:
28:           return (LRESULT) 1;
29:       }
30:       ...
```

After initialization, every time we select a command item from the property page window, a `WM_COMMAND` item is emitted that is intercepted by the `OnReceiveMessage` function. As we have described earlier in this section, there are three buttons on the property page window that we can press to communicate our wishes to the CD filter. These are the reference, the CD, and the audio alarm buttons. We will describe here the code that handls the CD button (Listing 4.5.0.11). Similar logic applies to the other two. When a `WM_COMMAND` is issued, the `wParam` input parameter of the `OnReceiveMessage` holds the identifier for the GUI control (button) that was depressed. If this identifier happens to belong to the CD button (line 8, Listing 4.5.0.11), then we examine if the CD algorithm is active or inactive (line 11, Listing 4.5.0.11) and take appropriate action. If we find that the CD

LISTING 4.5.0.11 The continuation of the `OnReceiveMessage` member function.

```
 1:        ...
 2:        // messages sent from the property page dialog items
 3:        case WM_COMMAND:
 4:        {
 5:           ...
 6:
 7:           // if the CD on/off button is pressed
 8:           if(LOWORD(wParam) == ID_CD_ON_OFF_BUTTON)
 9:           {
10:               // if the CD is currently on
11:               if(m_bPPRunCDFlg == TRUE)
12:               {
13:                   // turn it off
14:                   SetWindowText(m_CDOnOffButton,"Turn CD On");
15:                   m_bPPRunCDFlg = FALSE;
16:
17:                }
18:                // else if the CD is currently off
19:                else
20:                {
21:                    // turn it on
22:                    SetWindowText(m_CDOnOffButton,"Turn CD Off");
23:                    m_bPPRunCDFlg = TRUE;
24:                }
25:
26:                // set the flag for the CD status in the filter
27:                m_pICDFilter->IManageCD(m_bPPRunCDFlg);
28:           }
29:
30:           ...
31:  }
```

algorithm is currently on, then we implement a toggle action by turning it off (line 15, Listing 4.5.0.11) and posting on the button the message `Turn CD On`. The latter message invites the user to again click on the CD button if he/she wants to turn the CD algorithm back on. We take a symmetrically reverse action if the CD algorithm is off. Finally, we communicate the newly set status of the CD algorithm to

the filter's interface in line 27 to take effect during the next frame-processing cycle. This is a representative example of the mechanism we use to have the commands issued by the user at the GUI level affect the filter processing. The secret is making the connection of these commands with the respective interface functions.

4.6 Create Registry Information

Our filter, like any other filter, needs to communicate with the filter graph manager. This communication is realized through the filter's registry entries. The registry entries encompass three data structures:

1. `AMOVIESETUP_MEDIATYPE`
2. `AMOVIESETUP_PIN`
3. `AMOVIESETUP_FILTER`

We insert all the definitions of the `AMOVIESETUP` structures in the beginning of the *ChangeDetFilter.cpp* file. The `AMOVIESETUP_MEDIATYPE` structure (see Listing 4.6.0.1) holds registry information about the media types our filter supports. The major type is a GUID value that describes the overall class of media data for a data stream. Since the CD filter processes video data, we set the major type to `MEDIATYPE_Video`. Some other possible major types include `MEDIATYPE_Audio` for filters that process audio information and `MEDIATYPE_Text` for filters that process text information. The minor type is also a GUID value that describes the media subtype. In our case, we choose `MEDIASUBTYPE_RGB24`, which applies to

LISTING 4.6.0.1 The `AMOVIESETUP_MEDIATYPE` structure of the CD filter.

```
1:    // Describe the pin media type.
2:    const AMOVIESETUP_MEDIATYPE sudPinTypes =
3:    {
4:        &MEDIATYPE_Video,       // major type
5:        &MEDIASUBTYPE_RGB24     // minor type
6:    };
```

uncompressed RGB samples encoded at 24 bits per pixel. This is the typical bitmap frame format provided by most live video sources. Other popular video subtypes include `MEDIASUBTYPE_MJPG` for motion JPEG compressed video and `MEDIASUBTYPE_dvsd` for standard DV video format.

The `AMOVIESETUP_PIN` structure holds registry information about the input and output pins our filter supports. In lines 4–12 of Listing 4.6.0.2 we define the input pin data. In lines 14–23 we define the output pin data. Part of our description includes pointers to the filter's media types that we defined earlier. Therefore, we build a progressive set of structures from the most detailed (media types) to the most abstract level (filter).

LISTING 4.6.0.2 The `AMOVIESETUP_PIN` structure of the CD filter.

```
 1:    // Desrcibe the pins.
 2:    const AMOVIESETUP_PIN sudPins[] =
 3:    {
 4:        { L"Input",         // pin's string name
 5:          FALSE,            // is this pin rendered?
 6:          FALSE,            // is it an output?
 7:          FALSE,            // can the filter create zero instances?
 8:          FALSE,            // can the filter create mutliple instances?
 9:          &CLSID_NULL,      // obsolete
10:          NULL,             // obsolete
11:          1,                // number of pin media types
12:          &sudPinTypes      // pointer to pin media types
13:        },
14:        { L"Output",        // pin's string name
15:          FALSE,            // is this pin rendered?
16:          TRUE,             // is it an output?
17:          FALSE,            // can the filter create zero instances?
18:          FALSE,            // can the filter create mutliple instances?
19:          &CLSID_NULL,      // obsolete
20:          NULL,             // obsolete
21:          1,                // number of pin media types
22:          &sudPinTypes      // pointer to pin media types
23:        }
24:    };
```

The AMOVIESETUP_FILTER structure holds registry information about the filter object. In line 3 of Listing 4.6.0.3 we provide the class ID of the CD filter. In line 4 we provide the filter's name. In line 5 we provide the filter's merit. The merit controls the order in which the filter graph manager tries filters when it is building automatically a filter graph. For the CD filter we use the value MERIT_DO_NOT_USE. Filters registered with this value will never be tried by the filter graph manager when in automatic graph-building mode. This means that to register the CD filter we must add it explicitly by using the IFilterGraph::AddFilter method. The MERIT_DO_NOT_USE value is typical for custom-made transform filters such as CD, where absolute user control is the safest mode of operation. In contrast, standard rendering filters use the merit value MERIT_PREFERRED. Such filters are always tried first by the filter graph manager in a proactive effort to establish suitable connections with other filters in the graph. In line 6 of Listing 4.6.0.3 we provide the number of pins. Finally, in line 7 we provide a pointer to the filter's pins, thus linking the information from the filter level all the way to the media type level.

LISTING 4.6.0.3 The AMOVIESETUP_FILTER structure of the CD filter.

```
// Describe the filter.
1:    const AMOVIESETUP_FILTER sudChangeDetFilter =
2:    {
3:        &CLSID_ChangeFilter,            // filter CLSID
4:        L"Change Detection Filter",     // filter name
5:        MERIT_DO_NOT_USE,               // filter merit
6:        2,                              // number of pin media types
7:        sudPins                         // pointer to pin information
8:    };
```

4.7 Summary

In this chapter we have developed our first DirectShow filter from scratch. The filter performs a simple differencing and thresholding operation on each incoming frame. If the residual blobs in the resulting binary image are of significant size an alarm is issued. The filter can serve as a basic component of a video-based security application

and exemplifies the power of DirectShow programming for developing professional grade Computer Vision software. Specifically, in creating the filter we have mastered how to define and instantiate our filter's class, override the appropriate base class member functions, access additional interfaces, create the property page, and create registry information. The same general methodology can be applied for the creation of any filter. The only part that changes significantly from case to case is the code of the processing algorithm. In this chapter, along with the general methodology, we have also described the specific processing algorithm (Change Detection algorithm). In subsequent chapters that describe new filters, we will concentrate only on the algorithmic code, since the general methodology remains the same.

CHAPTER 5

A Simple Video-Based Security Application

In the previous chapter we designed and developed our first filter from scratch. Now it is time to use this filter within an application. We will develop a simple application step-by-step. The application will provide the user with the opportunity to invoke the CD filter. Very little functionality will be provided beyond that. In particular, the application will also provide an exit function and the capability to select the camera of choice, if more than one camera is connected to the computer. Our purpose is to convey the basics about building a filter-invoking application. Then the user can expand this core application using standard Visual C++ methodologies. ■

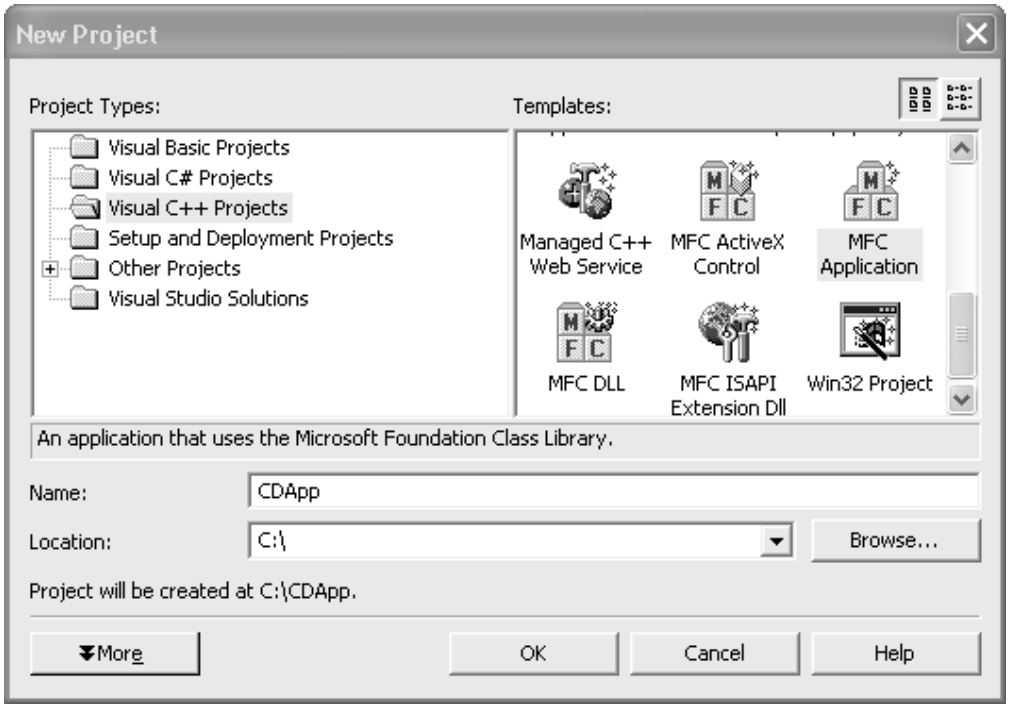
5.1 Building the Application Shell

We are now taking our first step to building the application, which we appropriately call `CDApp` for Change Detection Application.

1. We select `File->New->Project...`. This opens the New Project wizard shown in Figure 5.1.
2. Under the `Visual C++ Projects` type, we select the `MFC Application` template (Microsoft Foundation Class Application).
3. We type the name `CDApp` for the project in the name field.
4. We click OK. This causes the New Wizard to do two things: (a) create a project directory named `CDApp`, and (b) launch the MFC Application Wizard.

The MFC Application Wizard allows us to customize the application we intend to build. It uses this information to create a

FIGURE 5.1 The New Project wizard.



customized shell that we can immediately compile and run. This shell provides us with the basic infrastructure necessary to build our own computer vision application. We can create the shell by following these steps:

1. Under the `Application Type` tab of the wizard, we choose to create a single document application. The project style remains as MFC standard, and we maintain the shared DLL option. We also leave the `Document/View` option checked (see Figure 5.2).
2. Under the `Compound Document Support` tab of the wizard we keep the default selection of no support (see Figure 5.3).
3. Under the `Document Template Strings` tab of the wizard we keep the default settings (see Figure 5.4).
4. Under the `Database Support` tab of the wizard we keep the default selection for no database support (see Figure 5.5).

FIGURE 5.2 The `Application Type` tab in the MFC Application Wizard.

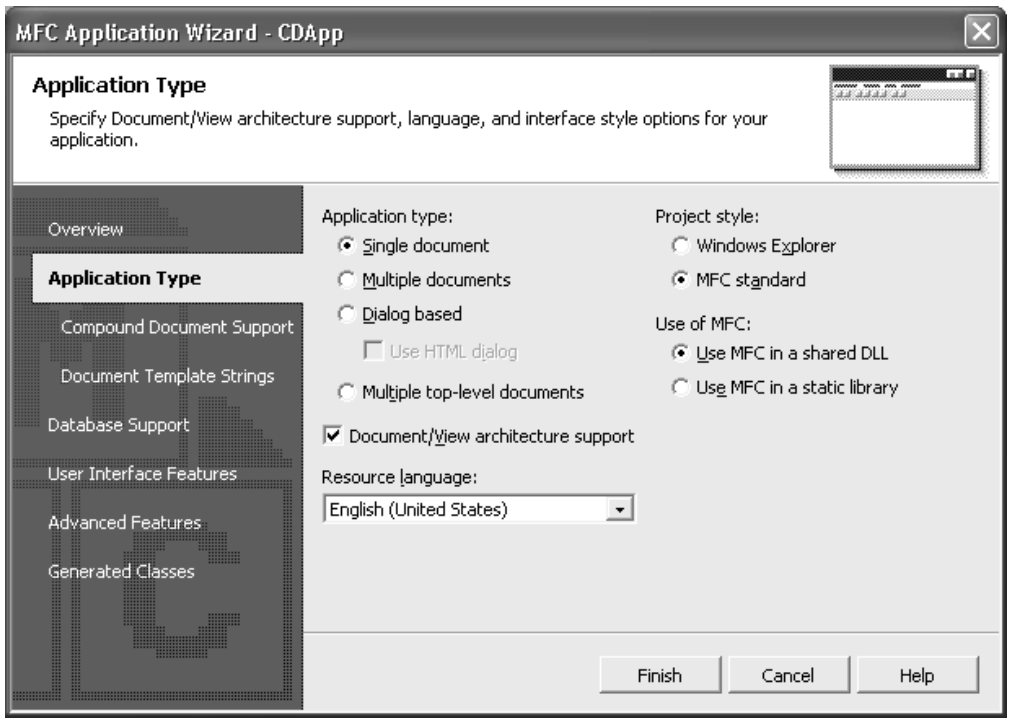
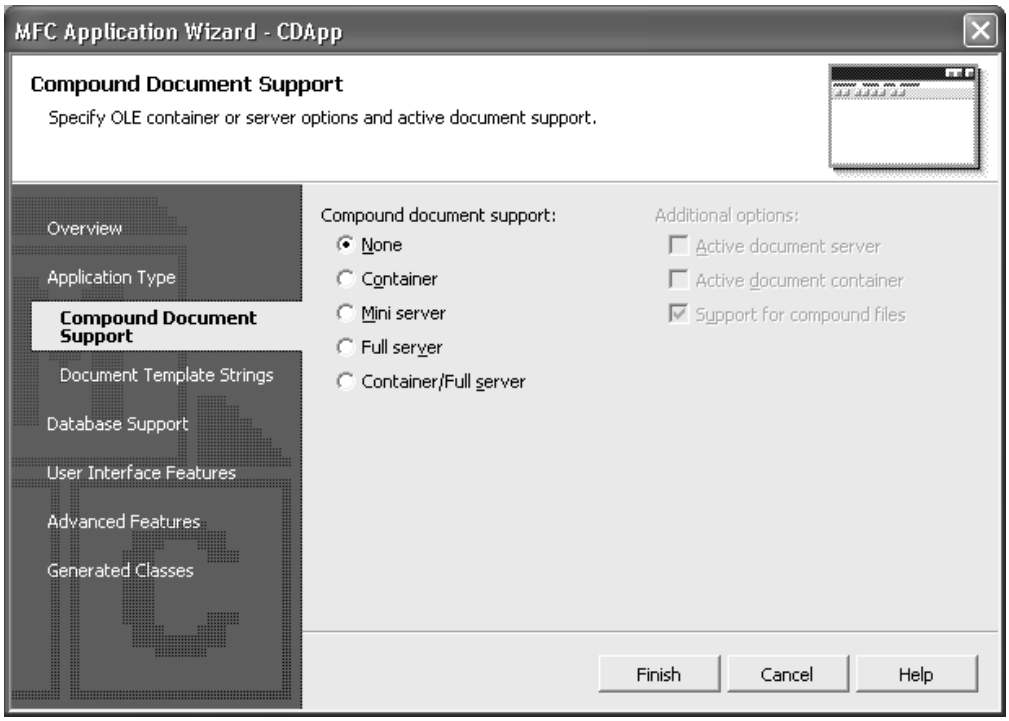
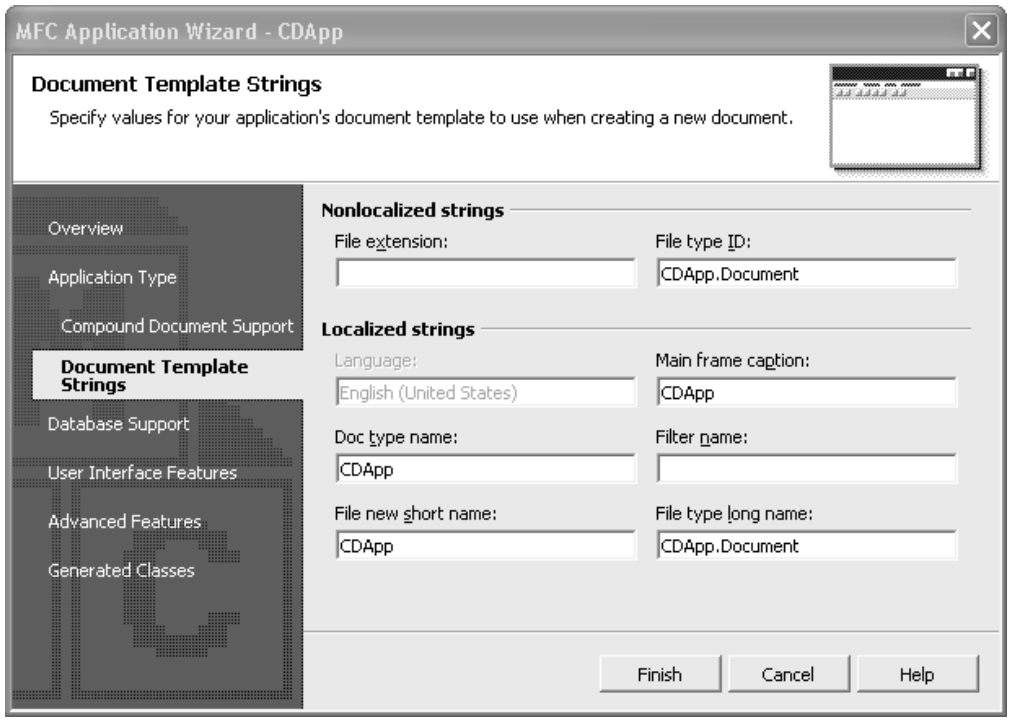


FIGURE 5.3 The `Compound Document Support` tab in the MFC Application Wizard.



5. Under the `User Interface Features` tab of the wizard we keep all the default check marks for the main frame styles while we choose the no toolbar option (see Figure 5.6).
6. Under the `Advanced Features` tab of the wizard we uncheck the `Printing and print preview` and `Active X controls` boxes. We then set the number of files to be maintained in the recent file list to zero. For the remaining boxes we leave the control selections (see Figure 5.7).
7. Finally, under the `Generated Classes` tab we can see a listing of all the files that the wizard is about to generate (see Figure 5.8). Now, click the finish button to generate the application shell.
8. After the MFC Application Wizard finishes generating our application shell, we find ourselves in the Developer Studio environment. We click the + next to the CDApp classes icon under the

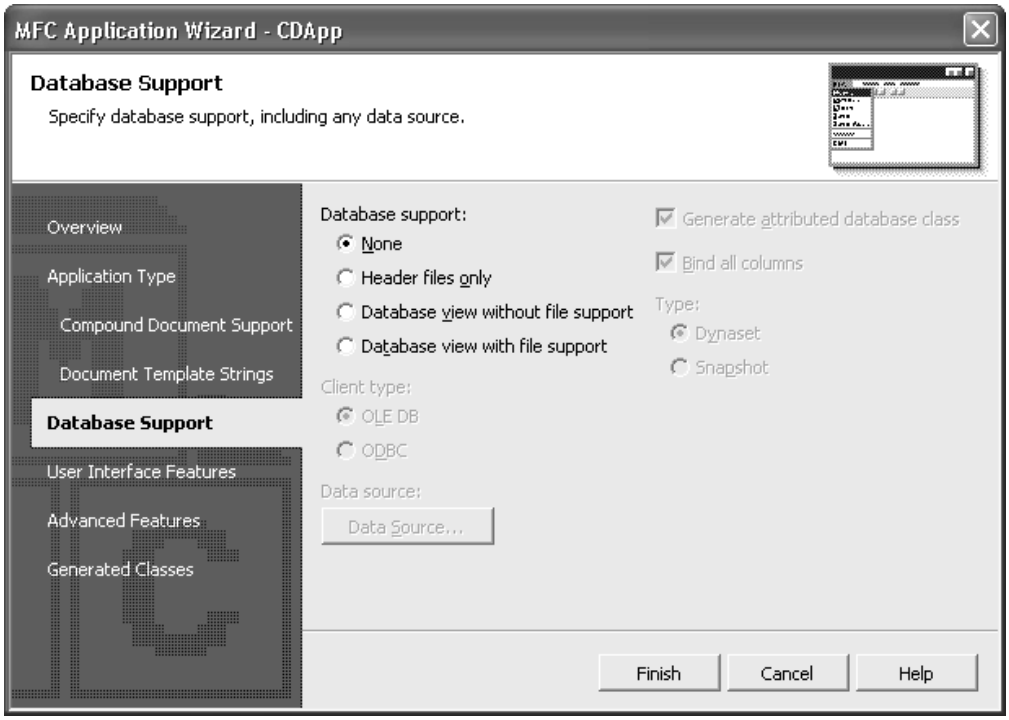
FIGURE 5.4 The `Document Template Strings` tab in the MFC Application Wizard.



class view. We notice that the Class View pane now presents us with a tree view of the classes used in the application shell, as in Figure 5.9.

9. We select the `Build->Build CDApp.exe` to compile our application.
10. As the Visual C++ compiler builds the `CDApp` project, we see progress, and other compiler messages scroll by in the `Output` pane. After our application is built, the `Output` pane should display a message stating that there were no errors or warnings as in Figure 5.9.
11. Now the `CDApp` project is ready to run. Choose `Debug->Start` to start the program.

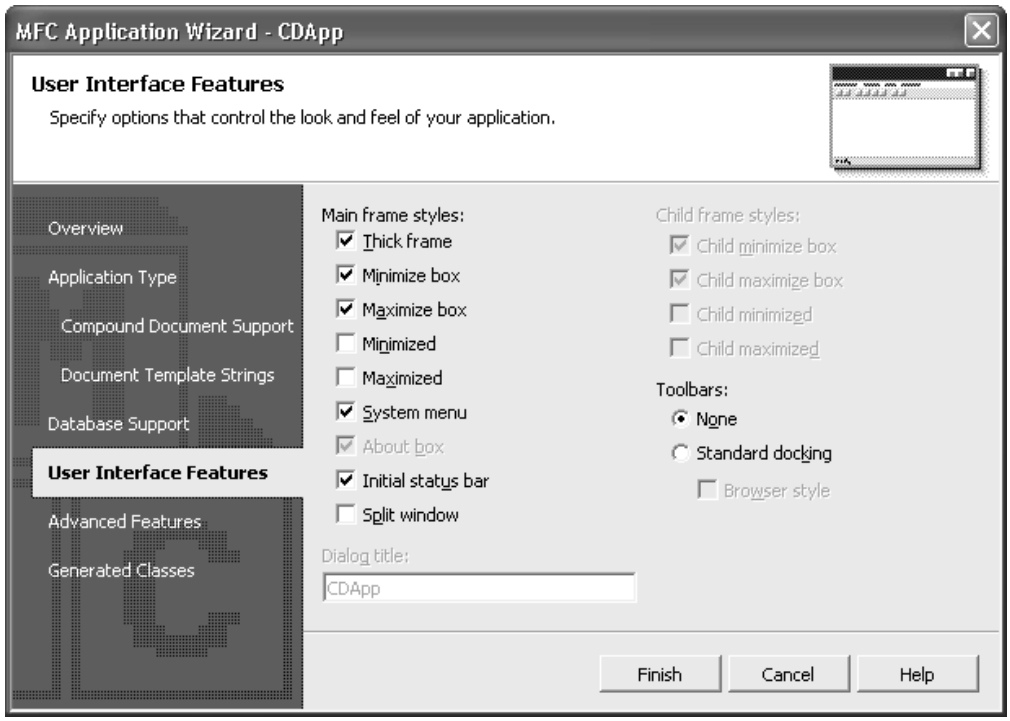
FIGURE 5.5 The `Database Support` tab in the MFC Application Wizard.



5.2 The Document-View Programming Architecture

The shell that was generated through the MFC Application Wizard conforms to the Document-View programming paradigm. The Document-View programming paradigm extends the original Windows programming style by separating the generation of data from their rendering into an output view. Based on the number of documents allowed to be opened concurrently, there exist the *Single Document Interface* (SDI) and the *Multiple Document Interface* (MDI) applications. One good example of an SDI application is Paint. Paint allows you to perform a number of tasks on a single image (document) at a time. In contrast, Photoshop is an MDI application, since it allows multiple images (documents) to be opened at one time. During step 1 of the MFC Application Wizard we chose `CDApp` to be an SDI application to keep things simple.

FIGURE 5.6 The `User Interface Features` tab in the MFC Application Wizard.

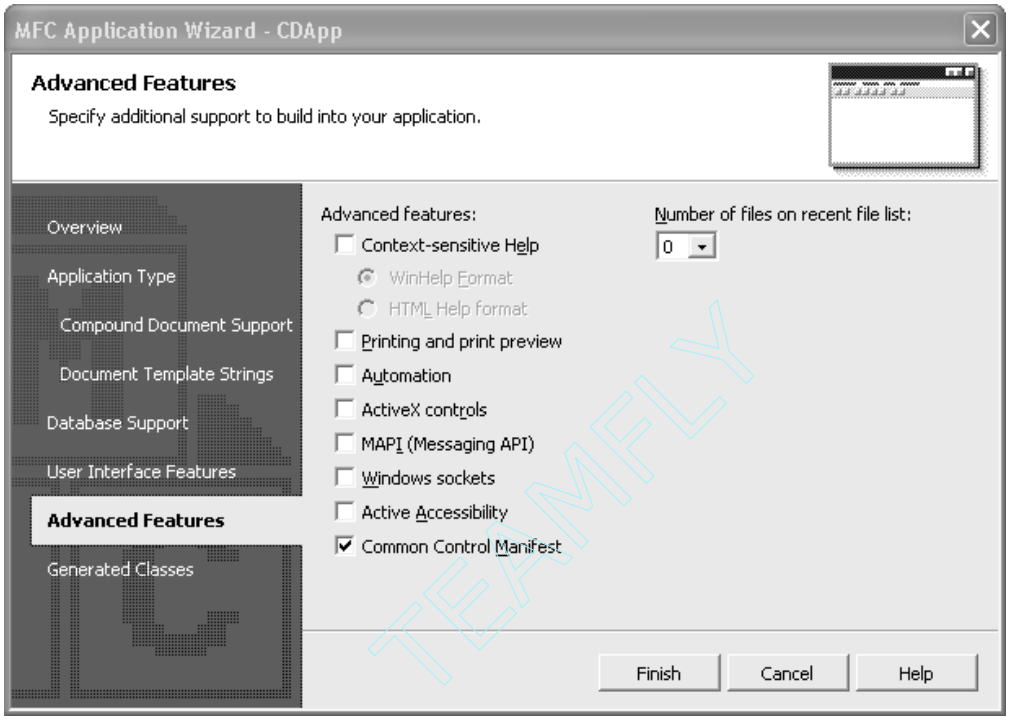


Evidently, the term *Document* is misleading. It does not mean that you can only create applications such as word processors and spreadsheets. Document refers to the data that is processed by your application. In the case of the `CDApp` application this data is the pixel data of the incoming video frames. The term *View* refers to the visual representation of the data. In the case of the `CDApp` project, the View refers to the display of the video and the change detection result in the main window.

The MFC Application Wizard creates five classes for the `CDApp` application. These can be seen under Class View of the Developer Studio environment. These classes are

1. the `CCDAppApp` class,
2. the `CMainFrame` class,

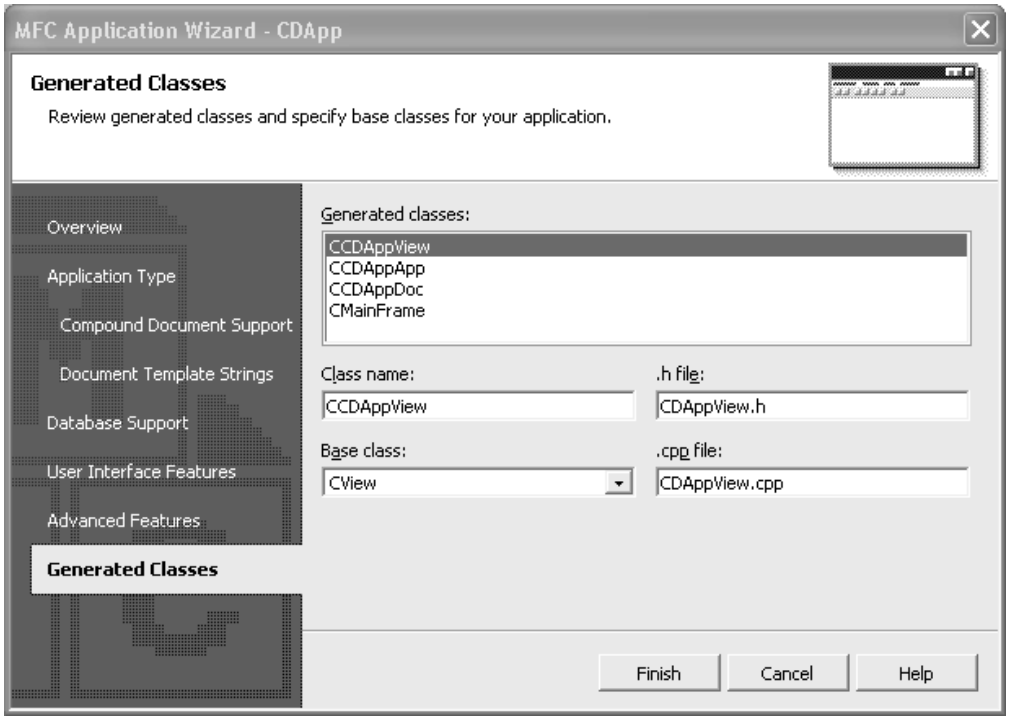
FIGURE 5.7 The `Advanced Features` tab in the MFC Application Wizard.



3. the `CCDAppDoc` class,
4. the `CCDAppView` class, and
5. the `CAboutDlg` class.

The `CAboutDlg` class produces the About CDApp . . . dialog box that is accessible under the `Help` menu. This is a relatively unimportant class. The other four classes are very important and work together to create the functionality of `CDApp` project. Figure 5.10 shows schematically how these classes interact. The `CCDAppApp` is the main class for the application. Its main function is to communicate event messages to the `CMainFrame` and `CCDAppView` classes. The `CMainFrame` class is the frame of the application window. The `CCDAppView` is the class that displays the video frames (View class). The `CCDAppDoc` is the class that houses the pixel data of the video frames (Document class). In the next few sections we will study each one of these classes in detail.

FIGURE 5.8 The `Generated Classes` tab in the MFC Application Wizard.



5.3 The `CCDAppApp` Class

The `CCDAppApp` class creates all the application components not related to data or displaying. It is the class that receives all the event messages from the user interface and devices and passes the messages to the `CMainFrame` and `CCDAppView` classes. The `CCDAppApp` class features three methods:

- `CCDAppApp`
- `InitInstance`
- `OnAppAbout`

The `CCDAppApp` is the constructor method for the class. The `OnAppAbout` method is called whenever the `About Dialog` window is invoked from the application menu. The only function of this method is to display the `About Dialog` window to the user. The `InitInstance` method is called when the application starts up.

FIGURE 5.9 The workspace with a tree view of the CDApp project classes. The `Output pane` displays any compiler errors.

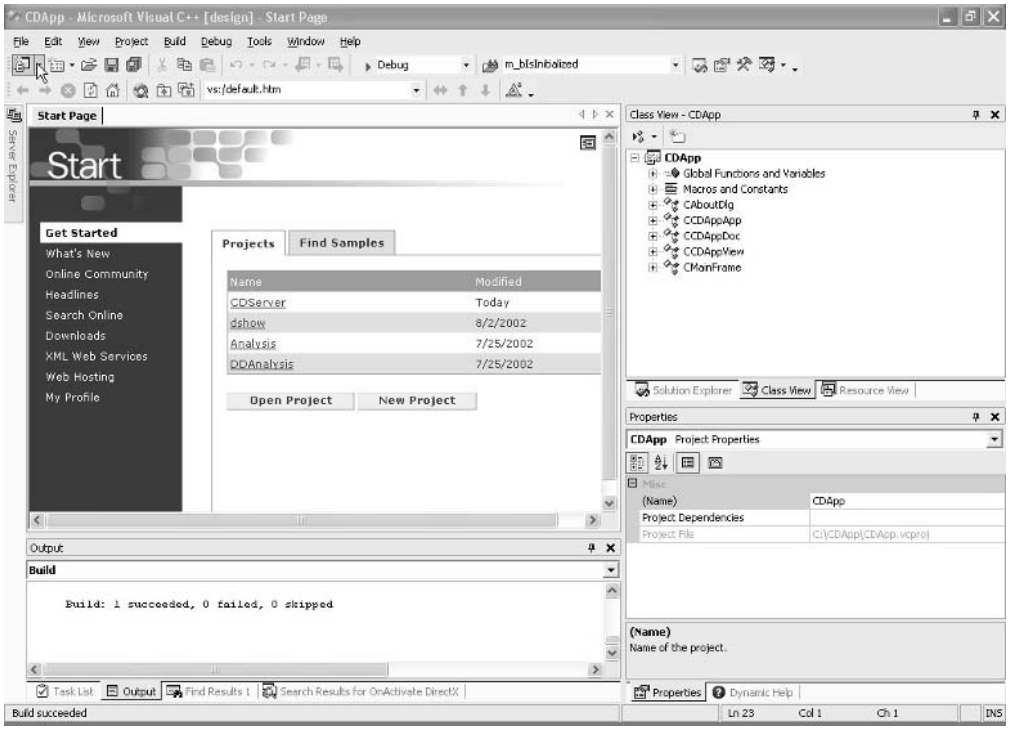
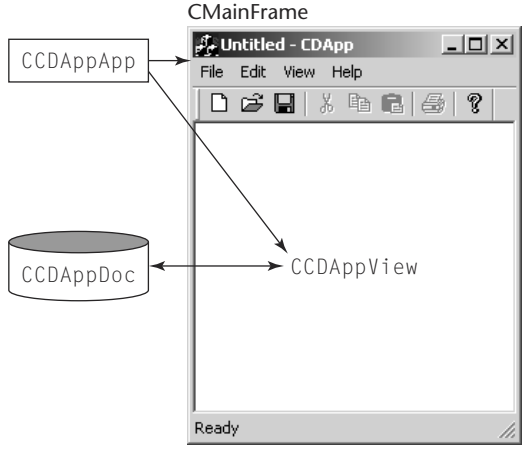


FIGURE 5.10

Interaction between the four main classes of the `CDApp` project.



The code for the method appears in Listing 5.3.0.1 The ellipsis in lines 2 and 13 stand for portions of the original code that have been omitted because they are not important to our presentation here.

LISTING 5.3.0.1 The `InitInstance` function definition.

```
 1: {
 2:     ...
 3:     // Register the application's document templates.
 4:     // Document templates serve as the connection between
 5:     // documents, frame windows, and views.
 6:     CSingleDocTemplate* pDocTemplate;
 7:     pDocTemplate = new CSingleDocTemplate(
 8:         IDR_MAINFRAME,
 9:         RUNTIME_CLASS(CCDAppDoc),
10:         RUNTIME_CLASS(CMainFrame),       // main SDI frame window
11:         RUNTIME_CLASS(CCDAppView));
12:     AddDocTemplate(pDocTemplate);
13:     ...
14:     //Set the title of the main window
15:     m_pMainWnd->SetWindowText("Change Detection App");
16:
17:     //Set initial position of the window.
18:     m_pMainWnd->SetWindowPos(NULL,0,0,660,320,NULL);
19:
20:     //Show the main window
21:     m_pMainWnd->ShowWindow(SW_SHOW);
22:     m_pMainWnd->UpdateWindow();
23:
24:     return TRUE;
25: }
```

Upon the start of the program, the document template creates a new frame window through which to view the document. The document-template constructor (line 7 in Listing 5.3.0.1) specifies what types of documents, windows, and views the template will be able to create. This is determined by the arguments we pass to the document-template constructor. Arguments to the

CSingleDocTemplate constructor (lines 8–11 in Listing 5.3.0.1) include the resource ID associated with the document type's menus and three uses of the RUNTIME_CLASS macro. RUNTIME_CLASS returns the CRuntimeClass object for the C++ class named as its argument. The three CRuntimeClass objects passed to the document-template constructor supply the information needed to create a new object of the specified class during the document creation process. The code in Listing 5.3.0.1 shows creation of a document template that creates a CCDAppDoc object with a CCDAppView object attached. The view is framed by a standard SDI frame window.

In line 21 of Listing 5.3.0.1 we activate the main application window and show it in its current size and position. Then, in line 22 we cause the view to be redrawn through the UpdateWindow method. If we run the application, we will notice that the main window occupies some default amount of space. Our intention is to accommodate within the main window two video streams: the live video stream and the CD video stream. For the X10 USB camera that we use in our prototype system, the frame dimensions of the live stream are 320 × 240 pixels. The same frame dimensions apply to the CD stream. Accordingly, we interject the following window positioning and sizing statement in line 18 of Listing 5.3.0.1:

```
m_pMainWnd->SetWindowPos(NULL,0,50,660,320,NULL);
```

This statement sizes the window to fit the two video streams, one next to the other. Finally, we give a meaningful title to the main window by adding the following statement in line 15 of Listing 5.3.0.1:

```
m_pMainWnd->SetWindowText("Change Detection App");
```

5.4 The CMainFrame Class

The main application window, which we resized within the CCDAppApp::InitInstance function, is also known as the view window. This is the area where the application data ("document") is drawn. In the case of the CD application data is the two video streams (live and CD stream). The view window sits inside another window—the application's main frame window. The main frame window has

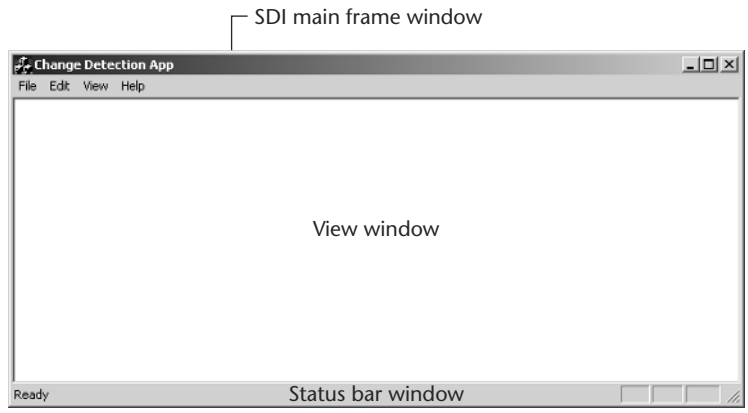


FIGURE 5.11
The application's main frame window and its child windows.

the title and menu bars. The main frame window is the parent window for the application. The view and status bar windows are its children. As can be seen in Figure 5.11 the view and status bar windows occupy the client (inner) area of the main frame window. The application framework controls the interaction between the frame and the view by routing messages from the frame to the view (see Figure 5.10).

The `CMainFrame` class is associated with the project's main frame window. The most interesting among the member functions of the `CMainFrame` class are the `PreCreateWindow` and `OnCreate` functions. `PreCreateWindow` is a virtual function (declared at the `CWnd` level) that we can override to change the characteristics of the frame window before it is displayed. The code for `PreCreateWindow` is shown in Listing 5.4.0.2. The function returns `TRUE` if the window creation process is to continue. It returns `FALSE` in the event of a failure. The `PreCreateWindow` function provides us the opportunity to change the styles of the main frame window. For example, we can choose to create a window without the Minimize and Maximize buttons or a sizable border. We can override the default window styles provided by the `PreCreateWindow` function through the modification of the `CREATESTRUCT` structure. In the present application we do not intend to change the default window styles. Therefore, we leave the compiler generated code in Listing 5.4.0.2 as it is.

LISTING 5.4.0.2 The `PreCreateWindow` function definition.

```
1: BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
2: {
3:     if( !CFrameWnd::PreCreateWindow(cs) )
4:         return FALSE;
5:     // TODO: Modify the Window class or styles here by modifying
6:     //  the CREATESTRUCT cs
7:
8:     return TRUE;
9: }
```

The `CREATESTRUCT` structure information established in the `PreCreateWindow` function is passed to the `OnCreate` function, which continues the window-creating process. `OnCreate` must return 0 to continue the creation of the window object. If the application returns 1, the window will be destroyed (lines 3–4 in Listing 5.4.0.3). Along with the main frame window the member function `OnCreate` creates also a secondary (child) window: the `StatusBar`

LISTING 5.4.0.3 The `OnCreate` function definition.

```
 1: int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
 2: {
 3:     if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
 4:         return -1;
 5:
 6:     if (!m_wndStatusBar.Create(this) ||
 7:         !m_wndStatusBar.SetIndicators(indicators,
 8:           sizeof(indicators)/sizeof(UINT)))
 9:     {
10:         TRACE0("Failed to create status bar\n");
11:         return -1;      // fail to create
12:     }
13:     ...
14:
15:     return 0;
16: }
```

window. The StatusBar window is created in lines 6–12 in Listing 5.4.0.3 by invoking the Create function.

The CMainFrame class also features two diagnostic functions: AssertValid and Dump. Calls to these functions make sense only in the Debug version of the MFC Library. The AssertValid function performs a validity check on the frame object by checking its internal state. It may assert and thus terminate the program with a message that lists the line number and filename where the assertion failed. The Dump function writes a textual representation of the object's member variables to a dump context (CDumpContext). The dump context is similar to an I/O stream. We will not use neither AssertValid nor CDumpContext very much in our examples. You may find them very useful, however, when you start developing your own video application programs from scratch.

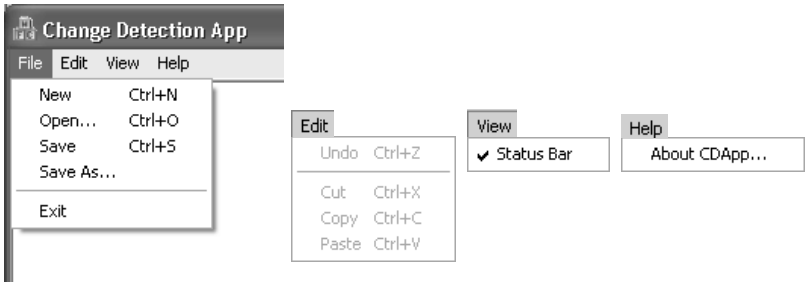
5.5 The CCDAppView Class

A view is a C++ object of a class derived from the MFC library CView class. Like any C++ object, the view object's behavior is determined by its member functions and its data members. The member functions are divided into two categories: (a) the standard functions inherited from the base classes, and (b) the application-specific functions that we will add in the derived class. When we run the CD application program, the MFC library application framework constructs an object of the derived view class and displays a window that is tightly linked to the C++ view object.

Our first priority is to establish the means by which the user will instruct the program to receive and display the live video stream from the PC camera. In Document-View applications the designated means of user input is the menu system. The AppWizard has already established a skeleton menu system in the frame window. The default menu includes the standard top level items: File, Edit, View, and Help (see Figure 5.12).

Some menu entries, such as the About CDApp..., seem to be fully functional. Some others, such as the entry New under the File menu item, seem to be doing nothing. Yet other entries, such as those under the Edit menu item, are completely disabled (grayed). What makes a menu entry useful from the programming point of view is

FIGURE 5.12 The default menus created by the `AppWizard`.



the message-handling system. Three things must happen to get a fully functional menu entry:

1. Create a message map entry.
2. Create a corresponding function prototype.
3. Create the function itself.

A message is produced in response to a user-initiated event. Such an event could be the selection of the menu entry About CDApp . . . . A handler function is programmed to perform certain actions should the event occur. The message map associates the handler function with the corresponding message. In the case of About CDApp . . menu entry, the message mapping is defined in the *CDApp.cpp* file (see Listing 5.5.0.4). This definition is produced automatically by

LISTING 5.5.0.4 The Message Map for `AppAbout`.

```
 1: BEGIN_MESSAGE_MAP(CCDAppApp, CWinApp)
 2:     //{{AFX_MSG_MAP(CCDAppApp)
 3:     ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
 4:     // The ClassWizard will add and remove mapping macros here.
 5:     // DO NOT EDIT what you see in these blocks of generated code!
 6:     //}}AFX_MSG_MAP
 7:     // Standard file based document commands
 8:     ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
 9:     ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
10: END_MESSAGE_MAP()
```

the `AppWizard` and maps the message `ID_APP_ABOUT` to the `OnAppAbout` function.

We use the WYSIWYG resource editor of Visual Studio to add one more top level menu entry and its associated items. We call this top level entry Video and associate it with three items: Video Start, Filter Page, and Video Devices..., as shown in Figure 5.13. The `Video Start` item will initiate the display of the live video feed from the camera. The Filter Page item will invoke the property page of the CD filter. The Video Devices... item will bring up the list of cameras connected to the PC for the user to choose. To view

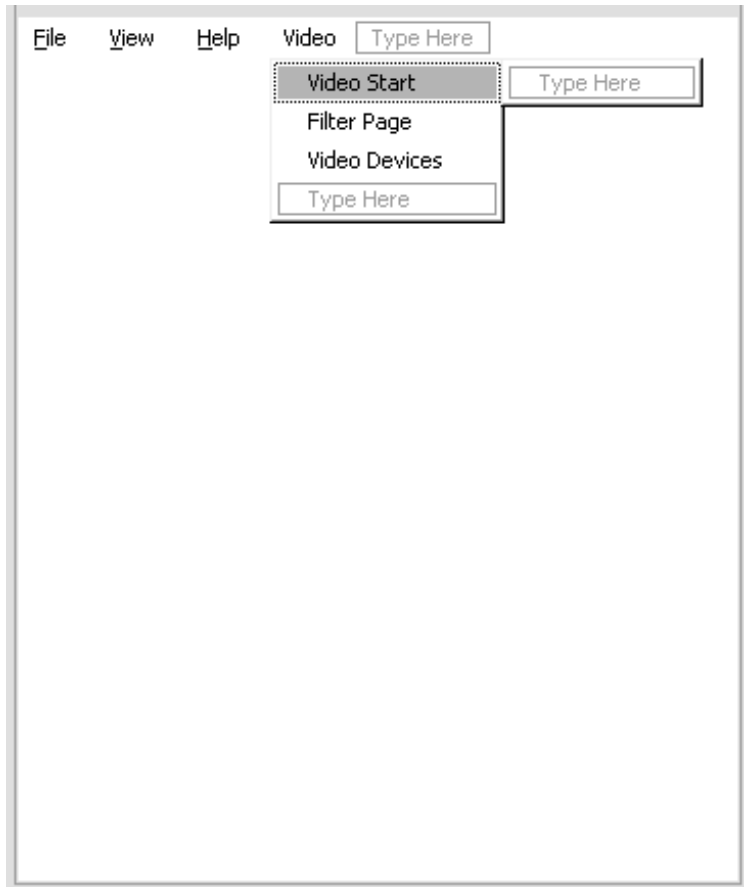


FIGURE 5.13
The `Video` menu and its entries.

the properties pane for the `Video Start` item we right click on it in the resource editor and select properties as shown in Figure 5.14. We need to provide three pieces of information in the properties window: first, the ID of the message, which we name ID_VIDEO_START; second, the menu item `caption`, which we name `Video`

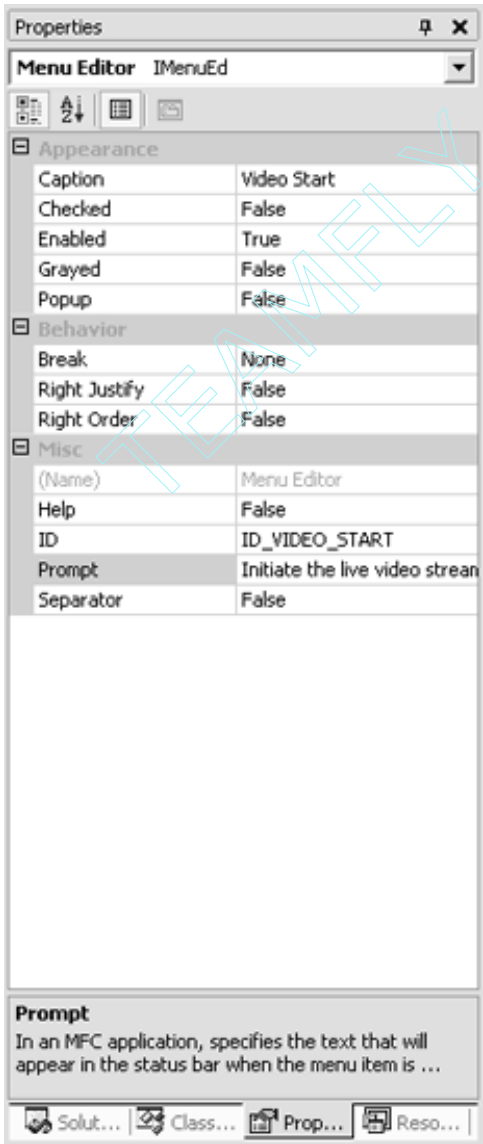


FIGURE 5.14
The Properties window for the `Video Start` menu item.

`Start`; third, the prompt that appears in the status window when the user moves the pointer over the menu item. The accompanying prompt for the `Video Start` item is "`Initiate the live video stream`." Figures 5.15 and 5.16 show the properties window for the `Filter Page` and `Video Devices...` menu items, respectively.

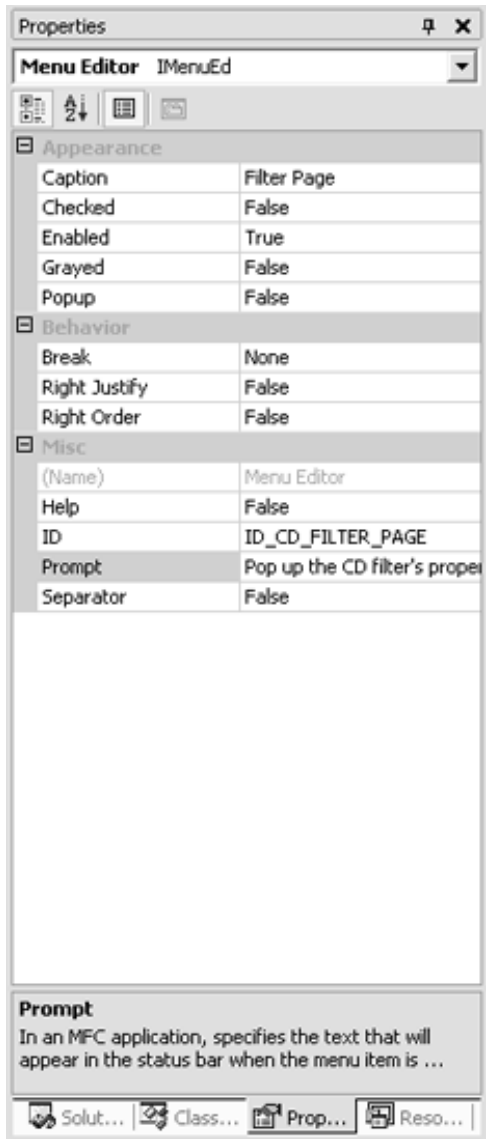


FIGURE 5.15
The Properties window for the CD Filter `Property Page` menu item.

FIGURE 5.16
The Properties window for the `Video Devices` menu item.

5.6 Summary

In this chapter we have described a customized application that uses the CD filter developed in Chapter 4. The application is based on the Document-View architecture associated with the Microsoft Foundation Class (MFC). The general programming methodology outlined in this chapter can serve as a model for application programming using any type of filter. In a nutshell, the data processed by the filter is handled through the Document class of the application while the display of the visual information through the View class of the application.

CHAPTER 6

A Custom Filter for Pan-Tilt Manipulation

In this chapter we will describe the development of a DirectShow filter that will enable the initialization of a pan-tilt device, the gauging of its position and its movement at specified increments. We will also develop a *device driver interface* (DDI) to graphically control the motions of the device. This filter will facilitate the development of an object-tracking application in a later chapter. We should note that this filter implements the functionality of a particular computer controlled pan-tilt unit manufactured by Directed Perception Inc. It is our hope that at the end of this chapter the reader will be able to develop other types of DirectShow filters that will realize "communication" with other peripherals, as well. ■

6.1 Device Driver Categorization

The pan-tilt filter we are to develop in this chapter belongs to the category of *source filters* since it allows the application program to communicate with an external device, (i.e., with the pan-tilt unit). From a programmer's viewpoint the software layer that lies between an application and the electronic device is called a *device driver*. Device drivers are compatible with both the workings of the *kernel code* associated with the particular operating system, which the computer is running, and the hardware structure of the device to be controlled. Often there is a distinction between a *high-level* and a *low-level* device driver. The latter represents only the part of the software that allows the programmer to actually exchange data with the device. The former is comprised of software that allows the user or the application to invoke certain functionality of the device, e.g., in the case of the pan-tilt mechanism, tilting the device up or down. There exists a classification of low-level device drivers that is more or less consistent across many different computing platforms (i.e., *MS Windows, UNIX, Linux,* and *Mac OS X* systems). Low-level device drivers are grouped into the following:

- *Character device drivers:* A character (`char`) device is accessed like a file, and the respective driver implements such a behavior. Functions to access character devices include `open`, `close`, `read`, and `write`.
- *Block device drivers:* Block devices host *filesystems* such as disks. These devices are accessed by blocks of data usually of 1 Kb in size. "Block" and "char" devices differ only in the way data is managed internally by the kernel.
- *Network interfaces device drivers:* Network interfaces enable the exchange of data with other hosts. Associated functions include *sending* and *receiving* of data packets driven by the network subsystem of the kernel. Example of high-level drivers associated with network interfaces include the `ftp` and `telnet` connections.

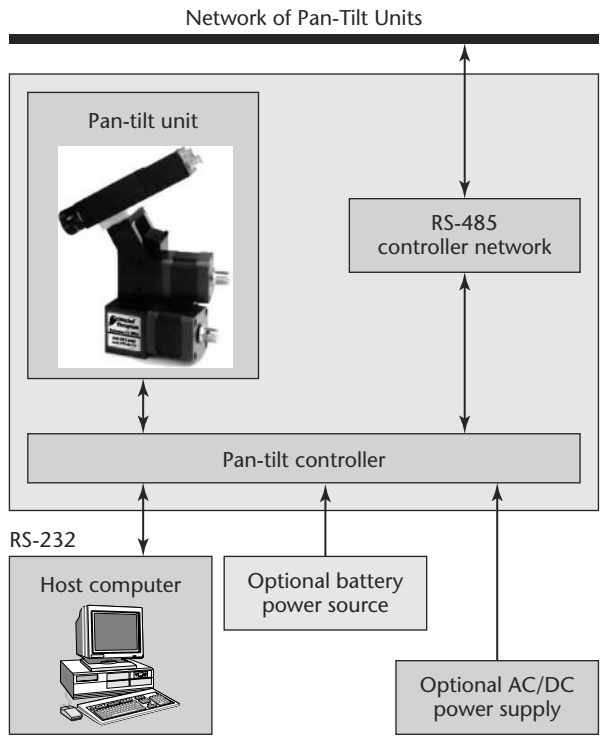
Having provided the above clarifications, we embark on developing a serial device driver for the Directed Perception Inc. pan-tilt unit. This device comprises two parts:

- *The low-level driver:* This is the device-independent part of the driver and implements exchange of data through a *serial* port.

- *The high-level driver:* This part will enable the user/application to invoke certain functions of the device (e.g., setting of default values for the pan and tilt angles, selection of different modes of operations such as position vs. velocity control, etc.)

A block diagram that shows the various components of a Directed Perception pan-tilt unit as it is connected to a host computer is shown in Figure 6.1. The pan-tilt unit (PT) is connected to the host computer via the *controller unit* through a serial port. The pan-tilt unit can be powered by either a battery or an AC/DC power supply. The pan-tilt controller includes an RJ22 handset that allows the formation of an RS-485 network of pan-tilt units that are all controlled by the host computer. We hope that Figure 6.1 will be helpful in understanding the functional interface of the pan-tilt device driver that we are about to describe.

FIGURE 6.1
Directed Perception's pan-tilt system design architecture.



6.2 Class `CSerial`: The Low-Level Serial Device Driver

In the Microsoft Windows environment the term *communications resource* is used to denote a *medium* through which exchange of data occurs. Serial ports are instantiations of such a medium. Other communications resources include parallel ports, etc. The serial port device driver we will develop provides the means with which an application program can access the communications resource. According to an earlier classification, this driver belongs to the category of *char device drivers*. We have created a C++ class object named `CSerial` that provides this capability. Its functional interface is shown in Table 6.1. Please refer to *Serial.h* and *Serial.cpp* for the complete software development of the low-level device driver.

In order for the application program to use the serial port it must first open it and obtain a handle to it (in other computing environments a handle is also called a *file descriptor*—`fd`). The Windows API function `CreateFile` does exactly this. It can also be used to open other objects such as *disk devices, files, communication*

TABLE 6.1 The functional interface of the `CSerial` class device driver.

`Class CSerial`	
Return Value	*Function Name*
`portstream_fd`	`portstream_fd SetSerial(...)`
`char`	`openserial(char *portname)`
`char`	`closeserial(portstream_fd)`
`char`	`SerialBytesOut(portstream_fd, ···)`
`char`	`SerialBytesIn (portstream_fd, ···)`
`char`	`PeekByte(portstream_fd, unsigned char *)`
`char`	`FlushInputBuffer(portstream_fd)`
`char`	`SerialStringOut(portstream_fd, unsigned char*)`
`char`	`ReadSerialLine(portstream_fd, ···)`
`char`	`GetSignedShort(portstream_fd, ···)`
`char`	`PutSignedShort(portstream_fd, ···)`
`char`	`GetUnsignedShort(portstream_fd, ···)`
`char`	`PutUnsignedShort(portstream_fd, ···)`
`char`	`GetSignedLong(portstream_fd, ···)`
`char`	`PutSignedLong(portstream_fd, ···)`
`void`	`do_delay(long)`

pipes, etc. The actual code for opening a serial port used inside the `CSerial::SetSerial` function is shown in Listing 6.2.0.1.

LISTING 6.2.0.1 Opening and obtaining a HANDLE to a serial port.

```
1: typedef HANDLE portstream_fd;
2:        ...
3: portstream_fd pstream;
4: DCB dcb;
5: COMMTIMEOUTS timeout_info;
6: pstream =
7:    CreateFile( portname,            // Port name e.g., COM1
8:         GENERIC_READ|GENERIC_WRITE, // Data access mode
9:         0,                          // Share mode
10:        NULL,                       // No security attributes
11:        OPEN_EXISTING,              // How to create the HANDLE
12:        FILE_ATTRIBUTE_NORMAL,      // File attributes
13:        NULL                        // Handle to comm device
14:    );
```

When a communications resource, like a serial port, is opened, it inherits the configuration that the communications resource had during its last operation. The inherited configuration values are comprised of the configuration settings of the device control block (which is a `DCB structure`; see line 4 of Listing 6.2.0.1) and the time-out values of the `COMMTIMEOUTS` structure used in input/output (I/O) operations (see line 5 of Listing 6.2.0.1). If the respective resource has never been opened, default settings are used. Listing 6.2.0.2 outlines coding for modifying the configuration values of the serial communication port.

LISTING 6.2.0.2 Modifying the cofiguration settings of the serial port communications resource.

```
1:     if ( !(GetCommState(pstream, &dcb)) ) {
2:         printf("GetCommState error(%d)\n", GetLastError());
3:         return NULL; }
4:
5:     dcb.BaudRate = Speed;
6:     dcb.ByteSize = (unsigned char) Bits;
7:     dcb.Parity = NOPARITY;
8:     if (StopBit == 2)
```

```
9:          dcb.StopBits = TWOSTOPBITS;
10:     else
11:         dcb.StopBits = ONESTOPBIT;
12:
13:     switch (Parity) {
14:         case NO_PARITY:   dcb.Parity = NOPARITY;    break;
15:         case EVEN_PARITY: dcb.Parity = EVENPARITY;  break;
16:         case ODD_PARITY:  dcb.Parity = ODDPARITY;   break;
17:         default:
18:         return NULL;}
19:
20:     dcb.fOutxCtsFlow        = TRUE;
21:     dcb.fOutxDsrFlow        = TRUE;
22:     dcb.fDsrSensitivity     = TRUE;
23:     dcb.fOutX               = TRUE;
24:     dcb.fInX                = FALSE;
25:     dcb.fDtrControl         = DTR_CONTROL_ENABLE;
26:     dcb.fRtsControl         = RTS_CONTROL_ENABLE;
27:     dcb.fTXContinueOnXoff   = TRUE;
28:
29:     if ( ! (SetCommState(pstream, &dcb)) ) {
30:         printf("SetCommState err(%d)\n", GetLastError());
31:         return NULL; }
32:
33:     // It terminates pending of read/write operations
34:     if ( ! (PurgeComm(pstream, PURGE_TXABORT | PURGE_RXABORT |
35:         PURGE_TXCLEAR | PURGE_RXCLEAR)) ) {
36:         printf("PurgeComm err(%d)\n", GetLastError());
37:         return NULL; }
38:
39:     if ( ! GetCommTimeouts(pstream, &timeout_info) ) {
40:         printf("GetCommTimeouts err(%d)\n", GetLastError());
41:         return NULL; }
42:
43:     timeout_info.ReadIntervalTimeout = MAXDWORD;
44:     timeout_info.ReadTotalTimeoutMultiplier =
45:     timeout_info.ReadTotalTimeoutConstant   = 0;
46:
47:     if (! SetCommTimeouts(pstream, &timeout_info) ) {
48:         printf("SetCommTimeouts err(%d)\n", GetLastError());
49:         return NULL; }
```

Some explanations of the code are in order. First and foremost the current values of the configuration settings must be obtained. This is done in lines 1–3 of Listing 6.2.0.2, whereby calling the function `GetCommState` the values of the device control block structure `dcb` are filled. Lines 5–27 describe the assignment of new values to the `dcb`'s member variables. Calling of `SetCommState` finalizes the assignment of the new settings (line 29 of Listing 6.2.0.2). Similarly, one can modify the time-out settings for I/O operations. Lines 33–49 (Listing 6.2.0.2) describe how it is implemented. First, a function call to the `PurgeComm` is required to terminate pending of any read/write operations on the serial port. Similarly, current time-out values are filled in by calling the `GetCommTimeouts` function (see line 5 of Listing 6.2.0.1 for the respective variable type used in the argument list). New values are assigned in lines 43–45 (Listing 6.2.0.2). A call to `SetCommTimeouts` finalizes the process.

One high-level call to the `CSerial::openserial` function achieves the collective opening of a serial port with certain settings for the baud rate, the number of data bits per byte, the number of stop bits per byte, and disabling parity checking (Listing 6.2.0.3).

LISTING 6.2.0.3 Function definition for opening a serial communication port with certain configurations.

```
1:    portstream_fd CSerial::openserial(char *portname) {
2:
3:    // Serial communications parameters
4:    int        speed    = 9600;
5:    int        parity   = NO_PARITY;
6:    int        bits     = 8;
7:    int        stopbits = 1;
8:
9:    portstream_fd pstream;
10:
11:   pstream = SetSerial(portname, speed, parity, bits, stopbits);
12:
13:   return pstream;}
```

Closing the serial communication resource is another important function that a device driver must implement. The `closeserial` performs the following tasks:

- It disables event notification by calling `SetCommMask`.
- It clears the DTR (data-terminal-ready) signal (pin no. 4 of the standard 9-pin serial port) by calling the `EscapeCommFunction` function.
- It purges any outstanding reads/writes through `PurgeComm`.
- It closes the serial port via a function call to `CloseHandle`.

The complete code appears in Listing 6.2.0.4.

LISTING 6.2.0.4 Function definition for closing a serial communication port.

```
1:    char CSerial::closeserial(portstream_fd portstream) {
2:      if(portstream != NULL) {
3:
4:        // Disable event notification.
5:        SetCommMask( portstream, 0 ) ;
6:
7:        // Create DTR signal.
8:        EscapeCommFunction( portstream, CLRDTR );
9:
10:       // purge any outstanding reads/writes
11:       PurgeComm( portstream, PURGE_TXABORT | PURGE_RXABORT |
12:                              PURGE_TXCLEAR | PURGE_RXCLEAR );
13:
14:       // close the device
15:       err = CloseHandle( portstream ) ;
16:       portstream = (HANDLE) -1;
17:       if (err < 0)
18:          return -1;
19:     }
20:     return 0;
21:   }
```

6.2.1 *Reading and Writing Data into a Serial Port*

In this section, we will describe two functions that form the basis for exchanging data with an external serial device through a serial

communications port. They are

- `char CSerial::SerialBytesOut` for writing data out to a serial port, and
- `char CSerial::SerialBytesIn` for reading data in from a serial port.

Writing data to a serial port is mainly performed by the `WriteFile` windows API function. A detailed description of its argument list is shown in Table 6.2. The code of the custom developed `CSerial::SerialBytesOut` function appears in Listing 6.2.1.1 and it is self-explanatory. It consists of a check that validates the successful opening of the serial port (lines 5–6) and continues with the actual writing of data (lines 7–10).

LISTING 6.2.1.1 Implementation of the `SerialBytesOut` function.

```
1: char CSerial::SerialBytesOut(portstream_fd portstream,
2:                              unsigned char *buffer, int charCnt) {
3:    int BytesWritten;
4:
5:    if (portstream == NULL)
6:        return FALSE;
7:    if ( WriteFile(portstream, (LPSTR) buffer, charCnt,
8:       (unsigned long *) &BytesWritten, NULL) != TRUE )
9:    { printf("\nWriteFile error of %d\n", GetLastError());
10:       return -1; }
11:
12:   return TRUE;
13: }
```

TABLE 6.2 Description of the `WriteFile` functions argument list.

Windows API Function `BOOL WriteFile(···)`	
Argument Type	*Description*
`HANDLE hFile`	`Handle to a serial port.`
`LPCVOID lpBuffer`	`Data buffer.`
`DWORD nNumberOfBytesToWrite`	`Number of bytes to write.`
`LPDWORD lpNumberOfBytesWritten`	`Number of bytes written.`
`LPOVERLAPPED lpOverlapped`	`Overlapped buffer.`

LISTING 6.2.1.2 Function declaration of `SerialBytesIn` along with a list of local variables used for reading data from a serial port.

```
1: char CSerial::SerialBytesIn (portstream_fd portstream,
2:               unsigned char *buffer, unsigned int charCnt,
3:               long timeout) {
4:    COMSTAT COMstatus;
5:    UINT NumCharsAvail;
6:    DWORD ErrCode;
7:    unsigned long BytesRead;
8:    time_t start_time, end_time;
9:    int timeout_in_secs;
10:    . . .
11: }
```

Reading bytes of data from a serial port is performed in the function `SerialBytesIn`. The argument list for this function includes a handle to a serial port, a data buffer in which the data will be stored, the number of data bytes to be read, and finally an optional time-out period (i.e., `timeout`) (Listing 6.2.1.2). The following tasks are performed inside `SerialBytesIn`. First, timing of the reading operation is initiated only when `timeout` $\neq -1$. Then a check on whether the serial port has been successfully opened is performed. Once this is the case, the actual reading process begins by a call to `ClearCommError`. This function prepares the serial port for reading when a communication error has occurred. Along with a returned flag regarding the occurrence of an error, the status of the communication (lines 4–8 of Listing 6.2.1.3) is also recorded. The `cbInQue` member variable of the `COMSTAT` structure (line 10 of Listing 6.2.1.3) denotes the number of bytes that have been received but not yet read off of the serial port. If this number is greater or equal to the number of bytes that are to be read by this operation, then reading is performed by calling the Windows API function `ReadFile`. The argument list of this function is similar to that of `WriteFile` function (see Table 6.2). Lines 10–23 of Listing 6.2.1.3 describe the above steps in detail. Finally, if the time-out period has elapsed, the function returns with the appropriate flag.

LISTING 6.2.1.3 Software code included in the `SerialBytesIn` function.

```
1:   . . .
2:   else {
3:         for (; charCnt > 0 ;) {
4:           err = ClearCommError(portstream, &ErrCode, &COMstatus);
5:           if (err != TRUE) {
6:              printf("\nERROR(SerialBytesIn):
7:              ClearCommError error (%d)\n", GetLastError() );
8:              return((char) err);}
9:
10:           NumCharsAvail = COMstatus.cbInQue;
11:
12:           if ( NumCharsAvail >= charCnt ) {
13:
14:               err = ReadFile(portstream, (LPSTR) buffer,
15:                     charCnt, &BytesRead, NULL);
16:               if (err == TRUE) {
17:                   charCnt -= BytesRead;
18:                   if (charCnt ==0) return(TRUE); }
19:               else {
20:                   printf("\nERROR(SerialBytesIn): ReadFile
21:                           error (%d)\n", GetLastError() );
22:                   return((char) err); }
23:           }
24:           else { if ( timeout != -1 )
25:                  { time(&end_time);
26:                    if ((end_time - start_time)>timeout_in_secs)
27:                       return TIMEOUT_CHAR_READ;
28:                  }
29:                  Sleep(0);  // give up the remaining timeslice
30:           }
31:       }
32:    }
33:    return TRUE;
34:  }
```

We will finish the part of the low-level serial driver by noting that almost all the other components of the `CSerial` functional interface represent variations of the `SerialBytesIn` and `SerialBytesOut`

functions. They describe I/O operations that involve specific sizes of data (e.g., `GetsignedShort` is defined for reading 4 bytes of data serially, etc.). In addition, these operations reflect the data that must be communicated to the pan-tilt device for the application to control the device's movements. Having outlined the development of the low-level device driver, we will continue in the next section with the implementation of its high-level portion that relates to the control of the pan-tilt peripheral.

6.3 Class `CPanTiltDevice`: The High-Level Driver

We will follow a bottom-up approach in explaining the development of the high-level driver as we did with the low-level counterpart. The details of the software implementation can be found in the *OPCODES.h, PanTiltDevice.h,* and *PanTiltDevice.cpp* files. In the *OPCODES.h* header file, a number of variables are defined (i.e., they are called `opcodes`) whose values, when received by the pan-tilt device, command it to execute particular movements or to query the status of the device; that is, the pan-tilt device decodes the opcodes values and performs the corresponding functions. The reader should keep in mind that these "codes" are different for different devices. The other two files implement a C++ class object named `PanTiltDevice` with a function interface that is largely based on the low-level driver. This class includes a pointer to a `CSerial` class object that is the gateway of communication between the pan-tilt device and the computer (see line 4 of Listing 6.3.0.1).

LISTING 6.3.0.1 The `CPanTiltDevice` class includes a member variable that points to a `CSerial` class object `m_Serial`.

```
1: class CPanTiltDevice
2: {
3:    public:
4:    CSerial *m_Serial;
5:
6:    CPanTiltDevice();
7:    virtual ~CPanTiltDevice();
8:
```

```
9:    // The class's member functions
10:    . . .
11: }
```

We will now explain the functional interface of the `CPanTilt-Device` class in detail.

6.3.1 *Opening and Closing a Pan-Tilt Communication Port*

The function `open_host_port` initializes the serial port through which the pan-tilt device will be controlled. It performs the following:

- It selects and opens a serial port for communication (line 4 of Listing 6.3.1.1).
- It clears the selected serial port by writing a `NULL` character to it (lines 7–8 of Listing 6.3.1.1).

Clearing the serial communication port is important especially when the pan-tilt device is used in a networked architecture, as we described earlier. The related `close_host_port` function closes the serial port and is always an integral part of the process that ends communication with the pan-tilt device.

LISTING 6.3.1.1 The `open_host_port` function selects, opens, and clears a serial port for communication.

```
1: portstream_fd CPanTiltDevice::open_host_port(char *portname)
2: {
3:     char out_string[10] = "    ";
4:     m_Serial->current_host_port = m_Serial->openserial(portname);
5:
6:     // It clears serial ports for networked startup
7:     m_Serial->SerialBytesOut(m_Serial->current_host_port,
8:               (unsigned char *) out_string, strlen(out_string));
9:     m_Serial->do_delay(2000);
10:
11:    return m_Serial->current_host_port;
12: }
```

6.3.2 *Setting the Mode and Executing a Pan-Tilt Movement*

Before using the pan-tilt unit certain parameters that configure the mode of operation must be set. For example, the user may choose particular speed or position limits for either the pan or the tilt axis beyond which command errors will be returned. We have chosen the factory default settings for operating the device, as we will explain later when we describe the initialization of the pan-tilt filter. The `set_mode` function allows the user to set the mode of operation. It takes two arguments, as Figure 6.2 shows. A sample of the code in Listing 6.3.2.1 shows a nested `switch` structure by which the mode is first selected (lines 7 or 10 or 13), and then the respective opcode is sent to the serial port (lines 8 or 11 or 14, respectively).

LISTING 6.3.2.1 A representative portion of the `set_mode` function consisting of a doubly nested `switch` statement on the function's arguments.

```
1: char CPanTiltDevice::set_mode(char mode_type, char mode_parameter)
2: {
3:     switch (mode_type) {
4:         case DEFAULTS:
5:             switch (mode_parameter) {
6:
7:                 case SAVE_CURRENT_SETTINGS:
8:                     SerialOut(SAVE_DEFAULTS);
9:                     break;
10:                case RESTORE_SAVED_SETTINGS:
11:                    SerialOut(RESTORE_SAVED_DEFAULTS);
12:                    break;
13:                case RESTORE_FACTORY_SETTINGS:
14:                    SerialOut(RESTORE_FACTORY_DEFAULTS);
15:                    break;
16:                default:
17:                    return(PTU_ILLEGAL_COMMAND_ARGUMENT);
18:                    break;
19:            }
20:        . . .
21:    }
22: }
```

FIGURE 6.2
The options for the argument list of the `set_mode` function.

char CPanTiltDevice::set_mode (Arg #1, Arg #2)	
Arg. #	Options
1	[COMMAND_EXECUTION_MODE | ASCII_VERBOSE_MODE | ASCII_ECHO_MODE | POSITION_LIMITS_MODE | DEFAULTS]
2	[EXECUTE_IMMEDIATELY | EXECUTE_UPON_IMMEDIATE_OR_AWAIT | VERBOSE | TERSE | QUERY_MODE ON_MODE | OFF_MODE | SAVE_CURRENT_SETTINGS | RESTORE_SAVED_SETTINGS | RESTORE_FACTORY_SETTINGS]

FIGURE 6.3
The options for the argument list of the `set_desired` function.

char CPanTiltDevice::set_desired (Arg #1, Arg #2, Arg #3, Arg #4)	
Arg. #	Options
1	[PAN | TILT]
2	[POSITION | SPEED | ACCELERATION | BASE | | UPPER_SPEED_LIMIT | LOWER_SPEED_LIMIT | | HOLD_POWER_LEVEL | MOVE_POWER_LEVEL]
3	[< position value > | < speed value > | <acceleration value> | | < power mode > | PTU_REG_POWER | PTU_LOW_POWER | | PTU_OFF_POWER]
4	[RELATIVE | ABSOLUTE | NULL]

We are now ready to describe how a desired movement can be executed by the pan-tilt unit. The function that commands the pan-tilt unit to perform various movements and to work under different power modes is `set_desired`. The main body of the function is again a nested `switch(Argument)` structure operating on the parameters of the function. The four input parameters are shown in Figure 6.3, along with the list of values they take. The first parameter chooses among the two degrees of freedom of the device, i.e., pan or tilt. The second argument identifies the dynamic properties of movement, i.e., position vs. speed vs. acceleration, etc. The third argument denotes the size of the movement, i.e., turn θ_0

clockwise. Finally, the fourth argument sets the mode of movement, i.e., absolute vs. relative.

A representative portion of the actual code appears in Listing 6.3.2.2. This code shows a nested structure of `switch` statements that work on the second, first, and fourth arguments of the function. Their values are used to set the characteristics (mode) of movement, e.g., rotate the device around its pan axis, relative to its current position. To execute such a movement, the respective opcode: PAN_SET_REL_POS is first sent to the pan-tilt unit via `SerialOut(PAN_SET_REL_POS)` (line 11 of Listing 6.3.2.2). Once the mode of movement has been set, we need to define its size, e.g., rotation by ϕ degrees. The size (represented by a pointer to a signed short integer) is finally sent to the device as it is shown in lines 35 and 36 (via a call to the low-level driver interface function `PutSignedShort`). In the same way that the operator configures the characteristics of a pan-tilt movement, we can also retrieve the respective values by calling `get_current(Arg1, Arg2|RESOLUTION)`. You should note that the latter function takes two arguments that are exactly the same as the first two arguments of the `set_desired`. The only exception is the inclusion of RESOLUTION in the list of options for the second argument.

LISTING 6.3.2.2 The body of the `set_desired` function consists of nested `switch` statements on the function's arguments.

```
1: char CPanTiltDevice::set_desired(char axis, char kinematic_property,
2:                                  PTU_PARM_PTR *value, char movement_mode)
3: {
4:   . . .
5:     switch (kinematic_property) {
6:         case POSITION:
7:             switch (axis)  {
8:                 case PAN:
9:                     switch (movement_mode) {
10:                        case RELATIVE:
11:                            SerialOut(PAN_SET_REL_POS);
12:                            break;
13:                        case ABSOLUTE:
14:                            SerialOut(PAN_SET_ABS_POS);
```

```
15:                             break;
16:                         default:
17:                             return(PTU_ILLEGAL_COMMAND_ARGUMENT);
18:                     }
19:                     break;
20:                 case TILT:
21:                     switch (movement_mode) {
22:                         case RELATIVE:
23:                             SerialOut(TILT_SET_REL_POS);
24:                             break;
25:                         case ABSOLUTE:
26:                             SerialOut(TILT_SET_ABS_POS);
27:                             break;
28:                         default:
29:                             return(PTU_ILLEGAL_COMMAND_ARGUMENT);
30:                     }
31:                     break;
32:                 default:
33:                     return (PTU_ILLEGAL_COMMAND_ARGUMENT);
34:             }
35:         m_Serial->PutSignedShort(m_Serial->current_host_port,
36:                                  (signed short *) value);
37:         break;
38:         . . .
39:     }
40: }
```

We will conclude the development of the high-level portion of the driver with a functional description of two functions that are used as a test to ensure successful communication with the pan-tilt unit. The first is the `firmware_version_OK` function that verifies that the pan-tilt controller firmware supports the binary interface we have encoded. The second is the `reset_PTU_parser` function that ensures that the pan-tilt device responds to user commands. In summary, once successful communication with the pan-tilt unit has been established, we can set the mode of operation (by calling `set_mode`) before invoking pan-tilt movements (via calls to `set_desired` with the appropriate arguments).

6.4 The `CPanTiltFilter` Source Filter

We are now ready to develop a DirectShow filter for the manipulation of the pan-tilt device via a GUI. The software development of the filter is broken down into three parts:

- the development of the GUI with the implementation of the associated callback functions (We referred to this in a previous chapter as the filter's property page).
- the implementation of the interface functions for enabling user-initiated commands to the pan-tilt device through the GUI.
- the implementation of the pan-tilt filter itself.

We will conclude this chapter with an example of how to run an application to control the pan-tilt device using the `GraphEdit` utility.

6.4.1 *Developing the GUI through a Property Page*

The property page provides the means that allow the user to interact directly with a filter object (e.g., the pan-tilt filter) via a graphics widget. A graphics widget may be any of a multitude of object controls such as `buttons, edit boxes, sliders,` etc. We have chosen to implement a GUI that consists of four buttons (two buttons per degree of freedom for the pan-tilt device) (see Figure 6.4).

The property page shown in Figure 6.4 is a class object whose detailed definition and functionality is described in files *PanFilterProp.h* and *PanFilterProp.cpp* files. A piece of the source that defines the respective class object is shown in Listing 6.4.1.1.

The four buttons that will be deployed to control the pan-tilt device movements have an associated `ID` as depicted in lines 1 to 4 of Listing 6.4.1.1. The `CPanTiltFilterProperties` class object is defined in lines 6–34 of Listing 6.4.1.1, with the class constructor shown in line 29 and the `CreateInstance` function that is used by the pan-tilt filter's *class factory* in lines 9 and 10 (Listing 6.4.1.1). We remind you that a class factory lists the class object IDs that will be used by the filter and the associated functions that are called to create them (we will revisit this point when we will describe the pan-tilt filter). Lines 19–27 (Listing 6.4.1.1) are associated with the control of the device's pan position. In particular, `m_PANMIN` and `m_PANMAX`

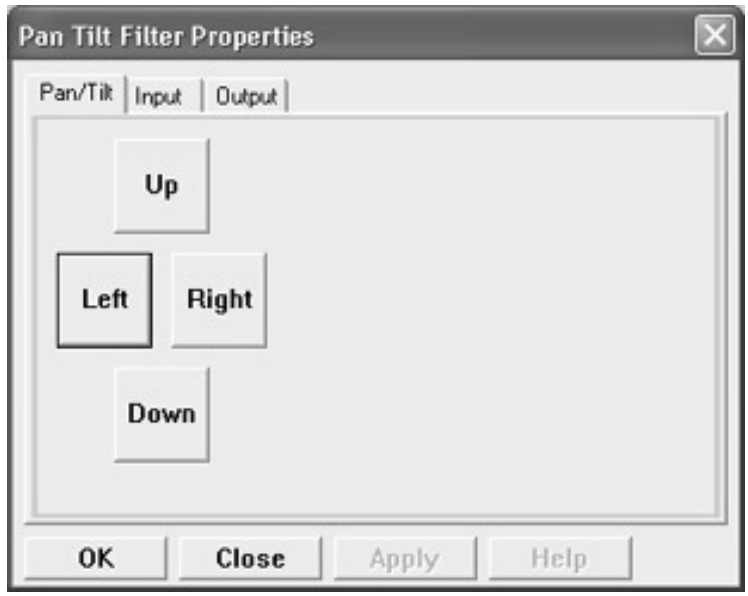


FIGURE 6.4
The pan-tilt property page for controlling the movement of the pan-tilt device.

LISTING 6.4.1.1 Definition of the `CPanTiltFilterProperties` class object.

```
1:  #define ID_LEFT  1
2:  #define ID_RIGHT 2
3:  #define ID_UP    3
4:  #define ID_DOWN  4
5:
6:  class CPanTiltFilterProperties : public CBasePropertyPage {
7:
8:  public:
9:     static CUnknown * WINAPI CreateInstance(
10:                         LPUNKNOWN lpunk, HRESULT *phr);
11: private:
12:    BOOL OnReceiveMessage(HWND hwnd, ...);
13:    HRESULT OnConnect(IUnknown *pUnknown);
14:    HRESULT OnDisconnect();
15:    HRESULT OnDeactivate();
16:    HRESULT OnApplyChanges();
17:    .  .  .
18:    //Pan Buttons
19:    void OnPanButtonClick(signed short int movement);
```

```
20:    HWND CreatePanButton(HWND hwndParent, signed short int movement);
21:    HWND m_panRightButton;
22:    HWND m_panLeftButton;
23:    signed short int m_PANMAX;
24:    signed short int m_PANMIN;
25:    signed short int m_rightMovement;
26:    signed short int m_leftMovement;
27:    signed short int m_panLocation;
28:    .  .  .
29:    CPanTiltFilterProperties(LPUNKNOWN lpunk, HRESULT *phr);
30:
31:    IPanTiltFilter *pIPanTiltFilter() {
32:        ASSERT(m_pPanTilt);
33:        return m_pPanTilt; };
34: };
```

define the minimum and maximum pan position limits and the `m_rightMovement` and `m_leftMovement` represent the change in the pan position commanded by clicking once the Right and Left buttons, respectively. Finally, `m_panLocation` holds the current value of the pan position, and the two handles to the left and right buttons are declared in lines 21 and 22. The two buttons that will be used to control the left and right position of the pan axis are created via a call to `CreatePanButton`. The body of this function appears in Listing 6.4.1.2. The input arguments are first a handle to the parent window of which the button will be a child and second a `signed short int` argument whose absolute value signifies the change of pan position caused by a single button click. Its sign denotes the sense of direction commanded by the clicking of the respective button. Lines 9–15 show the creation of a left button.

LISTING 6.4.1.2 The `CreatePanButton` function definition.

```
1:  HWND CPanTiltFilterProperties::CreatePanButton(
2:                    HWND hwndParent, signed short int movement) {
3:     HWND panButton;
4:
5:     //Styles for the pan buttons
```

```
6:     ULONG Styles = WS_CHILD | WS_VISIBLE | WS_TABSTOP |
7:                    BS_CENTER | BS_PUSHBUTTON;
8:
9:     //If we were given a positive movement.  Create a left button
10:    if(movement > 0) {
11:        panButton = CreateWindow("BUTTON", "Left", Styles,
12:           10, 70, 50, 50, hwndParent, (HMENU)ID_LEFT, g_hInst, NULL);
13:
14:        //Set the amount of movement for the button
15:        m_leftMovement = movement; }
16:
17:    //Else if we were given a negative movement. Create a right button
18:    else {
19:        panButton = CreateWindow("BUTTON", "Right", Styles,
20:           70, 70, 50, 50, hwndParent, (HMENUID_RIGHT, g_hInst, NULL);
21:
22:        //Set the amount of movement for the button
23:        m_rightMovement = movement; }
24:
25: return panButton; }
```

A single clicking of one of the pan buttons (left or right) invokes the execution of the callback function `OnPanButtonClick` described in detail in Listing 6.4.1.3. In line 6 the class member variable `m_panLocation` is updated, and after a check on the pan bounds is performed (i.e., lines 8–13), the interface function `put_PanLocation` is executed. Interface functions will be defined and discussed in greater detail in the following sections.

LISTING 6.4.1.3 The callback function `OnPanButtonClick` is invoked with a button click.

```
1: void CPanTiltFilterProperties::OnPanButtonClick
2:                 (signed short int movement) {
3:     SetDirty();
4:
5:     //Add the movement input to the current pan location
6:     m_panLocation += movement;
7:
```

```
8:     //Do not allow the new pan location to exceed the pan bounds
9:     if(m_panLocation < m_PANMIN)
10:        m_panLocation = m_PANMIN;
11:
12:    else if(m_panLocation > m_PANMAX)
13:        m_panLocation = m_PANMAX;
14:
15:    //Set the new pan location
16;    pIPanTiltFilter()->put_PanLocation(m_panLocation);
17: }
```

Changes in the properties of dialog controls (button clicks) are handled via a message control loop defined inside the `OnReceiveMessage` function. Three different events are handled by this loop:

- dialog controls initialization
- dialog controls commands
- dialog controls destruction

Dialog Controls Initialization

During initialization of the filter's property page the creation of the dialog control buttons takes place as it is depicted in Listing 6.4.1.4.

LISTING 6.4.1.4 Portion of the `OnReceiveMessage` function invoked during initialization creates the dialog control buttons.

```
1: BOOL CPanTiltFilterProperties::OnReceiveMessage(
2:         HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
3:
4:     switch (uMsg)
5:     {
6:         case WM_INITDIALOG:
7:         {
8:             //Create the pan buttons
9:             m_panLeftButton = CreatePanButton(hwnd,100);
10:            ASSERT(m_panLeftButton);
11:
```

```
12:             m_panRightButton = CreatePanButton(hwnd,-100);
13:             ASSERT(m_panRightButton);
14:
15:             // Create the tilt buttons
16:             ...
17:             return (LRESULT) 1;
18:         }
19:         . . .
20:     }
21:   return CBasePropertyPage::OnReceiveMessage(hwnd,uMsg,wParam,lParam);
22: }
```

Dialog Controls Commands

Changes in the dialog control buttons caused by user interaction are handled by the message control loop according to the code appearing in Listing 6.4.1.5.

LISTING 6.4.1.5 Portion of the `OnReceiveMessage` function handles user commands (button clicks).

```
1: BOOL CPanTiltFilterProperties::OnReceiveMessage(
2:          HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
3:
4:      switch (uMsg)
5:      {
6:          . . .
7:          case WM_COMMAND:
8:          {
9:              if (LOWORD(wParam) == IDB_DEFAULT)
10:                  SetDirty();
11:
12:             //If the pan right button was pressed
13:             else if (LOWORD(wParam) == ID_RIGHT)
14:                 OnPanButtonClick(m_rightMovement);
15:             //If the pan left button was pressed
16:             else if (LOWORD(wParam) == ID_LEFT)
```

```
17:                 OnPanButtonClick(m_leftMovement);
18:
19:             // handling of tilt buttons
20:             ...
21:             return (LRESULT) 1;
22:         }
23:         . . .
24:     }
25:     return CBasePropertyPage::OnReceiveMessage(hwnd,uMsg,wParam,lParam);
26: }
```

Dialog Controls Destruction

Quitting the property page window results in the destruction of dialog controls (lines 7–15) of Listing 6.4.1.6.

LISTING 6.4.1.6 Destruction of the dialog control buttons occurs after quitting the property page window.

```
1: BOOL CPanTiltFilterProperties::OnReceiveMessage(
2:          HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
3:
4:      switch (uMsg)
5:      {
6:          . . .
7:          case WM_DESTROY:
8:          {
9:              //Destroy the buttons
10:             DestroyWindow(m_panRightButton);
11:             DestroyWindow(m_panLeftButton);
12:             DestroyWindow(m_tiltUpButton);
13:             DestroyWindow(m_tiltDownButton);
14:             return (LRESULT) 1;
15:         }
16:     }
17:     return CBasePropertyPage::OnReceiveMessage(hwnd,uMsg,wParam,lParam);
18: }
```

6.4.2 *The Pan-Tilt Interface*

The pan-tilt filter interface allows the communication between the filter and its property page(s). The interface functions are part of the `IPanTiltFilter` interface and are declared in the header file *iPanTiltFilter.h*. For the pan-tilt filter we develop in this chapter, we define eight interface functions (four for setting and getting the pan-tilt locations of the device, two for getting the minimum and maximum limits of the two axes, and two for setting default values of pan and tilt positions.) A typical interface function declaration (to set the device's tilt position) appears below.

```
DECLARE_INTERFACE_(IPanTiltFilter, IUnknown) {
 STDMETHOD(get_PanLocation) (THIS_ signed short int
 *PanLocation)
 PURE; };
```

The same interface functions are also declared as member functions of the `CPanTiltFilter` class and defined in detail in the files *PanTiltFilter.h* and *PanTiltFilter.cpp*.

6.4.3 *Development of the Pan-Tilt Filter*

In this section we describe the last part of the software that completes the development of the pan-tilt filter. To develop this filter we have followed the steps that we described in Chapter 4 during the development of the CD filter. Therefore, in this section we will only describe the points that are distinct to the development of the pan-tilt filter. The complete code appears in files *PanTiltFilter.h* and *PanTiltFilter.cpp*. The filter is defined as a class object named `CPanTiltFilter` that inherits from the DirectShow classes of `CTransInPlaceFilter`, `IPanTiltInterface`, and `ISpecifyPropertyPages`. A portion of the code that defines the `CPanTiltFilter` class is shown in Listing 6.4.3.1.

LISTING 6.4.3.1 The `CPanTiltFilter` class definition.

```
1:  class CPanTiltFilter : public CTransInPlaceFilter,
2:   public IPanTiltInterface, public ISpecifyPropertyPages {
3:
```

```
4: . . .
5:   // These implement the custom IPanTiltFilter interface
6:   STDMETHODIMP I_GetPanLocation(signed short int *PanLocation);
7:   STDMETHODIMP I_GetTiltLocation(signed short int *TiltLocation);
8:   STDMETHODIMP I_PutDefaultPanLocation(void);
9:   STDMETHODIMP I_PutDefaultTiltLocation(void);
10:  STDMETHODIMP I_PutPanLocation(signed short int PanSliderLoc);
11:  STDMETHODIMP I_PutTiltLocation(signed short int TiltSliderLoc);
12:  STDMETHODIMP I_GetTiltMaxMinValues(signed short int *TiltMax,
13:                                      signed short int *TiltMin);
14:  STDMETHODIMP I_GetPanMaxMinValues(signed short int *PanMax,
15:                                     signed short int *PanMin);
16:
17:  //Pan / Tilt Members
18:  void SerialPortInit();
19:
20:  //Pan-Tilt Device
21:  CPanTiltDevice* m_PanTiltDevice;
22:
23:  //Pan members
24:  const signed short int m_DefaultPanLocation;
25:  long             m_max_PAN;
26:  long             m_min_PAN;
27:  signed short int m_PAN;
28:  BOOL m_bPanChangeFlag;
29:  . . .
30: }
```

As we explained in the previous section, the interface functions are declared (see lines 6–15 of Listing 6.4.3.1) and implemented as part of the filter's functional interface. In Listing 6.4.3.2, we have chosen to describe the `I_PutTiltLocation` interface function, which performs two tasks: (1) it assigns the new value of tilt position to the member variable `m_TILT` (line 4 of Listing 6.4.3.2), and (2) it sets the `m_bTiltChangeFlag` (line 5 of Listing 6.4.3.2).

LISTING 6.4.3.2 The `I_PutTiltLocation` interface function.

```
1:  STDMETHODIMP CPanTiltFilter::I_PutTiltLocation(
2:                    signed short int newTiltLocation) {
```

```
3:      CAutoLock cAutoLock(&m_PanTiltLock);
4:      m_TILT = newTiltLocation;
5:      m_bTiltChangeFlag = TRUE;
6:      return NOERROR;
7:  } // I_PutTiltLocation
```

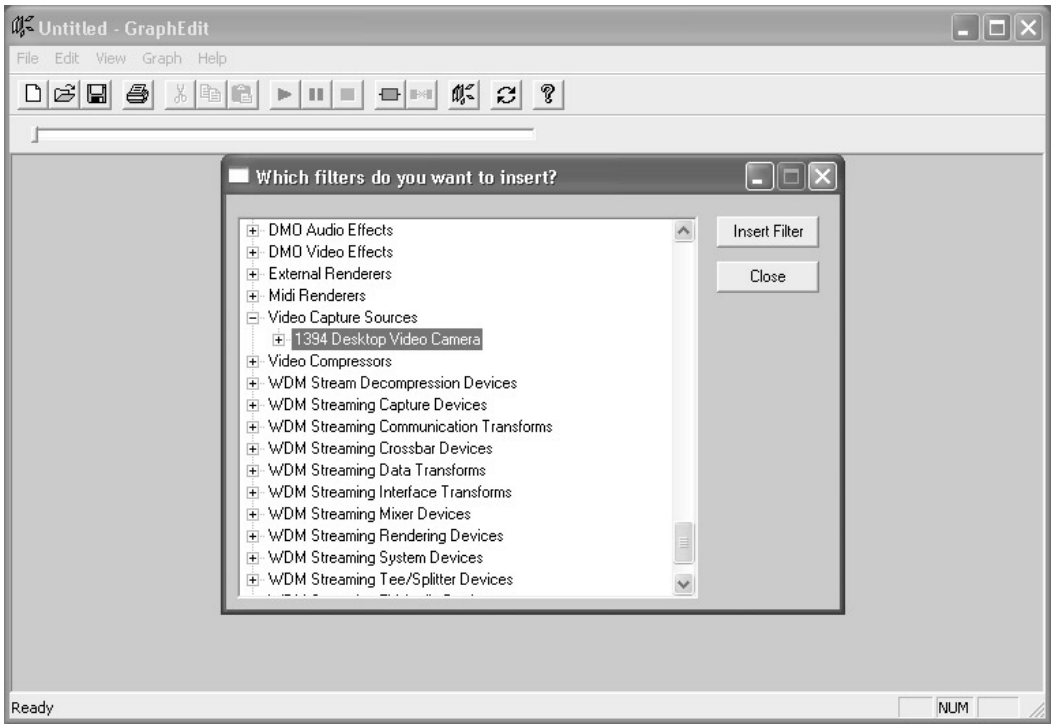
Once a user command has been entered (via the interface functions), the filter passes it on to the filter's `m_PanTiltDevice` object (line 21 of Listing 6.4.3.1), which eventually executes it. A group of member variables appearing in lines 24–28 of Listing 6.4.3.1 are associated with the control of the pan axis of the pan-tilt device. A similar group is defined for the tilt axis. The pan-tilt filter we are developing has one input and one output pin, and video is passed from one end to the other unchanged. Lines 4–9 of Listing 6.4.3.3 are detailing the `Transform` function that describes this functionality. In addition to this, code from lines 12–21 respond to changes made in the filter's properties that signal controlling commands of the pan-tilt filter.

LISTING 6.4.3.3 The `Transform` function definition.

```
1:  HRESULT CPanTiltFilter::Transform(IMediaSample
2:                                    *pMediaSample) {
3:
4:     AM_MEDIA_TYPE* pType = &m_pInput->CurrentMediaType();
5:     VIDEOINFOHEADER *pvi = (VIDEOINFOHEADER *) pType->pbFormat;
6:     BYTE *pData;    // pointer to the data from the input pin
7:     long lDataLen;  // holds length of any given sample
8:     pMediaSample->GetPointer(&pData);
9:    lDataLen = pMediaSample->GetSize(); //get the size of input
10:
11:    //Respond to changes made in the pan-tilt property page
12:    if(m_bPanChangeFlag = TRUE) {
13:        m_PanTiltDevice->set_desired(PAN, POSITION,
14:        (PTU_PARM_PTR *) &m_PAN, ABSOLUTE);
15:        m_PanTiltDevice->await_completion();
```

```
16:         m_bPanChangeFlag = FALSE; }
17:     if(m_bTiltChangeFlag = TRUE) {
18:         m_PanTiltDevice->set_desired(TILT, POSITION,
19:         (PTU_PARM_PTR *) &m_TILT, ABSOLUTE);
20:         m_PanTiltDevice->await_completion();
21:         m_bTiltChangeFlag = FALSE; }
22:
23:     return NOERROR;
24: }
```

FIGURE 6.5 The first step in creating a graph for running the pan-tilt filter is to include a filter for capturing video from an external source (via a USB or FireWire interface).



6.5 Running the CPanTiltFilter Using the GraphEdit Utility

We conclude this chapter by outlining a step-by-step process for running the pan-tilt filter using the GraphEdit utility.

1. First and foremost, we need to make sure that the filter (which as we mentioned earlier represents a COM object) is registered. This is done by calling regsvr32.exe on the *.ax* file we created during the compilation of the PTFilter project.
2. Next, we launch the GraphEdit utility and create a new *graph* for running the filter. We begin creating a new graph by incorporating a filter for getting a video input. This is shown in Figure 6.5. In our

FIGURE 6.6 The second step in creating a graph for running the pan-tilt filter is to include the pan-tilt filter we have developed.

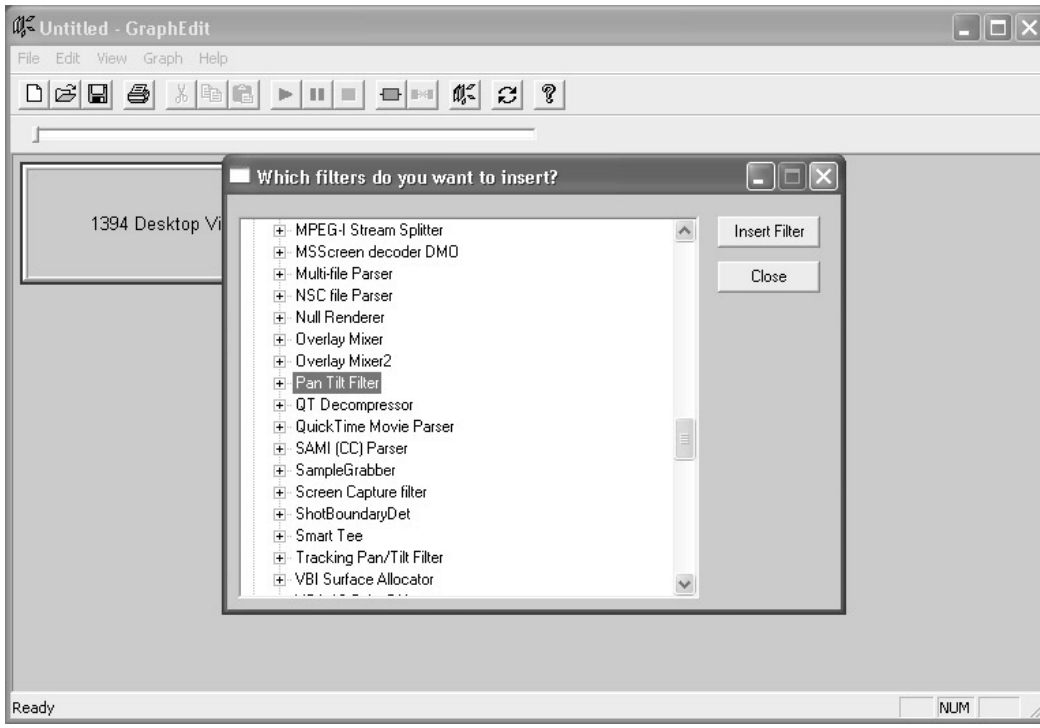
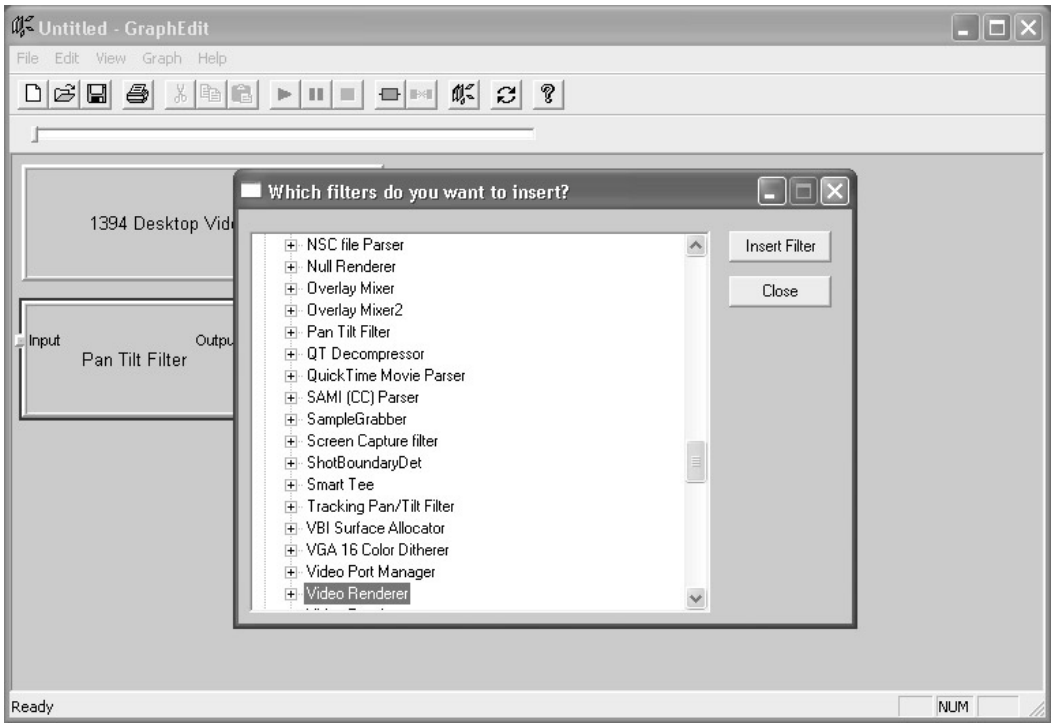


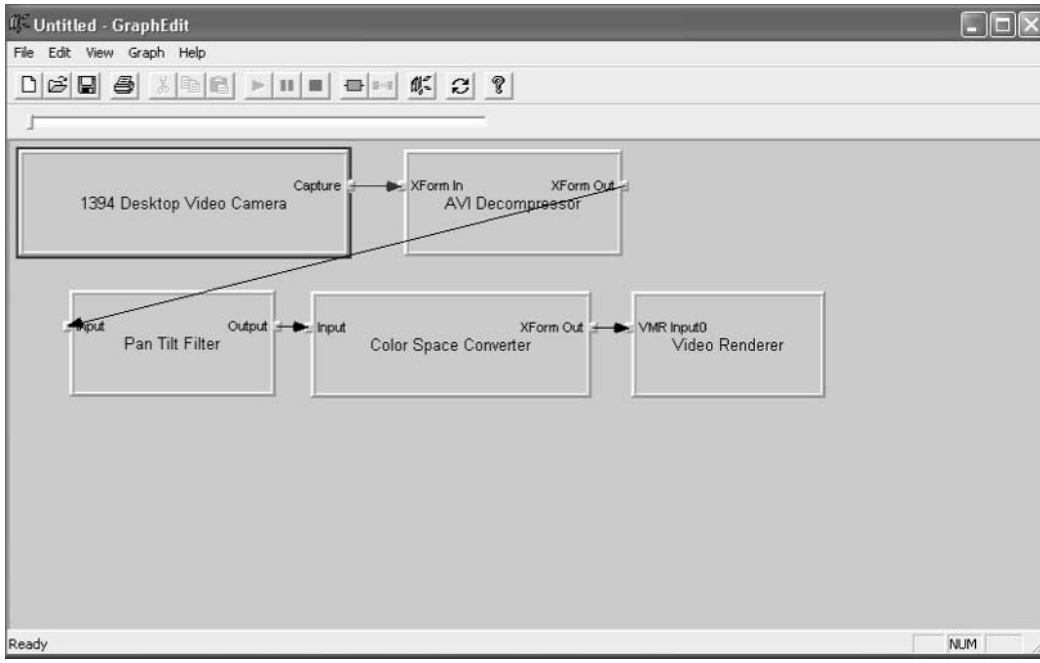
FIGURE 6.7 The third step in creating a graph for running the pan-tilt filter is to include a filter for rendering the video data.



case video is taken from a IEEE-1394 (or FireWire) iBot (Orange Micro) camera.

3. We continue by including the pan-tilt filter we developed earlier (Figure 6.6.)
4. A filter for displaying the output video sequence, the video renderer (in our case the input video taken by the camera will be output unchanged), is included as shown in Figure 6.7.
5. Finally, we connect the various filter objects as shown in Figure 6.8. The graph that we have created can be saved for later use.
6. We can now run the graph by pressing the `Run` button (a green triangle in the task bar). By right clicking on the pan-tilt filter

FIGURE 6.8 The final step in creating a graph for running the pan-tilt filter is to define the data flow between the various filter objects by connecting them appropriately.



object of the graph we open the property page of the filter (see Figure 6.4), and the user can control the pan-tilt position of the device by pressing the appropriate buttons.

6.6 Summary

In this chapter, we showed how to connect a pan-tilt device to a computer-camera system and control it using a DirectShow filter. In the next chapter, we will build upon the pan-tilt filter to create a new filter that will track a moving person in the camera's field of view and move the camera, via the pan-tilt device, to "follow" the person as he/she moves away from the camera's field of view.

CHAPTER 7

A Custom Filter for Object Tracking

Based upon the mean-shift algorithm, we describe step-by-step the implementation of a filter that follows a moving object. Specifically, the user draws a rectangle around the object to be tracked, and the filter tracks the particular object by attaching a bounding rectangle around it as it moves within the video image. Moreover, when the object moves toward the video image's boundaries, a computer-controlled pan-tilt mechanism is automatically engaged to keep the object of interest within the camera's field of view. A modified version of such a filter can be used in sophisticated videoconference or surveillance applications to keep the subject always in the center of the camera's field of view even when he or she is moving around. We will provide the theoretical foundations of the mean-shift algorithm and describe the detailed forms of those parameters that are essential in a practical implementation of the mean-shift procedure. We will finally describe the development of an object-tracking filter based on the mean-shift methodology. ■

7.1 The Theory of Mean-Shift Algorithm

The object-tracking filter that we will develop in this chapter is based on the mean-shift theory. Mean shift is an iterative process that in its most standard form is equivalent to a clustering methodology. That is, a group of neighboring points in a Euclidean space (i.e., the RGB color space of a video image) is represented by a single point located at the neighborhood's average. Mean-shift is considered a generalization of the popular *k-means* clustering algorithms. Some definitions that will help you understand the mean-shift algorithm will be discussed next. For more details the reader is referred to the work of Cheng [2].

Definition 1. If X is an n-dimensional Euclidean space, then a function $K : X \to R$ is called a kernel function, if there exists a *profile function* $k : [0, \infty] \to R$, such that $K(x) = k(\|x\|^2)$, with the following properties:

1. $k(x) \geq 0$.
2. $k(x)$ is nonincreasing: $k(a) \geq k(b)$ if $a < b$.
3. $k(x)$ is piecewise continuous and $\int_0^\infty k(u)\, du < \infty$.

Definition 2. Based on the first definition we can define the mean-shift algorithm in its most generalized form as follows. Let $U \subset X$ be a finite sample data set. Let also K be a kernel and $w : U \to (0, \infty)$ a weight function. The sample mean for an initial point $x \in X$ can then be defined as

$$m(x) = \frac{\sum_{u \in U} K(u - x)w(u)u}{\sum_{u \in U} K(u - x)w(u)} \tag{7.1.1}$$

The mean-shift algorithm can now be seen as the generation of the sequence of points: $u, m(u), m(m(u)), \ldots$, forming the trajectory that starts from the initial point u and ends at the fixed point u^* [i.e., $m(u^*) = u^*$] (Figure 7.1). The fixed point is also called the cluster center. For a special case of kernels called *shadow* kernels the mean-shift quantity, which will be defined later, provides an estimate of the gradient of an associated data density.

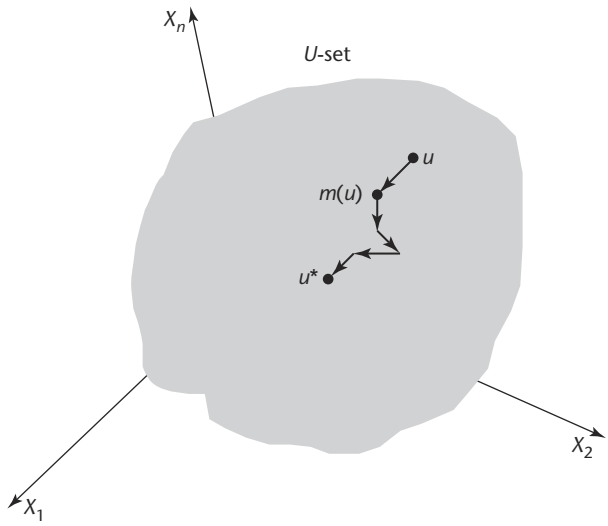


FIGURE 7.1
Visualization of the mean shift iterations. Starting from an initial point u mean shift iterations lead to the fixed point u^* (i.e., the center of the data cluster $U \subset X$).

Definition 3. The kernel H is a shadow of kernel K, if the mean-shift quantity defined by

$$m(x) - x = \frac{\sum_{u \in U} K(u - x)w(u)u}{\sum_{u \in U} K(u - x)w(u)} - x \tag{7.1.2}$$

points to the direction (at the point x) of the density estimate $g(x)$ defined using the shadow kernel H as follows:

$$g(x) = \sum_{u \in U} H(u - x)w(u) \tag{7.1.3}$$

With some mathematical manipulation one can prove that

$$m(x) - x = \frac{\nabla g(x)}{2cp(x)} \tag{7.1.4}$$

where $p(x) = \sum_{u \in U} K(u - x)w(u)$. Gaussian kernels are characterized by the fact that they are shadows of themselves. In this case $g(x) = p(x)$, and since $\frac{\nabla g(x)}{g(x)} = \nabla(\log p(x))$ one can find that the mean-shift step becomes

$$m(x) - x = -\beta x, \quad \beta > 0 \tag{7.1.5}$$

and the mean shift points to the gradient direction of the density estimate.

7.2 A Practical Implementation of the Mean-Shift Algorithm

Recently, there has been a deployment of the mean-shift algorithm for tracking objects based on a suite of features that include their color content, gradient of pixel intensity, etc. In those cases tracking becomes robust in the presence of partial occlusions and changes in camera positions. This last property makes this methodology appropriate for use in tracking applications that use pan-tilt devices. Tracking using the mean-shift algorithm begins by building an appearance model (also called the *target model*) of the object to be tracked (the appearance model in our case is based solely on the color content of the object). An initial estimate of the position of the object must also be provided. Tracking of the object is then reduced to finding the trajectory produced by applying the mean-shift procedure starting from an initial point that represents the position of the object in the current image. Convergence of the mean-shift algorithm on each frame is decided via a matching criterion between the color histogram distribution of target model and that of a *target candidate*.

7.2.1 *The Color Histogram Distribution of the Target Model*

The object that will be tracked by the mean-shift algorithm must be localized first. Object localization can be either automatic (e.g., one may deploy a face detection procedure to locate a face in a video image and then deploy the mean shift for tracking) or manual. In this book we have chosen the latter. The user draws a rectangle around the object to be tracked indirectly determining the pixel coordinates of the center of the target object. Once the object of interest has been localized, the RGB probability distribution of the target model is formed as the 3D (RGB) color histogram of the weighted RGB values of the pixels contained in a rectangle of size (h_x, h_y) centered at the target model's center. We will now explain how to describe this histogram distribution mathematically. Let us denote $\bar{x}_j$ (with

$j = 1, \ldots, n$) as the relative coordinates (with respect to the center of the rectangle) of the target model's pixels contained in the rectangle. For computational reasons, we approximate the target model's distribution using m number of bins per color channel (we have actually used 32 bins per channel). Pixel color intensities are weighted using the *Epachelnikov* profile of radius r defined by Equation 7.2.1.

$$k_E(x) = \begin{cases} \frac{2}{\pi r^2}(1 - x), & \text{if } x < 1 \\ 0, & \text{otherwise.} \end{cases} \tag{7.2.1}$$

We now define the function $\lambda : \mathbb{R}^2 \rightarrow M \in [1, 2, \ldots, m^3]$ that maps the pixel at location $\bar{x}_j$ to the index $\lambda(\bar{x}_j)$ of the 3D color histogram. Given the above definitions, the color histogram distribution of the target model is given by Equation 7.2.2. We can easily prove that $\sum_{s=1}^{M} \theta_s = 1$. As usual $\delta(\cdot)$ represents the Dirac function.

$$\theta_s = \frac{\sum_{j=1}^{n} k_E(\|\bar{x}_j\|^2)\delta[\lambda(\bar{x}_j) - s]}{\sum_{j=1}^{n} k_E(\|\bar{x}_j\|^2)} \tag{7.2.2}$$

7.2.2 The Color Histogram Distribution of the Target Candidate

In a similar fashion we derive the color histogram distribution $\phi_s(y)$ of a target candidate centered at an image location y. This location is the estimate of the new image location to which the object being tracked has moved. The color histogram distribution will be formed similarly, as we did with that of the target model by considering those pixels x_j that belong to the circular region of radius h centered at the pixel location y. The probability distribution of the target candidate can then be represented by Equation 7.2.3. We should note that n^* represents the total number of pixels in the respective target candidate's rectangle.

$$\phi_s(y) = \frac{\sum_{j=1}^{n^*} k_E\left(\left\|\frac{y - x_j}{h}\right\|^2\right)\delta[\lambda(x_j) - s]}{\sum_{j=1}^{n^*} k_E\left(\left\|\frac{y - x_j}{h}\right\|^2\right)} \tag{7.2.3}$$

7.2.3 *Object Tracking via Maximization of Color Histogram Matching*

The mean-shift algorithm for object tracking discussed in this book is being proposed by Comaniciu et al. [3]. Given the representations of the target model and target candidate color histogram distributions, object tracking is treated as a distance minimization problem between the model's and candidate's weighted histograms. Specifically, the image location y_{next} that maximizes the *Bhattacharyya* coefficient $\rho(y_{\text{next}})$ is the estimate of the new location of the object being tracked in the next video frame. The reader should note that the Bhattacharyya coefficient is a location-dependent quantity (see Equation 7.2.4).

$$\rho(y_{\text{next}}) \equiv \rho(\phi(y_{\text{next}}), \theta) = \arg\max_{y} \rho(\phi(y), \theta) = \arg\max_{y} \sqrt{\phi(y)\theta} \tag{7.2.4}$$

Maximization of the Bhattacharyya coefficient is equivalent to the minimization of the quantity d that has the properties of a distance metric and is related to the Bhattacharyya coefficient as shown in Equation 7.2.5.

$$d = \sqrt{1 - \rho(\phi(y), \theta)} \tag{7.2.5}$$

The Bhattacharyya coefficient can be approximated using a Taylor series expansion around the location y_{curr} of the target model's location in the current image frame according to Equation 7.2.6. It is assumed that the image location of the object being tracked does not change dramatically from frame to frame.

$$\rho[\phi(y), \theta] \approx \frac{1}{2}\sum_{s=1}^{M} \sqrt{\phi_s(y_{\text{curr}})\theta_s} + \frac{1}{2}\sum_{s=1}^{M} \phi_s(y)\sqrt{\frac{\theta_s}{\phi_s(y_{\text{curr}})}} \tag{7.2.6}$$

Taking into account the form of the target candidate distribution given by Equation 7.2.3 the Bhattacharyya coefficient can be further approximated by Equation 7.2.7,

$$\rho[\phi(y), \theta] \approx \frac{1}{2}\sum_{s=1}^{M} \sqrt{\phi_s(y_{\text{curr}})\theta_s} + \frac{C}{2}\sum_{j=1}^{n^*} w_j k\left(\left\|\frac{y - x_j}{h}\right\|^2\right) \tag{7.2.7}$$

where the coefficient C and the weights w_j are given by Equations 7.2.8 and 7.2.9, respectively.

$$C = \frac{1}{\sum_{j=1}^{n^*} k\left(\left\|\frac{y - x_j}{h}\right\|^2\right)} \tag{7.2.8}$$

$$w_j = \sum_{s=1}^{M} \delta[\lambda(x_j) - s]\sqrt{\frac{\theta_s}{\phi_s(y)}} \tag{7.2.9}$$

Given the expression of the weights w_j and also that the *biweight* kernel is a shadow kernel of the Epachelnikov kernel we can prove that the mean-shift update of the algorithm points to the gradient direction of the distribution $g(x)$ (Equation 7.1.3) where H represents the biweight kernel.

7.3 The Mean-Shift Tracking Algorithm

In this section we will describe how the mean-shift tracking algorithm can be used to track an object based on its color appearance model within a video sequence. In particular we will show the steps involved in finding the location of the target object y_{next} in the next video frame given its location y_{curr} in the current frame.

1. Current Frame—(t): This frame can be either the initial frame (i.e., $t = 0$) or an intermediate frame (t) during an object-tracking task. If the former is the case, then as we explained earlier the user must specify the location of the object to be tracked by drawing a rectangle around it. However, if this is not the case, then y_{curr} represents the estimate of the position of the object being tracked as computed by the object-tracking algorithm. Once the position of the object in the current frame has been determined we can compute the weighted color histogram distribution of the object (Equation 7.2.2). In both cases we use the Epachelnikov profile for the computation of the color histogram distributions.
2. Next Frame—($t + dt$):
   (a) In this frame we compute the target candidate's color histogram distribution $\phi(y_{\text{curr}})$ (Equation 7.2.3). This is equivalent to making the assumption that the object has not moved between the two frames.

(b) We then compute the Bhattacharyya coefficient for the target model and candidate distributions $\rho[\phi(y_{\text{curr}}), \theta] = \sum_{s=1}^{M} \sqrt{\phi_s(y_{\text{curr}})\theta_s}$.

(c) The values of the weights w_j (Equation 7.2.9) are computed for $\phi(y_{\text{curr}})$.

(d) Assuming the use of the Epachelnikov profile, the mean-shift quantity y_{next}, which represents the first estimate of the position of the target object in frame $(t + dt)$, is being computed as a weighted average of the image pixels x_j contained in the circular region of radius h centered at the pixel location y_{curr} and is given by Equation 7.3.1.

$$y_{\text{next}} = \frac{\sum_{j=1}^{n^*} x_j w_j}{\sum_{j=1}^{n^*} w_j} \tag{7.3.1}$$

(e) For the estimated position y_{next} we compute a new weighted color histogram $\phi(y_{\text{next}})$ and the corresponding Bhattacharyya coefficient $\rho[\phi(y_{\text{next}}), \theta] = \sum_{s=1}^{M} \sqrt{\phi_s(y_{\text{next}})\theta_s}$.

(f) In case $\rho[\phi(y_{\text{next}}), \theta] \leq \rho[\phi(y_{\text{curr}}), \theta]$, we update the estimate of the position of the object via $y_{\text{next}} \leftarrow \frac{1}{2}(y_{\text{next}} + y_{\text{curr}})$ and repeat the process by going back to step 2e.

(g) In case the Bhattacharyya coefficient does not decrease any further, we check whether $\|y_{\text{next}} - y_{\text{curr}}\| \leq \varepsilon$. If this condition is satisfied, then convergence has been reached, the final estimate of the target's position is y_{next}, and the iteration process stops. However, if this is not the case, we assign $y_{\text{curr}} \leftarrow y_{\text{next}}$ and iteration continues from step 2b.

We will now describe the software components that implement a pan-tilt object-tracking filter based on the mean-shift theory, as described by the algorithm above. We begin the tracking filter's description by first considering supporting class objects and data structures.

7.4 The Color Histogram `CColorHistogramIndex` Class

As we explained earlier, object tracking using the mean-shift algorithm is reduced to the problem of matching color histogram distributions.

The class `CColorHistogramIndex` defines member variables and functions that store and manipulate the color histogram distribution of an area in a video image. The definition of this class is shown in Listing 7.4.0.1. The member variables `m_width` and `m_height` hold the dimensions of the video image while the variable `m_NumberOfBins` fixes the number of bins used per color channel. This latter variable also determines the size of the 3D (RGB) histogram (`m_RGBHistogram`) associated with a video image.

LISTING 7.4.0.1 Definition of the `CColorHistogramIndex` class.

```
1: class CColorHistogramIndex {
2:     public:
3:         CColorHistogramIndex();
4:         CColorHistogramIndex(int width, int height, int numberofbins);
5:         virtual ~CColorHistogramIndex();
6:
7:         // The RGB Color Histogram
8:         float*** m_RGBHistogram;
9:
10:        int m_width;
11:        int m_height;
12:        int m_NumberOfBins;
13:        float** m_Weights;
14:
15:        //bin indexes
16:        int** m_RedBinIndex;
17:        int** m_GreenBinIndex;
18:        int** m_BlueBinIndex;
19:
20:        void ClearRGBHistogram();
21:        void InitializeBinIndexes();
22:        void InitializeWeights();
23: };
```

A final set of member variables (i.e., `m_RedBinIndex`, `m_GreenBinIndex`,`m_BlueBinIndex`) indexes the position of a (i, j) pixel in the 3D (RGB) histogram space.

Finally, the functions `ClearRGBHistogram`, `InitializeBinIndexes`, and `InitializeWeights` initialize the histogram values the three (RGB) bin indexes, and the weight arrays, respectively.

7.5 The `CModelParameters` Class

The `CModelParameters` class is introduced to speed up the computation of mean-shift tracking. Mean shift examines rectangular areas in a video image. This class implements a look-up table for those lattice points contained in the rectangular area that are also within the unit circle centered at the center of the rectangle, assuming that the rectangle's dimensions have been normalized. For these points it computes the parameters that weigh the color histogram of the rectangular subimage. The `CModelParameters` class definition is shown in Listing 7.5.0.1. The class's member variables include the dimensions of the rectangular area for which we create the look-up table (lines 8 and 9), a 2D array of Booleans (`m_nonZeroMask`) masking out the lattice points outside the unit circle, and the 2D array of weights [`m_profileParams` representing $k_E(\| \cdot \|^2)$] used in the computation of the color histogram distributions (Equations 7.2.2 and 7.2.3). You

LISTING 7.5.0.1 Definition of the `CModelParameters` class.

```
1: class CModelParameters {
2:     public:
3:         CModelParameters(void);
4:         ~CModelParameters(void);
5:
6:         void Initialize(int half_height, int half_width);
7:
8:         int m_half_height;
9:         int m_half_width;
10:
11:        float** m_profileParams;
12:        BOOL** m_nonZeroMask;
13:        float m_profileParamsSum;
14: };
```

should keep in mind that in this implementation we are using the Epachelnikov profile to form weighted color histogram distributions. Finally, the value of the denominator of the same equations, which is the normalization parameter, is stored in `m_profileParamsSum`.

7.6 Implementation of the Mean-Shift Object-Tracking Filter

The pan-tilt tracking filter belongs to the category of transform filters and is largely based upon the `CPanTiltFilter` class object we developed in Chapter 6. It is actually an extension to it, and our description of the tracking filter will reflect that. The reader can follow the exact same steps (which we deliberately omitted in this section) described in Chapter 4 for defining the `CPTTrackingFilter` filter class. The complete code can be found in the *PTTracking.h* and *PTTracking.cpp* files. The `CPTTrackingFilter` class extends the structure and functionality of the `CPanTiltFilter` class by incorporating the color-based object-tracking component. We will concentrate on describing only these particular extensions. We begin with the filter's interface as we did in an earlier filter implementation.

7.6.1 *The `IPTTrackingInterface` of the Tracking Filter*

The `IPTTrackingInterface` builds upon the `IPanTiltInterface` and includes three additional interface functions as shown in Listing 7.6.1.1. The complete interface declaration can be found in *iPTTrackingFilter.h*. The `I_PutTrackingCenter` function (line 4 of Listing 7.6.1.1) facilitates user interaction with the filter, enabling the user to identify the object to be tracked. In addition to this, the initial location of object to be tracked is indirectly determined and used by the mean-shift initialization process. The other two functions (lines 7 and 9 of Listing 7.6.1.1) initiate and terminate a tracking task, respectively. In Chapter 6 the `IPanTiltInterface` mediated communication between the filter and its property page. In the case of the object-tracking filter, however, the `IPTTrackingInterface` will mediate interactions between the user and the filter not via property pages but via a dialog window that contains buttons, edit boxes, and

LISTING 7.6.1.1 Interface function definitions for implementing the object tracking.

```
1: DECLARE_INTERFACE_(IPTTrackingInterface, IUnknown) {
2:
3:  . . .
4:     STDMETHOD(I_PutTrackingCenter) (THIS_
5:         int x, int y )PURE;
6:
7:     STDMETHOD(I_PutStartTracking) (THIS)PURE;
8:
9:     STDMETHOD(I_PutStopTracking) (THIS)PURE;
10: };
```

other controls. This will be demonstrated in Chapter 8, where we develop the application that will run the tracking filter.

7.6.2 *The Structure of the `CPTTrackingFilter` Class*

In this section, we describe only those portions of the filter's member variable structure that are pertinent to the implementation of mean shift–based object tracking. The filter's member variables appear in Listing 7.6.2.1. The tracking filter is implemented as a filter that transforms data in place. It inherits the functionality described

LISTING 7.6.2.1 Listing of `CPTTrackingFilter`'s member variables required for implementing mean-shift object tracking.

```
1: typedef struct Pixel{
2:         int x;
3:         int y;
4: }Pixel;
5:
6: class CPTTrackingFilter : public CTransInPlaceFilter,
7:         public IPTTrackingInterface,
8:         public ISpecifyPropertyPages {
9:  . . .
10:        BOOL m_bInitialTrackingFlag;
11:        BOOL m_bTrackingBoxFlag;
12:        BOOL m_bStartTrackingFlag;
13:
```

```
14:        Pixel m_InitialBoxCOG;
15:        Pixel m_CurrentBoxCOG;
16:
17:        //Color Indexes for Target and Target Candidate Models
18:        CColorHistogramIndex* m_QTarget;
19:        CColorHistogramIndex* m_PCandidate;
20:        CColorHistogramIndex* pColorHist;
21:
22:        int m_CurrentBoxHalfWidth;
23:        int m_CurrentBoxHalfHeight;
24:        CModelParameters m_ModelParameter;
25:        CModelParameters m_CandidateParameter;
26:
27:        float** m_CandidateWeights;
28:    . . .
29: };
```

by the `IPTTrackingInterface` and the `ISpecifyPropertyPages` (although in this version of the implementation the user interface is not realized via property pages). Lines 1–4 define a data structure (called `Pixel`) that holds the (x, y) image coordinates of a pixel. The `m_bInitialTrackingFlag` Boolean engages automatic pan-tilt control to always keep the object being tracked in the camera's field of view. The Boolean variable `m_bTrackingBoxFlag`, which is *not* initially set (i.e., its value is `FALSE`), is set only after the user has indicated to the filter which object to track by drawing a rectangle around the object shown in a video window (Figure 7.2). Finally, another Boolean variable (`m_bStartTrackingFlag`) is set (or unset) via the `::I_PutStartTracking` (`::I_PutStopTracking`) interface function, which is invoked by the user through the respective button to start (or stop) object tracking. The variable `m_InitialBoxCOG` records the center of the bounding rectangle around the object of interest that the user draws. On the other hand, `m_CurrentBoxCOG` retains the continuously updated image coordinates of the object's bounding rectangle center that moves along with the object as the latter changes position within the image window.

The mean shift–based tracking filter we are implementing is based on matching color histogram distributions. A pointer to `m_QTarget` holds the values of the `CColorHistogramIndex` object class

FIGURE 7.2 The user initiates object tracking by drawing a rectangle around the object of interest.

associated with the pixels contained in a rectangular region drawn by the user around the tracked object. Similarly, a pointer to the `m_PCandidate` holds the respective data associated with a rectangular image area centered around a pixel location that represents an estimate of the new location of the tracked object. A third pointer of the same type `pColorHist` facilitates the implementation of the mean-shift algorithm described in Section 7.3. We conclude this section by introducing two integer variables that represent the dimensions of the rectangle that is automatically drawn around the moving object (`m_CurrentBoxHalfWidth`, `m_CurrentBoxHalfHeight`).

7.6.3 *The `CPTTrackingFilter` Tracking Methods*

We continue with the discussion of the methods that determine the functional behavior of the object-tracking filter. Our description will

LISTING 7.6.3.1 Methods of the `CPTTrackingFilter` class associated with the implementation of mean-shift object tracking.

```
1: class CPTTrackingFilter : public CTransInPlaceFilter,
2:         public IPTTrackingInterface,
3:         public ISpecifyPropertyPages {
4:  . . .
5:         /////  Filter Interface functions  /////
6:         STDMETHODIMP I_PutStartTracking(void);
7:         STDMETHODIMP I_PutStopTracking(void);
8:         STDMETHODIMP I_PutTrackingCenter(int x, int y);
9:
10:        /////     Tracking Functions  //////////
11:        void CaptureCandidateRGBHistogram(BYTE* pData,
12:                        CColorHistogramIndex* pIndex, Pixel COG);
13:        void CaptureModelRGBHistogram(BYTE* pData,
14:                        CColorHistogramIndex* pIndex, Pixel COG);
15:        void DrawCandidateTargetRegion(BYTE* pData,Pixel COG);
16:        void MeanShiftTracking(BYTE* pData);
17:        void TrackPanTiltDevice();
18:   . . .
19: };
```

reflect on the human–computer interaction requirements for running the filter, as we will demonstrate in Chapter 8. This list of functions shown in Listing 7.6.3.1 is composed of the filter's interface functions (lines 6–8) and the tracking functions (lines 11–17).

The `I_PutTrackingCenter` Interface

Mean-shift object tracking begins only when the center of the object to be tracked has been input. The `I_PutTrackingCenter` function assigns the initial pixel locations for the target model and target candidate. This happens in lines 3, 4 and 5, 6 respectively of Listing 7.6.3.2. Finally, the Boolean `m_bTrackingBoxFlag` is assigned the `TRUE` value to indicate that the above rectangle centers have been set.

LISTING 7.6.3.2 The `I_PutTrackingCenter` interface function.

```
1: STDMETHODIMP CPTTrackingFilter::I_PutTrackingCenter(int x, int y)
2: {
3:     m_InitialBoxCOG.x = x;
4:     m_InitialBoxCOG.y = y;
5:     m_CurrentBoxCOG.x = x;
6:     m_CurrentBoxCOG.y = y;
7:
8:     m_bTrackingBoxFlag = TRUE;
9:
10:    return NOERROR;
11: }
```

The `I_PutStartTracking` Interface

This interface function defines the dimensions (lines 3 and 4 of Listing 7.6.3.3) of the rectangular area upon which the computation of the weighted color histogram for the target candidate is based. Two Boolean variables are set next. The `m_bStartTrackingFlag` Boolean is set to `TRUE`, thus denoting that object tracking may begin. For a particular tracking task the target model's color histogram is computed only once. The `m_bInitialTrackingFlag` variable enables the computation of the target model's color histogram. This variable is set to `TRUE` and remains this way until the initialization has been completed inside the `Transform` function. Finally, an error message is displayed if the `m_bTrackingBoxFlag` is not set to `TRUE`. This means that the user has not drawn an initial box.

LISTING 7.6.3.3 The `I_PutStartTracking` interface function.

```
1: STDMETHODIMP CPTTrackingFilter::I_PutStartTracking()
2: {
3:     m_CurrentBoxHalfWidth = CANDIDATE_HW;
4:     m_CurrentBoxHalfHeight = CANDIDATE_HH;
5:     m_bStartTrackingFlag = TRUE;
```

```
6:      m_bInitialTrackingFlag = TRUE;
7:
8:      if(m_bTrackingBoxFlag == FALSE)
9:          MessageBox(NULL,"You Must Select An Initial Tracking Box",
10:                     "NOTICE",MB_OK);
11:
12:     return NOERROR;
13: }
```

The `I_PutStopTracking` *Interface*

Listing 7.6.3.4 shows the counterpart to the function above. A simple assignment of the `m_bStartTracking` Boolean occurs in line 3, disabling calls to `MeanShiftTracking` (inside `Transform`), thus preventing initiation of new object-tracking tasks.

LISTING 7.6.3.4 The `I_PutStopTracking` interface function.

```
1: STDMETHODIMP CPTTrackingFilter::I_PutStopTracking()
2: {
3:      m_bStartTrackingFlag = FALSE;
4:
5:      return NOERROR;
6: }
```

The `CaptureModelRGBHistogram` *Method*

This function is the realization of the color histogram distribution expression given by Equation 7.2.2. Its input argument list includes three variables. They are as follows:

1. The RGB pixel values of the current image, stored in `*pData`.
2. The output pointer `*pColorIndex`, which will hold the values of the `CColorHistogramIndex` class object associated with a rectangular area centered around the third input variable (i.e., `Pixel COG`).
3. The image coordinates (`Pixel COG`) of the rectangle's center that bounds the tracked object.

Lines 7 and 8 of Listing 7.6.3.5 clear the histogram values and initialize the bin indexes. A double `for` loop enables traversing all the pixels contained in the rectangular area specified by its center and its dimensions ($2 \times$ MODEL_HW, $2 \times$ MODEL_HH). A bound check on a selected pixel's image coordinates (line 14) is followed by the extraction of the pixel's `r, g, b` intensity values (lines 17–19). These values are used in lines 22–24 to determine the respective bin indexes to which the current pixel belongs. The $\lambda(i, j)$ function used in Equation 7.2.2 is then realized (lines 27–29) so that it maps a pixel at location (i, j) to a point in the 3D space of bin indexes. Lines 32 and 33 represent the computation of the weighted color histogram using the Epachelnikov profile (Equation 7.2.1). This is followed by the normalization process (lines 37–41) so that the weighted color histogram of the respected image region represents a probability distribution.

LISTING 7.6.3.5 The method of the `CPTTrackingFilter` class associated with the construction of the target model weighted color histogram.

```
1: void CPTTrackingFilter::CaptureModelRGBHistogram(BYTE* pData,
2:                     CColorHistogramIndex* pIndex, Pixel COG) {
3:   BYTE r,g,b;
4:   int rBinIndex = gBinIndex = bBinIndex = 0;
5:   float binSum = 0;
6:
7:   pIndex->ClearRGBHistogram();
8:   pIndex->InitializeBinIndexes();
9:
10:  //Fill the RGB histogram
11:  for(int i=(COG.x-MODEL_HH),m=0; i<(COG.x+MODEL_HH); i++,m++)
12:   for (int j=(COG.y-MODEL_HW),l=0; j<(COG.y+MODEL_HW); j++,l++)
13:   {
14:      if((i>= 0) && (i<= m_height) && (j>=0) && (j<=m_width))
15:      {
16:        //Get the pixel color values
17:        r = *(pData + 2 + 3*(i*m_width + j));
18:        g = *(pData + 1 + 3*(i*m_width + j));
19:        b = *(pData + 0 + 3*(i*m_width + j));
20:
21:        //find the bin that the current pix fits into
22:        rBinIndex = (int)((r*(pIndex->m_NumberOfBins - 1))/255);
```

```
23:         gBinIndex = (int)((g*(pIndex->m_NumberOfBins - 1))/255);
24:         bBinIndex = (int)((b*(pIndex->m_NumberOfBins - 1))/255);
25:
26:         //Record the binIndex for this pixel
27:         pIndex->m_RedBinIndex[i][j] = rBinIndex;
28:         pIndex->m_GreenBinIndex[i][j] = gBinIndex;
29:         pIndex->m_BlueBinIndex[i][j] = bBinIndex;
30:
31:         //increment the pixel count for the bin
32:         pIndex->m_RGBHistogram[rBinIndex][gBinIndex][bBinIndex] +=
33:                     (float)m_ModelParameter.m_profileParams[m][l];
34:       }
35:    }
36:    //Normalize the RGB histogram
37:    for(i = 0; i < pIndex->m_NumberOfBins; i++)
38:     for(int j = 0; j < pIndex->m_NumberOfBins; j++)
39:      for(int k = 0; k < pIndex->m_NumberOfBins; k++)
40:        pIndex->m_RGBHistogram[i][j][k]=(float)(pIndex->
41:         m_RGBHistogram[i][j][k]/m_ModelParameter.m_profileParamsSum);
42: }
```

A similar function, `CaptureCandidateRGBHistogram`, is also implemented for computing the color distribution of the target candidate area. The reason for implementing a separate function stems from the size of the image rectangle used for estimating the position of the tracked object in subsequent frames not being the same as the size of the rectangular area that was used to compute the target model's weighted color distribution.

The `MeanShiftTracking` Method

This method represents the implementation of mean-shift–based object tracking as it was outlined in Section 7.3. First, a color histogram distribution of the target model object to be tracked is formed by drawing a rectangle containing the object. Then, the tracking algorithm updates the center of gravity (CoG) of a rectangle centered around the estimated pixel location according to mean-shift iterations. The complete software code broken down into two parts

appears in Listings 7.6.3.6 and 7.6.3.7. The `MeanshiftTracking` method is called only after an initial location of an object to be tracked has been identified. Then the mean-shift iterations take place thereafter and are described next. After some initializations are performed on `m_PCandidate` (lines 10–11 of Listing 7.6.3.6), the

LISTING 7.6.3.6 Part I: Implementation of the `MeanShiftTracking` method for object tracking.

```
1: void CPTTrackingFilter::MeanShiftTracking(BYTE* pData) {
2:
3:     Pixel tempCandidatePosition, tmpcand;
4:     float sumXWeights, sumYWeights, sumWeights;
5:     int i,j,k,p,q, uR, uG, uB, cnt = 0;
6:     float PrevBhattacharyya = Bhattacharyya = 0.0;
7:
8: start: ;
9:
10:    m_PCandidate->InitializeWeights();
11:    m_PCandidate->InitializeBinIndexes();
12:    m_PCandidate->ClearRGBHistogram();
13:
14:    // Target Candidate Color Histograms
15:    CaptureCandidateRGBHistogram(pData,m_PCandidate, m_CurrentBoxCOG);
16:
17:    // Find the Bhattacharyya Coefficient that shows how close
18:    // are the pdfs of the target model and the target candidate.
19:    for(i = 0; i < m_PCandidate->m_NumberOfBins; i++)
20:     for(j = 0; j < m_PCandidate->m_NumberOfBins; j++)
21:      for(k = 0; k < m_PCandidate->m_NumberOfBins; k++)
22:       PrevBhattacharyya+=(float) sqrt((float)(m_QTarget->
23:     m_RGBHistogram[i][j][k] * m_PCandidate->m_RGBHistogram[i][j][k]) );
24:
25:    // Compute weights for CoG updating of the Candidate model
26:    for ( i=(m_CurrentBoxCOG.x - CANDIDATE_HH),
27:                    p=0; i<(m_CurrentBoxCOG.x + CANDIDATE_HH); i++, p++)
28:     for ( j=(m_CurrentBoxCOG.y - CANDIDATE_HW),
29:                   q=0; j<(m_CurrentBoxCOG.y + CANDIDATE_HW); j++, q++){
30:       if((i>= 0) && (i< m_height) && (j>=0) && (j< m_width)) {
31:         if (m_CandidateParameter.m_nonZeroMask[p][q]) {
```

```
32:          uR = m_PCandidate->m_RedBinIndex[i][j];
33:          uG = m_PCandidate->m_GreenBinIndex[i][j];
34:          uB = m_PCandidate->m_BlueBinIndex[i][j];
35:
36:          if ( m_PCandidate->m_RGBHistogram[uR][uG][uB] > 0.0 )
37:           m_CandidateWeights[i][j] = (float) sqrt((m_QTarget->
38:            m_RGBHistogram[uR][uG][uB]) /
39:             (m_PCandidate->m_RGBHistogram[uR][uG][uB])); }
40:       }
41:    }
42:      sumXWeights = sumYWeights = sumWeights = 0.0;
43:  . . .
44: }
```

LISTING 7.6.3.7 Part II: Implementation of the `MeanShiftTracking` method for object tracking.

```
1: void CPTTrackingFilter::MeanShiftTracking(BYTE* pData) {
2: . . .
42:   sumXWeights = sumYWeights = sumWeights = 0.0;
43:   for (i=(m_CurrentBoxCOG.x - CANDIDATE_HH), p=0;
44:         i<(m_CurrentBoxCOG.x + CANDIDATE_HH); i++, p++)
45:     for ( j=(m_CurrentBoxCOG.y - CANDIDATE_HW), q=0;
46:         j<(m_CurrentBoxCOG.y + CANDIDATE_HW); j++, q++) {
47:
48:      if((i>= 0) && (i< m_height) && (j>=0) && (j< m_width)) {
49:        if (m_CandidateParameter.m_nonZeroMask[p][q]) {
50:            sumXWeights += (float)(i*m_CandidateWeights[i][j]);
51:            sumYWeights += (float)(j*m_CandidateWeights[i][j]);
52:            sumWeights  += (float)(m_CandidateWeights[i][j]); }
53:    }}
54:    tempCandidatePosition.x = (int)(sumXWeights / sumWeights);
55:    tempCandidatePosition.y = (int)(sumYWeights / sumWeights);
56: iteration: ;
57:    pColorHist->InitializeWeights();
58:    pColorHist->InitializeBinIndexes();
59:    pColorHist->ClearRGBHistogram();
60:
61:   CaptureCandidateRGBHistogram(pData,pColorHist,tempCandidatePosition);
62:    Bhattacharyya = 0.0;
```

```
63:    for(i = 0; i < m_PCandidate->m_NumberOfBins; i++)
64:     for(j = 0; j < m_PCandidate->m_NumberOfBins; j++)
65:       for(k = 0; k < m_PCandidate->m_NumberOfBins; k++)
66:        Bhattacharyya += (float) sqrt( (float) (m_QTarget->
67:         m_RGBHistogram[i][j][k]*pColorHist->m_RGBHistogram[i][j][k]));
68:
69:    if (Bhattacharyya < PrevBhattacharyya) {
70:      tmpcand.x = tempCandidatePosition.x;
71:      tmpcand.y = tempCandidatePosition.y;
72:      PrevBhattacharyya = Bhattacharyya;
73:      tempCandidatePosition.x=(int)(0.5*(m_CurrentBoxCOG.x+tmpcand.x));
74:      tempCandidatePosition.y=(int)(0.5*(m_CurrentBoxCOG.y+tmpcand.y));
75:      goto iteration; }
76:    m_CurrentBoxCOG.x = tempCandidatePosition.x;
77:    m_CurrentBoxCOG.y = tempCandidatePosition.y;
78:    if((sqrt(((m_CurrentBoxCOG.x - tempCandidatePosition.x)*
79:        (m_CurrentBoxCOG.x - tempCandidatePosition.x)) +
80:        ((m_CurrentBoxCOG.y - tempCandidatePosition.y)*
81:        (m_CurrentBoxCOG.y - tempCandidatePosition.y)))) > 2)
82:      goto start;
83: }
```

target candidate's color histogram is computed in line 15 of Listing 7.6.3.6, according to Equation 7.2.3. Lines 19–23 of Listing 7.6.3.6 describe how the Bhattacharyya coefficient is computed. As we have explained earlier, it represents a measure of closeness between the target model's and target candidate's color histogram distributions. Please refer to step 2b of the algorithmic steps described in Section 7.3. We proceed from lines 26–41 of Listing 7.6.3.6 with the computation of the weights w_j given by Equation 7.2.9. The numerical computation of the weights that are associated with the rectangular area around the current location of the object relies on the Boolean mask `m_nonZeroMask` that was initially computed for the specified Epachelnikov profile. Once the weights w_j are computed, the new estimate of the object's location y_{next} is calculated (lines 43–55 of Listing 7.6.3.7) according to Equation 7.3.1. The color histogram of the rectangular area centered around pixel location y_{next} is calculated in line 61 of Listing 7.6.3.7, and a new respective Bhattacharyya coefficient is computed in lines 23–28 of the same listing

(step 2e of Section 7.3). If the new Bhattacharyya coefficient is smaller than the previous one, a new pixel location is calculated according to the equation described in step 2f, and the process continues from line 56. However, if this is not the case, the image location is recorded and the function exits if a convergence criterion is reached. Otherwise, the algorithm repeats the steps from the label `start` (line 8 of Listing 7.6.3.6).

The `DrawCandidateTargetRegion` *Method*

This function enables drawing of a rectangle in a video image centered at a specified location by accessing and setting the *r*, *g*, *b* values of the pixels in the rectangle's boundary. The details of the implementation can be found in Listing 7.6.3.8. The argument list of the function contains the RGB pixel data of the current video image (`BYTE* pData`) and the center of the rectangle (`Pixel COG`). The function features two double `for` loops for coloring the top-bottom (lines 4–16, Listing 7.6.3.8) and left-right (lines 19–31, Listing 7.6.3.7) sides of the rectangle, respectively. A check that ensures that the boundary pixels are within the video image is performed in lines 10 and 25.

LISTING 7.6.3.8 Drawing a rectangle center around a pixel location by accessing and setting the r, g, b values of the boundary pixels.

```
1: void CPTTrackingFilter::DrawCandidateTargetRegion(
2:                                          BYTE* pData, PixelCOG){
3:     //Draw the top and bottom of the rectangle
4:     for (int i=(COG.x - m_CurrentBoxHalfHeight);
5:         i<(COG.x + 2*m_CurrentBoxHalfHeight);
6:             i += 2*m_CurrentBoxHalfHeight)
7:       for (int j=(COG.y - m_CurrentBoxHalfWidth);
8:         j<(COG.y + m_CurrentBoxHalfWidth); j++) {
9:
10:        if((i>= 0) && (i<= m_height) && (j>=0) && (j<=m_width))
11:        {
12:            *(pData + 0 + 3*(i*(m_width) + j)) = (BYTE)0;
13:            *(pData + 1 + 3*(i*(m_width) + j)) = (BYTE)255;
14:            *(pData + 2 + 3*(i*(m_width) + j)) = (BYTE)255;
15:        }
```

```
16:     }
17:
18:     //Draw the left and right of the rectangle
19:     for (i=(COG.x - m_CurrentBoxHalfHeight);
20:         i<(COG.x + m_CurrentBoxHalfHeight); i ++)
21:       for (int j=(COG.y - m_CurrentBoxHalfWidth);
22:         j<(COG.y + 2*m_CurrentBoxHalfWidth);
23:             j+= 2*m_CurrentBoxHalfWidth) {
24:
25:         if((i>= 0) && (i<= m_height) && (j>=0) && (j<=m_width))
26:         {
27:             *(pData + 0 + 3*(i*(m_width) + j)) = (BYTE)0;
28:             *(pData + 1 + 3*(i*(m_width) + j)) = (BYTE)255;
29:             *(pData + 2 + 3*(i*(m_width) + j)) = (BYTE)255;
30:         }
31:     }
32: }
```


The `Transform` Method

As we explained in Chapter 4, the `Transform` function provides the main functionality of the filter. The function is called continuously as new video frames are fetched from the capture device. The details of the object tracking's filter `Transform` function are provided in Listings 7.6.3.9 and 7.6.3.10. The `CPTTrackingFilter` belongs to

LISTING 7.6.3.9 Part I: Definition of the `Transform` function.

```
1: HRESULT CPTTrackingFilter::Transform(IMediaSample *pMediaSample)
2: {
3:     AM_MEDIA_TYPE* pType = &m_pInput->CurrentMediaType();
4:     VIDEOINFOHEADER *pvi = (VIDEOINFOHEADER *) pType->pbFormat;
5:
6:     BYTE *pData;    // pointer to the data from the input pin
7:     long lDataLen;  // holds length of any given sample
8:
9:     pMediaSample->GetPointer(&pData);
10:
11:    //get the size of the input data
12:    lDataLen = pMediaSample->GetSize();
```

```
13:
14:    //if this is the first time the transform function is called,
15:    //get the media size and allocate space for the members.
16:    if(m_bInitializeFlag == FALSE)
17:    {
18:        //get the image properties from the BITMAPINFOHEADER
19:        m_colors = 3;
20:        m_width  = pvi->bmiHeader.biWidth;
21:        m_height = pvi->bmiHeader.biHeight;
22:
23:        AllocateFilterMembers();
24:        m_bInitializeFlag = TRUE;
25:    }
26:
27:    // Perform the tracking
28:    if(m_bStartTrackingFlag==TRUE && m_bTrackingBoxFlag==TRUE){
29:      if(!m_bInitialTrackingFlag) {
30:        MeanShiftTracking(pData);
31:
32:        m_Temp.x =  120 - m_CurrentBoxCOG.x;
33:        m_Temp.y =  160 - m_CurrentBoxCOG.y;
34:
35:        if (( m_Temp.y > 0) && (m_Temp.y > 60) ) {
36:           m_PAN += (signed short int) (+20);            //Pan Left
37:           m_PanTiltDevice->set_desired(PAN, POSITION,
38:                                (PTU_PARM_PTR *) &m_PAN, ABSOLUTE);
39:           m_PanTiltDevice->await_completion(); }
40:        else if (( m_Temp.y < 0) && (m_Temp.y < -60) ) {
41:           m_PAN += (signed short int) (-20);           //Pan Right
42:           m_PanTiltDevice->set_desired(PAN, POSITION,
43:                                (PTU_PARM_PTR *) &m_PAN, ABSOLUTE);
44:           m_PanTiltDevice->await_completion(); }
45:   . . .
46: }
```

LISTING 7.6.3.10 Part II: Definition of the `Transform` function.

```
1: HRESULT CPTTrackingFilter::Transform(IMediaSample *pMediaSample)
2: {
3:     . . .
45:       if (( m_Temp.x > 0) && (m_Temp.x > 60) ) {
```

```
46:          m_TILT += (signed short int) (-20);        // Tilt down
47:          m_PanTiltDevice->set_desired(TILT, POSITION,
48:                                (PTU_PARM_PTR *) &m_TILT, ABSOLUTE);
49:          m_PanTiltDevice->await_completion(); }
50:       else if (( m_Temp.x < 0) && (m_Temp.x < -60) ) {
51:          m_TILT += (signed short int) (+20);        //Tilt up
52:          m_PanTiltDevice->set_desired(TILT, POSITION,
53:                                (PTU_PARM_PTR *) &m_TILT, ABSOLUTE);
54:          m_PanTiltDevice->await_completion();
55:       }
56:    }
57:    else if
58:    {
59:        //Get the color signature of the target model
60:        CaptureModelRGBHistogram(pData,m_QTarget, m_InitialBoxCOG);
61:        m_bInitialTrackingFlag = FALSE;
62:    }
63:
64:    //Draw candidate model region
65:    DrawCandidateTargetRegion(pData,m_CurrentBoxCOG);
66:  }//End tracking if statement
67:
68:  //Respond to changes made in the pan/tilt property page
69:  if(m_bPanChangeFlag = TRUE)
70:  {
71:    m_PanTiltDevice->set_desired(PAN, POSITION,
72:                          (PTU_PARM_PTR *) &m_PAN, ABSOLUTE);
73:    m_PanTiltDevice->await_completion();
74:    m_bPanChangeFlag = FALSE;
75:  }
76:
77:  if(m_bTiltChangeFlag = TRUE)
78:  {
79:    m_PanTiltDevice->set_desired(TILT, POSITION,
80:                          (PTU_PARM_PTR *) &m_TILT, ABSOLUTE);
81:    m_PanTiltDevice->await_completion();
82:    m_bTiltChangeFlag = FALSE;
83:  }
84:
85:  return NOERROR;
86: }
```

the class of transform filters and has one input and one output pin. Before performing any transformation on the input data, first we need to know the type of the incoming media data, and second we must get a handle on this information for subsequent data processing. The type of the media input is obtained in line 3 (Listing 7.6.3.9) as the `m_pInput` protected member variable points to the input pin's object class. A handle on the input data is passed via a `GetPointer` call to the `pMediaSample` input argument (line 9, Listing 7.6.3.9). In lines 16–25 the dimensions of the input frame are determined, and space is allocated dynamically for the filter's members. Setting the Boolean variable `m_bInitializeFlag` ensures that memory is allocated only once during the time that the first input frame is fetched from the media source.

The function proceeds by carrying out mean-shift object tracking. Object tracking begins only after the Boolean variables `m_bStartTrackingFlag` and `m_bTrackingBoxFlag` become `TRUE` (line 28, Listing 7.6.3.9). These values ensure first that the user has drawn a rectangle around the object, thus determining the target model's initial location. Second, they ensure that the user has communicated to the filter his/her intention to track an object (by pressing the Start button of the GUI). The first frame after the user interaction for initiating tracking is completed is used only for computing the color histogram of the target model (line 60, Listing 7.6.3.10). Therefore, calls to the `MeanShiftTracking` function occur thereafter. This process is regulated by the Boolean variable `m_bInitialTrackingFlag` (line 29 of Listing 7.6.3.9 and line 61 of Listing 7.6.3.10).

In lines 32–33 of Listing 7.6.3.9 we use the `Pixel` type variable `m_Temp` to create a virtual rectangular region of interest (ROI). Its size is user programmable (in this case it is 120×120), and it is placed in the center of the video image. If the CoG of the object being tracked falls outside the ROI, then a mechanism that automatically controls the pan-tilt position of the pan-tilt device is engaged to bring the object inside the ROI. The pan-tilt control procedure is shown in lines 35–44 of Listing 7.6.3.9 for the pan axis and in lines 45–55 of Listing 7.6.3.10 for the tilt axis. The controller that we have implemented is a simple open loop proportional controller, as lines 31, 36 (Listing 7.6.3.9) and 46, 51 (Listing 7.6.3.10) indicate.

Finally, the code found in lines 69–83 of Listing 7.6.3.10 handle the response to user pan-tilt commands entered via the GUI. The GUI, which is part of the application software discussed in the following chapter, allows the user to enter pan-tilt commands through a set of four control buttons independently from object tracking.

7.7 Summary

In this chapter we described the implementation of a DirectShow transform filter for object tracking based on the color content of the object. The methodology deployed is known in the literature as mean-shift tracking. The tracking filter is built upon the pan-tilt DirectShow source filter developed in Chapter 6 for controlling a Directed Perception pan-tilt unit. As opposed to the previously developed CD and pan-tilt filters that are designed to run via `GraphEdit`, this filter is designed to run through a separate application, which will be described in Chapter 8. In addition to object tracking, a capability that automatically controls the pan-tilt motions of the device to always keep the moving object of interest in the camera's field of view has also been integrated.

CHAPTER 8

An Object-Tracking Application

We will describe an application that uses the pan-tilt and object-tracking filters developed in Chapters 6 and 7 to form a composite pan-tilt object-tracking capability. Specifically, the user will identify, via a GUI, the object to be tracked, and the system will track the object continuously as it moves around the video image. Moreover, pan-tilt camera control will be automatically engaged when the object of interest appears in a rectangular zone close to the image boundaries. As a result, the system not only tracks the object, but it also allows for its tracking when the latter moves outside the camera's field of view. ■

8.1 Introduction

The object tracking application PTTrackingApp is the controlling program for the CPTTracking filter described in Chapter 7. The structure of PTTrackingApp should look very familiar to you. In Chapter 5, we created a similar application for the CD filter. In this chapter, we will walk you though the key functionality of the PTTrackingApp. We will not go into the step-by-step details of creating this application because these details were described in Chapter 5.

8.2 Running the PTTracking Application

In order to motivate you about this application, we will begin with a demonstration of the PTTracking application. To run the PTTracking application certain requirements must be guaranteed. First, a camera is definitely needed (e.g., USB, FireWire) and connected to the computer. Second, the CPTTracking filter developed in Chapter 7 must be compiled and registered on the DirectShow COM object list. A third optional, but recommended, component is the Directed Perception pan-tilt unit.

Once the required components associated with the PTTrackingApp application are in place, you may run the executable file PTTrackingApp.exe. When the program gets started, the user will see the screen shown in Figure 8.1. While the application is running, certain configurations should occur. First, you must choose the external video capture device. From the menu the user must select Camera → Choose Camera. This option will bring up a dialog such as the one shown in Figure 8.2. The user must select one of the devices in the list of connected video capture devices. The selection is ascertained by pressing the OK button. Now that a camera has been selected, we can start live video streaming by pressing the Start Camera button in the Main Controls dialog window. The user should now see a live video feed inside the PTTrackingApp window (see Figure 8.3).

As noted above, this application supports the Directed Perception pan-tilt unit. The tracking algorithm we developed in Chapter 7 deploys the particular pan-tilt unit to keep the tracking object within the camera's field of view. However, we can comment out the

FIGURE 8.1 Main windows for the PTTracking application.

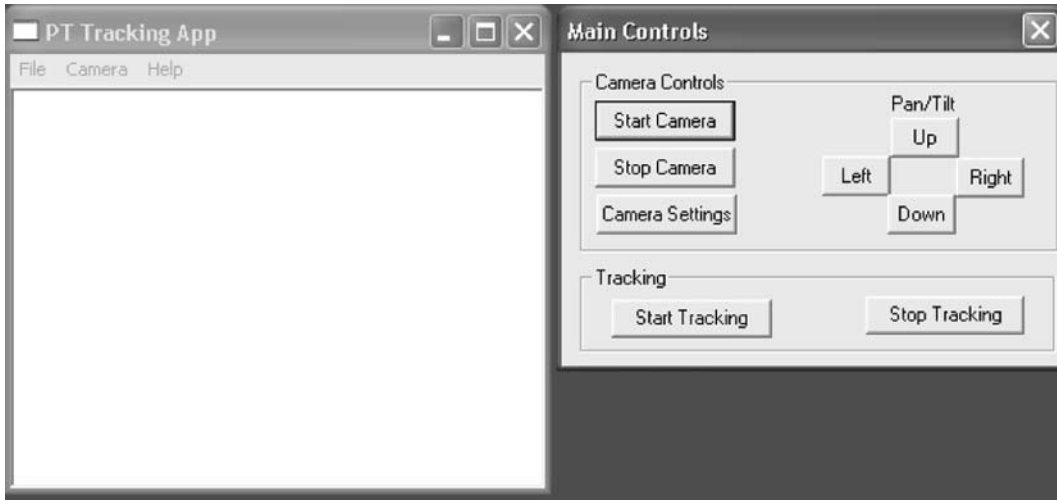


FIGURE 8.2 Video device selection dialog window.

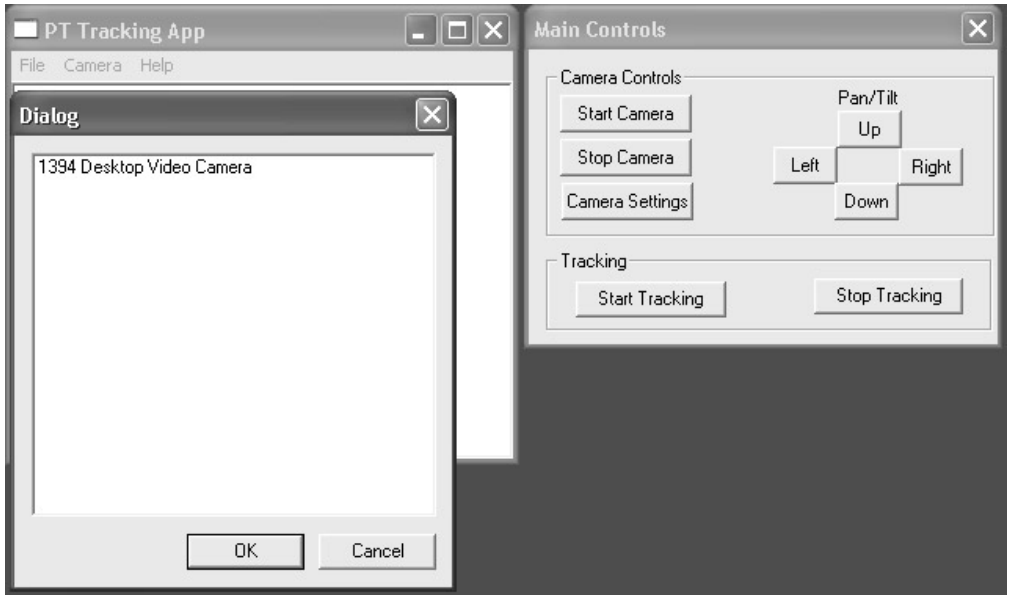


FIGURE 8.3
Live video display.

respective code of the `PTTracking` filter (lines 69–83, Listing 7.6.3.10) to disable this functionality in the absence of the pan-tilt unit. The `Main Controls` dialog also contains buttons for moving the pan-tilt unit manually (as seen in Figure 8.1).

This application also allows the user to change some video and camera control properties associated with the live video stream. You may click on the `Camera Settings` button inside the `Main Controls` dialog to see a typical camera property dialog (Figures 8.4 and 8.5). In doing so, the user can make changes to things such as, `zoom`, `focus`, `brightness`, and `sharpness`, to name just a few.

The main functionality of the `PTTrackingApp` application is color-based object tracking. To start the tracking algorithm we must first select an object within the camera's field of view. Next, the user must identify the object to be tracked. This is done by pressing the right mouse button and holding it down so that a rectangle is drawn around the desired object. Next, we need to press the `Start Tracking` button, and a yellow rectangle will appear around the selected object as shown in Figure 8.6. As the object moves, the yellow

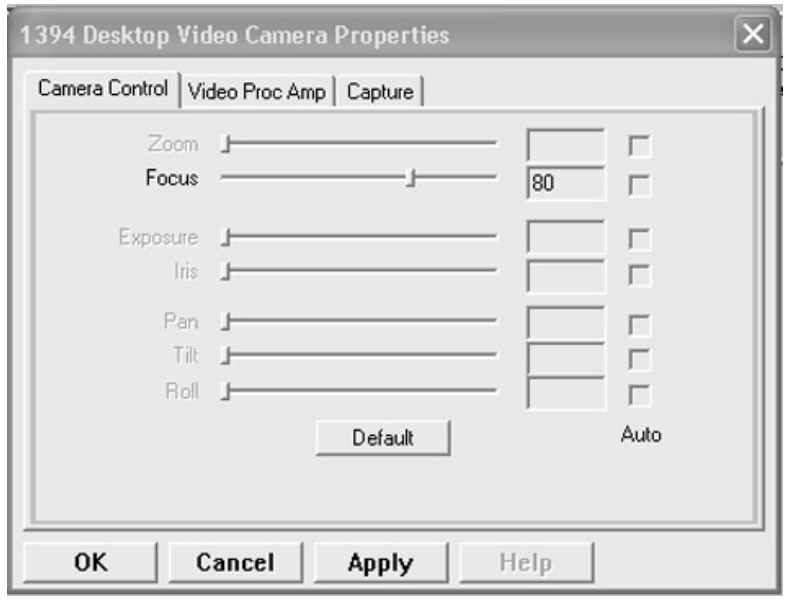


FIGURE 8.4
Camera Properties Page: Camera Control.

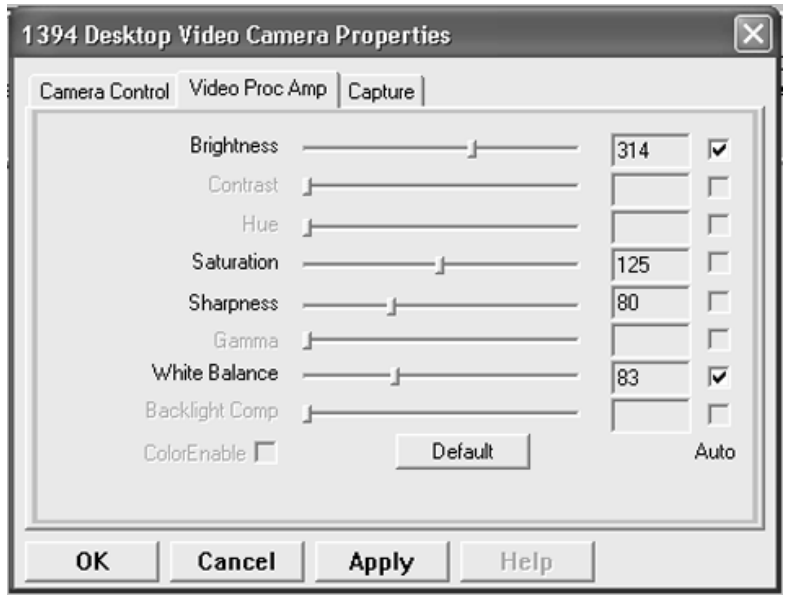
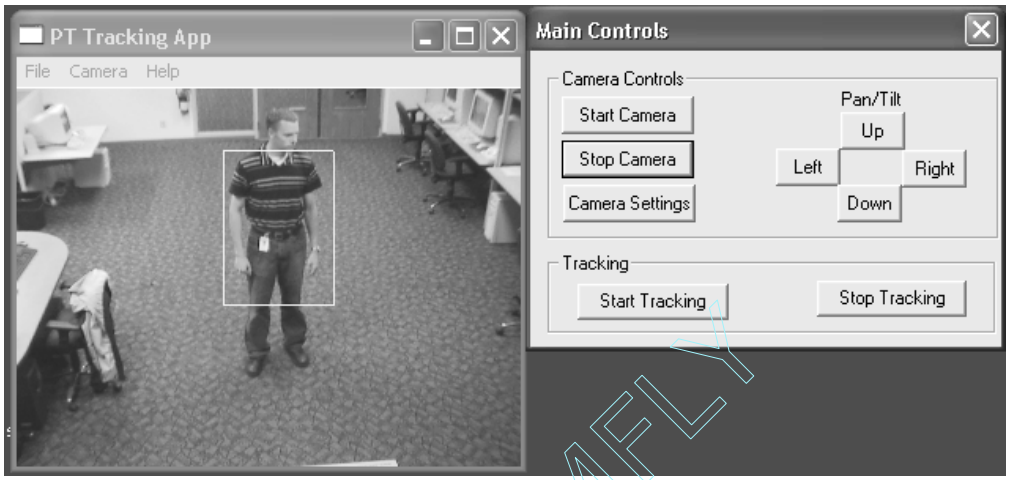


FIGURE 8.5
Camera Properties page: Video Properties.

FIGURE 8.6 Object tracking in action.



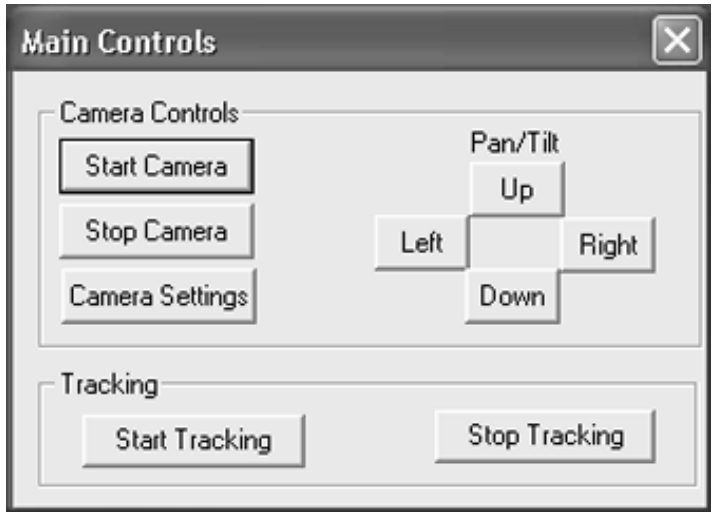
rectangle will move along with the object as long as the latter is within the camera's field of view. If you have the pan-tilt unit connected to the computer and the object moves toward the edges of the current video frame, the pan-tilt control mechanism will move automatically so that the object is kept inside the camera's field of view. To stop the tracking session we simply press the `Stop Tracking` button.

8.3 Main Control Dialog

The `PTTrackingApp` application contains two main windows: the `Main Controls` dialog and the `PTTrackingApp` window. The `Main Controls` dialog window is used to control various parts of the application such as starting and stopping video capture, controlling the pan-tilt device, and controlling the tracking algorithm. This dialog is implemented by a single class called `CMainControlDlg`. Figure 8.7 shows the main controls dialog window for this application.

The most important member of the `CMainControlDlg` class is its pointer to the `PTTrackingApp`'s application window, `m_pApp`. As it is shown in Listing 8.3.0.1, each of the buttons on the dialog

FIGURE 8.7
`Main Controls` dialog window.



LISTING 8.3.0.1 Main Control Dialog button callback functions.

```
1: BEGIN_MESSAGE_MAP(CMainControlDlg, CDialog)
2:  //{{AFX_MSG_MAP(CMainControlDlg)
3:  ON_BN_CLICKED(ID_START_CAMERA_BUTTON, OnStartCameraButton)
4:  ON_BN_CLICKED(ID_STOP_CAMERA_BUTTON, OnStopCameraButton)
5:  ON_BN_CLICKED(IDC_TILT_UP_BUTTON, OnTiltUpButton)
6:  ON_BN_CLICKED(IDC_TILT_DOWN_BUTTON, OnTiltDownButton)
7:  ON_BN_CLICKED(IDC_PAN_LEFT_BUTTON, OnPanLeftButton)
8:  ON_BN_CLICKED(IDC_PAN_RIGHT_BUTTON, OnPanRightButton)
9:  ON_BN_CLICKED(IDC_CAMERA_SETTINGS, OnCameraSettings)
10: ON_BN_CLICKED(IDC_START_TRACKING_BUTTON, OnStartTrackingButton)
11: ON_BN_CLICKED(IDC_STOP_TRACKING_BUTTON, OnStopTrackingButton)
12: //}}AFX_MSG_MAP END_MESSAGE_MAP()
13:
14: //CMainControlDlg message handlers
15: void CMainControlDlg::OnStartCameraButton()
16: { m_pApp->OnStartCamera(); }
17:
18: void CMainControlDlg::OnStopCameraButton()
19: { m_pApp->OnStopCamera();  }
20:
21: void CMainControlDlg::OnTiltUpButton()
```

```
22: { m_pApp->TiltUp(); }
23:
24: void CMainControlDlg::OnTiltDownButton()
25: { m_pApp->TiltDown();  }
26:
27: void CMainControlDlg::OnPanLeftButton()
28: { m_pApp->PanLeft(); }
29:
30: void CMainControlDlg::OnPanRightButton()
31: { m_pApp->PanRight(); }
32:
33: void CMainControlDlg::OnCameraSettings()
34: { m_pApp->OnCameraControls(); }
35:
36: void CMainControlDlg::OnStartTrackingButton()
37: { m_pApp->StartTracking(); }
38:
39: void CMainControlDlg::OnStopTrackingButton()
40: { m_pApp->StopTracking();}
```

window contains a single callback function. The callback functions are very simple at this point. All they really do is pass the responsibility onto the application by making a call to functions located within the `PTTrackingAppApp` class. Lines 1–13 of Listing 8.3.0.1 represent code generated automatically by Visual Studio. This code maps the actual buttons drawn on the dialog to the callback functions. Lines 15–40 show the callback functions for each of the buttons in the `Main Controls` dialog window. In the next section we will examine the functionality of the application in regard to these button commands.

8.4 `PTTracking` Application Window

The main application window, `PT Tracking App`, is created using the Document-View architecture. Three main classes are used. They are `CPTTrackingDoc`, `CPTrackingView`, and `CPTTrackingApp`. We will now discuss the roles and contents of each class.

8.4.1 *CPTTrackingDoc*

Typically, in a Document-View architecture the `Doc` class contains the data for the application. In this application we are not dealing with a great deal of data. The main role of our application is to provide communication between the user and the filter. All of the image and tracking data is actually contained within the `PTTracking` filter, and therefore the `CPTTrackingDoc` class has no additional information to describe.

8.4.2 *CPTTrackingView*

The `View` class contains all of the code related to displaying information to the user as well as handling input from the user. The `CPTTrackingView` class creates an instance of the `CMainControlDlg` for handling the user input. The view class also handles the mouse button and mouse movement inputs, which are used to create the input rectangle for the tracking algorithm.

The `OnDraw` function is probably the most important function with regard to drawing. In lines 8–20 of Listing 8.4.2.1 the `m_MainCtrlDlg` member variable of type `CMainControlDlg` is created and configured so that it displays information to the user. This is accomplished by calling the dialog's `Create` function with the `IDD` that has been assigned to the main controls dialog window inside

LISTING 8.4.2.1 Definition of the `OnDraw` function.

```
1: void CPTTrackingAppView::OnDraw(CDC* pDC)
2: {
3:   CPTTrackingAppDoc* pDoc = GetDocument();
4:   ASSERT_VALID(pDoc);
5:
6:   CPTTrackingAppApp* AppP = (CPTTrackingAppApp*) AfxGetApp();
7:
8:   if(m_bDlgCreatedFlg == FALSE)
9:   {
10:    //Create and place the main controls dialog in the window
11:    m_MainCtrlDlg.Create(IDD_MAIN_CONTROLS);
12:    m_MainCtrlDlg.m_pApp = AppP;
```

```
13:    CRect mainRect;
14:    this->GetClientRect(&mainRect);
15:    m_MainCtrlDlg.SetWindowPos(NULL,(mainRect.right + 15)
16:                                              ,0,305,220,NULL);
17:    m_MainCtrlDlg.ShowWindow(SW_SHOW);
18:
19:    m_bDlgCreatedFlg = TRUE;
20:  }
21:
22:  if(AppP-> m_bInitializedFlag == TRUE &&
23:                      AppP->m_bFirstStartFlag == FALSE &&
24:                                  m_bSetMessageFlg == TRUE)
25:  {
26:    //Set PT Tracking App video window to send a input message
27:    //to the main application window
28:    AppP->m_pPTTrackingVidWin->put_MessageDrain((OAHWND)this->m_hWnd);
29:
30:    m_bSetMessageFlg = FALSE;
31:  }
32: }
```

the resource editor (line 11, Listing 8.4.2.1). Next, the `CMainControlDlg`'s pointer (`m_pApp`) is directed to point to the `AppP` variable (line 12, Listing 8.4.2.1). This is very important because we use this pointer for handling all of the button callbacks inside the dialog. After the dialog is created, we must specify a place to display it on the screen. In line 15 of Listing 8.4.2.1 a call to the `SetWindowPos` function is made, which is an inherited member function from the `CWnd` class. The upper-left corner of the new window and its width and height are entered as input arguments to the function. Finally, the `Main Controls` dialog is configured to the displaying mode. This results in a call to the member function `ShowWindow` in line 17.

Note that the `OnDraw` function does not handle any of the video display. The live video from the camera is actually drawn inside a member of the `CPTTrackingApp` class, namely `m_pPTTrackingVidWin`. This member is a DirectShow interface of type `IVideoWindow`. The `m_pPTTrackingVidWin` is connected to the Video Renderer of the filter graph and then placed on top of the `CPTTrackingAppView`. In this way, we are actually creating an

illusion that makes the video appear to be drawn on the CPTTrackingAppView. Since this application uses mouse input on the video display (i.e., drawing a rectangle around the desired tracking object), we need to send the messages through the m_pPTTrackingVidWin to the View class. In line 28 of Listing 8.4.2.1, a function call to the put_MessageDrain is made. This is a member function of the IVideoWindow interface. In doing so, m_pPTTrackingVidWin sends its mouse message down to the CPTTrackingView level.

The Main Control dialog handles all its button functionality by itself. Therefore, the only thing that the View class needs to handle is the mouse click and mouse move commands that are performed on its window. As we explained earlier, there is a message-draining process that passes the mouse commands through the DirectShow video window down to the View. The CPTTrackingView class handles the mouse commands with the functions shown in Listings 8.4.2.2–8.4.2.4.

LISTING 8.4.2.2 Definition of the OnRButtonDown callback function.

```
1: void CPTTrackingAppView::OnRButtonDown(UINT nFlags, CPoint point)
2: {
3:  CPTTrackingAppApp* AppP = (CPTTrackingAppApp*) AfxGetApp();
4:
5:  AppP->m_pMediaControl->Pause();
6:  AppP->m_bFirstTime = TRUE;
7:  AppP->m_bDrawRect = TRUE;
8:  AppP->m_upperLeft.x = point.x;
9:  AppP->m_upperLeft.y = point.y;
10:
11: CView::OnRButtonDown(nFlags, point);
12: }
```

LISTING 8.4.2.3 Definition of the OnMouseMove callback function.

```
1: void CPTTrackingAppView::OnMouseMove(UINT nFlags, CPoint point){
2:
3:   CPTTrackingAppApp* AppP = (CPTTrackingAppApp*) AfxGetApp();
4:
```

```
5:   //Get the device drawing context
6:   CDC* pDC = GetDC();
7:   if(pDC->m_hDC == NULL)
8:      return;
9:
10:  if(AppP->m_bDrawRect == TRUE) {
11:    //if this is the first time the last rectangle is null
12:    if(AppP->m_bFirstTime == TRUE) {
13:      AppP->m_LastRect = NULL;
14:      AppP->m_CurrRect.TopLeft().x = AppP->m_upperLeft.x;
15:      AppP->m_CurrRect.TopLeft().y = AppP->m_upperLeft.y;
16:      AppP->m_CurrRect.BottomRight().x = point.x;
17:      AppP->m_CurrRect.BottomRight().y = point.y;
18:      AppP->m_bFirstTime = FALSE; }
19:    //otherwise the last rect gets the old current rect and the
20:    //current rect is drawn by the current position.
21:    else {
22:     //last rectangle gets the old current rectangle
23:     AppP->m_LastRect.TopLeft().x = AppP->m_CurrRect.TopLeft().x;
24:     AppP->m_LastRect.TopLeft().y = AppP->m_CurrRect.TopLeft().y;
25:     AppP->m_LastRect.BottomRight().x=AppP->m_CurrRect.BottomRight().x;
26:     AppP->m_LastRect.BottomRight().y=AppP->m_CurrRect.BottomRight().y;
27:
28:     //current rectangle
29:     AppP->m_CurrRect.TopLeft().x = AppP->m_upperLeft.x;
30:     AppP->m_CurrRect.TopLeft().y = AppP->m_upperLeft.y;
31:     AppP->m_CurrRect.BottomRight().x = point.x;
32:     AppP->m_CurrRect.BottomRight().y = point.y;
33:
34:     //draw the drag rect
35:     pDC->DrawDragRect(AppP->m_CurrRect, AppP->m_CurrRect.Size(),
36:                         AppP->m_LastRect, AppP->m_LastRect.Size());
37:    }
38:
39:    CView::OnMouseMove(nFlags,point);
40:  }
41:
42:  CView::OnMouseMove(nFlags, point);
43: }
```

LISTING 8.4.2.4 Definition of the `OnRButtonUp` callback function.

```
1: void CPTTrackingAppView::OnRButtonUp(UINT nFlags, CPoint point)
2: {
3:     CPTTrackingAppApp* AppP = (CPTTrackingAppApp*) AfxGetApp();
4:
5:     AppP->m_bDrawRect = FALSE;
6:     AppP->m_lowerRight.x = point.x;
7:     AppP->m_lowerRight.y = point.y;
8:     AppP->m_pMediaControl->Run();
9:
10:    int ulx = AppP->m_upperLeft.x;
11:    int uly = AppP->m_upperLeft.y;
12:    int lrx = AppP->m_lowerRight.x;
13:    int lry = AppP->m_lowerRight.y;
14:
15:    //Set the selected box
16:    AppP->m_ScreenCenter.x = (int)(AppP->m_upperLeft.x +
17:               ((AppP->m_lowerRight.x - AppP->m_upperLeft.x)/2));
18:    AppP->m_ScreenCenter.y = (int)(AppP->m_upperLeft.y +
19:               ((AppP->m_lowerRight.y - AppP->m_upperLeft.y)/2));
20:
21:    //Since the image in directX start with the origin in the lower
22:    //left corner and the x - axis is vertical and the y - axis is
23:    //horizontal.  We need to change the x,y values before entering
24:    //them into the filter.
25:    AppP->m_DirectXCenter.x = (AppP->m_videoHeight - 1) -
26:                               AppP->m_ScreenCenter.y;
27:    AppP->m_DirectXCenter.y = AppP->m_ScreenCenter.x;
28:
29:    AppP->UpdateTrackingBox();
30:
31:    CView::OnRButtonUp(nFlags, point);
32: }
```

The `OnRButtonDown` is a system callback function for responding to the click of the right mouse button. We use this command to signal the starting point of the rectangle the user will draw around a tracking object. Before we start drawing this rectangle, we pause the live video as shown in line 5 of Listing 8.4.2.2. The video has been paused to

allow drawing a region around a still object rather than a moving one. A couple of flags are being set denoting the start of our drawing routine (lines 6–7, Listing 8.4.2.2). Finally, we record the coordinates of the location at which this mouse was at the time the right button was clicked. These image coordinates represent the upper-left corner of the rectangle (lines 8–9, Listing 8.4.2.2).

The `OnMouseMove` is the next function that enables the user to dynamically draw the object-tracking rectangle. This function tracks the mouse as it moves across the window with the right mouse button still being held down. The mouse movement tracking starts by making sure the right mouse button is still down. This is evident by the `m_bDrawRect` flag that was set in the `OnRButtonDown` function (line 7, Listing 8.4.2.2). Because the `OnMouseMove` function is called for each new position of the mouse pointer, we need to continually update the current rectangle size. If this is the first time this function has been called since the right button was pressed, the `m_CurrRect` member must be initialized (lines 14–17, Listing 8.4.2.3). If this is the second or consequent time that this function has been called, the previous rectangle is recorded (lines 23–26, 8.4.2.3). We next update the current rectangle, `m_CurrRect`, with the current position of the mouse pointer (lines 29–32, Listing 8.4.2.3). At the end the updated rectangle is drawn to the window (lines 35–36, Listing 8.4.2.3).

Finally, the `OnRButtonUp` is called when the user has released the right mouse button to finish the rectangle drawing. In line 5 of Listing 8.4.2.4, we set the `m_bDrawRect` flag to `FALSE` because we have completed the drawing of the rectangle dynamically. The current position of the mouse at the lower-right corner of the rectangle is recorded as shown in lines 6–7 of Listing 8.4.2.4. This is followed by a process whereby we can restart the video as seen in line 8 of Listing 8.4.2.4 by a call to `Run`. Lastly, the object for tracking that was selected by the user following the above process is sent to the filter.

An explanation regarding the differences of image coordinate in DirectX and MFC coordinate frames is in order and absolutely necessary for writing other image-processing application programs. This clarification is required since we need to translate the coordinates of our rectangle before we can send it to the filter. Figures 8.8 and 8.9 give you a visual description of the difference in the respective image coordinate systems.

FIGURE 8.8
Definition of the MFC image coordinate system.

FIGURE 8.9
Definition of the DirectX image coordinate system.

8.4.3 *The `CPTTrackingApp` Application*

In other chapters we showed how to run a filter by creating a filter graph using the `GraphEdit` DirectShow utility. In this section we will show how to create a filter for running the pan-tilt tracking filter using DirectShow functions.

Building the Filter Graph

The `CPTTrackingAppApp` class ascribes the core functionality to the filter, namely the pan-tilt object tracking. This class is the messenger between the user interface and the filter.

When the application is loaded, the first function to be called is `InitInstance`. This function handles all of the initialization. Listing 8.4.3.1 shows the major parts of the `InitInstance` function.

LISTING 8.4.3.1 Definition of the `InitInstance` function.

```
1: BOOL CPTTrackingAppApp::InitInstance()
2: {
3:     . . .
4:
5:     //Set the size and position of the window.
6:     //(Only need to show the menu bar)
7:     m_pMainWnd->SetWindowPos(NULL,0,0,328,294,NULL);
8:
9:    //Set the title for the window.
10:    m_pMainWnd->SetWindowText("PT Tracking App");
11:    m_pMainWnd->ShowWindow(SW_SHOW);
12:    m_pMainWnd->UpdateWindow();
13:
14:    ////////////////////////////////////
15:    //Initialization
16:    //Initialize the interfaces to null
17:    m_pGraph = NULL;
18:    m_pMediaControl = NULL;
19:    m_pEvent = NULL;
20:    m_pPTTrackingVidWin = NULL;
21:    m_pDevEnum = NULL;
22:    m_pClassEnum = NULL;
23:    m_pMoniker = NULL;
24:    m_pSrc = NULL;
25:    m_pPTTrackingFilter = NULL;
26:    m_pColorSpaceConverter = NULL;
27:    CoInitialize(NULL);
28:
29:   . . .
30:  }
```


Lines 5–7 detail the steps for setting the position of the main application window and the assignment of the name appearing in the title bar. A series of initialization statements follow in lines 17–27 (Listing 8.4.3.1) of all the filter graph members that will be used to create the `PTTracking` Application.

Once the initialization is done, we can start constructing the filter graph. The `OnStartCamera` function contains all of the code

that allows us to get the filter up and running. This function is called when the `Start Camera` button is pressed on the main dialog or chosen from the menu `Camera` → `Start Camera`. Listing 8.4.3.2 shows the details of `OnStartCamera`. We have split this function into two main sections. The `if` statement determines whether we are in the initialization state, `m_bFirstStartFlag == TRUE`, or in the restart state, `m_bFirstStartFlag == FALSE`. While in the initialization state, we start off by creating an instance of the filter graph builder, `m_pGraph` (lines 9–10 of Listing 8.4.3.2). Next, we query our graph builder for a media control and media event control as seen in lines 12–18. Then, a check is performed by a call to the `CheckDevice` function. This check ensures that a video capture device is connected

LISTING 8.4.3.2 Definition of the `OnStartCamera` function.

```
1: void CPTTrackingAppApp::OnStartCamera()
2: {
3:   HRESULT hr;
4:
5:   //If we have not started before.
6:   if(m_bFirstStartFlag == TRUE)
7:   {
8:     //create the interface for the graph builder
9:     CoCreateInstance(CLSID_FilterGraph, NULL,
10:    CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void**)&m_pGraph);
11:
12:    //Get a media control for the filter graph
13:    m_pGraph->QueryInterface(IID_IMediaControl,
14:                             (void**)&m_pMediaControl);
15:
16:    //Get an event handler for the filter graph
17:    m_pGraph->QueryInterface(IID_IMediaEventEx,
18:                             (void**)&m_pEvent);
19:    //Make sure we have a device
20:    CheckDevice();
21:
22:    //Instantiate and add the filters to the graph builder
23:    InstantiateFilters();
24:
```

```
25:    //Connect the video source to the graph
26:    ConnectVideoSource();
27:
28:    //Connect the PT Tracking filter
29:    ConnectPTTrackingFilter();
30:
31:    //Connect the output video renderer
32:    ConnectVideoRenderer();
33:
34:    //run the output of the PT Tracking filter
35:    hr = m_pPTTrackingRender->Run(0);
36:
37:    //switch the graph into running mode
38:    m_pMediaControl->Run();
39:
40:    m_bFirstStartFlag = FALSE;
41:  }
42:  //If we have already started once
43:  else {
44:    //switch the graph into running mode
45:    m_pMediaControl->Run();
46:   //start the PT Tracking renderer
47:   hr = m_pPTTrackingRender->Run(0);
48:  }
49: }
```

to the computer and has been chosen. This function works the same way as the `DefaultDevice` function from the CD application of Chapter 5. We simply check to see if a device has been selected yet. If so, we exit the function. If not, we take the first item in the list of connected devices as the default choice.

Now that a connection to an external device is ascertained, we proceed with instantiating and adding filters to the graph builder (line 23, Listing 8.4.3.2). Listing 8.4.3.3 shows the details of the `InstantiateFilters` function. Creation of filter instances occurs in lines 5–18 (Listing 8.4.3.3). First in the list is the `ColorSpace-Converter` filter (we followed the same process in Chapter 5). This is followed by the `PTTracking` filter and a video renderer.

LISTING 8.4.3.3 Definition of the `InstantiateFilters` function.

```
1: void CPTTrackingAppApp::InstantiateFilters()
2: {
3:     HRESULT hr;
4:
5:     //Create the base filter for the color space converter filter
6:    hr = CoCreateInstance(CLSID_Colour, NULL, CLSCTX_INPROC_SERVER,
7:        IID_IBaseFilter, (void **)&m_pColorSpaceConverter);
8:
9:    //create the base filter for the PT Tracking renderer
10:    hr = CoCreateInstance(CLSID_VideoRenderer, NULL,
11:                          CLSCTX_INPROC_SERVER, IID_IBaseFilter,
12:                                (void **)&m_pPTTrackingRender);
13:
14:    //Create the base filter for the PT Tracking filter
15:    hr = CoCreateInstance(CLSID_PTTrackingFilter, NULL,
16:                          CLSCTX_INPROC_SERVER, IID_IBaseFilter,
17:                                 (void **)&m_pPTTrackingFilter);
18:
19:    //add the capture filter
20:    m_pGraph->AddFilter(m_pSrc,L"Video Capture");
21:
22:    //add the color space converter filter
23:    hr = m_pGraph->AddFilter(m_pColorSpaceConverter,
24:                                L"Color Space Converter");
25:    //add the PT Tracking filter
26:    hr = m_pGraph->AddFilter(m_pPTTrackingFilter,
27:                                L" PT Tracking Filter");
28:    //add the filter renderer filter to the graph
29:    hr = m_pGraph->AddFilter(m_pPTTrackingRender,
30:                                L"PT Tracking Renderer Filter");
31: }
```

While the `CoCreateInstance` function is used for filter creation, adding filters to the filter graph is obtained by calls to the `AddFilter` function. This is shown in lines 20–30 (Listing 8.4.3.3). The reader should note that in line 20 we added a filter named `m_pSrc`. We never called `CoCreateInstance` with this filter. The `m_pSrc` filter represents the input video source and was actually created when the

`CheckDevice` function was called. These steps precede the formation of the final connection topology of the filters in the graph. This is discussed next. Connecting the input pin of a particular filter with the output pin of another is accomplished through custom function calls. The `ConnectVideoSource` function is introduced to describe the connection of the output pin of the video source to the input pin of the `ColorSpaceConverter`. The details of the function are shown in Listing 8.4.3.4. The actual event of connecting the respective filter instances occurs in line 26 of Listing 8.4.3.2 (inside the `OnStartCamera` function).

LISTING 8.4.3.4 Definition of the `ConnectVideoSource` function.

```
1: void CPTTrackingAppApp::ConnectVideoSource()
2: {
3:     HRESULT hr;
4:
5:     //Connect the capture source and intermediate
6:     //filters needed (avi decompressor)
7:
8:     //Get the "output" capture pin from the device
9:     IPin *pCapOut = GetOutPin(m_pSrc, 0);
10:
11:    //Query the preview output pin to get the stream config filter
12:    pCapOut->QueryInterface
13:             (IID_IAMStreamConfig,(void**)&m_pStreamConfig);
14:
15:    //Create a media type to hold the pin media type
16:    AM_MEDIA_TYPE *MediaType = new AM_MEDIA_TYPE();
17:
18:
19:    //Get the current media type
20:    m_pStreamConfig->GetFormat(&MediaType);
21:
22:    //Get the video info for this pin
23:    VIDEOINFOHEADER *pvi = (VIDEOINFOHEADER *)MediaType->pbFormat;
24:
25:    //Get the width and height of the video
26:    m_videoWidth = pvi->bmiHeader.biWidth;
27:    m_videoHeight = pvi->bmiHeader.biHeight;
28:
```

```
29:     //if the dimensions are greater than 320x240.
30:     //Cap them at 320x240
31:     if(m_videoWidth > 320)
32:     {
33:         //change our dimensions to 320 x 240
34:         m_videoWidth = 320;
35:         m_videoHeight = 240;
36:     }
37:
38:     //Get the input pin for the color space converter filter
39:     IPin *pColorIn = GetInPin(m_pColorSpaceConverter,0);
40:
41:     hr = m_pGraph->Connect(pCapOut,pColorIn);
42:
43: }
```

In line 9 of Listing 8.4.3.4, we get the output pin of the video source (`m_pSrc`). Here we use the same pin access functions as we did in Chapter 5. Once we have a pointer to the pin, we query it for the stream configuration interface (line 12, Listing 8.4.3.4). We use the `m_pStreamConfig` to make sure that the video dimensions are no greater than 320×240. We would like to keep the video dimensions at this size since it is impossible to determine the display settings of every monitor as well as the maximum resolution of every possible camera (lines 20–35, Listing 8.4.3.4). Once we have changed the video dimensions we connect the capture output pin to the next filter in the graph, i.e., the color space converter (line 39–41, Listing 8.4.3.4). The color space converter filter object, `m_pColorSpaceConverter`, converts video data from one `RGB` color type to another `RGB` type, such as between 24-bit and 8-bit `RGB` color. The main use of the color space converter occurs when the stream source consists of uncompressed `RGB` frames, like the format provided by the Xcam2 and the iBot cameras.

As line 29 of Listing 8.4.3.2 indicates, the next filter to connect is our pan-tilt tracking filter. This is accomplished via a call to the `ConnectPTTrackingFilter` function described in Listing 8.4.3.5. The input pin pan-tilt tracking filter object is connected to the output of the color space converter object (line 15, Listing 8.4.3.5).

LISTING 8.4.3.5 Definition of the `ConnectPTTrackingFilter` function.

```
1: void CPTTrackingAppApp::ConnectPTTrackingFilter()
2: {
3:     HRESULT hr;
4:
5:     //Connect the PT Tracking filter to the graph
6:
7:     //Get the output pin for the color space converter
8:     IPin *pColorOut = GetOutPin(m_pColorSpaceConverter,0);
9:
10:    //Get the Input pin for the PT Tracking filter
11:    IPin *pPTTrackingIn = GetInPin(m_pPTTrackingFilter,0);
12:
13:    //connect the color space converter output pin to the
14:    //PT Tracking filter
15:    hr = m_pGraph->Connect(pColorOut,pPTTrackingIn);
16:
17: }
```

The process of creating the application's filter graph continues with the attachment of a video renderer. The video renderer filter will facilitate the display of video that depicts object tracking. The `ConnectVideoRenderer` function (shown in Listing 8.4.3.6) connects the output pin of the pan-tilt tracking filter object to the input pin of the video renderer object, i.e., `m_pPTTrackingRender` (line 12 of Listing 8.4.3.6). In order to actually see the video inside of the renderer filter, we need to attach a window for the video to be displayed in. In line 16 of Listing 8.4.3.6, we query the renderer for a pointer to its video window interface. The `CPTTrackingAppApp` has a DirectShow member of type `IVideoWindow` that we assign to this interface pointer, `m_pPTTrackingVidWin`. Once the video window interface is available, its ownership is passed on to an `MFC` window so that its contents will be nicely contained within the application window. In line 20 of Listing 8.4.3.6, we assign the `m_PTTrackingVidWin` to the main application window for the pan-tilt tracking application. Finally, we adjust the application window to fit the size of the video stream (lines 32–33, Listing 8.4.3.6).

LISTING 8.4.3.6 Definition of the `ConnectVideoRenderer` function.

```
1: void CPTTrackingAppApp::ConnectVideoRenderer()
2: {
3:     HRESULT hr;
4:
5:     //Get the output pin from the PT Tracking filter
6:     IPin *pPTTrackingOut = GetOutPin(m_pPTTrackingFilter,0);
7:
8:     //Get the input pin from the PT Tracking renderer filter
9:     IPin *pPTTrackingRender = GetInPin(m_pPTTrackingRender,0);
10:
11:    //Connect the PT Tracking filter to the video renderer
12:    hr = m_pGraph->Connect(pPTTrackingOut, pPTTrackingRender);
13:
14:    //Query the PT Tracking video window for the
15:    //PT Tracking video renderer
16:    m_pPTTrackingRender->QueryInterface
17:            (IID_IVideoWindow, (void**) &m_pPTTrackingVidWin);
18:
19:    //Attach the PT Tracking playback window to display window
20:    m_pPTTrackingVidWin->put_Owner((OAHWND)AfxGetMainWnd()->m_hWnd);
21:
22:    //Tracking the style of the window to a child window
23:    m_pPTTrackingVidWin->put_WindowStyle(WS_CHILD | WS_CLIPSIBLINGS);
24:    RECT grc2;
25:
26:    //set the position of the window
27:    ::GetClientRect(GetMainWnd()->m_hWnd, &grc2);
28:
29:    //Hold video size to the input media size
30:    //so that we can easily keep track
31:    //of the video frame buffer size
32:    m_pPTTrackingVidWin->SetWindowPosition
33:                                (0,0,m_videoWidth,m_videoHeight);
34:
35:}
```

This concludes the creation of the filter graph. We will now show how the filter graph will be activated. First, in line 35 of Listing 8.4.3.2, we set the video renderer to the run mode. Then in line 38 of the same listing, we set the graph into running mode. This will start the streaming video process. We mentioned earlier that the `OnVideoStart` has two states: those of *initialization* and *restart*. The remaining lines of Listing 8.4.3.2 (lines 42–48) handle the restart state by setting the states of the renderer and the graph to the run mode.

Controlling the Pan-Tilt Device

As we mentioned earlier in our description of the main control dialog class `CMainControlDlg`, the pan-tilt device can be controlled in two different ways. Either through the `PTTracking` algorithm or through the control buttons on the main control dialog. The main control dialog class discussed earlier made reference to four functions, namely, the `TiltUp`, `TiltDown`, `PanLeft`, and `PanRight`. Each of these has the same basic functionality, with the exception of two parameters: movement direction (*positive* or *negative*) and movement type (*pan* or *tilt*). Listing 8.4.3.7 shows the implementation details of the `TiltUp`

LISTING 8.4.3.7 Definition of the `TiltUp` function.

```
1: void CPTTrackingAppApp::TiltUp()
2: {
3:     //Get the PT Tracking Filter Interface
4:     IPTTrackingInterface *pInterface = NULL;
5:     HRESULT hr = m_pPTTrackingFilter->QueryInterface
6:         (IID_IPTTrackingFilterInterface, (void **)&pInterface);
7:     if (SUCCEEDED(hr))
8:     {
9:         //Set the new tilt location
10:        m_currTiltLoc += (signed short int)(100);
11:
12:        if(m_currTiltLoc > TILTMAX)
13:            m_currTiltLoc = TILTMAX;
14:
```

```
15:         //Set the new tilt location in the filter
16:         pInterface->I_PutTiltLocation(m_currTiltLoc);
17:
18:     }
19:
20:     if(pInterface != NULL)
21:     {
22:         pInterface->Release();
23:         pInterface = NULL;
24:     }
25: }
```

function. If you examine the pan-tilt tracking application closely, you will notice two things that are different from the CD application of Chapter 5. First, a copy of the `PTTracking` interface declaration is included inside the project (*iPTTrackingFilter.h*). Second, we define the `GUID` for the interface inside the *PTTrackingFilteruids.h* rather than in its header *iPTTrackingFilter.h*. The main reason for these differences is that in the `PTTrackingAppApp` application we want to have access to the filter's interface functions for performing operations such as controlling the pan-tilt device. In lines 4–6 of Listing 8.4.3.7, we create an instance of the pan-tilt tracking interface. If we are successful in creating the interface pointer (line 7), we execute the tilt-up command. Line 10 of Listing 8.4.3.7 updates a member of the `App`, which keeps the current tilt location. In lines 12–13 we make sure that the location is within the bounds of the tilt device. Finally, in line 16 we use our interface pointer to call the `I_PutTiltLocation` interface function. The only argument is the new location of the tilt device. Each of the remaining functions works the same way, with the exception being the location variable that is used and the interface function that is called. In lines 20–24 of Listing 8.4.3.7, we release the interface pointer. This is important, because a mishandling of the pointer will result in memory leaks.

Running the Pan-Tilt Tracking Algorithm

In an earlier section of this chapter we described how to run the tracking algorithm. Now we will show the reader more details on what

lies behind the buttons associated with running the application. The pan-tilt tracking algorithm requires the drawing of a rectangular region around the object to be tracked. This is the initial step. The user draws the rectangle around the desired tracking object using the mouse. Once the rectangle has been drawn (i.e., the right mouse button has been released), the `CPTTTrackingView` object calls `OnRButtonUp`. The center point of the rectangle is then converted automatically into DirectShow coordinates, then the `UpdateTrackingBox` function is called as seen in Listing 8.4.3.8. In line 4, we can acknowledge the use of the pan-tilt tracking interface pointer. This pointer is involved in calling the interface function, `I_PutTrackingCenter`. The input parameters are the DirectShow converted coordinates of the user-drawn rectangle. Inside the `I_PutTrackingCenter` function of the interface these input parameters are assigned to members of the filter. Finally, in lines 13–16, we release the resources of the interface pointer as we did earlier.

LISTING 8.4.3.8 Definition of the `UpdateTrackingBox` function.

```
1: void CPTTrackingAppApp::UpdateTrackingBox()
2: {
3:     //Put the tracking box into the filter
4:     IPTTrackingInterface *pInterface = NULL;
5:     HRESULT hr = m_pPTTrackingFilter->QueryInterface
6:         (IID_IPTTrackingFilterInterface, (void **)&pInterface);
7:     if(SUCCEEDED(hr))
8:     {
9:         pInterface->I_PutTrackingCenter
10:            (m_DirectXCenter.x,m_DirectXCenter.y);
11:    }
12:
13:    if(pInterface != NULL)
14:    {
15:        pInterface->Release();
16:        pInterface = NULL;
17:    }
18: }
```

Listings 8.4.3.9 and 8.4.3.10 handle the start and stop commands for the tracking algorithm. These functions are called by the button callback functions of the main control dialog. Again, the reader should notice that we are using an interface pointer to access functions of the filter interface. Inside these functions we are simply setting a flag, which is used in the filter's `Transform` function to denote whether or not to perform mean-shift tracking.

LISTING 8.4.3.9 Definition of the `StartTracking` function.

```
1: void CPTTrackingAppApp::StartTracking()
2: {
3:     //Get the PT Tracking Filter Interface
4:     IPTTrackingInterface *pInterface = NULL;
5:     HRESULT hr = m_pPTTrackingFilter->QueryInterface
6:     (IID_IPTTrackingFilterInterface, (void **)&pInterface);
7:     if (SUCCEEDED(hr))
8:     {
9:         pInterface->I_PutStartTracking();
10:    }
11:
12:    m_bTrackingRunning = TRUE;
13:
14:    if(pInterface != NULL)
15:    {
16:        pInterface->Release();
17:        pInterface = NULL;
18:    }
19: }
```

LISTING 8.4.3.10 Definition of the `StopTracking` function.

```
1: void CPTTrackingAppApp::StopTracking()
2: {
3:     //Get the PT Tracking Filter Interface
4:     IPTTrackingInterface *pInterface = NULL;
5:     HRESULT hr = m_pPTTrackingFilter->QueryInterface
```

```
6:      (IID_IPTTrackingFilterInterface, (void **)&pInterface);
7:      if (SUCCEEDED(hr))
8:      {
9:          pInterface->I_PutStopTracking();
10:     }
11:
12:     m_bTrackingRunning = FALSE;
13:
14:     if(pInterface != NULL)
15:     {
16:         pInterface->Release();
17:         pInterface = NULL;
18:     }
19: }
```

8.5 Summary

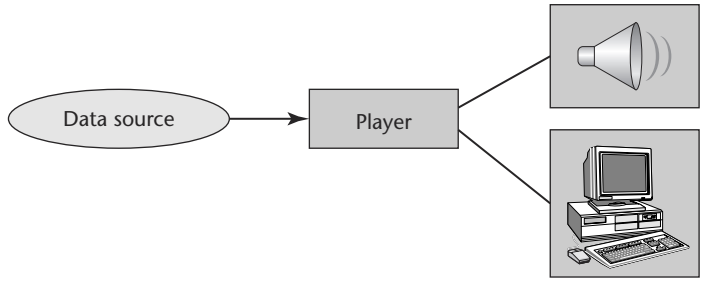
In this chapter, we showed how to create a complex application using a custom pan-tilt tracking filter. We attempted to promote modular programming of reusable software components (DirectX filters) through a computer programming exercise. We described how DirectX provides a framework in which integration of DirectX filters is rapid and done in a straightforward way. We hope that you can experience for yourself that the development of another application is both enjoyable and practical.

CHAPTER 9

Computer-Camera Programming in Java

Java has become a popular programming tool for the Internet. The introduction of Sun Microsystems's *Java Media Framework* (JMF) API, has made Java even more exciting with its support of streaming media, particularly video. You may download this API free at www.java.sun.com. This chapter will lead you through three fundamental concepts of JMF streaming applications: (1) acquiring and displaying live video, (2) performing image processing on Java video, and finally (3) transmitting video across a network. ■

FIGURE 9.1
Player architecture.



9.1 Understanding JMF

Programming using JMF requires that the software developer knows a number of key concepts about JMF architecture. The input to a JMF application is called a *DataSource*. DataSources are peripherals like videotapes, live video feeds, CDs, or live audio. A *player* provides the medium for processing and control of the data source.

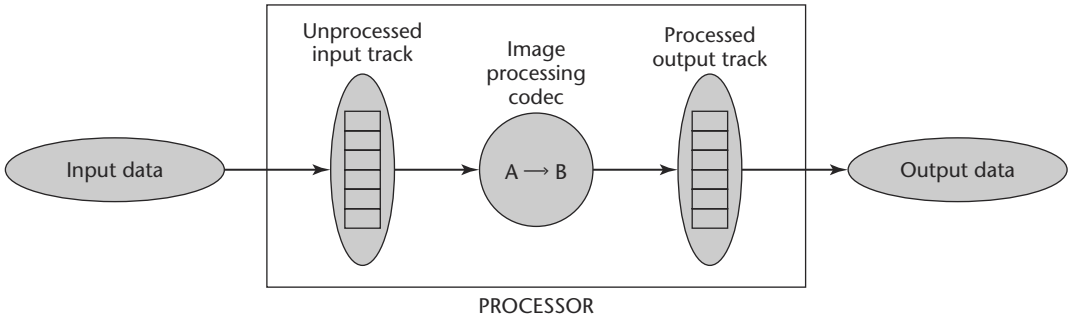
Processors can be used to present the input data source. A *processor* is a specialized type of player that can control what processing is performed on the media stream. Processors still support all of the same control mechanisms as a normal player. Figure 9.1 shows a diagram of the JMF data source/player architecture.

Another major component of a JMF application is the *Manager*. Managers handle the construction of players, processors, and data sources. Every application must begin with the introduction of a manager. Finally, to complete a JMF application a *media locator* must be deployed. A media locator is used to connect a specific media type with a manager. Media locators can be created with static files or a live feed. In the next section we will build our first JMF application.

9.2 The Java Live Viewer Application—Video Acquisition and Display Using Java

The first application we will walk you through is a live video viewer. This simple program will demonstrate the basics of JMF video programming. After finishing this section you should know how to create a processor, display video, and control a video feed.

FIGURE 9.2 Processor architecture.



9.2.1 *Creating a Processor/Player Architecture*

The processor is one of the critical components of the viewer application. Figure 9.2 shows a graphical diagram of the inner workings of the processor. It assumes complete control on the video stream. In order to implement a processor/player application the input media must be obtained using a media locator. In the code segment appearing in Listing 9.2.1.1 we create a `MediaLocator` object with an input string provided by the user on line 15. A typical input string represents a typical file path such as (*c:\Myvideo.mpg*) or the locator for the specific input device (*vfw://0*). To determine the locator for a live device see Appendix B. Once a valid media locator is available, a manager will be created. In the LiveViewer application we perform this step via a call to the `GetProcessor` function (line 24, Listing 9.2.1.1).

LISTING 9.2.1.1 The `main` function for the LiveViewer.

```
0:  // main function for the LiveViewer
1:  public static void main(String [] args)
2:  {
3:      . . .
4:      String url = args[0];
5:      //If the incorrect parameters were given
6:      //display the correct usage of the function
7:      if (url.indexOf(":") < 0)
```

```
8:      {
9:          CorrectUsage();
10:         System.exit(0);
11:     }
12:     //create a media locator
13:     MediaLocator ml;
14:
15:     if ((ml = new MediaLocator(url)) == null)
16:     {
17:        System.err.println("Cannot build media locator from: " + url);
18:        System.exit(0);
19:     }
20:
21:    //create the video filter
22:    LiveViewer lv = new LiveViewer()
23:    //try to get the video processor with the given input
24:    if (!lv.GetProcessor(ml))
25:        System.exit(0);
26:      . . .
27:  }
```

It is inside the `GetProcessor` function (shown in Listing 9.2.1.2) that a processor is created (namely `m_VideoProcessor`). This occurs via a call to the manager's `createProcessor` member function. The `MediaLocator` object (`ml`), created in the `main` function, is passed as an input argument.

LISTING 9.2.1.2 The `GetProcessor` function for the LiveViewer.

```
0:    //GetProcessor function definition
1:    public boolean GetProcessor(MediaLocator ml)
2:    {
3:     try
4:     {
5:         //try to create a processor with the MediaLocator
6:         m_VideoProcessor = Manager.createProcessor(ml);
7:     }
8:
9:     catch (Exception e)
```

```
10:    {
11:        System.err.println("Failed to create a processor from the
12:        given device: " + e);
13:        return false;
14:    }
15:    ...
16:   }
```

Once the processor has been created, we can start its configuration to meet our specific needs. Line 3 of Listing 9.2.1.3 describes how we add a *ControllerListener* to the video processor. A ControllerListener is used for *starting*, *pausing*, and *stopping* the streaming media. At a later point we will show you how a "listener" can be tied to the user interface.

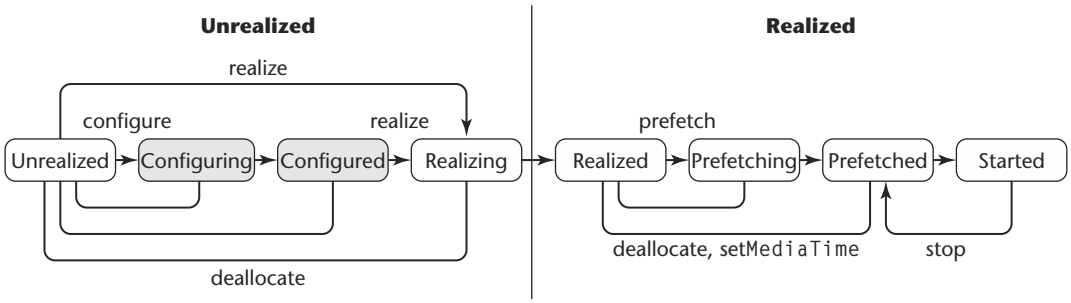
LISTING 9.2.1.3 Code for adding a controller listener.

```
1: ...
2: //add a controller listener to the video processor
3: m_VideoProcessor.addControllerListener(this);
4:
5:   ...
```

9.2.2 *Processor/Player States*

A processor/player goes through a series of states before it can use the input media. Processors start in an initial state called *unrealized*. At this time the processor exists, but it is not configured. Processor configuration is accomplished by calling the `Configure` member function (as shown in line 2 of Listing 9.2.2.1). In the configuring state the processor connects to the data source we determined earlier. Once configuration has been completed, the processor enters the configured state and posts a message as shown in line 3 of Listing 9.2.2.1. Finally, the processor is realized and its construction completes. Figure 9.3 shows the processor state diagram.

FIGURE 9.3 Processor states.



LISTING 9.2.2.1 Code for configuring the video processor.

```
1: // Put the Processor into configured state.
2: m_VideoProcessor.configure()
3: if(!waitForState(m_VideoProcessor.Configured))
4: {
5:     System.err.println("Failed to configure the processor.");
6:     return false;
7: }
```

9.2.3 *Data Source Media Tracks*

Data sources present media in a series of tracks. Each track contains a specific media type. For example, an MPEG movie with audio has two tracks, an audio track and a video track. Specific processing can be performed on each track through the `TrackControls`. `TrackControls`, as their name describes, are used to control what processing operations are performed on a track. In Listing 9.2.3.1 we get all of the track controls for our processor by calling the `getTrackControls` function (as shown in line 2). In this specific application, we are only concerned with the video track; therefore, we extract only this track from the list (see lines 15–21, Listing 9.2.3.1). Each track can use its own *codec* or *plug-in* for any desired processing. Next, we will show you how this can be done.

LISTING 9.2.3.1 Track controls.

```
1:  //Obtain the track controls.
2:  TrackControl trackControls[] = m_VideoProcessor.getTrackControls();
3:
4:  if (trackControls == null)
5:  {
6:    System.err.println("Failed to obtain the track controls from the
7:                              processor.");
8:    return false;
9:
10:  }
11:
12:  // Search for the track control for the video track.
13:  TrackControl videoTrack = null;
14:
15:  for (int i = 0; i < trackControls.length; i++)
16:  {
17:     if (trackControls[i].getFormat() instanceof VideoFormat)
18:     {
19:         videoTrack = trackControls[i];
20:         break;
21:     }
22:  }
23:    . . .
```

The step of obtaining the media tracks from the processor is followed by assigning the tracks codecs for processing. Lines 6–10 of Listing 9.2.3.2 demonstrate how to assign your own processing codec to the video track. In this example, the name of the codec is `FrameAccessCodec`. The video track is associated with the `FrameAccessCodec` and takes place in line 12. This assignment ensures that each input video frame will be processed. For the purposes of the LiveViewer application, the `FrameAccessCodec` function does not perform any image processing on the video stream. Details regarding pixel manipulation will be given later in Section 9.3.

LISTING 9.2.3.2 Sample software for codec assignment.

```
1:
2:      // instantiate and set the frame access codec to the data flow
3:      // path.
4:      try {
5:          //Instantiate an array of video codecs
6:          Codec codec[] =
7:          {
8:              //frame access codec
9:              new FrameAccessCodec(),
10:         };
11:
12:         videoTrack.setCodecChain(codec);
13:     }
14:
```


9.2.4 *Adding the Components to the User Interface*

Each processor contains two components used for displaying and controlling its media. The ControlComponent is used to control the media displayed. Common media controls are play, pause, stop, fast-forward, and rewind. In lines 14–15 of Listing 9.2.4.1, we grab the

LISTING 9.2.4.1 Sample code for using `controlComponent` and `visualComponent`.

```
1:
2:      //The control component
3:      Component controlComponent;
4:      Component visualComponent;
5:
6:      //Add the visual component (live video) to the west
7:      if ((visualComponent = m_VideoProcessor.getVisualComponent()) !=
8:       null)
9:      {
10:         add("West", visualComponent);
11:     }
12:
13:     //Add the control component (video controls) to the south
```

```
14:     if ((controlComponent =
15:         m_VideoProcessor.getControlPanelComponent()) != null)
16:     {
17:         add("South", controlComponent);
18:     }
19:
```

control panel for the video processor. The control component (`controlComponent`) is internally tied to the processor's `ControllerListener`, as mentioned earlier. In line 17 of Listing 9.2.4.1 we add the control component to the "south" portion of the application window. The `visualComponent` of the processor is used for displaying the particular media. In line 7 (Listing 9.2.4.1) we get the visual component from the processor. The visual component will display our live video in the application window. Finally, in line 10 the visual component is added to the application window.

9.2.5 *Creating a Codec*

A codec is a special nested class that allows us to manipulate the input media stream. In video applications we are mainly concerned with manipulating pixel values of a particular video frame. Earlier, we set a codec to the video track of our media. The `FrameAccessCodec` introduced earlier consists of two important functions: the `process` function and the `SaveFrame` function. The `process` function is a native function of the inherited codec class. Its main responsibility is to ensure that each input video frame from the source is processed and then sent out to the display. Listing 9.2.5.1 shows our overloaded version of the process function. Its input arguments include an input buffer for the incoming data and an

LISTING 9.2.5.1 Description of the `process` function.

```
1: //this function passes in the new video frame and puts out
2: //the processed video frame.
3:
4: public int process(Buffer in, Buffer out)
5: {
```

```
6:     // get the current frame of the input video buffer
7:     SaveFrame(in);
8:
9:    // swap the data between the input and output buffers.
10:    Object data = in.getData();
11:    in.setData(out.getData());
12:    out.setData(data);
13:
14:    // copy the input buffer attributes to the output buffer
15:    out.setFormat(in.getFormat());
16:    out.setLength(in.getLength());
17:    out.setOffset(in.getOffset());
18:
19:    return BUFFER_PROCESSED_OK;
20: }
```

output buffer for the processed data. In line 7 of Listing 9.2.5.1 we perform processing on the input frame through a member function of the `FrameAccessCodec` class called `SaveFrame`.

The `SaveFrame` function is shown in Listing 9.2.5.2. It performs all of the image processing on a video frame. To keep things simple, this example will not perform any image processing. Instead,

LISTING 9.2.5.2 Definition of the `SaveFrame` function.

```
1: //Callback function to access individual video frames for saving
2: void SaveFrame(Buffer frame)
3: {
4:     //take the input frame buffer and convert it to an image
5:     BufferToImage buff2Image =
6:         new BufferToImage((VideoFormat)frame.getFormat());
7:
8:    //create the java image
9:    Image currFrame = buff2Image.createImage(frame);
10:   //if the current frame is null, display a message to the
11:   //console and then exit
12:   if(currFrame == null)
13:   {
```

```
14:         System.err.println("Input Image is empty!!");
15:         System.exit(0);
16:
17:         //if the save menu item was selected save the current
18:         //frame to a file
19:         if(m_saveFlag == true);
20:         {
21:           //create a jpeg file of the current frame
22:           File f = new File("Live.jpg");
23:           try{
24:             if (!f.exists ())
25:                f.createNewFile ();
26:
27:             BufferedImage outImage = new
28:              BufferedImage(320,240, BufferedImage.TYPE_3BYTE_BGR);
29:
30:             outImage.createGraphics().drawImage(currFrame, 0, 0, null);
31:
32:             OutputStream os = new FileOutputStream(f);
33:
34:             com.sun.image.codec.jpeg.JPEGImageEncoder encoder =
35:             com.sun.image.codec.jpeg.JPEGCodec.createJPEGEncoder(os);
36:             encoder.encode(outImage);
37:             os.close();
38:           }
39:         }
40:     }
41: }
```

we will simply take the current frame and save it to a JPEG file per the user's request. In order to save a frame of video data we must convert the input buffer of pixel data into a Java image. This process starts in line 5 of Listing 9.2.5.2, where we create an instance of JMF's `BufferToImage` class. The `BufferToImage` class allows us to convert a buffer of pixel data into a Java image through its member functions. Notice that in line 6 of Listing 9.2.5.2 the constructor requires the assignment of a video format. The video format is important for the conversion, because the `BufferToImage` instance needs to know what type of input it will be converting (i.e., JPEG,

RGB, etc.). We create the Java image using the `createImage` member function of the `BufferToImage` class (line 9, Listing 9.2.5.2).

The LiveViewer application allows the user to save a frame through the menu. If the save menu item has been selected, the `m_saveFlag` member variable will be true allowing the `if-statement` at line 19 to be true. Now that the user has requested the JPEG format, he/she needs to save the Java image into a file, i.e., *ImageName.jpg*. This is shown in lines 22–25. For this application we have chosen a file named *Live.jpg*. If this file does not exist in the current directory, it will be created, otherwise it will be overwritten with our new image.

Lines 27–28 of Listing 9.2.5.2 show how to create a `BufferedImage` object, which we call `outImage`. There is a block of memory allocated to the `BufferedImage` for storing pixel data. The class constructor requires that image *resolution* and *bit depth* of a pixel be the input arguments. As seen in line 28, the current frame is saved as a 320×240 RGB image.

Next the `outImage` buffer that was created is assigned the current's frame pixel data (line 30, Listing 9.2.5.2). The graphics member of the `BufferedImage` class enables drawing of the image into our `outImage` area. The `drawImage` function requires the current frame, the upper-left coordinate of the image, the upper-right coordinate of the image, and an *ImageObserver*. In our example, we want to use the whole frame so we set the coordinates to be (0, 0). The ImageObserver is used for notifying the drawing window of any updates occurring to the input image. In the example under consideration, the current frame is drawn to memory; therefore, the ImageObserver value may be null.

The image data is saved into a file using an output stream (line 32 of Listing 9.2.5.2). Since we are saving the image into a JPEG file, we need to perform the JPEG encoding on the image data. The JPEG encoding is done using a Sun Microsystems encoder, as shown in lines 34–35 of Listing 9.2.5.2. More specifically, the encoder is linked with the output stream. This allows writing of data while performing the JPEG encoding. Simultaneously line 36 of Listing 9.2.5.2 describes encoding of the `outImage` and writing to the output stream.

The step of saving one frame to a file is followed by a process whereby the stream is sent back out for display. This is shown in lines 10–12 of Listing 9.2.5.1 where we copy the modified input buffer

into the output buffer for display. After the copying is complete, we make one final check for compatible formatting of the output stream (lines 15–17 of Listing 9.2.5.1).

Up to this point we have discussed the basic concepts of Java video programming with JMF. We will now show you how to perform image processing on a live video stream through a programming example named *VideoFilter* application.

9.3 The Java Video Filter Application—Pixel Manipulation Using Java

Pixel manipulation is an important part of video processing. The following section will show you how to access image pixels and manipulate them in Java.

9.3.1 *Using the Frame Access Codec*

As described in Section 9.2, we will use a codec to access each frame of video data from a data source. The Frame Access Codec for the `VideoFilter` consists of two main functions named `GetFrame` and `ProcessImage`. `GetFrame` is similar to the `SaveFrame` function used in the LiveViewer application. The difference, however, is that instead of saving the current frame to a JPEG file we send it to the `ProcessImage` function for manipulating the associated pixels.

9.3.2 *Getting the Pixels*

The `ProcessImage` function we are about to describe acquires the pixel data from the input video frame and manipulates them with a specific filtering matrix. To perform any processing on an image we need to get the pixels for the video frame. To access the pixels we use the `PixelGrabber` class. The `PixelGrabber` constructor takes an input image and puts its pixel values into an integer array. In lines 6–7 of Listing 9.3.2.1, we create a `PixelGrabber` with our input image and an integer array (`m_Pixels`). Getting the pixels now becomes an easy task. In line 19 of Listing 9.3.2.1 a call to the `grabPixels` member function is made in order to grab each pixel and store it to the assigned array.

LISTING 9.3.2.1 Definition of the `ProcessImage` function.

```
1: public void ProcessImage(Image img, int x, int y,
2:                          int w, int h)
3: {
4:
5:     //Create a pixel grabber
6:     PixelGrabber pg = new
7:         PixelGrabber(img,x,y,w,h,m_Pixels,0,w);
8:
9:     int pix;
10:    int a,r,g,b;
11:    int asum = 0;
12:    int rsum = 0;
13:    int gsum = 0;
14:    int bsum = 0;
15:
16:    //Try to grab the pixels
17:    try {
18:
19:            pg.grabPixels();
20:    }
21:    catch(InterruptedException e)
22:    {}
23: }
```

9.3.3 *Manipulating the Pixels*

Now that we have an array of pixels, we can easily manipulate their `RGBA` values. Listing 9.3.3.1 is a segment similar to the code used in `ProcessImage` of the `VideoFilter` class. Here we will show you how to take a particular pixel from a 320 × 240 image and change the color of the pixel to red. We start things off by creating a double `for-loop` that will give us access to each pixel in the image. On line 8 of Listing 9.3.3.1, we take the current indexed pixel from the array that was filled by the `PixelGrabber`. Once we have the current pixel we split it into four bands: ambient, red, green, and blue, as shown in lines 11–14 of Listing 9.3.3.1. Here we used the bit-wise AND (&) operator to extract each band. The format of a pixel, as shown in

FIGURE 9.4
Java pixel format.

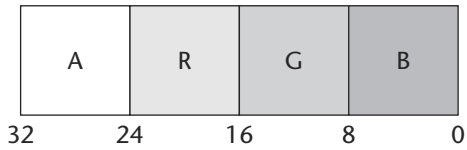


Figure 9.4, consists of 32 bits with each band consisting of 8 bits. The order of the bits are as follows; ambient, red, green, blue.

We decided earlier that we were going to set each pixel of the image to red. In lines 16–19 of Listing 9.3.3.1 we set the red band to full color (255) and the other bands to no color (0). This will give the pixel a bright red color. Once the pixel value has been set, we can put the four bands back into the pixel, as shown in line 23 of Listing 9.3.3.1. This operation consists of the bit-wise Or (|) operator. Once the bands are put back into a 32-bit format we can put the edited pixel back into its position in the array, as shown in line 26 of Listing 9.3.3.1.

LISTING 9.3.3.1 Pixel manipulation.

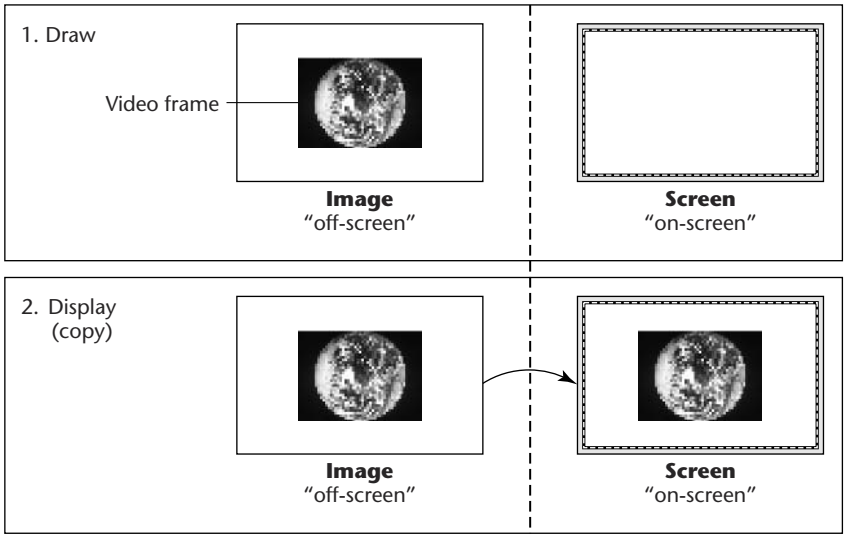
```
1:  //changing each pixel to red
2:  . . .
3:  //rows in the image (y)
4:  for(int j=1; j<239; j++)
5:  { //columns in the image (x)
6:     for(int i=1; i<319; i++)
7:     {   //get the current pixel at (x,y) = (i,j)from the array
8:         pix = m_Pixels[(j) * w + (i)];
9:
10:        //separate the pixel into ambient, red, green, and blue
11:        a = ((pix >> 24) & (0xff));
12:        r = ((pix >> 16) & (0xff));
13:        g = ((pix >>  8) & (0xff));
14:        b = ((pix ) & (0xff));
15:
16:        a = 0;
17:        r = 255;
18:        g = 0;
19:        b = 0;
```

```
20:          . . .
21:
22:         //put the edited color values back into the pixel
23:         pix = ((a<<24)|(r<<16)|(g<<8)|(b));
24:
25:         //put the edited pixel back into the array
26:         m_Pixels[j * w + i] = pix;
27:     }
28: }
```

9.3.4 *Displaying Using Double Buffering*

Processing successive frames of live video tends to add a flickering effect to the display. Double buffering is a technique often used in graphics programming to reduce the flickering effect found in high-speed animation. As shown in Figure 9.5, double buffering consists of having two buffers of video data, an *on-screen*, and an *off-screen* buffer. The currently displayed image comes from the on-screen buffer. While we are displaying the on-screen image, the next frame is loaded into the off-screen buffer. Then, when it is time for the

FIGURE 9.5 Picture showing how double buffering is performed



second frame, we transfer the off-screen buffer into the on-screen buffer and display it. This cycle continues for the duration of the clip. The off-screen buffer is the key component to eliminating the flickering. Since the next frame is loaded while the current is being displayed, we do not have to wait for loading when it is time for the new frame to be displayed.

In the VideoFilter application we use our own double-buffered canvas to display the manipulated image. Segments of this canvas class, named `VideoDisplay`, are shown in Listings 9.3.4.1 and 9.3.4.2. There are two essential functions to this class, `paint` and

LISTING 9.3.4.1 Definition of the `VideoDisplay` function.

```
1:
2:  public class VideoDisplay extends Canvas {
3:
4:  . . .
5:
6:      //Function for painting this canvas
7:      public void paint(Graphics g)
8:      {
9:          //Call the update function that uses double buffering.
10:
11:         update(g);
12:     }
13: }
```

LISTING 9.3.4.2 Definition of the `update` function.

```
1:
2: //Function for updating the display window using double buffering
3:
4: public void update(Graphics g)
5: {
6:      Dimension d = getSize();
7:
8:      //If this is the first time we are displaying the
```

```
9:      //image, load it.
10:     if(firstTime)
11:     {
12:         displayImage = createImage(d.width,d.height);
13:         offGraphics = displayImage.getGraphics();
15:         firstTime = false;
16:     }
17:
18:     //draw the image
19:     g.drawImage(this.displayImage,0,0,320,240,this);
20:
21:}
```

update. The paint function, as shown in lines 6–11 of Listing 9.3.4.1, can be called two different ways. Paint is often called by the system when the canvas has changed due to a resize or when a window that was covering the canvas region has been moved. Paint may also be called by the user application through the use of the repaint function call. Inside the paint function we make a single call to update.

The update function, as shown in Listing 9.3.4.2, performs the most important task in creating a double-buffered canvas. The first step involves getting the size of the canvas using the inherited function call to getSize. Next, we determine if this is the first time we are loading the image. If the flag is TRUE we, in effect, fill the off-screen buffer by creating an image of the correct size, as shown in line 12 of Listing 9.3.4.2, and then assign the off-screen graphics data, as shown in line 13 of Listing 9.3.4.2. Both displayImage and offGraphics are members of the VideoDisplay class. Once the image data has been loaded into the offGraphics member it can be drawn to the display using the drawImage function as shown in line 19 of Listing 9.3.4.2. The process of displaying the stored image in offGraphics to the screen simulates the idea of the on-screen buffer described above.

With the addition of pixel manipulation and double buffering to your JMF video program, you can create some really useful and

practical imaging applications. The development of a JMF application for a stand-alone PC follows the development of another application that enables the transmission of video across the Internet using JMF.

9.4 Video Data Transmission for the Web Using Real-Time Transfer Protocol

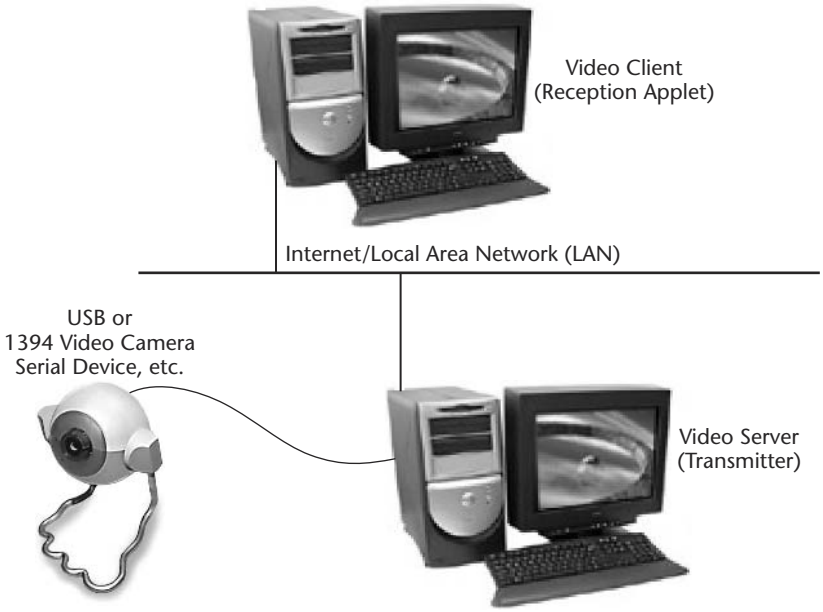
Java has become a very powerful programming tool for the Internet. The development of JMF has offered more than simple image-processing capabilities to the Java language. JMF is intended to be used for live video streaming across the Internet. In this section we will explain how to program a Java video application for Internet transmission.

9.4.1 *Real-Time Transfer Protocol*

We will begin this section by introducing the protocol used by JMF, called the *Real-Time Transfer Protocol* (RTP), for streaming video between remote computers on a *local area network* (LAN) or across the Internet.

To stream video data we need to establish a video server. A video server consists of a designated standard PC with a Web camera connected to it and a Java application for transmitting video data to client(s). The video server is responsible for managing the connections between itself and all of its clients. A connection link between the video server and a client is called a *Session*. A server can have many different sessions. In order for the video server to keep all of its sessions organized, it uses an *RTPManager*. RTPManagers manage the creation, maintenance, and closing of an RTP session. The receiving end of the system consists of a video client. The video client is essentially a Java applet that connects to the video server, then receives and displays the video data. Figure 9.6 shows the typical architecture of an RTP transmission/reception system. In the following sections we will show you how to create each of these components using JMF programming.

FIGURE 9.6 Video transmission using a client/server architecture.



9.4.2 *Creating the RTP Video Server*

The video server is responsible for setting up three necessary components for video transmission. First, the server must set up the input video from a connected camera (Section 9.4.3). Next, the server must establish communication between itself and the video client (Section 9.4.4). Finally, the server must allow the user to control the video transmission, allowing the server to start and stop transmitting (Section 9.4.5).

9.4.3 *Setup of the Video Input/Output*

Setting up the video input for the server is very similar to the methods used in the previous applications we have created. The whole process lies inside the `createProcessor` function. Lines 9–15 of Listing 9.4.3.1 should look familiar from Section 9.2.1. Here we use the media locator to find our video device at a specified location. As noted earlier, to find the location of the video device look at the steps described in Appendix B. Once we get the location of the

LISTING 9.4.3.1 Definition of the `createProcessor` function.

```
1:
2: // create a video processor that will take video input from
3: // the camera driver and create an output in the JPEG/RTP format
4:
5: private String createProcessor()
6: {
7:     //see if we have a valid media locator
8:    if(mediaLocator == null)
9:        return "Media Locator is NULL";
10:     DataSource ds;
11:     DataSource clone;
12:
13:    //create a data source with our media locator (the camera driver)
14:    try{
15:        ds = javax.media.Manager.createDataSource(mediaLocator);
16:    }
17:    catch (Exception e) {
18:        return "Couldn't create DataSource";
19:    }
20:
21:    //create the video processor for handling the input media
22:    try {
23:        VideoProcessor = javax.media.Manager.createProcessor(ds);
24:    }
25:    catch (NoProcessorException npe) {
26:        return "Could not create the video processor";
27:    }
28:
29:    catch (IOException ioe) {
30:        return "IO Exception creating processor";
31:    }
32:
33:    //wait for the processor to configure
34:    boolean result = waitForDesiredState(VideoProcessor,
35:                                         Processor.Configured);
36:
37:    if (result == false)
38:        return "Could not configure the processor";
39: }
```

media device from our GUI (in line 8 of Listing 9.4.3.1), we simply check to see if a location has been specified. Then, in lines 10–15 of Listing 9.4.3.1, we create a `DataSource` with the `mediaLocator`. Once the `DataSource` is set, the creation of the `VideoProcessor` follows, as shown in lines 22–24 of Listing 9.4.3.1. The processor can be used only when its state is configured appropriately. To determine when the processor is configured we have provided a function that checks the current state and waits if the state has not yet been achieved. As shown in line 34 of Listing 9.4.3.1, the function `waitForDesiredState` is used to make sure that the configuration process has been completed.

Listing 9.4.3.2 shows how the `waitForDesiredState` function works. In line 8 of Listing 9.4.3.2, note that we have added a `ControllerListener`, using our `ProcessorStateListener` inner class, to the processor.

LISTING 9.4.3.2 Definition of the `waitForDesiredState` function.

```
1:  // This function waits for the processor to reach the
2:  // given input state or for failure to be determined for
3:  // the current processor.
4:
5:  private synchronized boolean
6:  waitForDesiredState(Processor p, int state)
7:  {
8:     p.addControllerListener(new ProcessorStateListener());
9:     failed = false;
10:
11:    // call the required method on the processor
12:    if (state == Processor.Configured)
13:        p.configure();
14:    else if (state == Processor.Realized)
15:        p.realize();
16:
17:   //wait until we get an event that confirms the
18:   //success of the method, or a failure event.
19:   //see StateListener inner class
20:   while (p.getState() < state && !failed) {
```

```
21:         synchronized (getStateLock()) {
22:             try {
23:                     getStateLock().wait();
24:             }
25:             catch (InterruptedException ie) {
26:                     return false;
27:             }
28:         }
29:    }
30:
31:    if (failed)
32:     return false;
33:    else
34:     return true;
35: }
```

The `ProcessorStateListener` listens for `controllerEvents` for the current processor, then notifies the waiting thread when something has been received. This is shown in lines 15–18 of Listing 9.4.3.3, where all the threads are notified about the current event (i.e., a state change) that has occurred.

LISTING 9.4.3.3 Definition of the `ProcessorStateListener` inner class.

```
1:  //This inner class listens for states from the controllers.
2:  class ProcessorStateListener implements ControllerListener
3:  {
4:     public void controllerUpdate(ControllerEvent ce)
5:     {
6:         //If there was an error during configure or
7:         //realize, the processor will be closed
8:       if (ce instanceof ControllerClosedEvent)
9:        setFailed();
10:
11:       //All controller events, send a notification
12:       //to the waiting thread in waitForDesiredState method.
13:       if (ce instanceof ControllerEvent) {
14:            synchronized (getStateLock())
```

```
15:             {
16:                 getStateLock().notifyAll();
17:             }
18:         }
19:     }
20: }
```

Returning to Listing 9.4.3.2, in lines 8–15 we check the first possible state of the processor, "configured." If the desired state is not the configured state, we compare the current state of the processor with the desired state, as shown in line 20 of Listing 9.4.3.2. If the desired state has not yet been reached, the current thread is locked until a notification is received from the `ProcessorStateListener`. This notification wakes up the thread and brings us to the top of the loop again. There, we check the current state with the desired state. Once the comparison has been satisfied, a `TRUE` value is returned if the state has been reached and `FALSE` if it cannot be reached anymore.

Now that the processor has been created and is configured, it is time to look at the input video tracks the processor will be receiving from the `DataSource`. As we did in Section 9.2.3, we need to get the track controls for the video processor, as shown in line 4 of Listing 9.4.3.4. In order to make our search of the supported tracks

LISTING 9.4.3.4 Segment from the `createProcessor` function.

```
1:  .  .  .
2:  //Get the media tracks from the processor (We are
3:  //only concerned with the video track)
4:  TrackControl [] tracks = VideoProcessor.getTrackControls();
5:
6:  //Check for at least 1 valid track, likely to be the video
7:  if (tracks == null || tracks.length < 1)
8:    return "Could not find any tracks in video processor";
9:
10: //Set the output content descriptor to RAW_RTP
11: //This will limit the supported formats reported
12: //from Track.getSupportedFormats to only valid RTP
13: //formats, Since the main point of this program is
```

```
14: //to transmit RTP video data.
15: ContentDescriptor cd = new
16:           ContentDescriptor(ContentDescriptor.RAW_RTP);
17: VideoProcessor.setContentDescriptor(cd);
18:
19: Format supportedTracks[];
20: Format chosenTrack;
21: boolean oneTrackFlg = false;
22:
23: //Set the format of the tracks
24: for (int i = 0; i < tracks.length; i++) {
25:    //Get the format for the current track
26:    //Format format = tracks[i].getFormat()
27:
28:    if (tracks[i].isEnabled()) {
29:        //Get the formats supported by the current track
30:        supportedTracks = tracks[i].getSupportedFormats();
31:
32:        //We've set the output content to the RAW_RTP.
33:        //So all the supported formats should work with RTP.
34:        // We'll just pick the first one.
35:        if (supportedTracks.length > 0) {
36:            //We are only interested in the video,
37:            //so we need to see if this is a video track
38:            if(supportedTracks[0] instanceof VideoFormat) {
39:                //We need to check the video size since
40:                //not all formats work in all sizes.
41:                chosenTrack = VideoSize(tracks[i].getFormat(),
42:                                        supportedTracks[0]);
43:            }
44:        }
45:    }
46: }
47: .  .  .
```

quickly, we set up a `ContentDescriptor` in lines 15–17 of Listing 9.4.3.4. The `ContentDescriptor` allows us to specify the type of tracks we would like returned. In our case we want only raw video that can be sent using RTP. Therefore, the `ContentDescriptor` is set to `RAW_RTP`. Next, we loop through all of the tracks contained in

the video processor and store those that meet our supported format. As lines 28–30 of Listing 9.4.3.4 show, if the current track is enabled, we save its supported formats into an array. Later in line 35 of Listing 9.4.3.4, we check our array of supported tracks to see if we have found at least one supported track. If so, we check the track to see if it is a video track, since we are interested only in transmitting the video, as seen in line 38 of Listing 9.4.3.4. Once we have a video track we need to make sure that it is an appropriate size for RTP transmission. To check the size we make a call to our own function, `VideoSize`, in line 41 of Listing 9.4.3.4.

The `VideoSize` function takes the format of the track from the processor (original format) and the supported format as its input. Since JPEG and *H*263 formats will work only for certain sizes, we need to check the original format against these. In lines 9–11 of Listing 9.4.3.5, we create instances of each of these formats. Then we start by comparing the original format with the JPEG format. If it has been determined that the original format is of type JPEG, in lines 17–26 of Listing 9.4.3.5 we check its width and height. For JPEG video the

LISTING 9.4.3.5 Definition of the `VideoSize` function.

```
1:   //JPEG and H263 will only work for certain sizes. So, we do
2:   //some extra checking here to make sure the input sizes are OK.
3:   Format VideoSize(Format originalFormat,
4:                    Format supportedFormat) {
5:     //For the dimensions of the video int width, height;
6:     //Get the input video size
7:    Dimension size = ((VideoFormat)originalFormat).getSize();
8:    //Create a JPEG format
9:    Format JPEGFormat = new Format(VideoFormat.JPEG_RTP);
10:    //Create a H263 format
11:    Format H263Format = new Format(VideoFormat.H263_RTP);
12:    //Check to see if the supportedFormat is a JPEG format
13:    if (supportedFormat.matches(JPEGFormat)) {
14:        //For a valid JPEG we need to make sure width
15:        //and height are divisible by 8.
16:        //Check the width
17:        if((size.width % 8) == 0)
18:            width = size.width;
```

```
19:         else
20:             width = (int)((size.width / 8) * 8);
21:
22:         //Check the height
23:         if((size.height % 8) == 0)
24:             height = size.height;
25:         else
26:             height = (int)((size.height / 8) * 8);
27:     }
28:     //Else if the supportedFormat is a H263 format
29:     else if (supportedFormat.matches(H263Format)) {
30:         // For H263 we can only support certain sizes
31:         if(size.width < 128) {
32:             width  = 128;
33:             height = 96; }
34:         else if (size.width < 176) {
35:             width  = 176;
36:             height = 144; }
37:         else {
38:             width  = 352;
39:             height = 288; }
40:     }
41:     //Else just return the input, size it does not
42:     //need any special changes.
43:     else {
44:         return supportedFormat; }
45:
46:     //Return the format with the new size
47:     return (new VideoFormat(null,new Dimension(width, height),
48:               Format.NOT_SPECIFIED, null,Format.NOT_SPECIFIED)).
49:                  intersects(supportedFormat);
50:   }
```

width and height must be divisible by 8. In line 17 of Listing 9.4.3.5 we check to see if the width is divisible by 8, in which case we do nothing. If this is not the case we make it divisible by 8 by first dividing by 8, dropping the remainder, and then multiplying by 8, as shown in line 20 of Listing 9.4.3.5. The same process works for height in lines 22–26 of Listing 9.4.3.5. If the original format is not JPEG,

but found to be *H*263, in line 29 of Listing 9.4.3.5 we must choose the closest supported size. Lines 31–39 of Listing 9.4.3.5 show that the *H*263 format only supports the following dimensions: 128 × 96, 176 × 144, 352 × 288. Finally, if the original format does not match either JPEG or *H*263, we simply leave it alone because there is no need to change its dimensions.

Now that we have found a supported video track and set the dimensions, if needed, it is time to realize the processor. Realizing the processor will attempt to create an output `DataSource` for sending the RTP video frames to the client. In line 3 of Listing 9.4.3.6, we use the same function that we did above for synchronizing the processor state. Only this time we desire the "realized" state. Once the processor is realized, we get the output `DataSource` from the processor as shown in line 9 of Listing 9.4.3.6. To complete the setup of the video input/output we create a `frameGrabber` in lines 12–14 of Listing 9.4.3.6. The frame grabber will be used to grab the input video frame from the camera. Now that our input and output have been set up it is time to set up the communication link between the transmitter and receiver.

LISTING 9.4.3.6 Example on how to realize the `Processor` and get its output.

```
1:  //Realize the processor. This will create a flow graph and
2:  //attempt to create an output datasource for RTP video frames.
3:  result = waitForDesiredState(VideoProcessor,
4:                               Controller.Realized);
5:  if (result == false)
6:     return "Could Not realize the video processor";
7:
8: //Get the output data source of the video processor
9: dataOutput = VideoProcessor.getDataOutput();
10:
11: //Get the FrameGrabbingControl from the processor
12: frameGrabber = (FrameGrabbingControl)
13: VideoProcessor.getControl("javax.media.Control.
14:                           FrameGrabbingControl");
15: . . .
16:
```

9.4.4 *Setup Communication/Create Sessions*

The communication between the server and clients is the second essential component of a network-based video-streaming application. As described in the introduction, the connection between the client and server is called a session. Listing 9.4.4.1 focuses on our `createTransmitter` function. This function is used to set up the video transmission sessions.

LISTING 9.4.4.1 Part I: Definition of the `createTransmitter` function.

```
1: // This function creates the video transmitter
2: private String createTransmitter() {
3:
4:   //Create a push buffer data source from the output
5:   //data source
6:   PushBufferDataSource pbds = (PushBufferDataSource)dataOutput;
7:
8:   //Get the push buffer stream from the push buffer data source
9:   PushBufferStream pbs[] = pbds.getStreams();
10:
11:  //Create an array of rtpManagers equal to the number
12:  //of clients/streams we will be connecting to.
13:  rtpManagers = new RTPManager[pbs.length];
14:
15:  //Addresses for the client and server of the session
16:  SessionAddress serverAddress, clientAddress;
17:  InetAddress clientIPAddr;
18:
19:  //Stream we will use to send the data on
20:  SendStream sendStream;
21:
22:  //Port we will be sending the data from on this machine
23:  int localPort;
24:
25:  //Loop through all the streams we will be pushing data to.
26:  //In this simple example, we will only be using 1 client,
27:  //therefore we will have only 1 stream.
28:  for (int i = 0; i < pbs.length; i++) {
29:   try {
```

```
30:     //Create a new instance of the RTPManager
31:     rtpManagers[i] = RTPManager.newInstance();
32:
33:     //We will transmit and receive on the same port number,
34:     //since this is assumed by JMStudio.
35:     localPort = portBase + 2*i;
36:     //Get the client's address
37:     clientIPAddr = InetAddress.getByName (ClientIPAddress);
38:
39: . . .
```

To start, we instantiate all of the necessary members. These include the addresses for the sessions, a local port for the video, an `rtpManager` for managing each session, and finally a `PushBufferDataSource`, as shown in lines 6–13 of Listing 9.4.4.1. A `PushBufferDataSource` is used to manage the data in the form of PushStreams. PushStreams transfer data in entire media chunks to the clients, which ensures that only whole frames of video are sent at a time. Once we have all of the necessary members, we establish a loop to transmit data to all of the streams or clients in our system. In this beginner example, we will use only one video stream connected to one client; therefore, our loop will contain only one iteration. Inside the loop we start by instantiating the `rtpManagers`, as shown in line 31 of Listing 9.4.4.1. Next, in lines 35–45 of Listings 9.4.4.1 and 9.4.4.2, we assign the port number and IP addresses for the client and server as they were entered into the GUI. Now that we know where to look for everything, we initialize the `rtpManagers` with our server machine as shown in line 48 of Listing 9.4.4.2. In line 51 of Listing 9.4.4.2 we add the client machine to the same manager. With the client and server added to the `rtpManager` we now add a `sendStream`. The `sendStream` will be used for sending data from the server to the client, as shown in line 58 of Listing 9.4.4.1. This `sendStream` is connected to the output `DataSource` that we created earlier. Finally, in line 61 of Listing 9.4.4.2, we start the `sendStream`, which will allow the stream to send data when it is available. It is important to note that we are not actually sending any video data yet. Video transmission is not started until the `videoProcessor` allows, with its own `start` function.

LISTING 9.4.4.2 Part II: Definition of the `createTransmitter` function (continued).

```
1: // This function creates the video transmitter
2: private String createTransmitter() {
3:
4: . . .
39:     //Get this machine's address
40:     serverAddress = new SessionAddress
41:                          (InetAddress.getLocalHost(),localPort);
42:
43:     //Get the client machine's address
44:     clientAddress = new SessionAddress( clientIPAddr,
45:                                         localPort);
46:
47:     //Initialize the manager for the server machine
48:     rtpManagers[i].initialize(serverAddress);
49:
50:     //Add the client machine to the current manager
51:     rtpManagers[i].addTarget(clientAddress);
52:
53:     System.err.println( "Created RTP session: " +
54:                         ClientIPAddress + " " + localPort);
55:
56:    //Create a send stream with the output data source
57:    //for the current manager
58:    sendStream = rtpManagers[i].CreateSendStream(dataOutput, i);
59:
60:    //Start the send stream
61:    sendStream.start();
62:  }
63:  catch (Exception  e) {
64:    return e.getMessage();
65:  }
66: }
67: return null;
68:}
```

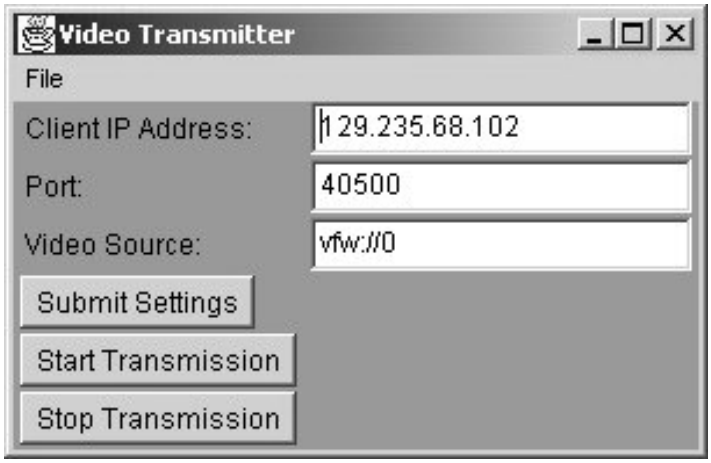


FIGURE 9.7
Java video transmitter dialog appearing at the server's display.

9.4.5 *Create a GUI for Transmission Controls*

To complete the server side of the streaming video application we need to create a GUI that will allow the user to enter important location information and control the transmission of video data. Figure 9.7 gives you an idea of what the GUI will look like on the server's display.

Since this book focuses on details of JMF rather than Java GUI programming, we will not go into the details on how to create the interface. We will, however, show you how the interface is related to controlling the video transmission. As seen in Listing 9.4.5.1, we have a button callback function to handle the `Start Transmission` and `Stop Transmission` buttons. In lines 16–19 of Listing 9.4.5.1 you will see that when you click on the `Start Transmission` button, the GUI application's instance of the `videoServer`, from above, calls the `start` function to begin transmission. Likewise, the `Stop` button has similar functionality in lines 22–25 of Listing 9.4.5.1.

LISTING 9.4.5.1 Definition of the `actionPerformed` function.

```
1:  //Listener function for buttons
2:  public void actionPerformed(ActionEvent e) {
4:    String command = e.getActionCommand();
5:
```

```
6:    //submit button pressed
7:    if(command == SUBMIT) {
8:
9:      Format fmt = null;
10:     videoServer.SetParameters(
11:      new MediaLocator(m_sourceText.getText()),
12:       m_clientIPText.getText(),m_portText.getText(), fmt);
13:   }
14:
15:   //submit button pressed
16:   else if(command == START) {
17:
18:     videoServer.start();
19:   }
20:
21:   //submit button pressed
22:   else if(command == STOP) {
23:
24:     videoServer.stop();
25:   }
26: }
```

Listing 9.4.5.2 shows how the `videoServer` starts the data transmission to the client. The function starts off by creating the `VideoProcessor` in line 13 of Listing 9.4.5.2, through the `createProcessor` function described earlier. Next, we create the RTP video transmitter in line 20 of Listing 9.4.5.2, by calling the `createTransmitter` function that we described above. Finally, we call the `VideoProcessor`'s `start` member function, as shown in line 31 of Listing 9.4.5.2. This will start the communication and video transmission between this video server and the connected clients.

LISTING 9.4.5.2 Function definition for initiating video transmission.

```
1:  // Create a processor to process the video input in
2:  // JPEG/RTP format
3:  // and then start the transmission to the client applet.
4:
```

```
 5:  public synchronized String start()
 6:  {
 7:      String result;
 8:
 9:      // Create a processor for the video input
10:      // specified by the media locator.  Set the output
11:      // from the processor to JPEG/RTP for transmission
12:      // to the client.
13:      result = createProcessor();
14:      if (result != null)
15:         return result;
16:
17:      // Create an RTP session to transmit the
18:      // processor's video output to the client applet
19:      // at the given IP address and port
20:      result = createTransmitter();
21:      if (result != null)
22:      {
23:         VideoProcessor.close();
24:         VideoProcessor = null;
25:         return result;
26:      }
27:
28:      // Begin the transmission of the video data from
29:      // the server this machine to the client
30:      // containing the reception applet
31:      VideoProcessor.start();
32:
33:      return null;
34: }
```

Stopping the video transmission is now very easy. First, we must stop the `VideoProcessor`, as shown in line 12 of Listing 9.4.5.3. This will halt all communication between the server and its clients, and it will stop the server from grabbing frames from the camera. Next, we must call the video processor's `close` member function to clean up all the necessary resources. This is very important because without this there is the potential that the video resources will not be freed for the next time you run the application. Finally, in lines 16–20

LISTING 9.4.5.3 Definition of the `stop` function.

```
 1:  // Stop the video data transmission.
 2:  public void stop()
 3:  {
 4:
 5:   synchronized (this)
 6:   {
 7:    //If the video processor still exists, stop and close the
 8:    //processor and dispose of any rtp session managers
10:    if (VideoProcessor != null)
11:    {
12:       VideoProcessor.stop();
13:       VideoProcessor.close();
14:       VideoProcessor = null;
15:
16:       for (int i = 0; i < rtpManagers.length; i++)
17:       {
18:         rtpManagers[i].removeTargets("Video Transmission Ended");
19:
20:         rtpManagers[i].dispose();
21:     }
22:   }
23:  }
24: }
```

of Listing 9.4.5.3, we release all objects allocated in `rtpManagers` during the course of the session to be "garbage collected" by calling the `dispose` function.

With the implementation of the controlling GUI, the server is now ready to start the video transmission process. Next we will discuss the video client's code.

9.4.6 *Creating the RTP Session Client*

Now that we have created the video server for sending the video it is time to create a client, which can display our video in an applet. We start by setting up the layout of the applet. In line 17 of Listing 9.4.6.1, we decide on a default applet size of 320 × 240. The

LISTING 9.4.6.1 Part I: Client applet.

```
1:  //This function initializes the applet, setting the
2:  //background color, size, and sets up the parameters
3:  // for the input stream path.
4:
5:  public void init()
6:  {
7:    setLayout(null);
8:
9:    //Set background color to white
10:   setBackground(Color.white);
11:
12:   //Create a 320x240 panel for displaying the received video
13:
14:   appletPanel = new Panel();
15:   appletPanel.setLayout( null );
16:   add(appletPanel);
17:   appletPanel.setBounds(0, 0, 320, 240);
18:
18:   //String for the network path of the client machine
19:   //running this applet
20:   String mediaFile = null;
21:
22:   //Media locator used for locating the
23:   //video from the rtp stream
24:   MediaLocator mrl = null;
25:
```

applet will resize itself if video has a different dimension or there are additional controls, which require more space. Starting on line 24 of Listing 9.4.6.1, we create a media locator for the video. This is similar to the method used by the server. The main difference for the client is that the media will not come from a location directly on the client machine. Instead, the media locator will be pointed to the IP address of the client machine, combined with the same video port that was used by the server. As shown in line 31 of Listing 9.4.6.2, the input media is determined by the Web page, which displays this applet. We will show you how this happens later. If we were successful in creating a media locator with the given address/port, we will now create

LISTING 9.4.6.2 Part II: Client applet.

```
1:  //This function initializes the applet, setting the
2:  //background color, size, and sets up the parameters
3:  // for the input stream path.
4:
5:  public void init()
6:  {
7:   . . .
26:   try {
27:
28:     // Create a media locator from the url given from
29:     // the web page that contains this applet.
30:
31:     mediaFile = getParameter("FILE");
32:
33:
34:     if ((mrl = new MediaLocator(mediaFile)) == null)
35:       Fatal("Cannot locate specified media: " + mediaFile);
36:
37:
38:     //Otherwise create an instance of a player for this media
39:
40:     try
41:     {
42:       videoPlayer = Manager.createPlayer(mrl);
43:     }
44:     catch (NoPlayerException e)
45:     {
46:
47:       System.out.println(e);
48:       Fatal("Could not create player for " + mrl);
49:     }
50:     //Add this applet as a listener to the player's events
53:     videoPlayer.addControllerListener(this);
54:   }
56:   catch (IOException e)
57:   {
58:
59:     Fatal("IO exception creating player for " + mrl);
60:   }
61: }
```

a videoPlayer in line 42 of Listing 9.4.6.2. The videoPlayer is responsible for controlling the input video stream and displaying it in the applet. The final bit of initialization needed for this applet is to add a controller listener to the videoPlayer. In line 53 of Listing 9.4.6.2, we call the videoPlayer's addControllerListener member function. As an input parameter we use "this," which corresponds to the applet. We want to have the applet as a controller to the videoPlayer so that the player responds appropriately to whatever happens to the applet. In the most important case, if the applet is closed, we want the videoPlayer to also be closed appropriately.

The start function for this applet is very simple. As shown in Listing 9.4.6.3, all we need to do is start the videoPlayer, so that it may receive video via the RTP stream and display it in the applet window.

LISTING 9.4.6.3 Start function for the client applet.

```
1:  // Get the player to start the video playback.
2:  //This function is called the first time the applet runs.
3:
4:
5:  public void start()
6:  {
7:    if (videoPlayer != null)
8:       videoPlayer.start();
9:  }
```

The Stop function for this applet is also very simple. As shown in Listing 9.4.6.4, we simply stop the videoPlayer, then deallocate any resources that were used by the player. It is very important that the deallocate function is called because it ensures that all of the resources are free for the next time the applet is used. The stop function can be called in two different cases:

- The user presses the Stop button on the Web browser.
- The user closes the Web browser.

LISTING 9.4.6.4 `Stop` function for the client applet.

```
1:  // Stop the player and deallocate any resources used.
2:  public void stop()
3:  {
4:    if (videoPlayer != null)
5:    {
6:      videoPlayer.stop();
7:      videoPlayer.deallocate();
8:    }
9:  }
```

The most important work of the video client applet occurs in the `controllerUpdate` function. This function is called for three specific media events that can occur during the life of the applet.

The first type of event is a `RealizeCompleteEvent`, which occurs at line 12 of Listing 9.4.6.5. A `RealizeCompleteEvent` occurs

LISTING 9.4.6.5 Part I: Definition of the `controllerUpdate` function.

```
1:  //This function is called when there is a media event, and
2:  //handles the particular media event appropriately
3:  public synchronized
4:  void controllerUpdate(ControllerEvent event)
5:  {
6:    // If the videoPlayer no long exists, return
7:
8:    if (videoPlayer == null)
9:      return;
10:   //When the videoPlayer is realized
11:
12:   if (event instanceof RealizeCompleteEvent)
13:   {
14:     int width = 320;
15:     int height = 0;
16:     //If the control component has not been created yet,
17:     //try to create it right now.
18:
```

```
19:     if (controlComponent == null)
20:       if (( controlComponent = videoPlayer.
21:              getControlPanelComponent()) != null)
22:       {
23:
24:          //If we successfully created the control component,
25:          //get the height of it and add it to the Panel
26:
27:
28:          controlPanelHeight = controlComponent.
29:                 getPreferredSize().height;
30:
31:          appletPanel.add(controlComponent);
32:          height += controlPanelHeight;
33:       }
34: . . .
```

when the `videoPlayer` has finished its realization process. At this time we are ready to set up the applet for displaying and controlling the video. Here, we will add the `controlComponent` and the `visualComponent` to the applet so that they will be displayed to the user. We start by adding the `controlComponent` to the applet, as shown in lines 19–32 of Listing 9.4.6.5. Provided that the controlComponent has not already been created, we get the `controlComponent` panel for the `videoPlayer`, as seen on line 21 of Listing 9.4.6.5. If we are successful in creating the `controlComponent`, we add it to the `appletPanel`. Simultaneously, we update the height of the `appletPanel` to accommodate the addition of the `controlComponent`. This step is necessary both for resizing the applet and for ensuring that there is enough room for the controls as seen in lines 30–32 of Listing 9.4.6.5.

Next, we create the `visualComponent`, starting on line 37 of Listing 9.4.6.6. If we were successful in creating the visualComponent, we also add this to the `appletPanel`. Next, we make sure that there is enough room in the `appletPanel` to include the `visualComponent`. In lines 47–54 of Listing 9.4.6.6 we determine the necessary dimensions of the `visualComponent` and adjust our local width and height appropriately. After we have added each component, it is

LISTING 9.4.6.6 Part II: Definition of the `controllerUpdate` function.

```
1:  //This function is called when there is a media event, and
2:  //handles the particular media event appropriately
3:  public synchronized
4:  void controllerUpdate(ControllerEvent event)
5:  {
6:   . . .
35:    //If the visual component has not been created yet,
36:    //try to create it now.
37:    if (visualComponent == null)
38:      if (( visualComponent = videoPlayer.
39:             getVisualComponent())!= null)
40:      {
41:
42:         //If the visual component was created successfully,
43:         //add it to the panel and determine the video size.
44:
45:
46:         appletPanel.add(visualComponent);
47:         Dimension videoSize = visualComponent.
48:                                 getPreferredSize();
49:         videoWidth = videoSize.width;
50:         videoHeight = videoSize.height;
51:         width = videoWidth;
52:         height += videoHeight;
53:         visualComponent.setBounds(0, 0,
54:                                 videoWidth, videoHeight);
55:     }
56:
57:   //Set the size of the panel to be the width of the video
58:   //by the height of the video plus, the height of
59:   //the control component.
60:   appletPanel.setBounds(0, 0, width, height);
61:
62:   //If the control component exists, set its size.
63:
64:   if (controlComponent != null)
65:   {
66:     controlComponent.setBounds(0, videoHeight, width,
67:                                  controlPanelHeight);
```

```
68:     controlComponent.invalidate();
69:   }
70:
71: }
```

time to resize the `appletPanel` to hold both of the components. In line 60 of Listing 9.4.6.6, we reset the bounds of the applet with the local variables, width, and height, that we have used throughout the process.

The second type of event is a `ControllerEventError`. This type of event occurs when there is some error condition that will cause the controller to cease functioning. Our only option at this point is to kill the `videoPlayer`, as seen in line 4 of Listing 9.4.6.7.

The final type of event is a `ControllerClosedEvent`. This type of event occurs when the `videoPlayer` has been closed and is no longer operational. As shown in line 12 of 9.4.6.7, the only thing that we can do in this case is clear off the appletPanel.

LISTING 9.4.6.7 Segment of `controllerUpdate` function.

```
1:  .  .  .
2:  else if (event instanceof ControllerErrorEvent) {
3:      //Kill the videoPlayer applet
4:      videoPlayer = null;
5:      Fatal(((ControllerErrorEvent)event). getMessage());
6:
7:  }
8:
9:  else if (event instanceof ControllerClosedEvent)
10: {
11:     //Remove the panel from the applet
12:     appletPanel.removeAll();
13: }
14:
15:}//end of controllerUpdate
```

Now that we have the video client applet complete, we will now be able to establish a connection from the video server and display the video we receive.

9.4.7 *Using the Client Applet on the Web*

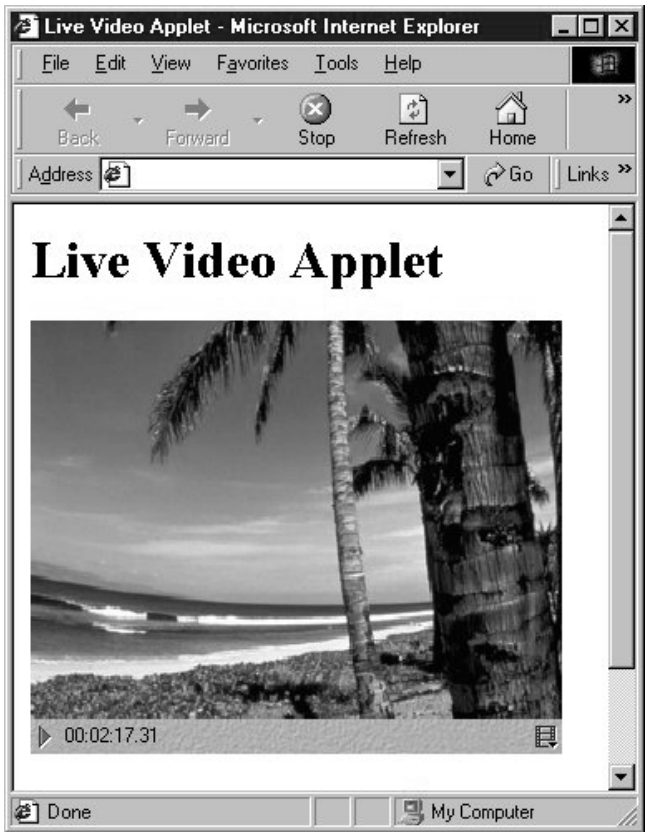
With the completion of the video server and video client, we need a way to view the results of our hard work. It is now time to embed the client applet in a Web page for your viewing pleasure!

The insertion of the applet code into the *.html* file is business as usual, except for one important thing. We must specify the media input that will be used by the applet. As seen in lines 16–17 of Listing 9.4.7.1, we need to provide the IP address of the client machine along with the video port of the server. Once this has been set up to meet your networking requirements, you will get an applet like the one in Figure 9.8. Since we specify the IP address and port number within the *.html* page, there is no need to recompile the applet code if you decide to move the client to another machine.

LISTING 9.4.7.1 Embedding the applet on a Web page.

```
1:  <html>
2:  <head>
3:  <title>Live Video Applet</title>
4:  </head>
5:  <body>
6:
7:  <h1>Live Video Applet</h1>
8:
9:  <!--Insert the RTPVideoApplet with the size of 320x340
10: The video feed will be 320 x 240, but we want the
11: applet to also have room for the player controls -->
12:
13:  <applet code=RTPVideoApplet.class width=320 height=340>
14:  <!--value="rtp://<Client Machine IP>:<Server's
15:  Port>/video/1" -->
16:  <param name=file value="rtp:
17:  //129.235.68.102:40500/video/1">
18:  </applet>
19:
20:  </body>
21:  </html>
```

FIGURE 9.8
Applet window.



9.5 Summary

In this chapter, we covered three main concepts of video programming in Java. We explained how to acquire and display video from a video device. We provided the details on how to perform image processing on video frames required by various applications. Finally, we showed how to transmit video results to other machines across the network. We focused mainly on JMF, which has added some very powerful features to the Java programming language. With the popularity and usefulness of streaming video media, the previous examples will be very valuable for years to come.

APPENDIX A

Component Software Setup

We present a step-by-step explanation for installing, configuring, and compiling the software components required for building the application software developed in this book. Software components include Microsoft's DirectX and Visual Studio .NET, software drivers for the FireWire iBot cameras from Orange Micro Inc., and the XCam2 USB cameras from X10. ■

A.1 Visual Studio .NET Installation

The first software component to be installed is Microsoft's Visual Studio .NET. Installing this development environment should not present any challenges because the associated wizard will guide you through the various steps. You should also note that the earlier version of Visual Studio such as C++ 6.0 will work as well as the latest .NET version.

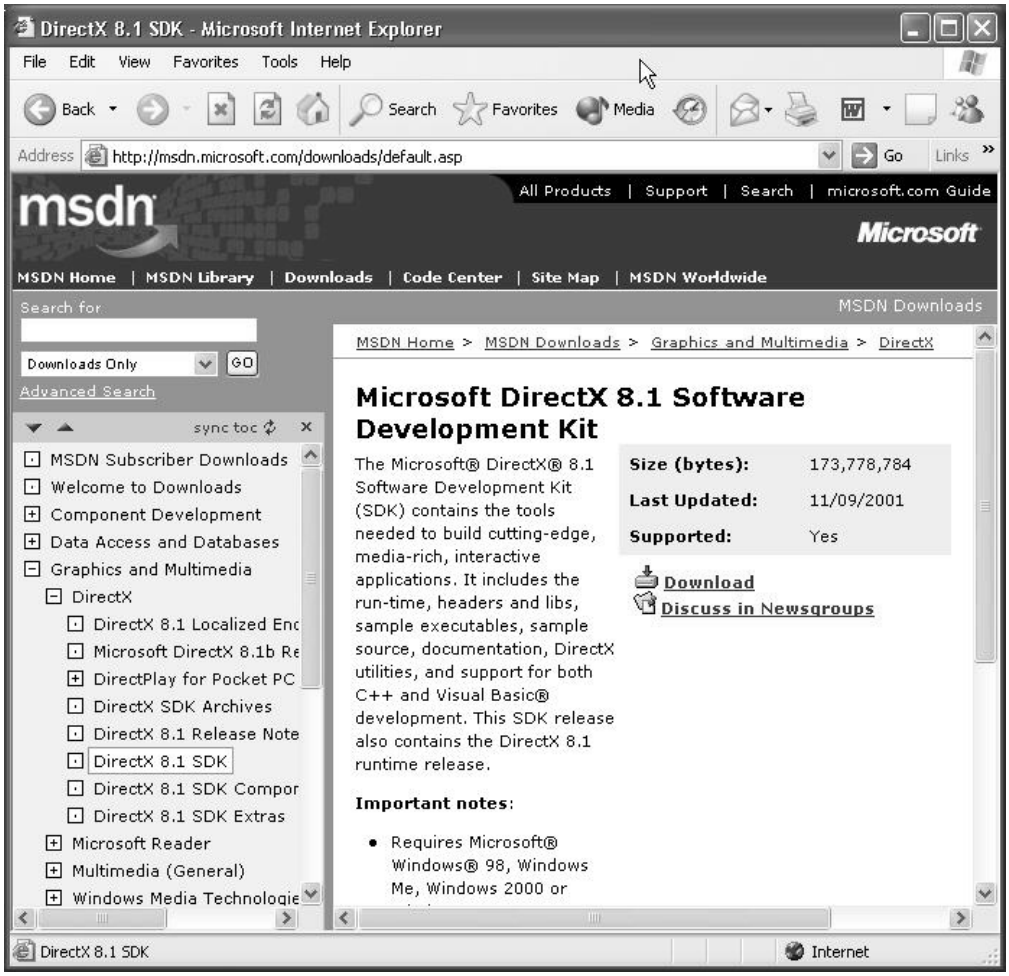
A.2 DirectX 8.1 Software Development Kit Installation

The DirectX Software Development Kit (SDK) provides the software tools for building multimedia applications and supports the latest

release of Microsoft's Visual Studio .NET. The following provides the steps from downloading to compiling successfully the DirectX SDK. We assume that installation of Visual Studio .NET has preceded this step.

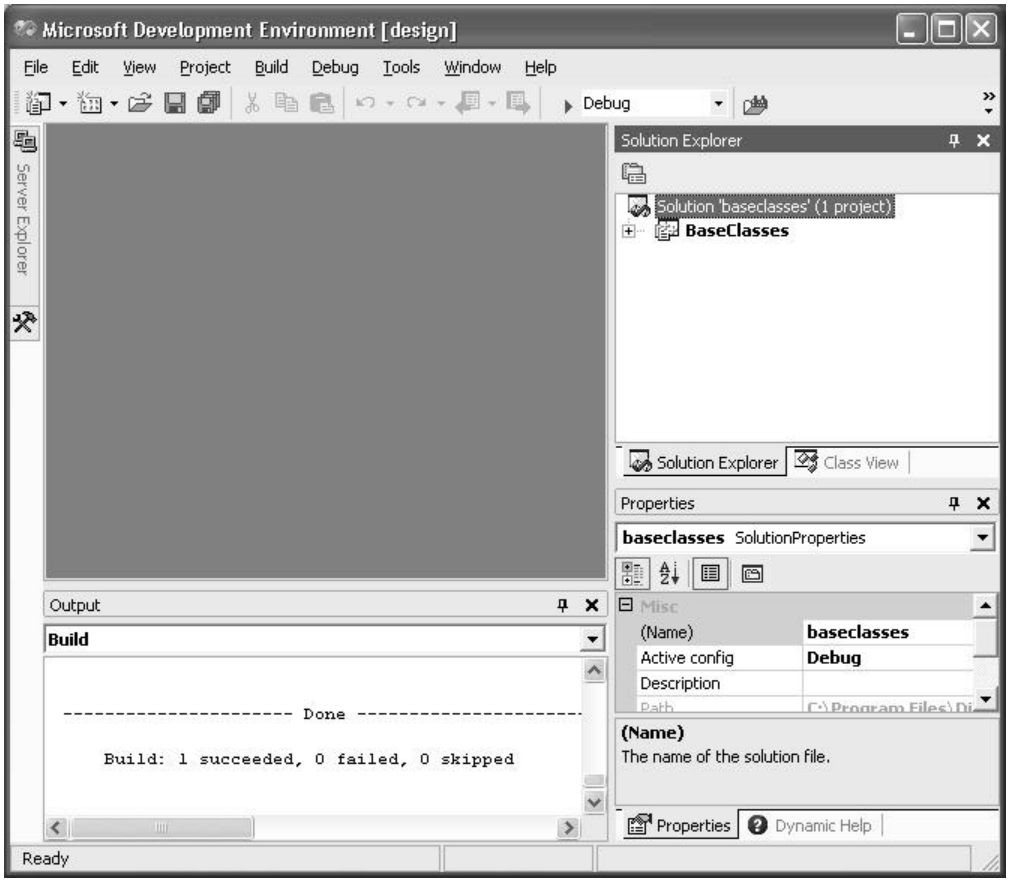
- Download the SDK from Microsoft's software download site located at the following URL: `http://msdn.microsoft.com/downloads/default.asp` (Figure A.1). Select `Graphics and Multimedia->DirectX->DirectX 8.1 SDK`.

FIGURE A.1 Downloads page of the DirectX 8.1 SDK.



- Doubleclick the setup file to install the DirectX SDK in your preferred directory. In our case the directory is: `C:\Program Files\DirectX8.1`.
- Using the .NET development environment, open the solution file *baseclasses.sln* for the `BaseClasses` project located in `C:\Program Files\DirectX8.1\samples\Multimedia\DirectShow\BaseClasses` and build the `baseClasses` library (Figure A.2).

FIGURE A.2 Building the `BaseClasses` library occurs before its use in a custom application.



A.3 Configuring the Application Environment

An application must link successfully to the appropriate libraries. Below we show step-by-step how to set the application environment for the `CDServer` project.

- Open the application in the .NET environment (Figure A.3).
- Right click on the highlighted project name (CDServer) and select `Properties`. A dialog window will appear as shown in Figure A.4.

FIGURE A.3 The `CDServer` project opened in visual studio .NET.

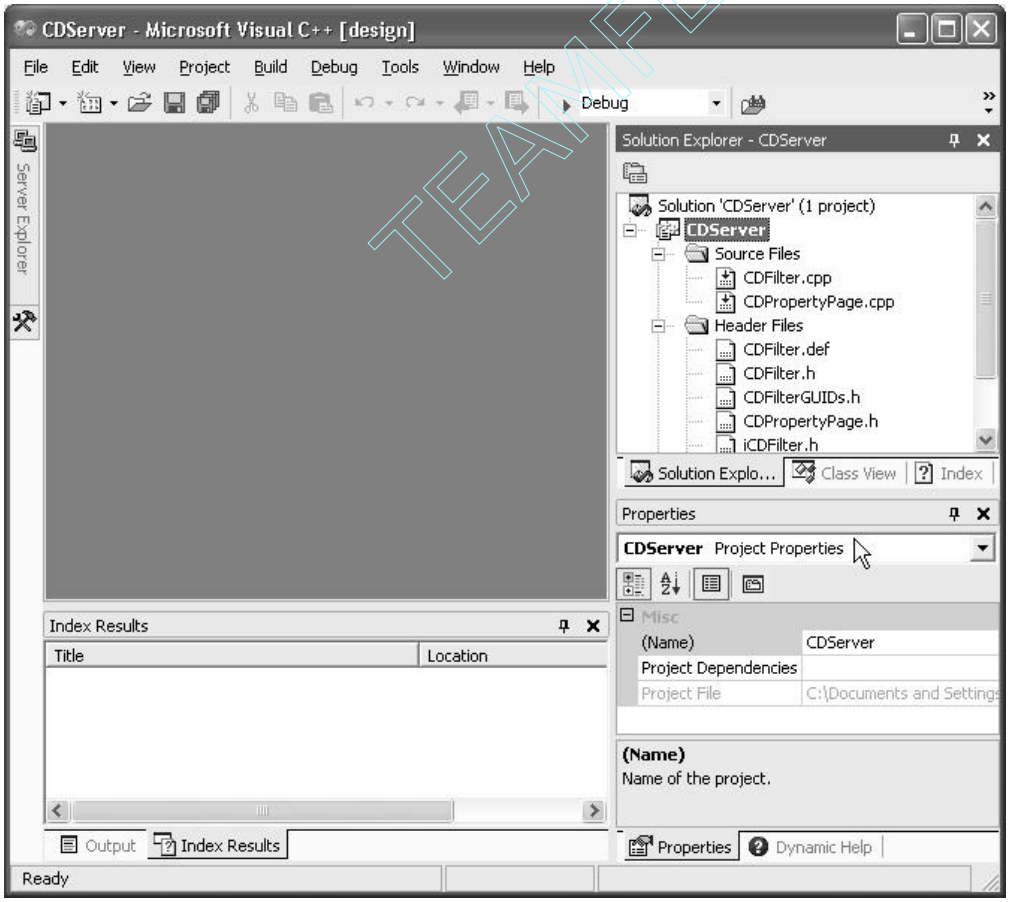
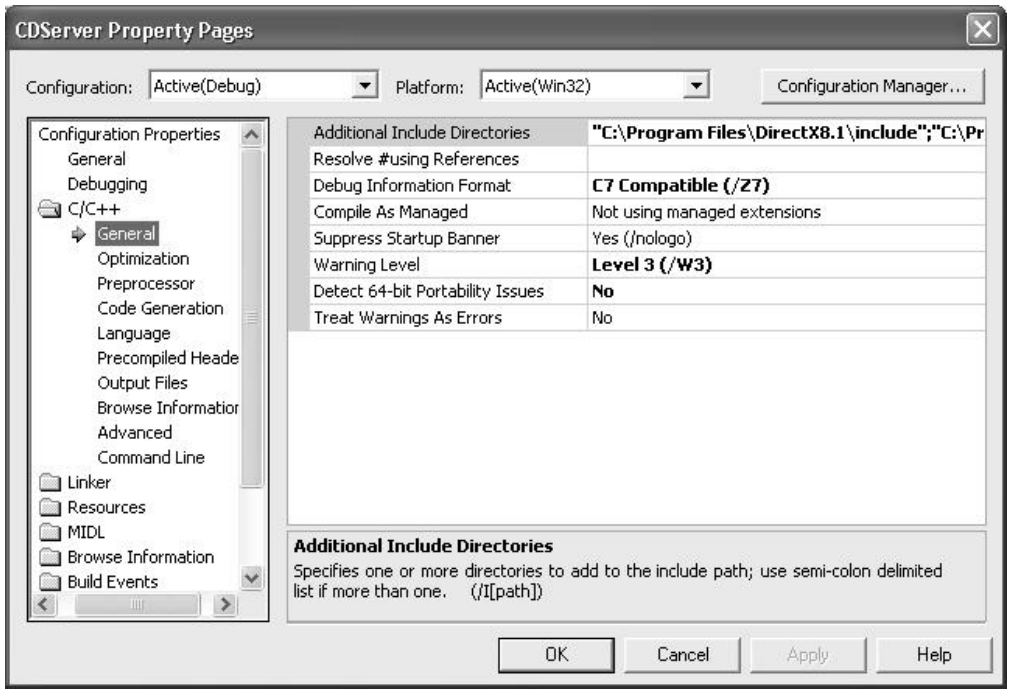


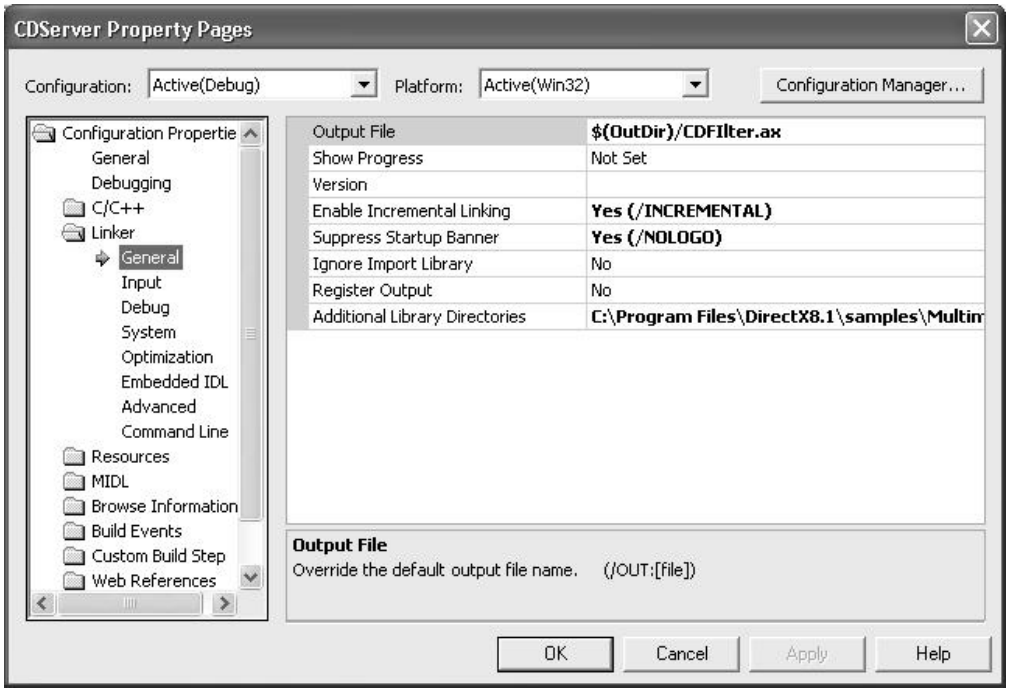
FIGURE A.4 Setting the appropriate Include directories.



- Select `C-C++ ->General` and in the `Additional Include Directories` slot include the following paths (Figure A.4):
  1. `C:\Program Files\DirectX8.1\include` and
  2. `C:\Program Files\DirectX8.1\samples\Multimedia \DirectShow\BaseClasses`

  (assuming you have installed DirectX8.1 under Program Files).
- Next, select `Linker->General` and in the `Additional Library Directories` slot include the following paths (Figure A.5):
  1. `C:\Program Files\DirectX8.1\lib` and
  2. `C:\Program Files\DirectX8.1\samples\Multimedia \DirectShow\BaseClasses\Debug`
- Finally, select `Linker->input`, and in the `Additional Dependencies` slot include the libraries to which your application software should link (Figure A.5). You should *definitely* include the `strmbasd.lib` library file.

FIGURE A.5 Setting the appropriate library directories.



A.4 Camera Drivers

The software developed in this book has been tested using an iBot FireWire camera from Orange Micro Inc. and an XCam2 USB camera from X10. These two camera drivers can be downloaded from their respective Web pages given below:

1. `http://www.orangemicro.com/softwarupdates.html`
2. `http://x10.com/xrv/software5.htm`

APPENDIX B

Accessing a Live Device in JMStudio

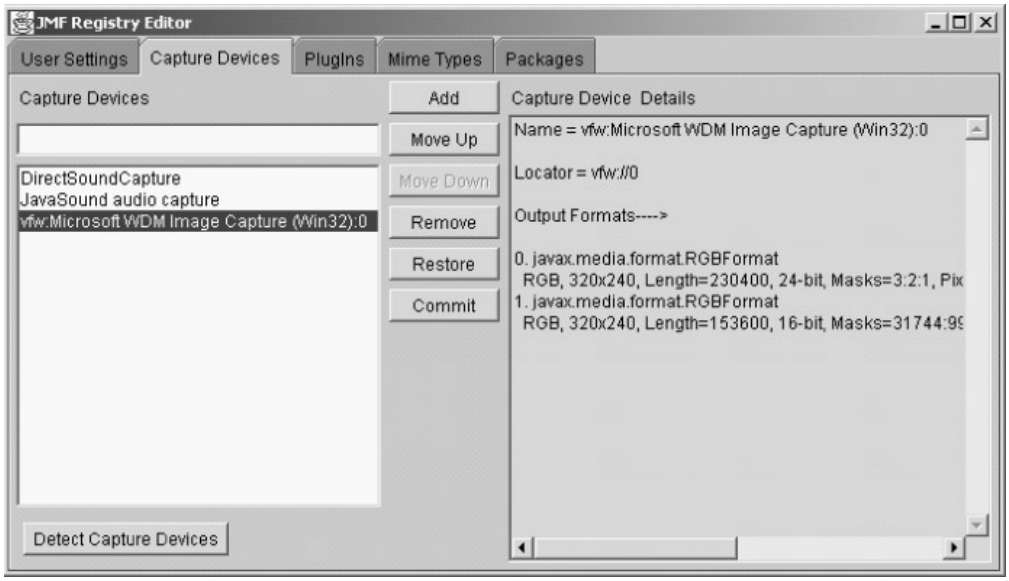
We present the steps for determining the locator for a live device by using JMStudio's Registry Editor. ■

B.1 Live Devices in JMStudio

In order to run many of our Java video applications we need to access a live video device. The following steps show how to determine the locator for a live device by using JMStudio's registry editor.

1. `Start->Programs->Java Media Framework 2.1.1->JMStudio`.
2. Select `File | Preferences`.
3. Click on the `Capture Devices` tab of the JMF Registry Editor window (Figure B.1).
4. Look for your device in the list on the left. Select it, and look at the list on the right.
5. The information of importance is the value of the locator. This string is used by the application's `MediaLocator` for locating the live device.

FIGURE B.1 The JMF registry editor.



Chapman, D. *Teach Yourself Visual C++ .NET in 21 Days*. SAMS Publishing, Indianapolis, Indiana, 2002.

Cheng, Y. Mean Shift, Mode Seeking, and Clustering. In *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 1995, vol. 17, pp. 790–799.

Comaniciu, D., Ramesh, V., and Meer, P. Real-Time Tracking of Non-Rigid Objects Using Mean Shift. In *Proceedings, Conference on Computer Vision and Pattern Recognition,* 2000, vol. 2, pp. 142–149.

Deitel, H. M., and Deitel, P. J. *Java How To Program* (4th Edition). Prentice Hall, Upper Saddle River, New Jersey, 2002.

Gonzalez, R. C., and Woods, R. E. *Digital Image Processing*. Addison-Wesley, Reading, Massachusetts, 1992.

Linetsky, M. *Programming Microsoft DirectShow*. Wordware Publishing, Inc., Plano, Texas, 2002.

Otsu, N. A Threshold Selection Method from Gray-Level Histograms. In *IEEE Transactions on Systems, Man, and Cybernetics,* 1979, vol. 9, pp. 62–66.

Templeman, J., and Olsen, A. *Microsoft Visual C++ .NET Step by Step*. Microsoft Press, Redmond, Washington, 2002.

Troelsen, A. *COM and .NET Interoperability*. Apress, Berkeley, California, 2002.

The letters *fig* following a page number indicate a figure.
The word *table* following a page number indicates a table.
The word *listing* following a page number indicates a listing of code.

About the CD-ROM

The accompanying CD-ROM contains the source code that illustrates the ideas in this book. All of the C++ source code was written using Microsoft Visual Studio.

Directory Structure

Software/Chapter2/AVIPlayer/ This folder contains a Direct Show application for playing AVI movie files.

Software/Chapter3/ This folder contains DirectX Graph Edit Utility files for displaying live video using the camera attached to the computer. These files can be opened with the Graph Edit Utility.

Software/Chapter4/CDServer/ This folder contains a Direct Show filter for Change Detection.

Software/Chapter5/CDApp/ This folder contains a Direct Show application for running the Change Detection filter of Chapter 4.

Software/Chapter6/PanTiltFilter/ This folder contains a Direct Show filter for controlling the Directed Perception's Pan/Tilt unit.

Software/Chapter7/PTTrackingFilter/ This folder contains a Direct Show filter for tracking moving objects in the camera's field of view. It also allows the Pan/Tilt unit to follow the object.

Software/Chapter8/PTTrackingApp/ This folder contains a Direct Show application for running the PTTrackingFilter of Chapter 7.

Software/Chapter9/LiveViewer/ This folder contains the Java source code for the Live Viewer application. VideoFilter/ This folder contains the Java source code for the video processing filter application. WebVideo/Reception/ This folder contains the source code for the Web Video client. Transmission/ This folder contains the source code for the Web Video server.

About the Authors

Ioannis Pavlidis holds a Ph.D. and an M.S. in computer science from the University of Minnesota, a M.S. in robotics from the Imperial College of the University of London, and a B.S. in electrical engineering from the Democritus University in Greece. He is currently an associate professor in the computer science department at the University of Houston.

Vassilios Morellas holds a Ph.D. in mechanical engineering from the University of Minnesota, an M.S. in mechanical engineering from Columbia University, and a Diploma in mechanical engineering from the National Technical University of Athens, Greece. Currently, he is a senior research scientist at Honeywell Laboratories, developing high-end automated surveillance systems.

Pete Roeber holds a B.S. in computer science from the University of Minnesota. He is currently a software engineer at Vital Images, Inc., in Plymouth, Minnesota, where he works on the development of 3D medical imaging software.