# Video Tracking System Based on DirectShow

Contents



## 1. Objective

An Object is moving in a video,we want to track it by computer,the video can be a video file,such as avi,wmv,rm,etc. or can be from a live video input such as a camera.Below is some images from videos.Our Objective is to track the light and the hand in the videos below.

A light is moving in the video

A hand holding a pen is moving in the video

## 2. Why Use DirectShow?

The Challenge of Multimedia

Working with multimedia presents several major challenges:

- ✧ Multimedia streams contain large amounts of data, which must be processed very quickly.
- ✧ Audio and video must be synchronized so that it starts and stops at the same time, and plays at the same rate.
- ✧ Data can come from many sources, including local files, computer networks, television broadcasts, and video cameras.
- ✧ Data comes in a variety of formats, such as Audio-Video Interleaved (AVI), Advanced Streaming Format (ASF), Motion Picture Experts Group (MPEG), and

Digital Video (DV).

- ✧ The programmer does not know in advance what hardware devices will be present on the end-user's system.

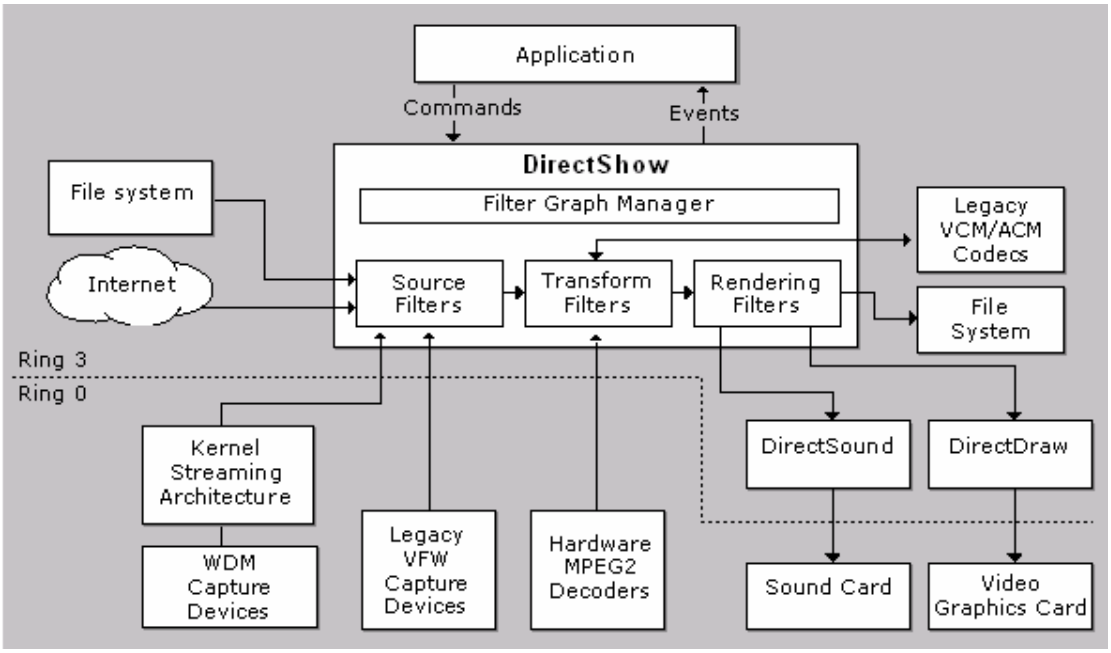
## The DirectShow Solution

DirectShow is designed to address each of these challenges.

- ✧ Its main design goal is to simplify the task of creating digital media applications on the Windows® platform, by isolating applications from the complexities of data transports, hardware differences, and synchronization.
- ✧ To achieve the throughput needed to stream video and audio, DirectShow uses DirectDraw® and DirectSound® whenever possible. These technologies render data efficiently to the user's sound and graphics cards.
- ✧ DirectShow synchronizes playback by encapsulating media data in time-stamped samples.
- ✧ To handle the variety of sources, formats, and hardware devices that are possible, DirectShow uses a modular architecture, in which the application mixes and matches different software components called *filters*.

DirectShow provides filters that support capture and tuning devices based on the Windows Driver Model (WDM), as well as filters that support legacy Video for Windows (VfW) capture cards, and codecs written for the Audio Compression Manager (ACM) and Video Compression Manager (VCM) interfaces.

The following diagram shows the relationship between an application, the DirectShow components, and some of the hardware and software components that DirectShow supports.



As illustrated here, DirectShow filters communicate with, and control, a wide variety of devices, including the local file system, TV tuner and video capture cards, VfW codecs, the video display (through DirectDraw or GDI), and the sound card (through DirectSound). Thus, DirectShow insulates the application from many of the complexities of these devices. DirectShow also provides native compression and decompression filters for certain file formats.

# 3. How to Use DirechShow For Video Tracking?

## 1) Preparation

If you want to write DirectShow programmes by yourself.You must follow the steps below to setup your development environment.. I just tell DX9.0

a) Install Visual C++ 6.0 or higher version

b) Install DirectX SDK 9.0,you can download it from Microsoft's website

c) Setup the Include Path and Lib Path of VC

In VC,Select menu tools→options,then select option group "directories",Select "Include files",Add the following path :" F:\DXSDK\Include" and "F:\DXSDK\Samples\C++\DirectShow\BaseClasses".Select "library files",and add the path below:" F:\DXSDK\Lib"

d) Open "F:\DXSDK\Samples\C++\DirectShow\BaseClasses\ baseclasses.dsw",compile it ,and copy the files "STRMBASE.lib"and "STRMBASD.lib" to F:\DXSDK\Lib

Now,we setup your development environment successfully.

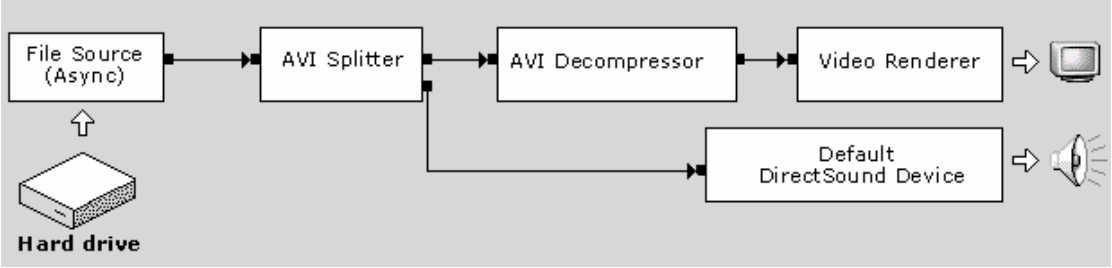
## 2) Filters and Filter Graphs

The building block of DirectShow is a software component called a *filter*. A filter is a software component that performs some operation on a multimedia stream. For example, DirectShow filters can

- ✧ read files
- ✧ get video from a video capture device
- ✧ decode various stream formats, such as MPEG-1,2,4 video ,RM video
- ✧ pass data to the graphics or sound card

Filters receive input and produce output. For example, if a filter decodes MPEG-1 video, the input is the MPEG-encoded stream and the output is a series of uncompressed video frames.

In DirectShow, an application performs any task by connecting chains of filters together, so that the output from one filter becomes the input for another. A set of connected filters is called a *filter graph*. For example, the following diagram shows a filter graph for playing an AVI file.



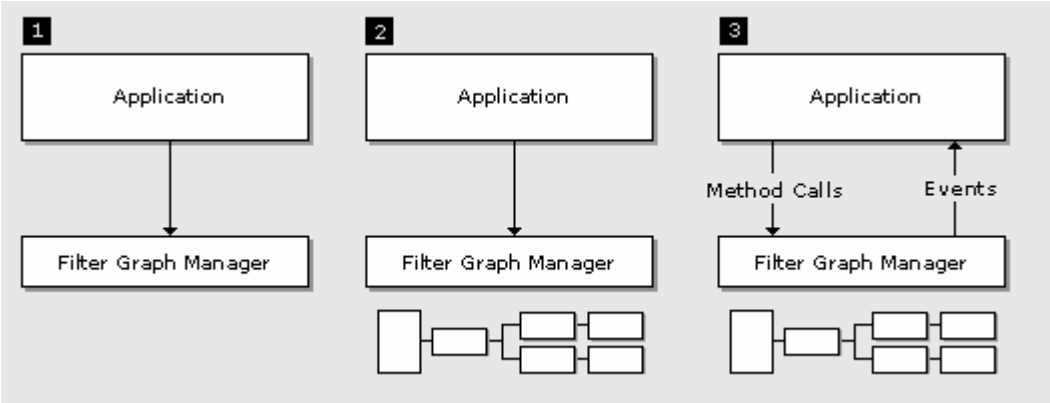

The File Source filter reads the AVI file from the hard disk. The AVI Splitter filter parses the file into two streams, a compressed video stream and an audio stream. The AVI Decompressor filter decodes the video frames. The Video Renderer filter draws the frames to the display, using DirectDraw or GDI. The Default DirectSound Device filter plays the audio stream, using DirectSound.

The application does not have to manage all of this data flow. Instead, the filters are controlled by a high-level component called the *Filter Graph Manager*. The application makes high-level API calls such as "Run" (to move data through the graph) or "Stop" (to stop the flow of data). If you require more control over the stream operations, you can access the filters directly through COM interfaces. The Filter Graph Manager also passes event notifications to the application.

The Filter Graph Manager serves another purpose as well: It provides methods for the application to build the filter graph, by connecting the filters together. (DirectShow also provides various helper objects that simplify this process. These are thoroughly described in the documentation.)

# 3) Writing a DirectShow Application

In broad terms, there are three tasks that any DirectShow application must perform. These are illustrated in the following diagram:




i. The application creates an instance of the Filter Graph Manager.
ii. The application uses the Filter Graph Manager to build a filter graph. The exact set of filters in the graph will depend on the application.
iii. The application uses the Filter Graph Manager to control the filter graph and stream data through the filters. Throughout this process, the application will also respond to events from the Filter Graph Manager.
iv. When processing is completed, the application releases the Filter Graph Manager and all of

the filters.

DirectShow is based on COM; the Filter Graph Manager and the filters are all COM objects. You should have a general understanding of COM client programming before you begin programming DirectShow.

## How To Play a File

A DirectShow application always performs the same basic steps:

i. Creates an instance of the Filter Graph Manager.
ii. Uses the Filter Graph Manager to build a filter graph.
iii. Runs the graph, which causes data to move through the filters.

- Include <dshow.h> and these lib files in your application.

strmbase.lib strmiids.lib amstrmid.lib winmm.lib

- Start by calling **CoInitialize** to initialize the COM library:

```
HRESULT hr = CoInitialize(NULL);
if (FAILED(hr))
{
    // Add error-handling code here. (Omitted for clarity.)
}
```

To keep things simple, this example ignores the return value, but you should always check the **HRESULT** value from any method call.

- Next, call CoCreateInstance to create the Filter Graph Manager:

```
IGraphBuilder *pGraph;
HRESULT hr = CoCreateInstance(CLSID_FilterGraph, NULL,
    CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);
```

As shown, the class identifier (CLSID) is CLSID_FilterGraph. The Filter Graph Manager is provided by an in-process DLL, so the execution context is CLSCTX_INPROC_SERVER. DirectShow supports the free-threading model, so you can also call **CoInitializeEx** with the COINIT_MULTITHREADED flag.

The call to **CoCreateInstance** returns the **IGraphBuilder** interface, which mostly contains methods for building the filter graph.

- Two other interfaces are needed for this example:

**IMediaControl** controls streaming. It contains methods for stopping and starting the graph.

**IMediaEvent** has methods for getting events from the Filter Graph Manager. In this example, the interface is used to wait for playback to complete.

Both of these interfaces are exposed by the Filter Graph Manager. Use the returned **IGraphBuilder** pointer to query for them:

```
IMediaControl *pControl;
IMediaEvent   *pEvent;
hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
hr = pGraph->QueryInterface(IID_IMediaEvent, (void **)&pEvent);
```

- Now you can build the filter graph. For file playback, this is done by a single method call:

```
hr = pGraph->RenderFile(L"C:\\Example.avi", NULL);
```

The **IGraphBuilder::RenderFile** method builds a filter graph that can play the specified file. The

first parameter is the file name, represented as a wide character (2-byte) string. The second parameter is reserved and must equal NULL.

This method can fail if the specified file does not exist, or the file format is not recognized. Assuming that the method succeeds, however, the filter graph is now ready for playback.

- To run the graph, call the IMediaControl::Run method:

```
hr = pControl->Run();
```

- When the filter graph runs, data moves through the filters and is rendered as video and audio. Playback occurs on a separate thread. You can wait for playback to complete by calling the **IMediaEvent::WaitForCompletion** method:

```
long evCode = 0;
pEvent->WaitForCompletion(INFINITE, &evCode);
```

This method blocks until the file is done playing, or until the specified time-out interval elapses. The value INFINITE means the application blocks indefinitely until the file is done playing.

- When the application is finished, release the interface pointers and close the COM library:

```
pControl->Release();
pEvent->Release();
pGraph->Release();
CoUninitialize();
```

## 4) About Video Captureing

## 5) How to Use a Filter

Before using the filter,You must know the GUID of the filter.you can use CreateInstance to get a pointer to the filter object,then use IGraphBuilder::AddFiter() method to add the filter to your filter graph,then use IGraphBuilder::RenderFile() to intelligently create a filter graph ,or you can create the graph by hand. Following is a sample:

```
#include <atlbase.h>
DEFINE_GUID(CLSID_EZrgb24,
0x8b498501, 0x1218, 0x11cf, 0xad, 0xc4, 0x0, 0xa0, 0xd1, 0x0, 0x4, 0x1b);

CoCreateInstance(CLSID_EZrgb24, 0, CLSCTX_INPROC_SERVER,
        IID_IBaseFilter, reinterpret_cast<void**>(&pFilter));
pGB->AddFilter(pFilter, L"RGB Filter");
pFilter->Release();
pGB->RenderFile(wFile, NULL);
```

## 6) How to write a filter

There a three kinds of filters:

Source filter

Transform filter

Renderer filter

Source Filters 主要负责取得数据，数据源可以是文件、因特网、或者计算机里的采集卡、数字摄像机等，然后将数据往下传输；

Transform Fitlers 主要负责数据的格式转换、传输；
Rendering Filtes 主要负责数据的最终去向，我们可以将数据送给声卡、显卡进行多媒体的演示，也可以输出到文件进行存储。
值得注意的是，三个部分并不是都只有一个 Filter 去完成功能。恰恰相反，每个部分往往是有几个 Fitler 协同工作的。比如，Transform Filters 可能包含了一个 Mpeg 的解码 Filter、以及视频色彩空间的转换 Filter、音频采样频率转换 Filter 等等。除了系统提供的大量 Filter 外，我们可以定制自己的 Filter，以完成我们需要的功能。

A video tracking filter is a Transform filter,we can use CTransformFilter as its base class.such as:

```
class CMyFilter : public CTransformFilter
{
private:
    /* Declare variables and methods that are specific to your filter.
public:
    /* Override various methods in CTransformFilter */
};
```

A filter is a COM object,we must Export some functions,we can create a def file.
Include a definition (.def) file that exports the DLL functions. The following is an example of a definition file. It assumes that the output file is named MyFilter.dll.

```
LIBRARY MYFILTER.DLL
EXPORTS
    DllMain                 PRIVATE
    DllGetClassObject       PRIVATE
    DllCanUnloadNow         PRIVATE
    DllRegisterServer       PRIVATE
    DllUnregisterServer PRIVATE
```

Declare the DLL entry point in your source code, as follows:

```
extern "C" BOOL WINAPI DllEntryPoint(HINSTANCE, ULONG, LPVOID);
BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, LPVOID lpReserved)
{
    return DllEntryPoint((HINSTANCE)(hModule), dwReason, lpReserved);
}
```



## Step 1. Choose a Base Class

Assuming that you decide to write a filter and not a DMO, the first step is choosing which base class to use. The following classes are appropriate for transform filters:

- **CTransformFilter** is designed for transform filters that use separate input and output buffers. This kind of filter is sometimes called a *copy-transform* filter. When a copy-transform filter receives an input sample, it writes new data to an output sample and delivers the output sample to the next filter.
- **CTransInPlaceFilter** is designed for filters that modify data in the original buffer, also called *trans-in-place* filters. When a trans-in-place filter receives a sample, it changes the data inside that sample and delivers the same sample downstream. The filter's input pin and output pin always connect with matching media types.
- **CVideoTransformFilter** is designed primarily for video decoders. It derives from **CTransformFilter**, but includes functionality for dropping frames if the downstream renderer falls behind.
- **CBaseFilter** is a generic filter class. The other classes in this list all derive from **CBaseFilter**. If none of them is suitable, you can fall back on this class. However, this class also requires the most work on your part.

**Important** In-place video transforms can have a serious impact on rendering performance. In-place transforms require read-modify-write operations on the buffer. If the memory resides on a graphics card, read operations are significantly slower. Moreover, even a copy transform can cause unintended read operations if you do not implement it carefully. Therefore, you should always do performance testing if you write a video transform.

For the example RLE encoder, the best choice is either **CTransformFilter** or **CVideoTransformFilter**. In fact, the differences between them are largely internal, so it is easy to convert from one to the other. Because the media types must be different on the two pins, the **CTransInPlaceFilter** class is not appropriate for this filter. This example will use **CTransformFilter**.



## Step 2. Declare the Filter Class

Start by declaring a C++ class that inherits the base class:

```
class CRleFilter : public CTransformFilter
{
    /* Declarations will go here. */
```

```
};
```

Each of the filter classes has associated pin classes. Depending on the specific needs of your filter, you might need to override the pin classes. In the case of **CTransformFilter**, the pins delegate most of their work to the filter, so you probably don't need to override the pins.

You must generate a unique CLSID for the filter. You can use the Guidgen or Uuidgen utility; never copy an existing GUID. There are several ways to declare a CLSID. The following example uses the **DEFINE_GUID** macro:

[RleFilt.h]

```
// {1915C5C7-02AA-415f-890F-76D94C85AAF1}
DEFINE_GUID(CLSID_RLEFilter,
0x1915c5c7, 0x2aa, 0x415f, 0x89, 0xf, 0x76, 0xd9, 0x4c, 0x85, 0xaa, 0xf1);
```

[RleFilt.cpp]

```
#include <initguid.h>
#include "RleFilt.h"
Next, write a constructor method for the filter:
CRleFilter::CRleFilter()
   : CTransformFilter(NAME("My RLE Encoder"), 0, CLSID_RLEFilter)
{
    /* Initialize any private variables here. */
}
```

Notice that one of the parameters to the **CTransformFilter** constructor is the CLSID defined earlier.

## Step 3. Support Media Type Negotiation

When two pins connect, they must agree on a media type for the connection. The media type describes the format of the data. Without the media type, a filter might deliver one kind of data, only to have another filter treat it as something else.

The basic mechanism for negotiating media types is the IPin::ReceiveConnection method. The output pin calls this method on the input pin with a proposed media type. The input pin accepts the connection or rejects it. If it rejects the connection, the output pin can try another media type. If no suitable types are found, the connection fails.
Optionally, the input pin can advertise a list of types that it prefers, through the IPin::EnumMediaTypes method. The output pin can use this list when it proposes media types, although it does not have to.

The **CTransformFilter** class implements a general framework for this process, as follows:

- The input pin has no preferred media types. It relies entirely on the upstream filter to propose the media type. For video data, this makes sense, because the media type includes the image size and the frame rate. Typically, that information must be supplied by an upstream source filter or parser filter. In the case of audio data, the set of possible formats is smaller, so it may be practical for the input pin to offer some preferred types. In that case, override **CBasePin::GetMediaType** on the input pin.
- When the upstream filter proposes a media type, the input pin calls the **CTransformFilter::CheckInputType** method, which accepts or rejects the type.
- The output pin will not connect unless the input pin is connected first. This behavior is typical for transform filters. In most cases, the filter must determine the input type before it can set the output type.
- When the output pin does connect, it has a list of media types that it proposes to the downstream filter. It calls the **CTransformFilter::GetMediaType** method to generate this list. The output pin will also try any media types that the downstream filter proposes.
- To check whether a particular output type is compatible with the input type, the output pin calls the **CTransformFilter::CheckTransform** method.

The three **CTransformFilter** methods listed previously are pure virtual methods, so your derived class must implement them. None of these methods belongs to a COM interface; they are simply part of the implementation provided by the base classes.

The following sections describe each method in more detail:



## Step 4. Set Allocator Properties

**Note** This step is not required for filters that derive from **CTransInPlaceFilter**.

After two pins agree on a media type, they select an allocator for the connection and negotiate allocator properties, such as the buffer size and the number of buffers.

In the **CTransformFilter** class, there are two allocators, one for the upstream pin connection and one for the downstream pin connection. The upstream filter selects the upstream allocator and also chooses the allocator properties. The input pin accepts whatever the upstream filter

decides. If you need to modify this behavior, override the CBaseInputPin::NotifyAllocator method.

The transform filter's output pin selects the downstream allocator. It performs the following steps:

1. If the downstream filter can provide an allocator, the output pin uses that one. Otherwise, the output pin creates a new allocator.
2. The output pin gets the downstream filter's allocator requirements (if any) by calling **IMemInputPin::GetAllocatorRequirements**.
3. The output pin calls the transform filter's **CTransformFilter::DecideBufferSize** method, which is pure virtual. The parameters to this method are a pointer to the allocator and an **ALLOCATOR_PROPERTIES** structure with the downstream filter's requirements. If the downstream filter has no allocator requirements, the structure is zeroed out.
4. In the **DecideBufferSize** method, the derived class sets the allocator properties by calling **IMemAllocator::SetProperties**.

Generally, the derived class will select allocator properties based on the output format, the downstream filter's requirements, and the filter's own requirements. Try to select properties that are compatible with the downstream filter's request. Otherwise, the downstream filter might reject the connection.

In the following example, the RLE encoder sets minimum values for the buffer size, buffer alignment, and buffer count. For the prefix, it uses whatever value the downstream filter requested. The prefix is typically zero bytes, but some filters require it. For example, the AVI Mux filter uses the prefix to write RIFF headers.

```
HRESULT CRleFilter::DecideBufferSize(
    IMemAllocator *pAlloc, ALLOCATOR_PROPERTIES *pProp)
{
    AM_MEDIA_TYPE mt;
    HRESULT hr = m_pOutput->ConnectionMediaType(&mt);
    if (FAILED(hr))
    {
        return hr;
    }

    ASSERT(mt.formattype == FORMAT_VideoInfo);
    BITMAPINFOHEADER *pbmi = HEADER(mt.pbFormat);
    pProp->cbBuffer = DIBSIZE(*pbmi) * 2;
    if (pProp->cbAlign == 0)
    {
        pProp->cbAlign = 1;
```

```
    }
    if (pProp->cBuffers == 0)
    {
        pProp->cBuffers = 1;
    }
    // Release the format block.
    FreeMediaType(mt);

    // Set allocator properties.
    ALLOCATOR_PROPERTIES Actual;
    hr = pAlloc->SetProperties(pProp, &Actual);
    if (FAILED(hr))
    {
        return hr;
    }
    // Even when it succeeds, check the actual result.
    if (pProp->cbBuffer > Actual.cbBuffer)
    {
        return E_FAIL;
    }
    return S_OK;
}
```

The allocator may not be able to match your request exactly. Therefore, the **SetProperties** method returns the actual result in a separate **ALLOCATOR_PROPERTIES** structure (the *Actual* parameter, in the previous example). Even when **SetProperties** succeeds, you should check the result to make sure they meet your filter's minimum requirements.

### Custom Allocators

By default, all of the filter classes use the **CMemAllocator** class for their allocators. This class allocates memory from the virtual address space of the client process (using **VirtualAlloc**). If your filter needs to use some other kind of memory, such as DirectDraw surfaces, you can implement a custom allocator. You can use the **CBaseAllocator** class or write an entirely new allocator class. If your filter has a custom allocator, override the following methods, depending on which pin uses the allocator:

- Input pin: **CBaseInputPin::GetAllocator** and **CBaseInputPin::NotifyAllocator**.
- Output pin: **CBaseOutputPin::DecideAllocator**.

If the other filter refuses to connect using your custom allocator, your filter can either fail the connection, or else connect with the other filter's allocator. In the latter case, you might need to set an internal

flag indicating the type of allocator. For an example of this approach, see CDrawImage Class.

## Step 5. Transform the Image

The upstream filter delivers media samples to the transform filter by calling the IMemInputPin::Receive method on the transform filter's input pin. To process the data, the transform filter calls the **Transform** method, which is pure virtual. The **CTransformFilter** and **CTransInPlaceFilter** classes use two different versions of this method:

- **CTransformFilter::Transform** takes a pointer to the input sample and a pointer to the output sample. Before the filter calls the method, it copies the sample properties from the input sample to the output sample, including the time stamps.
- **CTransInPlaceFilter::Transform** takes a pointer to the input sample. The filter modifies the data in place.

If the **Transform** method returns S_OK, the filter delivers the sample downstream. To skip a frame, return S_FALSE. If there is a streaming error, return a failure code.

The following example shows how the RLE encoder might implement this method. Your own implementation might differ considerably, depending on what your filter does.

```
HRESULT CRleFilter::Transform(IMediaSample *pSource, IMediaSample *pDest)
{
    // Get pointers to the underlying buffers.
    BYTE *pBufferIn, *pBufferOut;
    hr = pSource->GetPointer(&pBufferIn);
    if (FAILED(hr))
    {
        return hr;
    }
    hr = pDest->GetPointer(&pBufferOut);
    if (FAILED(hr))
    {
        return hr;
    }
    // Process the data.
    DWORD cbDest = EncodeFrame(pBufferIn, pBufferOut);
    KASSERT((long)cbDest <= pDest->GetSize());

    pDest->SetActualDataLength(cbDest);
```

```
    pDest->SetSyncPoint(TRUE);
    return S_OK;
}
```

This example assumes that EncodeFrame is a private method that implements the RLE encoding. The encoding algorithm itself is not described here; for details, see the topic "Bitmap Compression" in the Platform SDK documentation.

First, the example calls IMediaSample::GetPointer to retrieve the addresses of the underlying buffers. It passes these to the private EncoderFrame method. Then it calls IMediaSample::SetActualDataLength to specify the length of the encoded data. The downstream filter needs this information so that it can manage the buffer properly. Finally, the method calls IMediaSample::SetSyncPoint to set the key frame flag to TRUE. Run-length encoding does not use any delta frames, so every frame is a key frame. For delta frames, set the value to FALSE.

Other issues that you must consider include:

- Time stamps. The **CTransformFilter** class timestamps the output sample before calling the **Transform** method. It copies the time stamp values from the input sample, without modifying them. If your filter needs to change the time stamps, call **IMediaSample::SetTime** on the output sample.
- Format changes. The upstream filter can change formats mid-stream by attaching a media type to the sample. Before doing so, it calls **IPin::QueryAccept** on your filter's input pin. In the **CTransformFilter** class, this results in a call to **CheckInputType** followed by **CheckTransform**. The downstream filter can also change media types, using the same mechanism. In your own filter, there are two things to watch for:
    - Make sure that **QueryAccept** does not return false acceptances.
    - If your filter does accept format changes, check for them inside the **Transform** method by calling **IMediaSample::GetMediaType**. If that method returns S_OK, your filter must respond to the format change.

  For more information, see Dynamic Format Changes.

- Threads. In both **CTransformFilter** and **CTransInPlaceFilter**, the transform filter delivers output samples synchronously inside the **Receive** method. The filter does not create any worker threads to process the data. Typically, there is no reason for a transform filter to create worker threads.

## Step 6. Add Support for COM

The final step is adding support for COM.

## Reference Counting

You do not have to implement **AddRef** or **Release**. All of the filter and pin classes derive from CUnknown, which handles reference counting.

## QueryInterface

All of the filter and pin classes implement **QueryInterface** for any COM interfaces they inherit. For example, **CTransformFilter** inherits **IBaseFilter** (through **CBaseFilter**). If your filter does not expose any additional interfaces, you do not have to do anything else.

To expose additional interfaces, override the CUnknown::NonDelegatingQueryInterface method. For example, suppose your filter implements a custom interface named IMyCustomInterface. To expose this interface to clients, do the following:

- Derive your filter class from that interface.
- Put the **DECLARE_IUNKNOWN** macro in the public declaration section.
- Override **NonDelegatingQueryInterface** to check for the IID of your interface and return a pointer to your filter.

The following code shows these steps:

```
CMyFilter : public CBaseFilter, public IMyCustomInterface
{
public:
    DECLARE_IUNKNOWN
    STDMETHODIMP NonDelegatingQueryInterface(REFIID iid, void **ppv);
};
STDMETHODIMP CMyFilter::NonDelegatingQueryInterface(REFIID iid, void **ppv)
{
    if (riid == IID_IMyCustomInterface) {
        return GetInterface(static_cast<IMyCustomInterface*>(this), ppv);
    }
    return CBaseFilter::NonDelegatingQueryInterface(riid,ppv);
}
```

For more information, see How to Implement IUnknown.

## Object Creation

If you plan to package your filter in a DLL and make it available to other clients, you must support **CoCreateInstance** and other related COM functions. The base class library implements most of this; you just need

to provide some information about your filter. This section gives a brief overview of what to do. For details, see How to Create a DLL.

First, write a static class method that returns a new instance of your filter. You can name this method anything you like, but the signature must match the one shown in the following example:

```
CUnknown * WINAPI CRleFilter::CreateInstance(LPUNKNOWN pUnk, HRESULT *pHr)
{
    CRleFilter *pFilter = new CRleFilter();
    if (pFilter== NULL)
    {
        *pHr = E_OUTOFMEMORY;
    }
    return pFilter;
}
```

Next, declare a global array of CFactoryTemplate class instances, named **g_Templates**. Each **CFactoryTemplate** class contains registry information for one filter. Several filters can reside in a single DLL; simply include additional **CFactoryTemplate** entries. You can also declare other COM objects, such as property pages.

```
static WCHAR g_wszName[] = L"My RLE Encoder";
CFactoryTemplate g_Templates[] =
{
  {
    g_wszName,
    &CLSID_RLEFilter,
    CRleFilter::CreateInstance,
    NULL,
    NULL
  }
};
```

Define a global integer named **g_cTemplates** whose value equals the length of the **g_Templates** array:

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

Finally, implement the DLL registration functions. The following example shows the minimal implementation for these functions:

```
STDAPI DllRegisterServer()
{
```

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

## Filter Registry Entries

The previous examples show how to register a filter's CLSID for COM. For many filters, this is sufficient. The client is then expected to create the filter using **CoCreateInstance** and add it to the filter graph by calling IFilterGraph::AddFilter. In some cases, however, you might want to provide additional information about the filter in the registry. This information does the following:

- Enables clients to discover the filter using the Filter Mapper or the System Device Enumerator.
- Enables the Filter Graph Manager to discover the filter during automatic graph building.

The following example registers the RLE encoder filter in the video compressor category. For details, see How to Register DirectShow Filters. Be sure to read the section Guidelines for Registering Filters, which describes the recommended practices for filter registration.

```
// Declare media type information.
FOURCCMap fccMap = FCC('MRLE');
REGPINTYPES sudInputTypes = { &MEDIATYPE_Video, &GUID_NULL };
REGPINTYPES sudOutputTypes = { &MEDIATYPE_Video, (GUID*)&fccMap };

// Declare pin information.
REGFILTERPINS sudPinReg[] = {
    // Input pin.
    { 0, FALSE, // Rendered?
        FALSE, // Output?
        FALSE, // Zero?
        FALSE, // Many?
        0, 0,
        1, &sudInputTypes    // Media types.
    },
    // Output pin.
    { 0, FALSE, // Rendered?
        TRUE, // Output?
        FALSE, // Zero?
        FALSE, // Many?
```

```
            0, 0,
            1, &sudOutputTypes          // Media types.
    }
};

// Declare filter information.
REGFILTER2 rf2FilterReg = {
    1,                      // Version number.
    MERIT_DO_NOT_USE, // Merit.
    2,                      // Number of pins.
    sudPinReg               // Pointer to pin information.
};

STDAPI DllRegisterServer(void)
{
    HRESULT hr = AMovieDllRegisterServer2(TRUE);
    if (FAILED(hr))
    {
        return hr;
    }
    IFilterMapper2 *pFM2 = NULL;
    hr = CoCreateInstance(CLSID_FilterMapper2, NULL, CLSCTX_INPROC_SERVER,
            IID_IFilterMapper2, (void **)&pFM2);
    if (SUCCEEDED(hr))
    {
        hr = pFM2->RegisterFilter(
            CLSID_RLEFilter,                // Filter CLSID.
            g_wszName,                      // Filter name.
            NULL,                           // Device moniker.
            &CLSID_VideoCompressorCategory, // Video compressor category.
            g_wszName,                      // Instance data.
            &rf2FilterReg                   // Filter information.
            );
        pFM2->Release();
    }
    return hr;
}

STDAPI DllUnregisterServer()
{
    HRESULT hr = AMovieDllRegisterServer2(FALSE);
    if (FAILED(hr))
    {
        return hr;
```

```
    }
    IFilterMapper2 *pFM2 = NULL;
    hr = CoCreateInstance(CLSID_FilterMapper2, NULL, CLSCTX_INPROC_SERVER,
            IID_IFilterMapper2, (void **)&pFM2);
    if (SUCCEEDED(hr))
    {
        hr = pFM2->UnregisterFilter(&CLSID_VideoCompressorCategory,
            g_wszName, CLSID_RLEFilter);
        pFM2->Release();
    }
    return hr;
}
```

Also, filters do not have to be packaged inside DLLs. In some cases, you might write a specialized filter that is designed only for a specific application. In that case, you can compile the filter class directly in your application, and create it with the new operator, as shown in the following example:

```
#include "MyFilter.h"    // Header file that declares the filter class.
// Compile and link MyFilter.cpp.
int main()
{
    IBaseFilter *pFilter = 0;
    {
        // Scope to hide pF.
        CMyFilter* pF = new MyFilter();
        if (!pF)
        {
            printf("Could not create MyFilter.\n");
            return 1;
        }
        pF->QueryInterface(IID_IBaseFilter,
            reinterpret_cast<void**>(&pFilter));
    }

    /* Now use pFilter as normal. */

    pFilter->Release();    // Deletes the filter.
    return 0;
}
```

# 4. Video Tracking Algorithms

## A. Tracking Algorithms Based on Brightness

Before we start processing frames, it is important to understand the different representations for color spaces used in digitized video. There are many color spaces to choose from, and each of them has its own strengths and limitations. Choosing the right color space for a specific application simplifies computation significantly.

The feature that we will be looking at for this demo application is brightness, and we will track objects based on their brightness. A very natural approach is to make sure that the color space that we are dealing with has a brightness component. YUV is one color space that has this very component that we are seeking for. However, YUV is not necessarily one of the input formats that is available from the web camera. Therefore, a conversion is required from the typical RGB24 input format to YUV.

The relationship between RGB and YUV can be expressed simply as the following set of linear equations.

```
Y=0.299R+0.587G+0.114B
U=-0.148R-0.289G+0.437B
V=0.615R-0.515G-0.100B
```

This matrix results from the concept of change of basis in linear algebra, where in this case, corresponds to the rotation of the color cube such that the new basis has a component with the unique property R = G = B.

For more fast computing speed,we use Y=(R+G)>>1 to compute the brightness.

## Rectangle algorithm

Now that we have direct access to the brightness of each pixel, a simple algorithm can be used to track a bright object. The algorithm that will be introduced here is a fairly simple one, called the "rectangle algorithm". The rectangle algorithm keeps track of four points in each frame, the top most, left most, right most and bottom most points where the brightness exceeds a certain threshold value.

```
Initialize left=inf,right=0,top=inf,bottom=0
For each pixel p(x,y)
    Compute the brightness of the pixel use Y=(R+G)>>1
    If the brightness is lerger enough ,then
        If left>x then left=x
        If right<x then right=x
        If top>y then top=x
        If bottom<y then bottom=y
```

```
    End if
End for
```

A rectangle can be constructed from these points, which tells us where the bright object is. The border of the rectangle is then simply replaced by a predefined color.

```
For x=left to right
        Make pixel p(x,y:=top) with RED color.
        Make pixel p(x,y:=bottom) with RED color.
End if
For y=top to bottom
        Make pixel p(x=left,y) with RED color.
        Make pixel p(x=right,y) with RED color.
End if
```

This algorithm obviously has a lot of weaknesses.

- It only gives the position of the object as a whole on the screen.
- It does not keep any information about the shape of the object.
- It does not tell where the middle of the object is.
- It can never track multiple objects.

## An Improved Algorithm

This algorithm tracks objects by identifying segments that make up the object on the screen. Each segment consists of the head and the length of the segment. The object is constructed by grouping the segments together.
First we introduce a struct which consists three member:int x,y;which denote the left position of the bright pixels, and int length,which denotes the length of the bright pixels in one line.

```
Init an empty list which consists the struct QSEG .
For y=1 to heightof the image
        QSEG Qseg={0,0,0};
        For x=1 to width of the image
                If the brightness of pixel p(x,y) is larger than the threshold T,then
                        If sqseg.length==0 then
                                Qseg.x=x
                                Qseg.y=y
                                Qseg.length=1
                        Else
                                Qseg.length++
                        End if
                End if
        End for
        Add Qseg to the list.
```

```
End for
```

Now we can draw the object.

```
For each element qseg in the list
        Mark pixel p(qseg.x,qseg.y) with RED color
        Mark pixel p(qseg.x+length,qseg.y) with RED color
End for
```

## B. Tracking Algorithms Based on Static Background

If the background is not moving,then we can compute the brightness differences between the foreground and the background,If the difference is Large Enough,then we treat it as some thing stands there.

Main Frames of tracking algorithm:

```
For each pixel in one frame image
        Compute the difference of pixel values between background image and
foreground image
        If the difference is larger than the threshold value,then
                Make a mark on the pixel which denotes the pixel as object tracked
        End if
End for
```

There are some kinds of ways to get the background image:

1) One of the most common approaches is to compare the current frame with the previous one. It's useful in video compression when you need to estimate changes and write only the changes, not the whole frame. But it is not the best one for motion detection applications. But it is an effective and efficient method.

Assume that we have an original 24 bpp RGB image called current frame (image), a grayscale copy of it (currentFrame) and a previous video frame also gray scaled (backgroundFrame). First of all, let's find the regions where these two frames differ a bit. For this purpose, we can use Difference and Threshold methods,then we get an image with white pixels in the place where the current frame is different from the previous frame on the specified threshold value. It's possible to count the pixels, and if the amount is greater than a predefined alarm level we can signal a motion event.

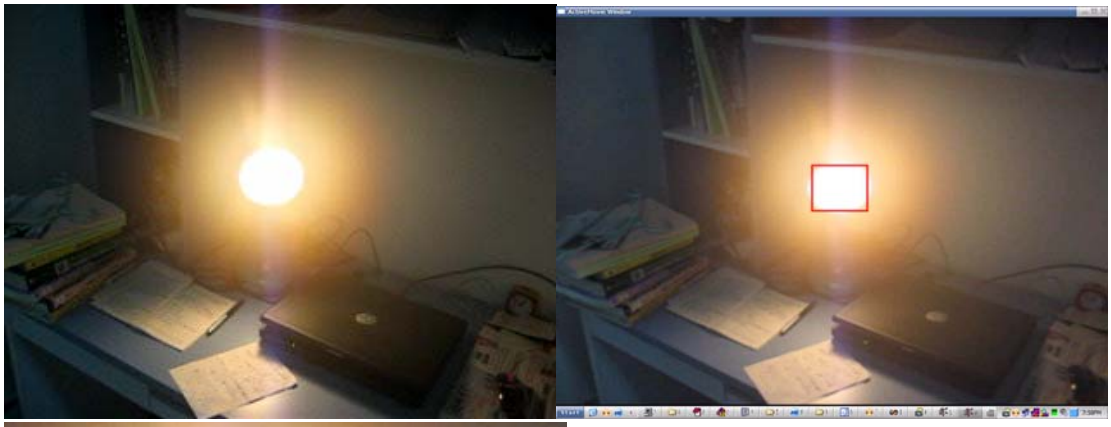
The disadvantages of the approach are quite clear. If the object is moving smoothly we receive small changes from frame to frame. So, it's impossible to get the whole moving object. Things become worse, when the object moves slowly, then the algorithms do not give any result at all.

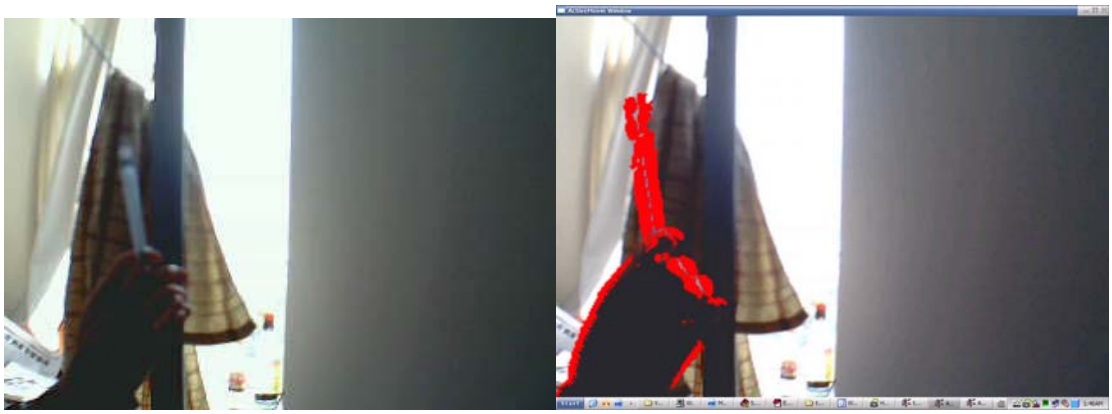
2) The background image is gaven before tracking,this is the easiest way.But it often works not as good as we think,since the image in the video is always changing for the changing of weather or the camera's parameters,for example,if the weather is clouding or sunny,the grayscale of the image may be much different.

3) The background image is calculated from the video by algorithm.There are many different algorithms for computing the background images,but most of them are time consuming.We use a relative simple algorithms to compute the background image,results shows the algorithms is effect,but still needs improving.

Assume that we have an original 24 bpp RGB image called current frame (image), a grayscale copy of it (currentFrame) and a background frame also gray scaled (backgroundFrame). In the beginning, we get the first frame of the video sequence as the background frame. And then we compare the current frame with the background one. Our approach is to "move" the background frame to the current frame on the specified amount . We slowly move the background frame in the direction of the current frame – we change the colors of the pixels in the background frame by one level per frame,By this algorithm,we can get a image which tends to the real background.

## 5. Experiment Results & Discuss





Above Figures are the experiment results,the left two images are the original video images,the right two images are the video tracking results,the object is marked with RED colors.

Experiment results shows that our video tracking algorithms can be used in many fields.The two kinds of algorithms are suit for different application fields.

The first algoritm is based on brightness,It can be applied in tracking very white or very black objects whose color are very distinct from the background.

The second algorithm is based on stable background and can only tracks moving objects.It can be used in such fields where the camera is not moving,such as traffic monitoring,hotel monitoring,and so on.

There are still many things need improving,such as:

The algorithms just use pixel value,do not use other information,such as object geometry structure,which is always used by our human's vision system.

In the second algorithm,if the background can not obtained directly,we must extract it from the video,This is a very hard work,The methods in this paper is VERY preliminary.The algorithms we introduce are still need improving.

# 6. Appendix

## Source code:

Videofeaturetracking.rar

## How to compile and test the programme:

- ✧ You must have installed VC6.0 and DirectXSDK and have setup the development environment.More detail,see 3.1)Preparation.
- ✧ Compile and build the project ezrgb24.dsw
- ✧ Use regsvr32.exe ezrgb24.ax to register the filter in the system
- ✧ Run graphedit.exe in DXSDK tools,click menu Graph→Insert filters… ,then select DirechShow filters / ImageEffects(EZRGB24).click button "Insert filters" to insert the filter in the graph.
- ✧ Click menu file→render media file..,open the video file,then we can see the filter graph be built,and the ImageEffects filter was connected in the graph

automatically.

- ✧ Now click menu graph→Play to enjoy the effects!!

# 7. Reference


1) www.codeproject.com
2) msdn
3) www.csdn.net