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`.

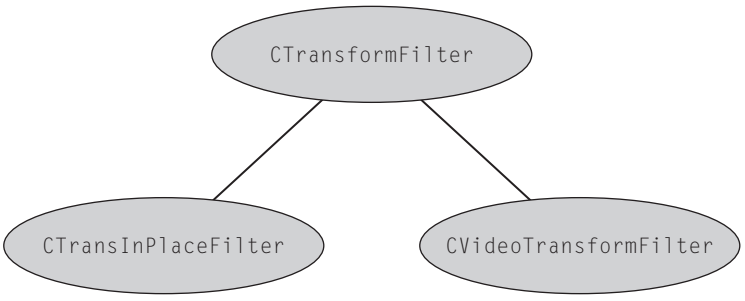


FIGURE 4.1
The inheritance tree for the transform filter family.

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.

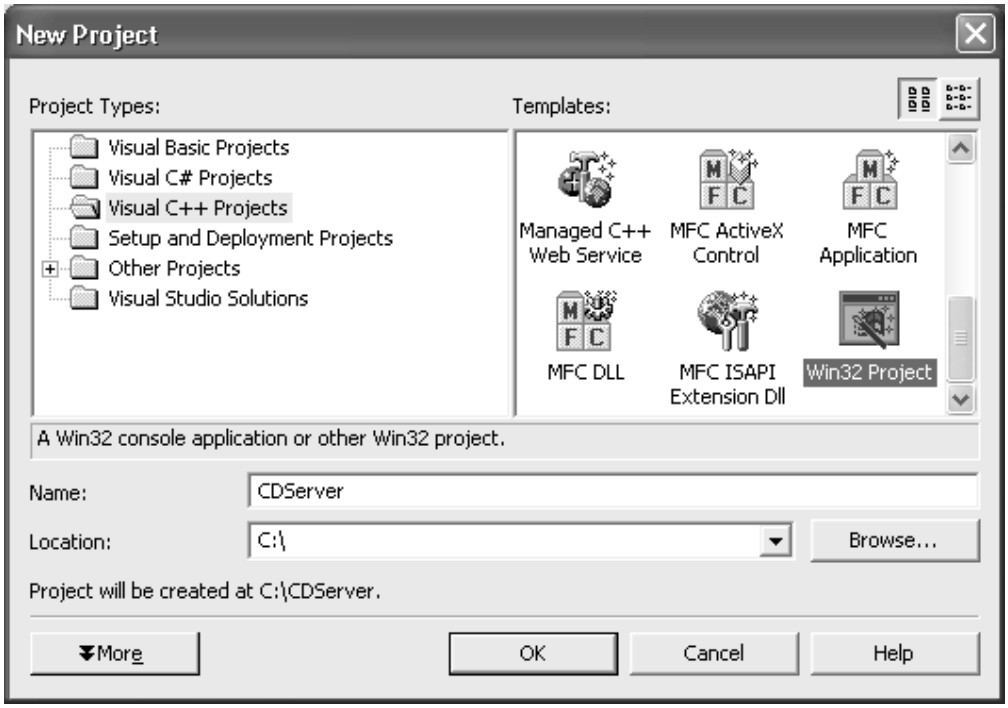
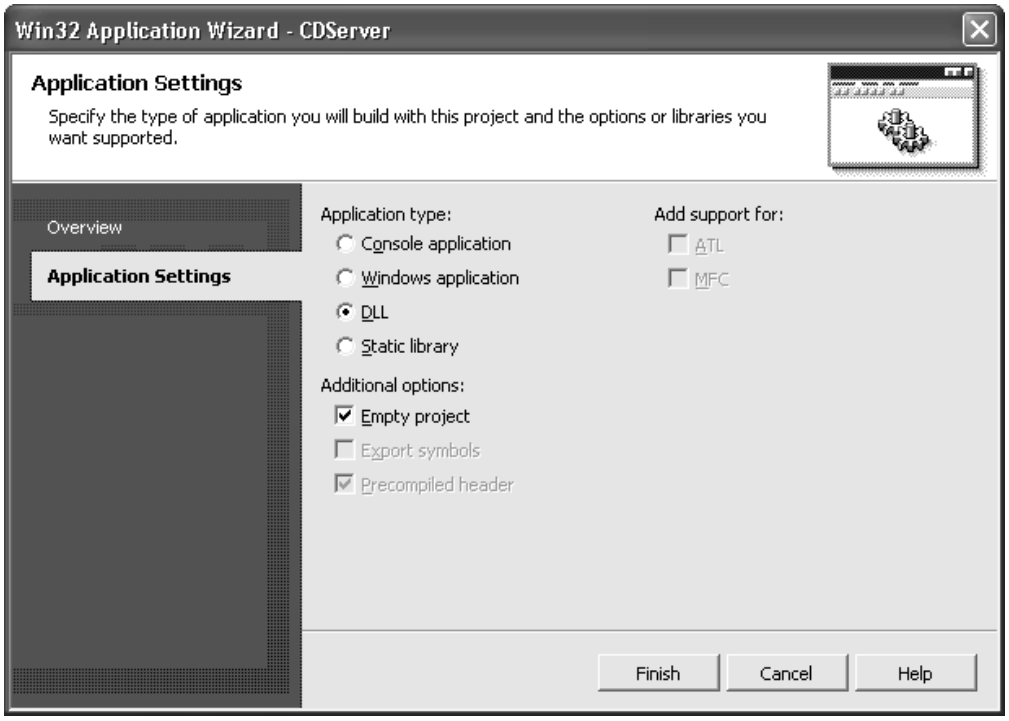


FIGURE 4.3 The wizard window for application definition.



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.

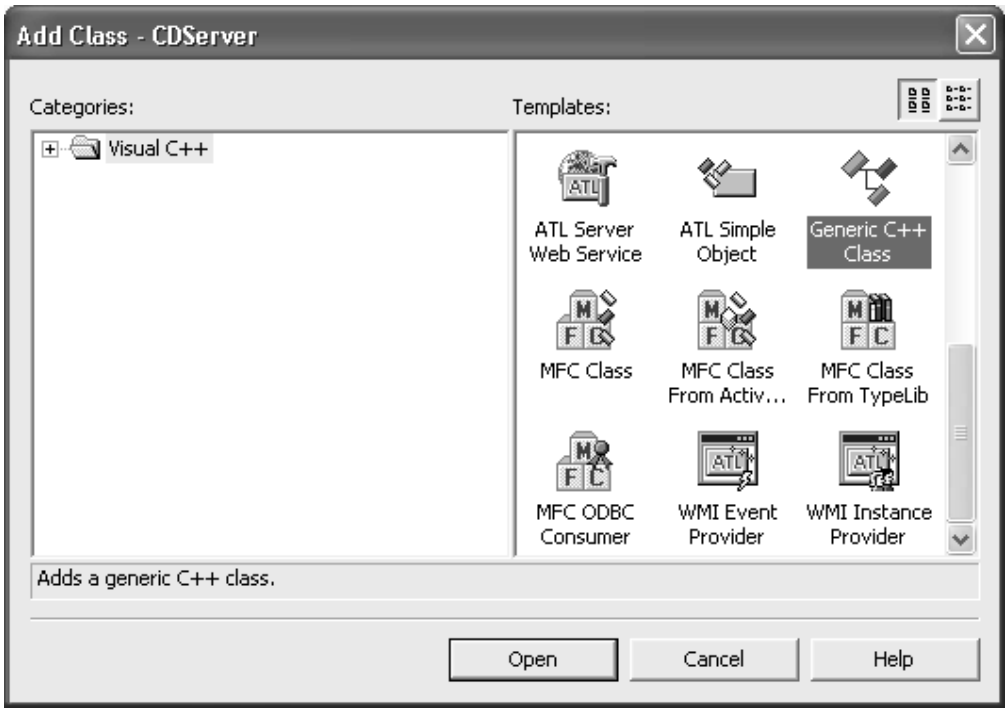
4.2 Define and Instantiate Our Filter's Class

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

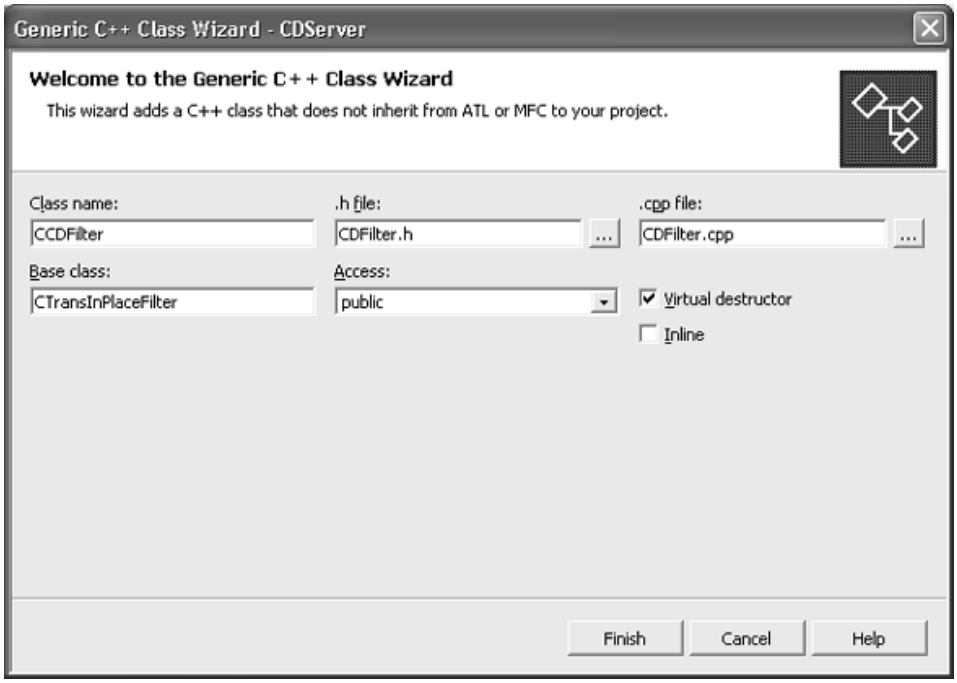
FIGURE 4.4 The class wizard window.



`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.

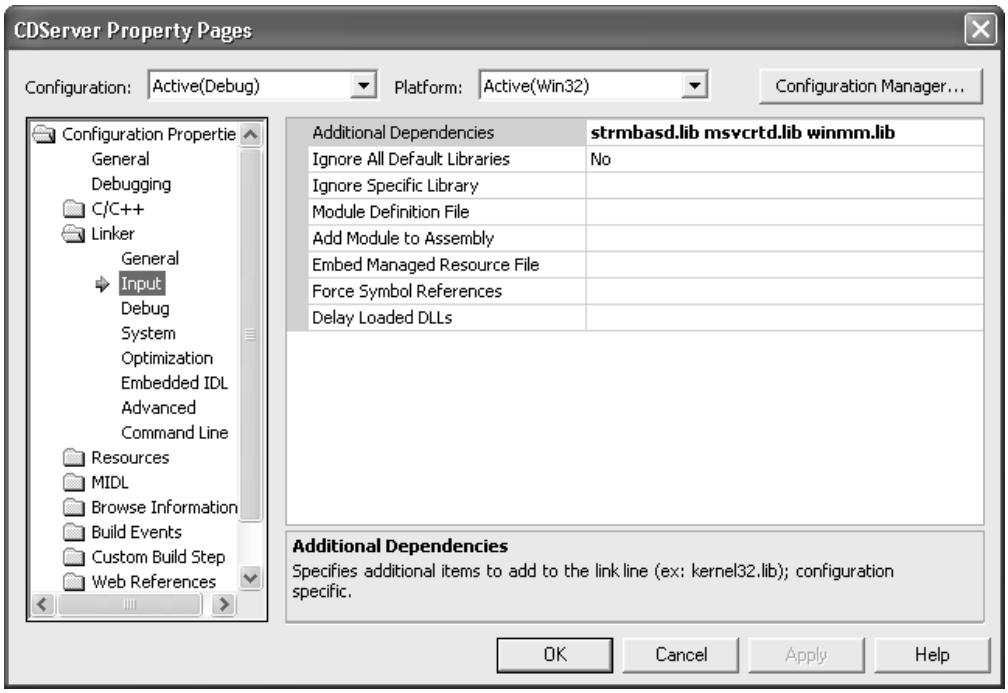


We switch to the `Solution Explorer` pane and open the *CDFilter.cpp* file. There we modify the definition of the class constructor as it appears in Listing 4.2.1.1. The initialization code will be added piece by piece as we enter the various member variables. For each variable type the Visual C++ compiler 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
```

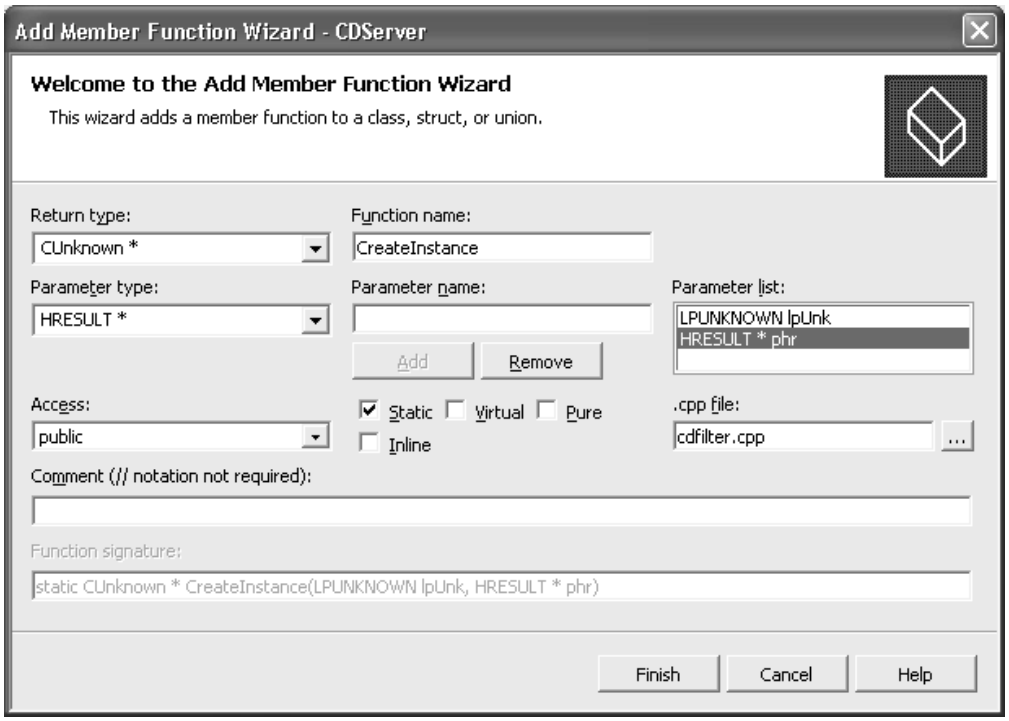
The *strmbasd.lib* library supports the DirectShow base classes. The *msvcrtd.lib* is the import library for dynamically linking the

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

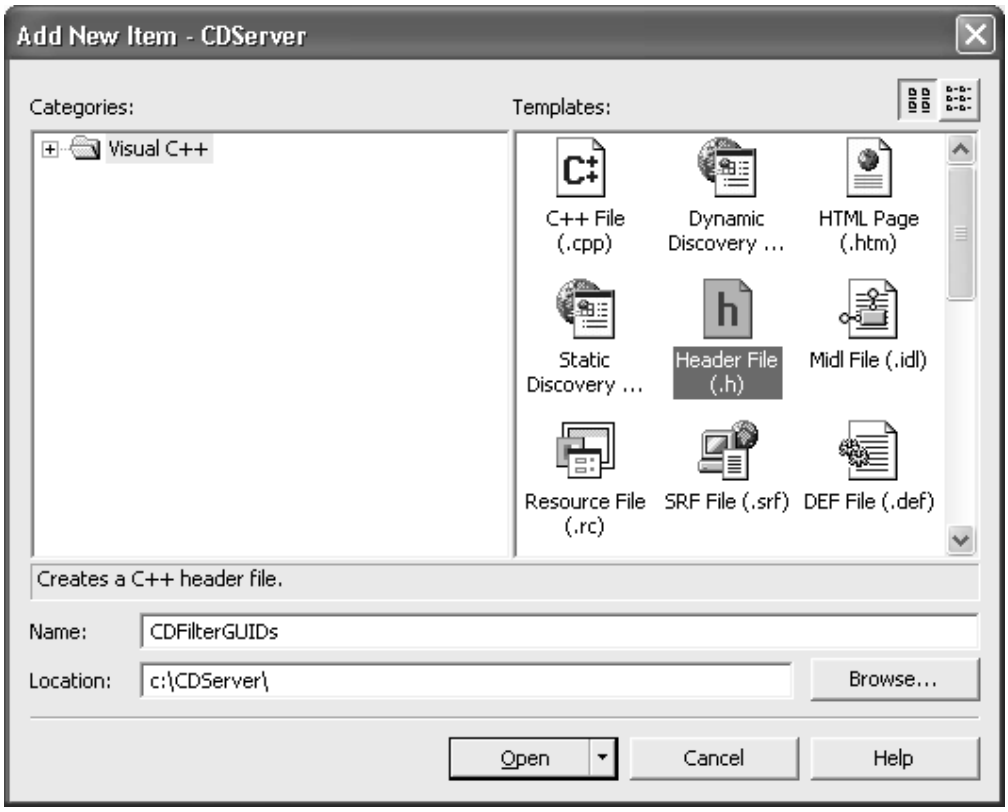
LISTING 4.2.2.2 A global array of objects to be communicated to the class factory.

```
 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.

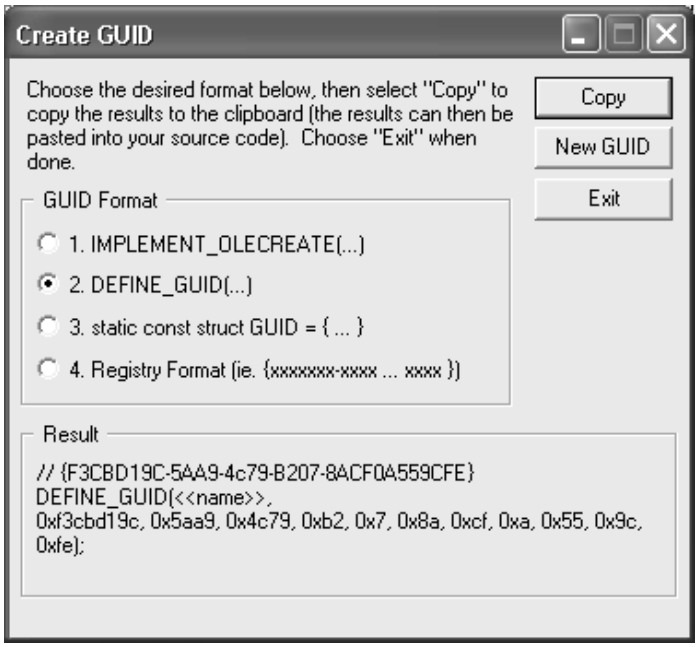


FIGURE 4.9
The dialog window of the `GUIDGEN.EXE` utility.

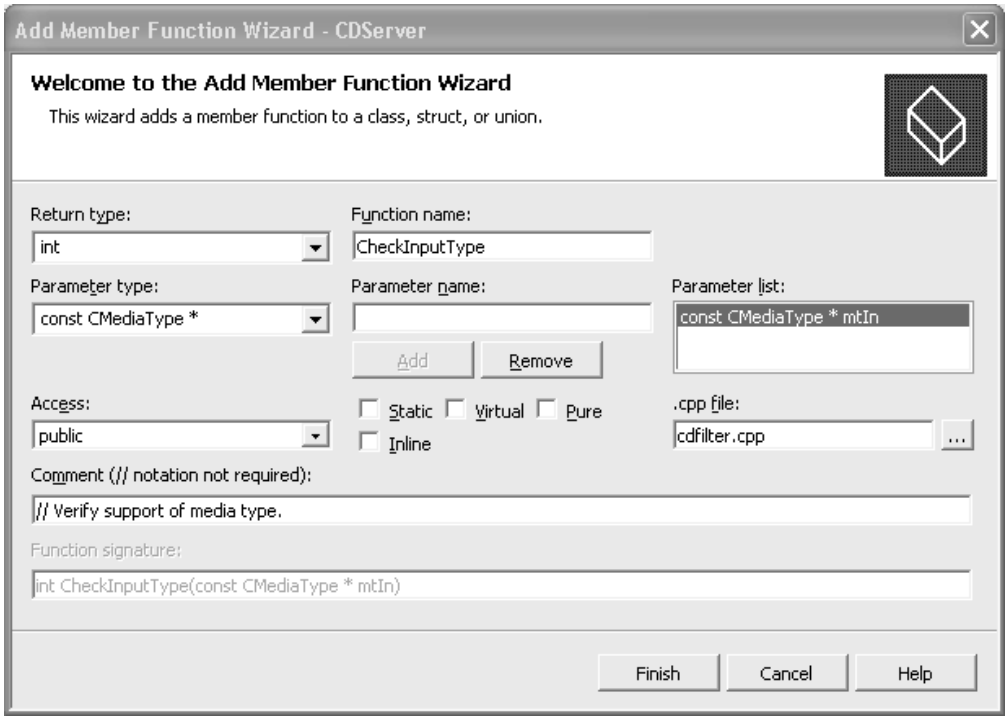
4.3 Override the Appropriate Base Class Member Functions

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:  }
```

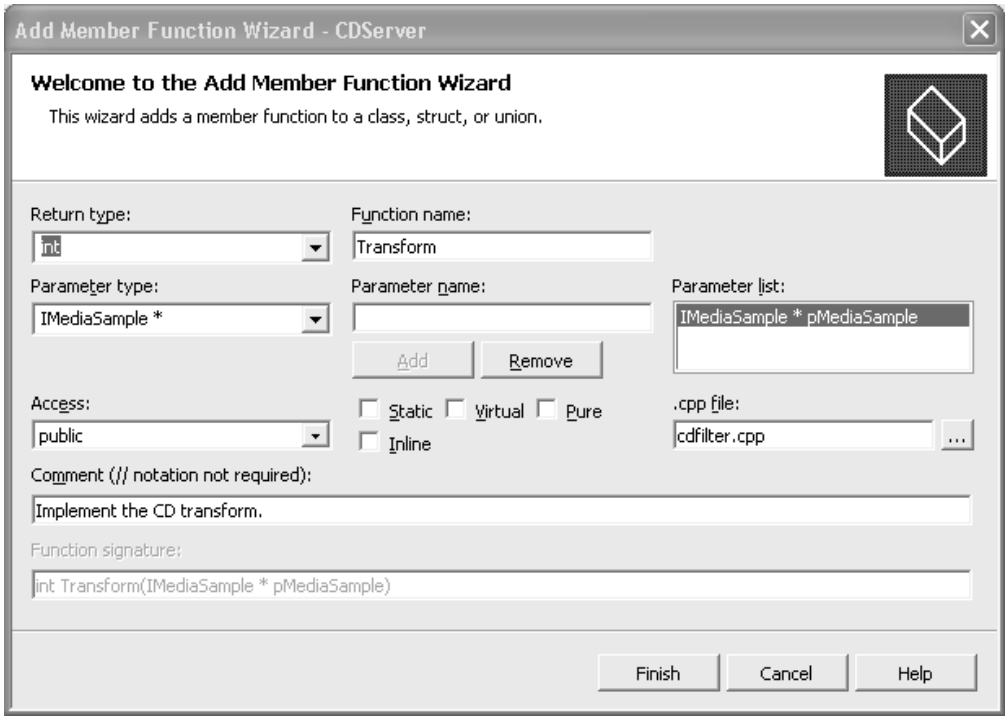
LISTING 4.3.1.2 The `CheckInputSubType` helper function definition.

```
 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.

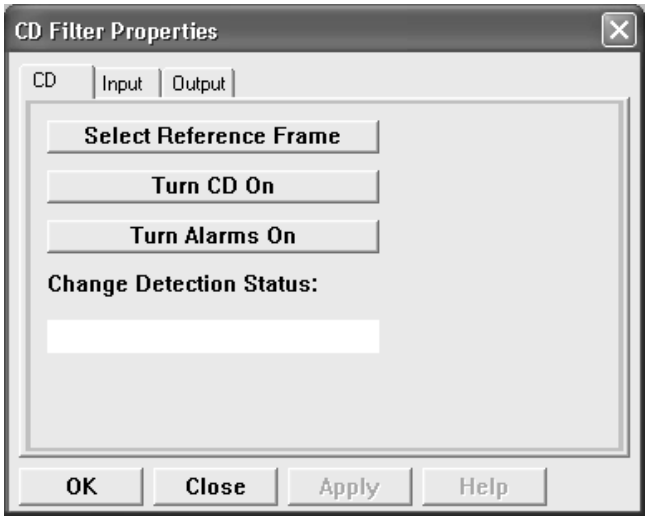


FIGURE 4.12
The property page of the CD filter.

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:  ...
```

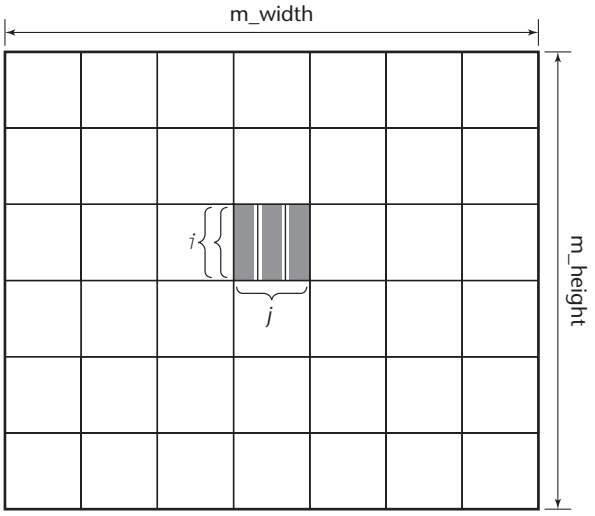


FIGURE 4.13
Organization of the bitmap frame. Each pixel (*i*, *j*) packs three byte numbers that represent the constituent primary colors for the pixel (red, green, and blue).

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:

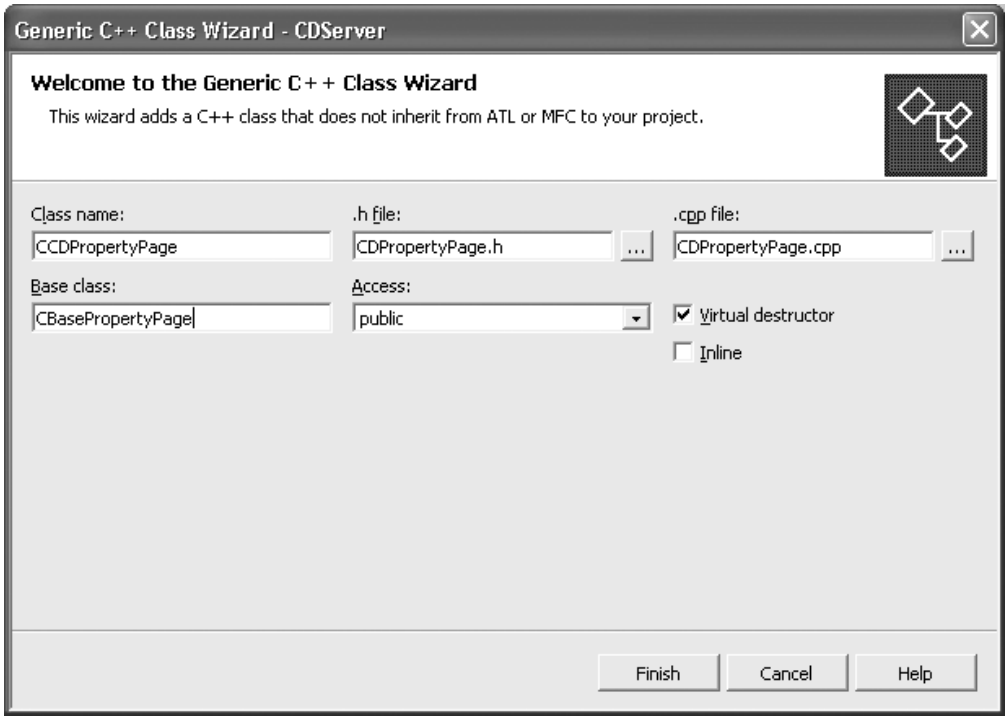
```
CCDPropertyPage(LPUNKNOWN lpUnk, HRESULT *phr);
```

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.