DIRECTSHOW TV DISPLAY
WITH REAL-TIME VIDEO PROCESSING

INDIVIDUAL PROJECT REPORT
ANDREW DOWSEY MENG

SUPERVISOR: IAN HARRIES
ABSTRACT

In general computing companies have been quick to bring the latest digital technologies to the market. There has been little action in harnessing the power of current PC processors for the improvement of analogue video, however. This can be attributed both to the lack of a uniform multimedia framework & the vertical integration policies of the TV Card manufacturers.

Recently, Microsoft’s multimedia framework DirectShow, part of DirectX, has matured enough for the release of television viewer applications & digital video recorders. Currently these programs are only of much use to OEM’s, as very few TV card manufacturers have developed drivers native to DirectShow. The current Video for Windows drivers lack vital features including the TV Tuner.

Whilst the demand for better quality display in televisions & AV equipment has fuelled innovation in video processing IC’s, the PC world remains blissfully unaware of the advancements. Typical PC TV viewers suffer from severe artefacts on moving scenes due to the interlaced video, & also noise from other equipment. An Open Source development, dScaler, attempts to remedy this situation, but suffers from the twin disadvantages of lack of academic involvement & little future proofing.

This project attempts to combine real-time techniques for noise reduction & deinterlacing derived from academic research papers, with a viewer application, both implemented in the Microsoft DirectShow framework. The video processing ‘filter’, as it is called, is a COM component, & therefore can be reused without modification in other DirectShow projects. The filter implements a series of noise reduction & deinterlacing algorithms for use on varying speeds of PC, as well as to show contrast with current methods. The viewer application facilitates the demonstration of the techniques & illustrates advantages of the DirectShow framework, such as simultaneous preview & recording of video.

From the outset the need for compatibility with legacy Video for Windows drivers was recognised. To this end a DirectShow Tuner filter for the Hauppauge line of WinTV cards was developed (Hauppauge sell over 80% of all TV cards in Europe). The application also works out of the box with any WDM complaint driver, including those for digital video & HDTV cards, as well as any AVI file if you have the requisite codec.

ACKNOWLEDGEMENTS

I wish to thank my supervisor Ian Harries [1], & my 2nd supervisor Guang-Zhong Yang [2], for their help, inspiration & encouragement during the course of the project & meetings.

I also wish to thank the development team behind dScaler [3], headed by John Adcock [4], for impressing me so much with their program, which consequently gave me the initial idea for this project. Their help on the dScaler developer mailing list was also invaluable, especially when they asserted that DirectShow was too slow for programs of dScaler’s purpose.

Thanks are also due to my personal tutor Pete Harrison [5], for our discussions over my years as a student; CSG, for supplying me with a project PC; Stuart Holliday & Co. for testing & feedback; & Gerald de Hann [6] for being so proficient in developing deinterlacing algorithms.
CONTENTS

Abstract...........................................................................................................................................................i
Acknowledgements ......................................................................................................................................i
Introduction .................................................................................................................................................. 1
Convergence between Computers & Video............................................................................................... 1
Video & Video Processing ......................................................................................................................... 2
Current Options ........................................................................................................................................ 3
The Realm of Electrical Engineers?......................................................................................................... 5
My Alternative ........................................................................................................................................... 6
Research ......................................................................................................................................................... 7
  Programming.............................................................................................................................................. 7
  Microsoft DirectX..................................................................................................................................... 7
  The Microsoft DirectShow Framework ................................................................................................... 8
  MMX, integer SSE & the Intel Performance Primitives .......................................................................... 10
  Hauppauge WinTV SDK ....................................................................................................................... 12
Noise Reduction ......................................................................................................................................... 13
  Early Noise Reduction techniques ......................................................................................................... 13
  Noise Reduction Algorithm 1: Weighted Recursive Temporal Averaging ........................................... 14
  Noise Reduction Algorithm 2: Adaptive Temporal Averaging ............................................................. 14
Deinterlacing ............................................................................................................................................... 15
  Early de-interlacing techniques ............................................................................................................. 15
  Deinterlacing Algorithm 1: Vertical-Temporal ...................................................................................... 16
  Deinterlacing Algorithm 2: VT with 5 Point Median Protection .......................................................... 17
  Deinterlacing Algorithm 3: 3D Edge Orientated .................................................................................. 18
Motion Estimation ..................................................................................................................................... 19
Design .......................................................................................................................................................... 21
  System Overview .................................................................................................................................... 21
  Hauppauge WinTV Tuner Filter ............................................................................................................. 22
  Interfaces................................................................................................................................................ 22
  Adaptors .................................................................................................................................................. 25
Video Processor Filter .............................................................................................................................. 26
  Flip Field Detector ............................................................................................................................... 26
  Video Processing Strategies .................................................................................................................. 28
  Strategy Factories................................................................................................................................ 29
  Image Buffer .......................................................................................................................................... 30
  Interfaces................................................................................................................................................ 31
Application ................................................................................................................................................ 32
  Features .................................................................................................................................................. 32
  GUI Prototyping ..................................................................................................................................... 33
  Commands............................................................................................................................................... 34
# Implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hauppauge WinTV Tuner Filter</td>
<td>35</td>
</tr>
<tr>
<td>CTransInPlaceFilter</td>
<td></td>
</tr>
<tr>
<td>Adaptors</td>
<td>37</td>
</tr>
<tr>
<td>TV Tuner Property Page</td>
<td>38</td>
</tr>
<tr>
<td>TV Audio Property Page</td>
<td>40</td>
</tr>
<tr>
<td>Filter Registration</td>
<td></td>
</tr>
<tr>
<td>Assembly Optimisation</td>
<td>41</td>
</tr>
<tr>
<td>Detecting MMX, 3DNow! &amp; SSE</td>
<td></td>
</tr>
<tr>
<td>Vectoring with MMX &amp; iSSE</td>
<td>42</td>
</tr>
<tr>
<td>Other Optimisations</td>
<td>43</td>
</tr>
<tr>
<td>Video Processing Filter</td>
<td>44</td>
</tr>
<tr>
<td>CTransformFilter</td>
<td></td>
</tr>
<tr>
<td>Image Buffer</td>
<td>47</td>
</tr>
<tr>
<td>Strategies</td>
<td>47</td>
</tr>
<tr>
<td>Property Pages</td>
<td>49</td>
</tr>
<tr>
<td>Application</td>
<td>50</td>
</tr>
<tr>
<td>GUI</td>
<td></td>
</tr>
<tr>
<td>Filter Graph Construction</td>
<td>51</td>
</tr>
<tr>
<td>GUI</td>
<td></td>
</tr>
<tr>
<td>Evaluation</td>
<td>52</td>
</tr>
<tr>
<td>Unit Testing</td>
<td></td>
</tr>
<tr>
<td>Video Processing</td>
<td>53</td>
</tr>
<tr>
<td>Performance Analysis</td>
<td></td>
</tr>
<tr>
<td>Subjective Analysis</td>
<td>55</td>
</tr>
<tr>
<td>Comparative Analysis</td>
<td>61</td>
</tr>
<tr>
<td>Achievements</td>
<td>63</td>
</tr>
<tr>
<td>Limitations</td>
<td>64</td>
</tr>
<tr>
<td>Conclusions</td>
<td>65</td>
</tr>
<tr>
<td>Proposed Extensions</td>
<td>66</td>
</tr>
<tr>
<td>Improving Edge Orientated Deinterlacing</td>
<td></td>
</tr>
<tr>
<td>3D Recursive Search Block Based Motion Estimation</td>
<td>67</td>
</tr>
<tr>
<td>Appendix A - Bibliography</td>
<td>69</td>
</tr>
<tr>
<td>Appendix B - Links</td>
<td>71</td>
</tr>
</tbody>
</table>

## SOURCE CODE

Source code, executable binaries & a pdf version of this document are available from:

http://keeper.warhead.org.uk/awd97.zip
INTRODUCTION

The aim of this project was to create real-time video processing components under the Microsoft DirectShow framework for PC's with Microsoft Windows 98, ME or 2000 Professional & at least MMX support. The algorithms are also optimised for the AMD 3DNow! & Intel SSE instruction sets, & the most advanced algorithms will run in combination in real-time on a 1Gz processor.

A demonstration application was created to showcase the components & to illustrate the advantages of the DirectShow framework, such as:

- Compatibility with a wide range of current & future video input devices.
- A plugin architecture for compression
- Simultaneous video preview & AVI recording
- AVI processing.

A Hauppauge WinTV Tuner component was also created to continue the realisation of investment consumers have for these cards, which currently have no satisfactory WDM (DirectShow) drivers.

This report is structured as follows: The rest of this section introduces the reader to the history of multimedia in the computing industry, the progress of video processing outside of the computing industry, & the programs available which attempt to converge the two streams, both in real-time & in post-processing modes of operation; This section ends by explaining my motivation for this project & my contribution to this emerging field.

In chapter 2 the background of my project is explained. Research is given on noise reduction & deinterlacing algorithms, together with the underlying technologies for their implementation. Chapter 3 covers the top-level design & design patterns used & chapter 4 details the implementation & interfaces for understanding the code & for further future development of the work. Chapter 5 evaluates the work achieved, paying particular attention to the real-time performance & subjective quality of the algorithms implemented. It ends with a discussion of the achievements, limitations & conclusions of the project. Finally, a suggested improved deinterlacing algorithm for very fast computers is given, together with idea for an investigation of incorporating motion estimation into the system.

CONVERGENCE BETWEEN COMPUTERS & VIDEO

In the early 1990s, a digital video system capable of capturing full-screen video images would have cost several thousands of pounds. The biggest cost element was the compression hardware needed to reduce to a manageable size the huge files that result from the conversion of an analogue video signal into digital data. Less powerful 'video capture' cards were available, capable of compressing quarter-screen images - 320 x 240 pixels - but even these were far too expensive for the average PC user.

The turning point for PC video systems came as Intel released the Pentium processor with MMX technology [7] on Jan 8 1997. MMX (initially reported as MultiMedia eXtension, but later said by Intel to mean Matrix Math eXtension) is very similar to the earlier SPARC VIS instructions - they perform integer operations on vectors of 8,
16, or 32 bit words. Now with up to 4 pixels processed per instruction, video capture without dedicated hardware was possible.

Booktree (later Conexant, later Intel) offered a low cost IC onto the market for consumer television display & capture boards. The bt848 could stream full resolution video data onto the PCI bus for transfer direct into main memory through DMA or straight to the overlay engine of a graphics card. The bt848 was before its time, as it is still in widespread use today & has very few limitations even with today’s demanding needs. The chip was later surpassed by the bt878 & bt878a, with Nicam stereo support & basic digital TV functionality (ATSC).

With MMX, PCs could capture video streams of up to 320 x 240 without the need for expensive compression hardware. The advent of the Pentium 2 allowed real-time low compression of the video for full resolution recording; note that UK PAL video clocks in at 250Mbits per second & only a striped RAID array of hard disks could record this uncompressed even now.

In 1999 the Pentium !!! introduced SSE (Streaming SIMD Extensions) instructions. These were based mainly at vectoring floating point data for computer graphics use, but they did include some new integer instructions for facilitating such video processing as motion estimation & interpolation. This, together with the extra raw power of the CPU’s, led to heightened interest as the PC as a multimedia platform, & software such as DVD players [8], real-time compression codecs [9] & basic Digital Video Recorders (10) were released.

AV enthusiasts have now started asking questions at why more techniques they used to spend £1000s on could not be emulated by a PC. The AV Science Forum [11] hosts a bulletin board ‘Home Theatre Computers’ in which interested parties can discuss incorporating PC’s into their home cinema configurations. With software DVD players & Dolby Digital decoders in some cases surpassing the quality from high-end AV equipment, why cannot we also replace the video scaler, or a complete digital video recorder?

With Intel delivering the Pentium 4 with SSE2 instructions that double the vectoring capability to 128bits per instruction, multimedia convergence is becoming a big player in the computing industry.

The transmission of time-varying pictures, or video, needs a means to convert the sequence of 2D images into a 1D signal. The spatio-temporal information ordered as a function of time is prescribed by the scanning format [1]. The scanning format describes the number of scan lines per image & the number of images per second. The number of scan lines defines the maximum vertical resolution & the number of images per second defines the temporal resolution. The maximum perceivable horizontal resolution is determined by the video bandwidth & the video format, which leads to the pixel sampling frequency.

The illusion of motion can be achieved with more than 10 images per second, as an observer retains the impression of an image after it has been withdrawn for 0.1 seconds [2]. Higher rates than this are used to reduce flicker, whose perception varies wildly with viewing conditions.

To avoid artefacts caused by the cycling frequency of mains power, the picture update frequency was set to 60 images per second in America & Japan (NTSC with 525 scan lines), & 50 images per second in Europe (PAL with 625 scan lines). The transmission of 50 or 60 images per second was not economically attractive at the time, so the bandwidth was reduced by only transmitting even scan lines in one frame (even field), followed by odd scan lines in the next (odd field) –
interlacing. Whilst interlacing preserves vertical resolution & avoids large area flicker, a study by Engstrom [3] in the beginning of the thirties revealed the effect of line flicker – flicker often due to horizontal edges in the picture. High definition displays show visible serrations under swift horizontal motion. It is these artefacts that we attempt to reduce by the video processing discipline of deinterlacing.

Noise is defined as an unwanted component of the image. Modelled by the Gaussian distribution, thermal noise is the additive effect of transport mechanism imperfections & interference from the surrounding equipment. In the case of broadcast video, noise appears through the interference of unwanted signals, whether from other sources or the superposition of a delayed version of itself from refraction of buildings (ghosting). Noise reduction through video processing involves firstly modelling the noise, secondly estimating where it occurs, & finally eliminating it. This is non-trivial; as image detail & fast motion exhibit the same high frequencies we associate noise with. Noise reduction can be extended to the study of missing data reconstruction. Here, through blotches, digital dropout, or otherwise, we wish to recover the missing data through reconstruction. Stochastic techniques become important here.

For historical reasons, NTSC & PAL transmit the images as 3 signals: 1 luminance (Y) signal usually occupies the maximum bandwidth, while the chrominance signals (U & V) will each occupy half that bandwidth, since the eye is less sensitive to colour detail. These U & V signals are 'difference' signals; they are constructed by extracting the luminance signal alternatively from the colour signals (Red & Blue). The U colour-difference signal is Blue minus Y & the V is Red minus Y. Digital video processing is the theory & application of filtering, coding, transmitting, estimating, detecting, analysing, recognizing, synthesizing, recording, & reproducing the YUV 3D video, with particular attention to the 1D signal it may be derived from.

I will focus towards the current alternatives consumers have for displaying enhanced television & video on their computer monitors, televisions & projectors. Whilst there are many candidates that could be used to solve my project in the production & studio field, their costs are prohibitive & therefore their analysis is beyond the scope of this report.

**Hardware Video Processors**

- These include the home theatre video processors used in conjunction with home & cinema projectors. They invariably include line doublers, or quadruplers.
- These terms are a little misleading, as the doublers perform advanced deinterlacing & quadruplers also resample the resulting progressive frame vertically. More advanced processors include noise reduction & linear processing controls (i.e. contrast, sharpness, colour correction).

**Advantages:** Best performance?

**Disadvantages:** Very costly: Runco & Faroujda models retail for around £10,000
100Hz Televisions

The latest 100Hz Televisions that promise reduction in flickering artifacts must do so by up-converting the 50Hz field rate of PAL to 100Hz. This is achieved by deinterlacing to 50Hz frame rate & splitting the even/odd fields.

Advantages:
A great deal of research into the area has been conducted by Philips & Dr. Gerald de Hann (Senior member of the IEEE) in particular. The ICs used in these televisions now incorporate motion compensated deinterlacing techniques & noise reduction.

Consumer interest has confirmed the effectiveness of the technology.

Disadvantages:
Philips has intellectual rights on the methods.
There are rumors that the technology will be incorporated into future Philips PC DTV card reference designs, but this will come at a license price & it will not rectify the omission of this technology in current video capture sources.

Software Video Editing Suites

Software video editing suites such as Adobe Premier [12] or VirtualDUB have a plugin architecture, & it is known that deinterlacing & noise reduction plugins exist for them.

Advantages:
VirtualDUB is free.

Disadvantages:
Adobe Premiere costs £400.
They lack TV tuner support since they are not Television viewers. They do not process video in real time.

PC TV Card Vertical Integration

When you buy a PC TV Card, bundled with it is a basic application to allow you to view, change channel & possibly record the video stream. Examples include Hauppauge[14], ATI [15] & AVerMedia [16].

Advantages:
Free with your TV Card

Disadvantages:
No video processing whatsoever. Whilst technically a deinterlacing algorithm, weave is used to interleave the current field with the previous field. This causes very visible serrations under horizontal motion.
The applications are low on features & are created with proprietary APIs that limit their future extensibility.
PC TV Card manufacturers do not, despite there being some obvious benefits, take the effort to support the latest multimedia frameworks as they are not required for their own TV applications.

DirectShow Digital Video Recorders

Recently, companies such as Intervideo [8] & Cyberlink [10] have released Digital Video Recorders to supplement their software DVD player products. PowerVCR & WinDVR allow real-time viewing & recording of the stream from a WDM compliant capture source. It is the facilitation of the Microsoft DirectShow framework that has allowed this progress to take place.

Advantages:
First programs that give PC users the real functionality associated with consumer set-top-box hard disk recording devices.
Have TIVO [17] like functionality to pause & replay live video.
The DirectShow framework allows compatibility with all future capture sources & TV tuners that comply with WDM specifications.
The resultant recorded programmes are compressed highly.

Disadvantages:
Very poor support for b8x8 TV Cards, which make the bulk of those in existence. WDM drivers do exist for the chipset, but they are limited, buggy & do not have PAL tuner support. These problems are not due to the underlying hardware.
Because of this, these programs are of most interest to OEM’s, & their sales are targeted in this way.
The recording technology is based on MPEG & MPEG2 compression, which is not ready for prime-time quite yet as quality is only acceptable on 1Gz+ machines, & even then the quality can only be argued as comparable to VHS. Deinterlacing support is limited to basic bobbing (see later), which is implicitly supported by DirectShow.

**dScaler**

dScaler is an interesting OpenSource project born by a movement from the AV Science Forum. Frustrated with the price of conventional hardware scalers, they resolved to create a bt8x8 chipset television viewer with video processing technology such as deinterlacing & noise reduction.

**Advantages:**
Deinterlacing algorithms used are miles ahead of PC alternatives. They use the modern graphics card’s ability to stretch the overlay surface for line quadrupling purposes.

Believe it or not, a combination of dScaler & an ATI Radeon graphics card (with 10bit Video DACs) has outperformed a £20,000 Faroujda video processor in subjective tests.

**Disadvantages:**
At its inception, for want of a better architecture & for performance concern reasons, they ported the linux bt8x8 driver for use in the project. The driver runs in kernel ring 0 on Windows NT systems. Problems with this approach are:

- The software does not, & cannot work with any other video capture card.
- The developers are committed to the architecture, & refuse to believe a DirectShow version is viable due to performance reasons.
- The deinterlacing algorithms are born out of the minds of the programmers, & not from academic research. The algorithms are in some cases impressive, but they cannot compare to the decades of research on the subject by academics.
- Only an administrator can run the program on a Windows NT system.

Trawling through research on the subject of motion estimation & deinterlacing reveals that it is almost completely confined to the electrical engineering community. Therefore the algorithms are targeted towards hardware or VLSI (Very large- scale integration) implementation. The papers I identified as viable for real-time software implementation are over 10 years old. All recent papers require some form of motion estimation that has proved to be beyond current PC’s potential. Also, the algorithms aim to minimise memory at the expense of performance for hardware cost reasons. A current PC has the opposite cost model: sacrifice memory to gain performance. The algorithms proposed in the next section come from academic sources but with certain noted modification for PC use.

Very recently (too late for implementation in the core project), new papers with simpler algorithms have been proposed. The main motivation for this is for integration into low-end widespread consumer applications. The proposed extensions section discusses these.
What did I contribute to the field?

My motivation was to create a television viewer that could enhance the viewing experience for television & other video sources on the PC platform. I felt that there was a hole in the market in PC real-time video processing that could maintain the attractiveness of current video capture cards as well as add to the worth of future devices, without sacrificing usability. Why should the CPU remain idle during television viewing?

I felt that the research & experience from the same problems in the electrical engineering field could be drawn upon to create more effective techniques whilst keeping performance cost low to keep the potential market large. These algorithms draw upon greater analysis, model use & stringent evaluation criteria that current 'available' (not secret or patented) algorithms do in the PC community.

So why did I create an alternative product rather than collaborating in dScaler development or create plugins, which they support?

I felt that dScaler was too limited by a non-standard architecture to be commercially attractive. OEM's & specialist companies would be mostly concerned with the ease of integration & coexistence with their own products, which dScaler cannot allow for. Through initial discussions with the development team I understood their beliefs that DirectShow limited the performance too much to be viable as an architecture to migrate dScaler to. Whilst I was certain framework's such as the Java Media Framework were too slow, I refused to accept DirectShow would be.

Microsoft DirectShow gives me a component architecture, with integrated recording out of the box, & support for any compliant compression codec or video capture card without modification. It would not be feasible to get dScaler to do this.

Central to the ethos of my project was to keep backwards compatibility with the majority of current devices, by developing DirectShow wrappers for specific TV tuners & cards, if it became necessary. This protects the investment of consumers.

In a nutshell, my project attempt to bring to the field of PC multimedia the best aspects of dScaler & existing DirectShow TV viewers. It then goes on to improve them with my own innovations & those from the electrical engineering field. Hopefully this will act as a catalyst for further research into the area of real-time PC video processing.
Microsoft DirectX is a group of technologies to make Microsoft Windows-based computers the platform for running & displaying applications like multimedia & games that are rich in real-time full-color 3-D graphics, video, interactive music, & surround sound. Built directly into the Windows family of operating systems, DirectX is an integral part of Microsoft Windows 9x & Microsoft Windows 2000.

DirectX gives software developers a consistent set of APIs (application programming interfaces) that provides them with improved access to the advanced features of high-performance hardware such as 3-D graphics accelerators & sound cards. These APIs allow direct access to low-level functions that are disallowed by the standard Win32 API. The low-level functions are grouped into components that make up DirectX: Direct3D, DirectDraw, DirectInput, DirectMusic, DirectPlay, DirectSound, & DirectShow.

DirectX is based on the Component Object Model (COM) [4]. COM is Microsoft’s technology for defining a language independent interface for an object (or ‘component’ since COM is not restricted to Object Orientated languages). The interface (the set of method signatures) for a component is defined in IDL (Interface Definition Language, developed by the Open Software Foundation). IDL is not a programming language; it merely declares the signatures of the methods. Programs in languages such as Visual C++, Visual Basic & Java can then access these components through these interfaces.

Component models for multimedia are used to address the diversity & heterogeneity of technologies used for multimedia. For example, the display of streaming video can be based on one of many different video compression/decompression technologies. Commercial frameworks for media software components, such as Microsoft DirectX, Apple Quicktime [17], Video4Linux2 [18] & the Java Media Framework (JMF) [19], have all evolved to address this technological diversity.

These existing frameworks provide an impressive array of capabilities that will accelerate the use of continuous media in common applications. They address the issue of orchestrating diverse media hardware sound devices, video capture devices, & competing compression/decompression technologies. They also improve system support for the real time nature of continuous media: they provide mechanisms for time management & stream synchronization & also introduce low overhead communication channels to media hardware necessary to cope with the extremely high volume of data transfer that accompanies continuous media.
DirectShow Filters

The basic building block of DirectShow is a software component called a filter. A filter usually performs a single operation on a multimedia stream. For example, there are filters that read files, get video from a video capture device, decode stream formats such as MPEG video & pass data onto the graphics card. To perform a given task, an application connects several filters so that

![Filter Graph Diagram]

Figure 1: DirectShow filter graph for playing an AVI file.

the output from one filter becomes the input for another. A set of connected filters is called a filter graph. Figure 1 shows a filter graph for playing an AVI file.

Filters are categorized into three broad headings:

1. **Source filters** present the raw multimedia data for processing. They may get it from a file on a hard disk, or from a CD or DVD drive, or from a “live” source such as a television receiver card or a capture card connected to a digital camera.

2. **Transform filters** accept either raw or partially processed data & process it further before passing it on. They include parsers that split raw byte streams into samples or frames, compressors & decompressors, & format converters.

3. **Renderer filters** generally accept fully processed data & play it on the system’s monitor or through the speakers or possibly through some external device. Also included in this category are "file-writer" filters that save data to disk or other persistent storage.

One of the main advantages of DirectShow is that already included are a range of filters for playing, converting, & capturing many different media formats. Popular codecs such as MPEG-1 & ASF are included. Many 3rd party developers have built their own custom filters for other formats. Of note are the advanced DVD movie player filter from InterVideo, & the real-time Motion JPEG codec from Pegasus Imaging.

**Filter Graph Manager**

You do not have to construct the graphs yourself as DirectShow provides a Filter Graph Manager. When you instantiate two filters that cannot join together due to incompatible formats, the Filter Graph Manger will search the registry for intermediate filters & construct the graph you intended. The Filter Graph Manager also controls the flow of data through the graph & passes event notifications on to you.

**Pins**

The multimedia data in a filter graph moves downstream from a source filter through zero or more transform filters & finally to a renderer filter. Pins handle the low-level details of the data transfer between filters. A pin represents the point of connection with another filter. Pins know what media types they can support & they negotiate the media type when two filters initially connect.
Media Types & Allocators
A media type represents an interchange format between filters. They can be physical format (e.g. YUY2 Video), or logical (representing a hardware connection or relationship). Physical media types use memory to transfer media samples downstream. These memory buffers (frame buffers when referred to images) are created & destroyed by DirectShow’s Allocators. At any point a developer may call for a new buffer to be allocated, or add a reference to a current buffer so it is not destroyed when it reaches the end of the filter graph.

Clocks
In any operation related to multimedia, it is vital to synchronize the samples so that video frames & audio samples are displayed at the proper rate. A DirectShow filter graph contains exactly one clock that all filters use to either timestamp, process or render media samples. The Filter Graph Manager selects a clock & instructs all the filters to use that clock as the synchronization source.

Base Classes
DirectShow is a component framework that merely defines interfaces for filter components to guarantee their interoperability, & implements the streaming filter graph system to aid application developers. It does not provide any functionality to aid developers create filter components. This is left to a further package, the DirectShow Base Classes. These define (rather minimal) templates for popular filter types (source, transform, renderer), together with classes that extend some of Directshow’s interfaces & structures to provide methods that aid in filter development.

VfW & WDM Capture Drivers
Microsoft released Video for Windows 1.x in November 1992 for Windows 3.1 & optimised it for capturing movies to disk. The Video for Windows architecture lacks features important to video conferencing, television viewing, capture of video fields, & additional data streams such as vertical blanking interval (VBI). 3rd parties have extended Video for Windows by implementing proprietary extensions.

WDM (Windows Driver Model) video capture was designed to resolve these problems. One WDM driver works on Windows 98, NT & 2000 platforms. There is a single driver architecture for hardware, including support for TV tuners, fields & VBI. But most importantly, there is direct synergy between DirectShow & WDM. Figure 2 illustrates a WDM Video Capture driver – notice the DirectShow Filter Graph!
The MMX technology [6] is designed to accelerate multimedia & communications applications by including new instructions & data types that allow applications to achieve a new level of performance. It exploits the parallelism inherent in many multimedia & communications algorithms, yet maintains full compatibility with existing operating systems & applications. A wide range of software applications, including graphics, MPEG video & music synthesis show many common, fundamental characteristics [7]:

- Small integer data types (for example: 8-bit pixels, 16-bit audio samples).
- Small, highly repetitive loops.
- Frequent multiplies & accumulates.
- Compute-intensive algorithms.
- Highly parallel operations.

The MMX technology is designed as a set of general-purpose integer instructions. The highlights of the technology are:

- Single Instruction, Multiple Data (SIMD) technique.
- 57 new instructions.
- 8 64-bit wide MMX registers.
- 4 new data types.

MMX technology introduces four new data types: three packed data types & a new 64-bit entity. Each element within the packed data types is an independent fixed-point integer. The architecture does not specify the place of the fixed point within the elements, because it is up to the developer the control of its place within each element throughout the calculation. The four MMX technology data types are:

- Packed byte -- 8 bytes packed into one 64-bit quantity.
- Packed word -- 4 16-bit words packed into one 64-bit quantity.
- Packed doubleword -- 2 32-bit double words packed into one 64-bit quantity.
- Quadword -- one 64-bit quantity.

The degree of parallelism that can be achieved with the MMX technology depends on the size of data, ranging from 8 when using 8-bit data to 1, i.e. no parallelism, when using 64-bit data.

The MMX technology is integrated into Intel x86 architecture in a way that maintains full compatibility with existing operating systems [8]. Aliasing MMX registers & state upon the x86 floating-point registers & state obtain this (figure 3). This does not preclude applications from executing both MMX routines & floating point routines, but the developer cannot interleave MMX & floating point instructions.

Figure 3: MMX registers aliased upon the floating-point registers.
The MMX instructions cover several functional areas including:

- Basic arithmetic operations such as add, subtract, multiply, arithmetic shift & multiply-add.
- Comparison operations.
- Conversion instructions to convert between the new data types: pack data together, & unpack from small to larger data types.
- Logical operations such as AND, AND NOT, OR & XOR.
- Shift operations.
- Data transfer instructions for MMX register-to-register transfers, or 64-bit & 32-bit load/store to memory.
- State management instruction to handle MMX to floating point transitions.

The AMD 3D Now! technology provides 21 additional instructions to support high-performance 3D graphics & audio processing. The 3D Now! instructions are vector instructions that operate on 64-bit registers, divided into two 32-bit single-precision floating-point words that are compatible with the IEEE-754, single-precision format. 3D Now also includes a single extra integer based instruction for averaging 2 vectors.

Intel SSE is a competitor to 3D Now! The Streaming SIMD Extensions enhance the Intel x86 architecture in four ways:

- 8 new 128-bit SIMD floating-point registers that can be directly addressed;
- 50 new instructions that work on packed floating-point data;
- 8 new instructions designed to control cacheability of all MMX & 32-bit x86 data types, including the ability to stream data to memory without polluting the caches, & to prefetch data before it is actually used;
- 12 new instructions that extend the MMX instruction set.

The 12 new MMX instructions are mostly specialized to particular subjects, e.g. the instruction for calculating the summed absolute difference is used mainly in block matching motion estimation. However, useful instructions for general video-processing use are `pavghw`, for averaging 2 vectors (same as 3D Now!) & `pmin /pmax` for calculating the pairwise minimum & maximum respectively (useful for sorting).

The Intel Performance Primitives provide a set of low-level image & signal manipulation functions. The functions are optimized for the Intel architecture, & are particularly effective at taking advantage of MMX & ISSE technology. Unfortunately despite the fact the AMD Athlon supports the whole iSSE instruction set, Intel sees it fit for them not to be used on AMD platforms. The Primitives contain functions that perform vector & image manipulation, conversion, filtering, windowing, thresholding & transforms, as well as arithmetic, statistic, & morphological operations.

Whilst the Intel Performance Primitives are expressive enough to perform most video processing tasks, they are not flexible enough to perform all tasks at optimum efficiency. Because of this, & its limitations on the AMD architecture, the Primitives are used in this project mainly for experimentation & prototyping use.
The Hauppauge WinTV SDK was originally created to supplement the Video for Windows driver &
provide the functionality ViW does not have. These include an API for the TV tuner, crossbar &
audio/video adjustment. The SDK was last updated in November 1997, & it is showing its age.

The SDK provides 4 methods & a large amount of structures. It is very much a message driven
architecture. You can initialise or free the TV Card, & send get or set requests to it.

AVHdl WINAPI AV_InitByDetail( WORD wBoardType, DWORD dFeatures, WORD wGetType, DWORD dMisc) ;
DWORD WINAPI hcwAV_Free ( AVHdl hdl ) ;
(DWORD WINAPI hcwAV_Get ( DWORD dDetail, AVFunction avFunc, PVOID dQualifier) ;
(DWORD WINAPI hcwAV_Set ( DWORD dDetail, AVFunction avFunc, PVOID dQualifier) ;

First you obtain a handler to an audiovisual device (AVHdl) with the
AV_InitByDetail method. AVFunction defines a large enumeration of possible messages, each one allowing you to get or set
the contents of a structure (message dependent), which you must send as dQualifier. For example,
to tune to a different channel, you must initialise an AVTune structure with the required channel,
country code, input source etc. & pass it in a call to hcwAV_Set with message type AVF_Tune. To
finalise, you must call the hcwAV_Free method to release the device for other applications’ use.
for more information, please see [14].

### NOISE REDUCTION

Noise reduction is an estimation problem. From an image \( I_c \) corrupted by unknown noise \( q \),
reproduce the original image \( I \). The two most common decompositions are additive (1) &
multiplicative (2) [9]:

\[
I_c(\cdot) = I(\cdot) + q(\cdot) \quad (1) \quad \quad \quad I_c(\cdot) = I(\cdot)q(\cdot) \quad (2)
\]

Gaussian noise is the most frequently occurring additive noise in video sequences. It is widely
used to model thermal noise & also approximates film grain & photon counting noise, since one
of the properties of the Gaussian distribution is the Central Limit Theorem. Its univariate
probability density distribution with mean \( \mu \) & variance \( \sigma^2 \) is given below:

\[
q(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (3)
\]

It is a characteristic bell shaped curve which extends to both \(+\infty \& -\infty\). But since an image is an
intensity map, values must be non-negative \& so the model is truncated at 0.

### EARLY NOISE REDUCTION TECHNIQUES

2D spatial noise filtering relies on the notion that Gaussian noise is mostly high frequency;
therefore a low pass filter will reduce it. Filters designed for this purpose include Gaussian filters,
Median smoothers & Morphological open-close filters. These methods have the side effect that
the original image is low-pass filtered as well, loosing definition.

We will assume a zero-mean additive white noise model. The noise is zero-mean if the average of
R arbitrary noise samples [9]
as \( R \) grows large. The term ‘white noise’ is an idealised model for noise that has, on average, a broad spectrum. Therefore the Gaussian distribution of this noise has a 0 mean but a high variance.

The goal of noise reduction is to recover an image that resembles the original image closely by reducing the noise image. But neither the noise image nor the original image is known. The solution is to attenuate each frequency in the noisy image by the probable amount of noise present, from the Gaussian distribution.

Luckily, to avoid a DFT (Discrete Fourier Transform) step, convolution is equivalent to multiplication in the frequency domain. Also, a broad Gaussian in the frequency domain is equivalent to a narrow Gaussian in the spatial domain. Therefore we can see that convolution with the following mask performs a Gaussian low-pass filter:

\[
\begin{array}{ccc}
\frac{1}{36} & \frac{1}{9} & \frac{1}{36} \\
\frac{1}{9} & \frac{4}{9} & \frac{1}{9} \\
\frac{1}{36} & \frac{1}{9} & \frac{1}{36}
\end{array}
\]

\[
\begin{array}{ccc}
\frac{1}{36} & \frac{1}{9} & \frac{1}{36} \\
\frac{1}{9} & \frac{4}{9} & \frac{1}{9} \\
\frac{1}{36} & \frac{1}{9} & \frac{1}{36}
\end{array}
\]

\[
\text{NOISE REDUCTION ALGORITHM 1: WEIGHTED RECURSIVE TEMPORAL AVERAGING}
\]

Gaussian noise in video sequences has no intrinsic motion. Object motion increases the object’s frequency & therefore it’s susceptibility to spatial low-pass filtering. 2D noise reduction therefore blurs fast motion. So it is better to consider the processing of image sequences as a full 3D problem rather than a repetitive 2D one [10]. The simplest temporal filter carries out a weighted averaging of successive frames. That is, the restored image sequence is obtained by [9]:

\[
R(x, y, t) = \sum_{l=1}^{2K+1} H(l) I(x, y, t-l)
\]

(5)

Here \( H(l) \) are the temporal filter coefficients used to weight \( 2K+1 \) consecutive frames, \( I \) is the noisy video sequence & \( R \) is the noise reduced video sequence. A disadvantage to this method is that you have to buffer several frames of an image sequence. Also, recursive filters are easier to adapt since there are fewer parameters to control. A recursive temporal filter, then, follows the form:

\[
R(x, y, t) = (1-\alpha)R(x, y, t-1) + \alpha I(x, y, t)
\]

(6)

The noise-reduced frame is a weighting of the current noisy frame & the previous noise reduced frame. The filter gain \( \alpha \) is controlled by the observation of motion for that pixel. Decreasing \( \alpha \) under large motion increases the weighting for the current frame. Though this reduces the noise reduction, it also reduces the filter’s comet trailing artifacts that would otherwise result.

To reduce the computation time for cheap implementation in software, the filter gain is controlled by a switching filter [9]. If the difference between the previous pixel & the current is greater than a threshold, the filter is switched off:

\[
\alpha(x, y, t) = \begin{cases} 
1 & \text{if } |R(x, y, t-1) - I(x, y, t)| > \epsilon \\
\frac{1}{4} & \text{otherwise}
\end{cases}
\]

(7)
A disadvantage of recursive temporal noise reduction is that it assumes the previous frame is perfectly noise reducer. This is very unlikely to be the case. This point reduces the effectiveness of the technique, but it doesn’t negate it totally.

Noise Reduction Algorithm 2: Adaptive Temporal Averaging

A more robust motion detector would give more confidence in applying stronger noise reduction techniques. A block based motion detector increases the amount of pixels used in the motion detecting process, therefore from (4) it follows that the noise element will be nearer to 0 mean. However, it also follows that moving details (high frequency) will also have less chance of being detected.

Each block in the current frame is compared with the same block in the previous frame for motion. The criterion is the MSE. The higher the value of the MSE, the higher the likelihood of motion. The MSE is linearly translated into a weight between 0 & 1 for each block:

\[
\alpha(x, y, t) = \frac{\text{MSE}(I(x, y, t), I(x, y, t - 1))}{\epsilon}
\]

\[
\alpha(x, y, t) = \begin{cases} 
1 & \text{if } \text{MSE}(I(x, y, t), I(x, y, t - 1)) > \epsilon \\
\alpha(x, y, t) & \text{otherwise}
\end{cases}
\]

Where \( \epsilon \) is a sequence wide parameter. For performance reasons, the recursive algorithm (6) is used for the noise reduction.

A post-processing erosion step performed on the motion detector map would smooth out blocking artifacts, but this is not feasible in real-time processing due to difficulty in vectoring this step.
DEINTERLACING

Human vision is less sensitive to ickering details than to large-area ickers [3]. Television displays are interlaced to take advantage of this for bandwidth reduction, while broadcast formats were originally defined to match the display scanning rate. The most prevalent broadcasting formats all employ interlacing (NTSC, PAL, HDTV).

The major flaw of interlacing is that it complicates many image processing tasks. The question is to interlace or not to interlace. This issue divides the TV & PC communities [11]. The latter are biased towards the opinion that modern technologies are powerful enough to produce progressively scanned video at high rate, so traditionally the PC community has almost ignored the deinterlacing problem – see Microsoft’s half-hearted paper on ‘state of the art’ PC deinterlacing techniques [12]. Given the expertise available in the TV community, it seems remarkable that PC’s still rely on deinterlacing technologies developed for use in the television chain in the 70s.

The TV world is more conservative. Their opinion is that present day technologies are powerful enough to deinterlace video material acceptably [13], which negates the need to introduce incompatible standards & sacrifice the investments of its customers. The recent HDTV standard proposes an interlaced format of 1920x1080 in 60 fields per second. Interlacing & Deinterlacing is therefore not a relic of the past.

Interlaced video can more formally defined as the vertical (spatial) sub-sampling without pre-filter of each frame depending on its temporal location [11] – each pair of fields contains the odd & then the even scan-lines. The de-interlacing problem is to reconstruct the sequence of video frames from these incomplete fields. In the following the interlaced sequence \( I \) is de-interlaced using itself & the interpolated missing data \( I_i \).

\[
I(x,y,t) = \begin{cases} 
I(x,y,t), & (y \mod 2 = n \mod 2) \\
I_i(x,y,t), & \text{otherwise}
\end{cases} \tag{12}
\]

Standard Televisions use a CRT coating of medium-persistent phosphors as a de-interlacing solution. Effectively, \( I(x,y,t) \) is a recursive addition of the same pel in previous fields, exponential decayed. This interpolation has the disadvantage of reduced spatial-temporal frequency response, or blurring.

The PC community uses CRTs with far higher refresh rates & less persistent phosphors; so physical de-interlacing is not an option. Currently the most prevalent techniques in PCs are Bob & Weave.

Weave is used out of basic necessity by simple video viewers such as the Hauppauge WinTV 2000 system et al. The missing data from one field is taken directly from the previous field:

\( I(x,y,t) = I(x,y,t-1) \). Whilst the best solution for stationary images, moving objects are not shown at the same position for odd & even lines of a single output frame. This causes serration of moving edges, as shown in figure 4.

Whilst Weave is a purely temporal filter, Bob is purely a spatial filter. Each \( I(x,y,t) \) is a linear interpolation between \( I(x,y-1,t) \) & \( I(x,y+1,t) \). This eliminates motion artifacts, but with a substantial loss of vertical resolution. However, human vision is less sensitive to a reduction in...
the frequency range than increased aliasing. See figure 5. Some PC Multimedia viewers give Bob as an option.

Better results should be obtained by filter selection from prior motion detection. Motion adaptive algorithms can move gracefully from temporal to spatial interpolation as motion increases.

\[ I(x, y, t) = aF_{\text{st}}(x, y, t) + (1-a)F_{\text{mov}}(x, y, t) \] (13)

\( \alpha \) is determined by a motion detector. \( \alpha = 0 \) means no motion & \( \alpha = 1 \) means very fast motion. \( F_{\text{st}} \) represents a deinterlacing function optimised for stationary parts of the image (usually weave), & \( F_{\text{mov}} \) represents a deinterlacing function for moving parts. A suitable motion detector can be a SAD (summed absolute difference) or MSE (mean squared error) between blocks in the current & last field or frame. Note that this detector is susceptible to blocking artifacts & therefore a post-processing step of smoothing is carried out on the detector map.

It must be pointed out that adaptive techniques do not lend themselves well to MMX vectoring. Each pixel must be tested individually for its motion weight \( \alpha \) to avoid blocking artifacts. Therefore we end our discussion of adaptive methods for this project, & concentrate on linear & implicitly adaptive methods to come.

A most popular method, incorporated into ICs [14] due to its low complexity & superior properties at vertical edges, is Median filtering - an implicitly adaptive method. The interpolated samples are found as the median luminance value of the 2 vertical neighbors & the temporal neighbor in the previous field (a 3 tap filter):

\[
I(x, y, t) = \begin{cases} 
I(x, y, t), & (y \text{ mod } 2 = n \text{ mod } 2) \\
\text{med}(I(x, y-1,t), I(x, y+1,t), I(x, y,t-1)) & (\text{otherwise})
\end{cases} \] (14)

Preferably noise reduction should be employed before Median filtering to avoid noise breakthrough near edges [15].
A vertical-temporal interpolation filter would theoretically solve the de-interlacing problem, if the signal were bandwidth-limited prior to interlacing [16]. The required pre-filter would be similar to a vertical up-conversion filter. Although it is missing, the VT interpolator combines the benefits from weave & bob, in that it prevents alias & blur in stationary images. Vertical detail is gradually reduced with increasing temporal frequencies.

The filter is designed so that the contribution from neighbouring fields is limited to the higher vertical frequencies [17]. Because of this, motion artifacts are absent for objects without vertical detail that move horizontally. From experimentation [17], the following linear combination is calculated for each missing pixel:

\[
I(x, y, t) = \frac{1}{18} (8I(x, y + 1, t) + 8I(x, y - 1, t) + I(x, y + 3, t) + 8I(x, y - 3, t) + 10I(x, y, t - 1) - 5I(x, y + 2, t - 1) - 5I(x, y - 2, t - 1))
\]

See figure for an illustration. However, this is poor for MMX vectoring performance, as we cannot reduce the equation to shifts, additions & subtractions - most notably the division by 18 is a big problem. The filter below approximates the above filter:

\[
I(x, y, t) = \frac{1}{16} (8I(x, y + 1, t) + 8I(x, y - 1, t) + 9I(x, y, t - 1) - \frac{1}{3} I(x, y + 2, t - 1) - \frac{1}{3} I(x, y - 2, t - 1))
\]

Unfortunately with VT deinterlacing, at vertical edges degradation can become visible. Lethonen & Renfors [18] combine a VT filter with a 5-point median. The output of the VT filter is one of the inputs of a five-point median. The remaining four inputs are the nearest neighbours on the VT sampling grid:

\[
I(x, y, t) = \text{MED}[I(x, y + 1, t), I(x, y - 1, t), I(x, y + 1, t), I(x, y, t + 1), I(x, y, t - 1), F_{vt}(x, y, t)]
\]
The 5-point median acts as a safety. If the candidate pixel intensity from the VT filter is wildly different from its nearest neighbours, a neighbour will be chosen instead of it. Whilst this removes aliasing artifacts, it does so at the expense of detail. Also, it is possible that a correct VT candidate will not be chosen when it is outside a possible monotonic interpolation of either its temporal neighbours xor its spatial neighbours. This could happen if the scene exhibits high temporal or spatial frequencies respectively.

So the median does not replace a better deinterlacing filter, but helps a poorer one. The question is, can we find a better real-time deinterlacing filter?

The de-interlacing algorithms discussed so far can only profit from information captured in the vertical direction &/or temporal direction. Especially at diagonal edges it can be advantageous to profit from the horizontal spatial direction as well [19]. If intra-field interpolation is necessary because of motion, then edge-orientation dependent algorithms can at least preserve low frequency spatial information through inspection of the diagonal (temporal-horizontal) edge gradient. The aperture of the edge-dependent deinterlacer is given in figure 2.26:

Oh et al. [20] suggest linearly interpolating between the pixels of minimum gradient, where the gradient is tested in 6 orientations:

\[
\begin{align*}
U_a &= |a - f|, \quad U_b = |b - e|, \quad U_c = |c - d| \\
U_a &= |i - n|, \quad U_b = |j - m|, \quad U_c = |k - l|
\end{align*}
\]

\[
\begin{align*}
X_a &= \frac{a + f}{2}, \quad X_b = \frac{b + e}{2}, \quad X_c = \frac{c + d}{2} \\
X_a &= \frac{i + n}{2}, \quad X_b = \frac{j + m}{2}, \quad X_c = \frac{k + l}{2}
\end{align*}
\]

\[
F_{step}(x,y,n) = \begin{cases} 
X_a, & U_a = \text{MIN}(U_a, U_b, U_c, U_d, U_e, U_f) \\
X_b, & U_b = \text{MIN}(U_a, U_b, U_c, U_d, U_e, U_f) \\
X_c, & U_c = \text{MIN}(U_a, U_b, U_c, U_d, U_e, U_f) \\
X_d, & U_d = \text{MIN}(U_a, U_b, U_c, U_d, U_e, U_f) \\
X_e, & U_e = \text{MIN}(U_a, U_b, U_c, U_d, U_e, U_f) \\
X_f, & U_f = \text{MIN}(U_a, U_b, U_c, U_d, U_e, U_f)
\end{cases}
\]

One noticeable step omitted from this procedure is that in [20] the video signal was low pass filtered before the edge gradients were tested. This makes the deinterlacer more robust at rejecting incorrect candidates due to noise.
Unfortunately this algorithm still requires a 5-point median safety to guard against incorrect candidates. A paper was released too late for implementation in this project [21], which offered more robust & accurate candidate selection. It is described in the proposed extensions section.

**Motion Estimation**

Except at scene cuts, the recording of each image in a video sequence occurs more rapidly than the change of information in a scene. We can take advantage of this temporal correlation between frames to reject more corruption than spatial methods can alone, or to compress the redundancies. Algorithms which track the motion of scene objects through the sequence can differentiate between details & fast motion (both are high energy phenomenon), & can interpolate the motion between frames. Motion Estimation [22] is the process of gauging this inter-frame motion, & the motion vectors generated specify the spatial displacement of each pixel or block or pixels in time (usually between one frame & the next). Motion Estimation is a useful tool in most video processing disciplines, including compression, artefact suppression, noise reduction, deinterlacing & format conversion.

Creating an image sequence model from the source, i.e. moving objects in a 3D world, is complex & currently intractable. Simple 3D object motion leads to translation, rotation & zooming in 2D projection. Analysing this is impractical so today the image sequence model assumes zooming & rotating between frames will be small & therefore approximated by translation:

\[
I_{\tau}(x, y, t) = I(x + d_1, (x, y, t), y + d_2, (x, y, t), t - 1)
\]

Where \((d_x, d_y) = d(x, y, t, t-1)\) is the displacement mapping the region in the current frame \(t\) into the previous frame \(t-1\) – the motion vector for the location \((x, y)\). This model avoids one important consideration – some objects will appear or become occluded as the sequence progresses. Also, only the luminescence data is available to track object motion, so its correlation with scene illumination is ignored.

The most popular technique for generating motion vectors is Block Matching (BM). Its basic assumption is that pixels in small regions undergo the same translational motion. The current frame is segmented into blocks of size \(N \times N\). Motion vectors are assigned to each block in turn by matching the block against one of the set of blocks in the previous frame (See figure 9). The smallest vector that can be estimated is determined by the size of a candidate block (integer accuracy), unless candidate blocks are extracted between locations (sub-pixel accuracy), by interpolation. Methods for determining the best matching block error include Mean Squared Error (MSE) & Summed Absolute Difference (SAD).

This technique arises from a direct solution of (21). The Displaced Frame Difference is defined below, where \(v\) is some displacement vector:

\[
DFD(x, v) = I_{\tau}(x) - I_{\tau}(x + v)
\]
The BM solution minimises the Mean DFD with respect to $v$ over the $N \times N$ block. An exhaustive search of all blocks is computationally demanding, but reducing the search space introduces the possibility of finding local minima. The BM algorithm is robust to noise in the same way block based motion detector was.
The project is implemented in Microsoft Visual C++ 6 under the Microsoft DirectShow framework.

- The Java Media Framework was deemed too slow for the real-time optimised algorithms that would be required. A work around involving implementing the video processing filters in C++ through a JNI interface was ascertained to be too risky to base a project around.
- The project was not implemented as a plugin for another system such as dScaler or VirtualDub due to the freedom & future direction restrictions this would bring.
- The algorithms are to be optimised for MMX processors initially, followed by SSE optimisations where time would allow. It was deemed through inspection of similar products that this could define the minimum type of processor capable to run the CPU-intensive algorithms.
- DirectShow was chosen, rather than Video for Windows, as Video for Windows is now unsupported by Microsoft, & there is no guarantee that future video capture sources or compression codecs will support it.
- It was deemed necessary to implement a DirectShow wrapper for the Hauppauge WinTV line of products, due to the lack of WDM drivers for any available TV card available to me at the start of the project, & as Hauppauge represent over 80% of sales of TV cards in Europe. This would guarantee a large user base of prospective customers.

Figure 10: Data & Control Flow overview diagram of the project
Figure 10 shows a data flow diagram (DFD) of the project. The Directshow Filter Graph is shown in grey. Video data moves downstream from the video input source to the graphics card overlay &/or hard disk. The TV Display application controls the parameters of each stage in the filter graph through the DirectShow public interfaces. The application then exports functionality to the user through a GUI. Not shown is the quality control data passed upstream from the video renderer /AVI file output filter to the source filter.

**HAUPPAUGE WINTV CARD FILTER**

Access to the Hauppauge WinTV Card is allowed with an ‘add-on’ transform filter that is intended to be connected directly to an output pin of the Video for Windows driver wrapper filter. The transform filter does not perform any processing of the video, but passes it on downstream unaffected.

It is a limitation of Microsoft DirectShow that a filter cannot interrogate an acquainted (in the same filter graph) Video for Windows wrapper for its underlying hardware device ID. Therefore we must recognize this fact when we come to adding the component to a filter graph, & carry out any preconditions before this happens. The most the filter does in this respect is to refuse to connect directly downstream from anything other than a Video for Windows capture source.

An application developer should use the `ICaptureGraphBuilder2::FindInterface` method to obtain a particular interface, such as TV Tuner support, from the graph, rather than query a specific filter for the interface. This is necessary for retrieving interfaces from a WDM capture source, as it is the job of `FindInterface` to dynamically add the WDM tuner, audio, crossbar etc. filters upstream of the capture source to the filter graph. `FindInterface` traverses the filter graph searching for the required interface. Because of this, after the special initialization routine remarked upon above, access to the Hauppauge WinTV Card filter occurs in the same homogeneous way as any other WDM filter.

The tuner filter uses the object adaptor design pattern [23] to morph the Hauppauge WinTV SDK interface to that of DirectShow. The tuner filter component exhibits all the functionality (exports all the interfaces) that a WDM video capture driver has above & beyond a Video for Windows capture driver.

![Figure 11: Hauppauge VFW capture driver connected to the Hauppauge TV Card Filter](image)

**INTERFACES**

The Video for Windows drivers export an API that only supports capturing of video. There is no API for other features of capture cards, such as TV tuners, crossbars (input/output switching) & audio configuration. The VFW mechanism for publishing these features is through the property pages of the driver. To upgrade the Hauppauge VFW driver to DirectShow functionality, we must export the following DirectShow interfaces to emulate a WDM capture source:
IAMTVAudio

- **GetHardwareSupportedTVAudioModes**
  Retrieves a bitmask of the formats available in the hardware.
  *This is a composition of the TVAudioMode enumeration: mono, stereo, language A, language B & language C.*

- **GetAvailableTVAudioModes**
  Retrieves a bitmask of the possible modes.

- **get_TVAudioMode**
  Retrieves the current TV audio mode.

- **put_TVAudioMode**
  Sets the current TV audio mode.

IAMTVTuner

- **put_Channel**
  Sets the TV channel.

- **get_Channel**
  Retrieves the TV channel that the encoder is currently set to.

- **ChannelMinMax**
  Retrieves the highest & lowest channels available.

- **put_CountryCode**
  Sets the country code to establish the frequency to use.

- **get_CountryCode**
  Retrieves the country code that establishes the current channel-to-frequency mapping.

- **put_TuningSpace**
  Sets a storage index for regional channel-to-frequency mappings.

- **get_TuningSpace**
  Retrieves the storage index for regional fine tuning.

- **Logon**
  Logs a user onto the system.

- **Logout**
  Logs a user off the system.

- **SignalPresent**
  Retrieves the strength of the signal on a given channel.

- **put_Mode**
  Sets the mode of operation for a multifunction tuner.

- **get_Mode**
  Retrieves the mode of operation for a multifunction tuner.

- **GetAvailableModes**
  Determines which video modes are available on the tuner.

- **RegisterNotificationCallBack**
  Allows an object that implements the IAMTunerNotification interface to receive event notifications when the tuner changes state.

- **UnRegisterNotificationCallBack**
  Unregister an object for notification of events.

- **get_AvailableTVFormats**
  Retrieves all the analog video TV standards that the tuners support.

- **get_TVFormat**
  Retrieves the current analog video TV standard in use.

- **AutoTune**
  Scans for a precise signal on the channel’s frequency.

- **StoreAutoTune**
  Saves the fine-tuning information for all channels.

- **get_NumInputConnections**
  Retrieves the number of TV sources plugged into the tuner filter.

- **put_InputType**
  Sets the tuner input type (cable or antenna).

- **get_InputType**
  Retrieves the input type.

- **put_ConnectInput**
  Sets the hardware tuner input connection.

- **get_ConnectInput**
  Retrieves the hardware tuner input connection.

- **get_VideoFrequency**
  Retrieves the current video frequency.

- **get_AudioFrequency**
  Retrieves the current audio frequency.

IAMCrossbar

- **get_PinCounts**
  Retrieves the number of input & output pins.

- **CanRoute**
  Determines if the crossbar filter can route the analog or digital signal.

- **Route**
  Routes an input pin to an output pin.

- **get_IsRoutedTo**
  Retrieves the input pin connected to a given output pin.

- **get_CrossbarPinInfo**
  Retrieves a pin that has audio or video data relating to a given pin.
IAMVideoProcAmp

Enables applications to adjust the qualities of the incoming video signal such as brightness, contrast, hue, saturation, gamma, & sharpness. It defines a uniform range for these settings regardless of whether the adjustment is made in the analog or digital domain.

GetRange

Retrieves minimum, maximum, & default values for setting properties. The VideoProcAmpProperty enumeration specifies the property. Values can be: Brightness, Contrast, Hue, Saturation, Gamma, Color Enable, White Balance, Backlight Compensation & Gain

Set

Sets video quality for a specified property.

Get

Retrieves video quality for a specified property.

IAMAnalogVideoDecoder

This interface specifies & retrieves information about the process of video. It contains methods for selecting the digitization format, indicating the horizontal lock status, & controlling the time constant on the digitizer phase lock loop (PLL).

get_AvailableTVFormats

Retrieves the currently available analog video formats.

put_TVFormat

Sets the current analog video format.

get_TVFormat

Retrieves the current analog video format.

get_HorizontalLocked

Determines whether the horizontal sync is locked.

put_VCRHorizontalLocking

Sets the condition for being connected to a VCR (changes PLL timing).

get_VCRHorizontalLocking

Determines whether horizontal locking has been set.

get_NumberOfLines

Retrieves the number of horizontal lines in the current video signal.

put_OutputEnable

Enables or disables the output bus.

get_OutputEnable

Determines whether the output bus is enabled.

Not all the methods shown above are applicable to a Hauppauge TV Card, & neither do they have to be. Also, in some cases the Hauppauge WinTV SDK does not support the functionality required, despite the TV Card or bt8x8 chip supporting it. A known way to report this is by returning a E_NOTIMPL Hresult to the client, since the client must always guard against this for the above reasons.

The WinTV SDK provides a C style package of get & set methods that take &/or return structures. To avoid polluting the global namespace of the project, the main WinTV SDK header file is encapsulated in the namespace ‘NWTVSDK’.

Like Video for Windows, WDM driver include property pages, so that application developers can support the basic functionality of the components they include without writing a GUI around the component’s API. It also facilitates the plugin architecture, as the developer need not know which components he supports before his application is deployed.

Video for Windows capture sources exports the following functionality in their property pages:

- Video Source:
  - Video crossbar
  - Video processing controls: Brightness, Contrast, Hue, Saturation
  - Video standard e.g. NTSC, PAL, SECAM
- Video Format:
  - Image format e.g. RGB 8bit, 16bit, 24bit, YUY2, YV12
  - Video dimensions
- Video Display (not applicable)
  - Camera control
Not wishing to re-invent the wheel, the Hauppauge TV Card Filter exports the following:

- **TV Tuner:**
  - Country selection
  - Input type: Cable, Antennae
  - Channel
- **TV Audio:**
  - Input source: Tuner, Aux
  - Input type: Mono, Stereo, Languages A, B or C.

The WinTV SDK heralds from the days of Windows 3.1 & ANSI C programming. To convert the WinTV SDK interface to the DirectShow SDK, I employed the Adapter (or Wrapper) design pattern. An adapter converts the interface of a class into another interface clients expect. It lets classes work together that couldn’t otherwise because of incompatible interfaces.

A UML diagram of the namespace adapter, a modified object adapter from, is given below:

![UML Diagram](image)

- The target defines the domain-specific interface that the Client uses - in our case the relevant DirectShow interface e.g. `IAMTVTuner`.
- The client is the DirectShow application.
- The adaptee is a namespace containing the legacy interface that needs adapting – the Hauppauge WinTV SDK.
- The adapter inherits from target, implementing the target interface using the adaptee’s API.

Ideally, we would have used a true object adapter in this situation, as wrapping the WinTV SDK in an object would have made its methods totally private to classes that do not & should not use them. Unfortunately, the WinTV SDK could not be encapsulated in an object without heavy modification of the supplied include file, due to a deficiency in Microsoft Visual C++’s ANSI C++ implementation (Visual C++ does not allow you to initialise attributes inside the class definition). Therefore it was decided that encapsulating the WinTV SDK in a namespace would be the next best option.
The Video Processor Filter consists of a noise reduction / deinterlacing ‘Strategy’ hierarchy, a flip field detector to allow legacy Video for Windows devices to be used with this project, & an image buffer to give the strategies the video input requirements they need. Equivalent exported interfaces & property pages for strategy selection complete the filter.

YUY2 was chosen as the video input & output format for the filter. YUY2 is a packed 4:2:2 format. The image is stored top-down. The Y (Luminescence) component is fully sampled. The U & V (Chromaticity) components are subsampled by a factor of 2 in the horizontal direction, & fully sampled vertically. The data is stored as (Y0-U0-Y1-V0)(Y2-U2-Y3-V2)(Y4-U4-Y5-V4) & so on.

The motivation for using YUY2 is as follows:

- It is the most popular output format for analog video input sources, as it mirrors the sampling components of PAL & NTSC video.
- It is the most widely supported format for graphics card overlays.
- It is Microsoft PC99 complaint.
- Rarely do the algorithms affect the colour components of the video. The human eye is less sensitive to colour than luma, therefore it is very beneficial for performance to have the luminescence data as a separate component (unlike RGB). The algorithms can concentrate on manipulating the luma values only.
- Whilst it might appear that a planar format, such as YV12, would be more efficient, there are two problems with this:
  - There is no standard planar format that is 4:2:2, only 4:2:0 & below. So we will be throwing colour information away needlessly.
  - Some of the MMX & SSE instructions only function on packed words, not packet bytes. So by using a planar format we would be restricted to a subset of the MMX instructions, or suffer the performance loss of expanding the luma values back out to words.

Media samples received from video source allocators are accompanied by an AM_MEDIA_TYPE header [24]. This structure stores statistics on the sample such as its major & minor types (e.g. YUY2 Video), the size of the sample & a pointer to a format block. The format block contains more detailed statistics relevant to major type of the sample. For uncompressed video (audio & MPEG video are alternatives) the format block is either a VIDEOINFOHEADER or VIDEOINFOHEADER2 structure.

Video for Windows & other legacy capture sources use the VIDEOINFOHEADER structure, which does not include support for interlaced video. Instead, the video is pre-weaved into 25Hz progressive video at source. This would not be a problem if we knew which field came first temporally in the woven sequence (is the even or the odd field the flip field?). Alas this is not the case. When the VfW source filter cannot process a field (perhaps due to not enough CPU time), it will skip that field, & the flip field will change.

This is a real problem for the deinterlacing task. Fields deinterlaced out of sequence will cause very noticeable judder. The remedy for this is a heuristic to choose the most likely flip field each time a new frame comes in. Given J(x,y,t), a pre-weaved sequence of frames, we calculate the distance \( \alpha \) between the latest frame’s odd field & the previous frame’s bobbed odd field derived
only from its even field. In the same way we obtain the distance $\beta$ between the latest frame's even field & the previous frame's bobbed even field derived only from its even field:

$$\alpha = \sum_{y=\text{mod}=1} J(x,y,t) \cdot \frac{J(x,y+1,t-1)+J(x,y-1,t-1)}{2}$$

$$\beta = \sum_{y=\text{mod}=0} J(x,y,t) \cdot \frac{J(x,y+1,t-1)+J(x,y-1,t-1)}{2}$$

(23)

Flip Field = \begin{cases} 
  \text{Even,} & \text{if } \alpha > \beta \\
  \text{Odd,} & \text{otherwise}
\end{cases}

The flip field is even if $\alpha > \beta$, otherwise the flip field is odd. Out of bounds parameters of $J$ are bounded to the nearest value in bounds. The motivation behind this method is better shown diagrammatically in figure 13.

The idea behind using a spatial bob deinterlacer for interpolation here is that we are deriving the temporal information, & therefore we cannot use it in the algorithm itself. The algorithm could have been made more efficient with assumptions of the flip field of the previous frame, but we have no information on how stale it is (how many fields the video driver skipped), & we would also need a 2nd algorithm for the very first frame in the video sequence.

The algorithm therefore requires 2 ‘new’ frames before a new flip field is detected. Note the algorithm could give the wrong result if there is noise at low contrast levels, or if the global motion of the scene reverses direction sharply.

![Diagram of flip field detection algorithm](image)

Figure 13: Flip field detection algorithm. If the distance $\alpha$ is greater than the distance $\beta$ then the flip field is even, otherwise the flip field is odd.
As you may recall from the research chapter, the video processing algorithms implemented are:

- Weighted Recursive Temporal Averaging Noise Reduction
- Adaptive Temporal Averaging Noise Reduction
- Vertical-Temporal Deinterlacing
- VT with 5 Point Median Protection Deinterlacing
- 3D Edge Orientated Deinterlacing

The following are also implemented for comparison with traditional PC techniques:

- Spatial Gaussian Noise Reduction
- Weave Deinterlacing
- Bob Deinterlacing

In essence, the only requirement for each one of these algorithms is for it to be passed a buffer with the current frame & an empty buffer for it to write into. Each algorithm should be encapsulated & made fully interchangeable. There is no reason why the client should accommodate the specifics of each processing technique; therefore I developed a plugin architecture using the Strategy [23] design pattern (see figure 14).

- The Strategy declares an interface common to all deinterlacing algorithms. This interface is as follows:

  ```cpp
  CStrategy(CImageBuffer* pImageBuffer);
  virtual HRESULT FieldProcess(IMediaSample* pSample);
  virtual HRESULT Update();
  ```

  - The constructor passes the strategy an image buffer object. See the Image Buffer section for details.
- The ConcreteStrategies implement the above interface:
  - The video-processing algorithm is implemented in the FieldProcess method. The pSample parameter points to the output framebuffer.
  - Update is called when the input media type changes, for example if the resolution is modified. This method should update all attributes dependent on the media type.
- The Context class, in our case the filter, configures the strategy object & maintains a reference to it.

Now we can use polymorphism to execute a video-processing filter at run-time - the strategies eliminate the conditional statements we would have needed.

The abstract Strategy class was sub-classed with further abstract classes, to group families of related strategies that share common functionality. The common functionality was migrated to the abstract parent. For instance, deinterlacing strategies share the need for scan-rate up-conversion – they convert ½ rate progressive weaved video into full rate progressive video. The DeinterlaceStrategy class handles these details, whilst all deinterlacing strategies inherit from it. Also, most deinterlacing strategies only interpolate the missing data – each cycle then copies one
half (one field) of the data from the input buffer to the output buffer. This functionality becomes the LineStrategy class. Strategies may inherit from it & implement the LineProcess method, which is called for every missing line that has to be interpolated. See figure 15.

For the client application to select the intended video-processing algorithm, it must use a criterion such as name, quality or performance rather than the implementation class. Each strategy requires multiple implementations for different instruction sets. Also, the instruction set of the particular machine the program is running on is immutable. Therefore the instruction set should be detected once only, at initialisation time, & the strategies requested throughout the execution of the program should be implicitly selected from the family of strategies designed for that instruction set.

A Singleton object [23] MachineType is implemented that detects the instruction set capabilities of the host machine. Each strategy factory, defined by the Abstract Factory design pattern [23], is configured at initialization with the MachineType object. Creation of a strategy is directed through its factory, which guarantees the most optimal strategy is returned. See figure 16.

- StrategyA is a descendent of Strategy (not necessarily its child).
- StrategyA defers the implementation of FieldProcess to its children MMXStrategy & SSEStrategy.
- StrategyAFactory is initialised with the instance of MachineTypeSingleton, & calls GetInstructionSet to make a decision on which strategy to return on an invocation of CreateStrategy.

Figure 15: Video processing strategy hierarchy.
The strategy factory handles more than just strategy creation. Clients may query the factory for the strategy’s name, GUID & property page, if it exists, with invocations to GetName, GetGUID & CreatePropertyPage respectively. GetGUID returns the globally unique identifier for the strategy. This is for the proposed extensions of a strategy repository in the registry, to allow 3rd party authors to integrate strategies with this project without access to the source code. The CreatePropertyPage method returns a property page for the strategy, if the strategy requires any manual selection of parameters. If this is the case, a specialist COM interface for API use should also be created in parallel.

The image buffer provides a filter wide solution for storage & buffering of past input frames for use by the video-processing strategies. It also provides statistics on the current media type through its composition of the MediaHeader class.

Figure 16: Factory encapsulated selection of strategies dependent on machine type.

Figure 17: Video Processing Filter flowdiagram
The filter has the image buffer as a member. The image buffer is passed in the constructor of each strategy. When a new input frame is received, it is appended to the buffer, & the noise reduction strategy is called to process it. The deinterlacing strategy is then called for the 1st time. The output frame is then sent on to the downstream filter. The deinterlacing strategy is called a 2nd time with a new output for the scan-rate up-conversion to take place. Finally, this 2nd frame is passed on to the downstream filter.

The image buffer both stores past frames & buffers incoming ones so that the strategies can use future information in processing the current (time-delayed) frame.

The noise reduction strategy processes the video sequence in-place, whilst the deinterlacing strategy copies & processes the noise reduced frame into empty frames allocated by the frame allocator.

### INTERFACES

The video-processing filter exports interfaces for selection of noise reduction & deinterlacing strategies. The external representation of the noise reduction & deinterlacing strategies are identical, so the selection interfaces should be identical to. Unfortunately there is no way to export the same interface twice with COM, so a parent interface `IStrategyEnumeration` is defined, & two interfaces `IDeinterlaceStrategyEnumeration` & `INoiseReductionStrategyEnumeration` are exported, which add no further methods to the `IStrategyEnumeration` interface:

<table>
<thead>
<tr>
<th><code>IStrategyEnumeration</code></th>
<th>Provides methods for creating strategy factories that produce strategies, &amp; using these factories to set &amp; get the current strategy the filter uses to process the video.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateFactory</td>
<td>Creates a strategy factory.</td>
</tr>
<tr>
<td>SetFactory</td>
<td>Sets the strategy factory the filter uses to create the current strategy.</td>
</tr>
<tr>
<td>GetFactory</td>
<td>Gets the strategy factory the filter uses to create the current strategy.</td>
</tr>
</tbody>
</table>

Two property pages, that give the user a basic GUI for the `IDeinterlaceStrategyEnumeration` & `INoiseReductionStrategyEnumeration` interfaces are also included. They allow the user to select the current noise reduction & deinterlacing strategies.
The TV Display application is created as an MFC exe application.

The Microsoft Foundation Class Library (MFC) [25] is an application framework for programming in Microsoft Windows. Written in C++, MFC provides much of the code necessary for managing windows, menus, & dialog boxes; performing basic input/output; storing collections of data objects; & so on.

At its root, an MFC application consists of a main frame window, a single document & multiple views. The main frame displays the GUI of your application. The document contains & manages your application-specific data; provides an interface consisting of document data variables for manipulating the data; & participates in reading, writing & printing files. The view’s responsibilities are to display the document's data graphically to the user & to accept & interpret user input as operations on the document. A document can support many views, in this respect the architecture is akin to the Observer design pattern [23].

You may choose which of the 3 classes handle each of your application’s commands & messages, depending on their nature.

The main requirements of the application are:

- Selection of input source, either:
  - Legacy Video for Windows driver.
  - Hauppauge Video for Windows driver with TV Card filter extensions.
  - WDM Video Capture filter & auxiliary filters.
  - AVI from disk, or in general, any URL.
  - Therefore there must be a system device enumerator for capture sources.
- Selection of output source, either:
  - Screen preview.
  - Screen preview + AVI file to disk, with any YUY2 compatible DirectShow compression filter installed.
  - Therefore there must be a system enumerator for installed compression codecs.
- Access to Hauppauge TV Card & Video Processing filter property pages.
- Full screen mode.
- Starting, stopping & pausing preview mode.
- For AVI playback, also the standard media control panel i.e. rewind, fast forward, random seek etc.

Currently the view & document architecture of the MFC is pretty meaningless to this project. In the future, however, one could envisage multiple video capture sources inside a single computer, together with features such as picture in picture & enhanced noise reduction through interpolation between multiple sources of the same video sequence. So the document/view framework shall be left in for future possibilities.
What type of GUI is best? Will any GUI satisfy 100% of people? Is therefore the 1st question too simplistic?

This project is targeted towards consumer use. The GUI must be designed with the needs of the customer in mind. However, the customer is likely not to be able to quantify, or even qualify his needs. Therefore we need to give the customer a taste of what the completed system will be like so he can ascertain his needs & comment on the physical entity before him. One way of doing this is through prototyping. The developer uses conventional theory to develop a few different GUIs, & then tests them on the customers to see what elements of which they prefer.

The following requirements [26] have been identified for the application:

- Selection of input source e.g. dialog box:
  - Selection of AVI file to open from disk or URL.
- Selection of output source:
  - Creating of AVI file to write to.
- Menuing system, toolbar &/or media control panel for controlling AVI file playback.
- Access to Hauppauge TV Card & Video Processing filters.

The testing procedure uses Use Case theory. Use cases testing involves identifying the actors (people) that interact with the system & then following them through the required uses of the system. At each point during the interaction you must record the steps the actor performs to reach his goal. With the use case you must also record as exceptions faced & how the actor receives feedback & then overcomes them.

The following use cases have been defined for this project’s application:

- Select legacy VIW input source → Select preview output.
- Select legacy VIW input source → Select compression codec → Select preview + file output → enter valid/invalid file name.
- Select Hauppauge WinTV input source → Select preview output → Change channel → Change to mono.
- Select Hauppauge WinTV input source → Select compression codec → Select preview + file output → enter valid/invalid file name.
- Select WDM input source → Select preview output → Change channel → Change to mono.
- Select WDM input source → Select compression codec → Select preview + file output → enter valid/invalid file name.
- Select legacy VIW input source → Select Hauppauge WinTV input source → Select preview output.
- Select Hauppauge WinTV input source → Select legacy VIW input source → Select preview output.
- Select AVI input → Select invalid/valid file → Select preview output.
- Select AVI input → Select invalid/valid file → Select compression codec → Select preview + file output → enter valid/invalid file name.
- Select AVI input → Select legacy VIW input source → Select preview output.

For debugging purposes the above list to not exhaustive. It illustrates the common user tasks that we wish to have an efficient GUI for.
Underneath the GUI lies a Command pattern [23] framework. Each request the user submits, whether it is by clicking on a menu item or an icon on the toolbar, is encapsulated as an object. This facilitates the queuing & undoing of operations, which considerably simplifies the construction of filter graphs, which is essentially serial in nature.

For instance, if we wish to playback video from an AVI file instead of a Hauppauge TV Card, we have to undo the construction of the video capture sub-graph, & then construct the AVI asynchronous read sub-graph instead. So we store the last ‘video input change’ command used, so we can undo it before executing the next ‘video input change’ command. Ditto with the output device command – we have to be able to insert & delete a Smart Tee filter as well as an AVI file writer. Figure 18 illustrates the concept.

- **Command** declares the interface for executing an operation.
- **InputCommand** extends Command to keep the last input command as state.
- **AVIReaderCommand** undoes the last command before constructing its own filter sub-graph. Finally it makes itself the last executed command.

![Figure 18: Application Command architecture.](image-url)
**IMPLEMENTATION**

**HAUPPAUGE WINTV CARD FILTER**

To avoid polluting the global namespace, the Hauppauge WinTV Card Filter was implemented in the \texttt{NTVCard::NHauppauge} namespace.

There are at last count 47 DirectShow interface methods that require adapting from the Hauppauge WinTV SDK. To promote modularity, & to eliminate linkage between the interface implementations & the filter, each interface (IAMTVAudio, IAMTVTuner, IAMCrossbar, IAMVideoProcAmp & IAMAnalogVideoDecoder) was implemented separately, before being multiply-inherited by the filter. The final class relationships are illustrated below:

![UML diagram of Hauppauge TV Card Filter interface](image)

**CTransInPlaceFilter** is a DirectShow Base Class, which provides a generic implementation of a transform filter that processes samples in the same buffer. That is, if it can avoid it, it uses the allocator from the upstream filter & passes it on to the down stream filter, rather than use its own allocator & copy samples between it & the upstream filter. If the upstream filter’s samples are declared read only, & the current filter is not declared read only, then **CTransInPlaceFilter** has no choice but to provide its own allocator.

**CTransInPlaceFilter** provides virtual methods for a derived class to override. The TV Card filter extends **CTransInPlaceFilter**, but does not implement methods such as Transform & Receive that are related to transforming the data. This is because the TV Card filter does not transform any data.
The TV Card filter must still implement some virtual methods of CTransInPlaceFilter, however. The public interface of our NTVCard::NHauppauge:CFilter is thus:

```cpp
namespace NTVCard { namespace NHauppauge {
  class CFilter : public CTransInPlaceFilter, public CAM AnalogVideoDecoder, public CAMVideoProcAmp,
                  public CAM TV Tuner, public CAM TV Audio, public CAM Crossbar,
                  public ISpecifyPropertyPages {
    public:
      static CUnknown *WINAPI CreateInstance(LPUNKNOWN punk, HRESULT *phr);
      DECLARE_IUNKNOWN;
      STDMETHODIMP QueryVendorInfo(LPWSTR *pVendorInfo);
      STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void **ppv);
      STDMETHODIMP GetPages(CAUUID *pPages);
      HRESULT Transform(IMediaSample *pSample);
      HRESULT CheckInputType(const CMediaType *mtIn);
      CBasePin* GetPin(int n);
      HRESULT CheckConnect(PIN_DIRECTION dir, IPin *pPin);
  }
}}
```

COM is a binary standard that does not use the inheritance structure of C++ to find the interfaces supported by an object. Instead, a COM component must implement the QueryInterface method of the root of the COM hierarchy IUnknown, which every COM component must inherit from. In this case the DECLARE_IUNKNOWN macro implements IUnknown, whilst its declaration is inherited from CTransInPlaceFilter. QueryInterface returns a pointer to the interface if it is supported. COM components usually use composition to implement multiple interfaces, but DirectX was built for efficiency & it uses the aggregation principle with NonDelegatingQueryInterface instead. This is why we use multiple inheritance of the interface implementations, rather than composition. CFilter defines as NonDelegatingQueryInterface as:

```cpp
STD_METHODIMP CFilter::NonDelegatingQueryInterface(REFIID riid, void **ppv) {
  CheckPointer(ppv, E_POINTER);
  if (riid == IID_IAM AnalogVideoDecoder) {
    return GetInterface((IAM AnalogVideoDecoder*) this, ppv);
  } else if (riid == IID_IAM VideoProcAmp) {
    return GetInterface((IAM VideoProcAmp*) this, ppv);
  } else if (riid == IID_IAM TV Tuner) {
    return GetInterface((IAM TV Tuner*) this, ppv);
  } else if (riid == IID_IAM TV Audio) {
    return GetInterface((IAM TV Audio*) this, ppv);
  } else if (riid == IID_IAM Crossbar) {
    return GetInterface((IAM Crossbar*) this, ppv);
  } else if (riid == IID_ISpecifyPropertyPages) {
    return GetInterface((ISpecifyPropertyPages*) this, ppv);
  }
  return CTransInPlaceFilter::NonDelegatingQueryInterface(riid, ppv);
}
```

The method checks to see if the passed REFIID (unique interface identifier) is for an interface we support, & if it is, returns a pointer to this casted to the required interface. If the interface is not supported, the method delegates to its base class CTransInPlaceFilter.

CreateInstance is required as a COM factory method to create components of type CFilter. QueryVendorInfo simply returns a string of the filter’s vendor (“awd97”). GetPages is the single method required of the ISpecifyPropertyPages interface, which tells the outside world that we export 1 or more property pages.

Required CTransInPlaceFilter implementations are CheckInputType & CheckConnect. CheckInputType verifies that we only accept a video media type from the input pin. CheckConnect
is called just before a pin of \textit{CFilter} is connected to another filter. In this case it checks whether the it is the input pin, \& if it is, only allows connection to a Hauppauge Video for Windows source filter.

In the \texttt{WTVSDK.h} header, The Hauppauge WinTV SDK is encapsulated into the namespace \texttt{NWTVSDK} as follows:

```cpp
namespace NWTVSDK {
    #include "hcwav.h"
}
```

The adaptor classes of \texttt{CAMTVAudio}, \texttt{CAMTVTuner}, \texttt{CAMCrossbar}, \texttt{CAMVideoProcAmp} \& \texttt{CAMAnalogVideoDecoder} all require the TV card to be initialised at construction, \& freed at destruction. A singleton class \texttt{CHCWImplSingleton} was implemented to provide the TV Card’s \texttt{AVHdl}. When the singleton instance is first requested the TV Card is initialised \& the handle is obtained. Further requests for the handle simply return the same handle. On program exit the TV Card is freed. The \texttt{CHCWImplSingleton} class also implements a static helper method to display an error message box.

The process of adapting the WinTV SDK to DirectShow is a mapping of the WinTV SDK messages \& structures to the DirectShow interfaces \& structures. For example, the adaptation to the \texttt{SignalPresent} method in \texttt{IAMTVTuner} is as follows:

```cpp
HRESULT CAMTVTuner::SignalPresent(long *plSignalStrength) {
    using namespace NWTVSDK;
    AVVideoSignalStatus hcwSignal;
    if (AVError_Success != AVError_hcwAV_Get(
        CHCWImplSingleton::Instance()->GetTVCardHdl(),
        AVF_VideoSignalStatus,
        &hcwSignal)) return E_UNEXPECTED;
    switch(hcwSignal) {
    case AVVideoSignalStatus_Good:
        *plSignalStrength = 1;
        break;
    case AVVideoSignalStatus_Bad:
        *plSignalStrength = 0;
        break;
    default:
        return E_UNEXPECTED;
    }
    return S_OK;
}
```

Firstly the \texttt{NWTVSDK} namespace is brought into the glocal namespace for the method’s scope only. An \texttt{AVF_VideoSignalStatus} getmessage is then sent to the TV Card to return the signal strength. If an error occurs the method returns with an \texttt{E_UNEXPECTED} \texttt{HRESULT}. What then follows is a switch statement that maps the WinTV SDK \texttt{AVVideoSignalStatus} enumeration to DirectShow’s representation: a \texttt{long} \texttt{0} meaning no signal present \& \texttt{1} meaning signal present.

The majority of the other methods follow the same routine. It must be mentioned that both \texttt{IAMTVTuner} \& \texttt{IAMTVAudio} include \texttt{RegisterNotificationCallback} \& \texttt{UnregisterNotificationCallback} methods that work a little differently. Their intention is that a client registers an \texttt{IAMTunerNotification} object with the filter, which is called back when the tuner state changes. This allows programs such as tuner GUI enhancements to keep in synchronisation with the hardware. In this project a \texttt{CNotificationCallbackList} was implemented, which includes methods...
for maintaining a list of IAMTunerNotification’s. Its NotifyAll method executes all the list item’s OnEvent methods. NotifyAll is called during every successful invocation on CAMTVTuner or CAMTVAudio.

The following methods were not implemented due to deficiencies in the WinTV SDK:

```
CAM Analog Video Decoder::get_HorizontalLocked
CAM Analog Video Decoder::get_OutputEnable
CAM Analog Video Decoder::put_OutputEnable
CAM Analog Video Decoder::get_VCRHorizontalLocking
CAM Analog Video Decoder::put_VCRHorizontalLocking
CAM TV Tuner::put_TuningSpace
CAM TV Tuner::get_TuningSpace
CAM TV Tuner::Logon
CAM TV Tuner::Logout
```

As DirectShow filters are lightweight COM components, they are meant to be as memory & processor efficient as possible. This precludes the use of heavyweight technologies such as the MFC in their construction. Therefore the property pages must be implemented using the raw Win32 messaging API.

The DirectShow Base Classes include a generic property page class, CBasePropertyPage, for filter developers to extend. CBasePropertyPage provides an entry point for messages sent to the property page, OnReceiveMessage, plus overrideable methods called on certain important events, such as when the property page is connected to the filter - OnConnect, or when the user confirms modification to the property page - OnApplyChanges.

The TV Tuner property page does not use OnApplyChanges since all changes are immediately carried out for user feedback. The tuner property page provides:

- Country code list box
- Input source list box
- Channel list box
- Signal present label

The property page could be used in conjunction with any other DirectShow component that implements IAMTVTuner, since it only calls methods in the IAMTVTuner interface passed to it by the filter.
An abridged definition of `CAMTVTunerPropertyPage` is as follows:

```cpp
class CAMTVTunerPropertyPage : public CBasePropertyPage {
  public:
    static CUnknown * WINAPI CreateInstance(LPUNKNOWN lpunk, HRESULT *phr);
  private:
    BOOL OnReceiveMessage(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
    HRESULT OnConnect(IUnknown *pUnknown);
    HRESULT OnDisconnect();
};
```

`OnConnect` the property page queries the `IAMTVTuner` interface from the filter it has just connected to. On `OnDisconnect` the property page frees this interface. The `OnReceiveMessage` method looks like this:

```cpp
BOOL CAMTVTunerPropertyPage::OnReceiveMessage(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
  HRESULT hr;
  switch (uMsg) {
  case WM_INITDIALOG: // Initialise the dialog box
    case WM_COMMAND:
      switch (LOWORD(wParam)) {
      case IDC_COUNTRYCOMBO: // Change WinTV Card country
        break;
      case IDC_INPUTCOMBO: // Change WinTV Card input source
        break;
      case IDC_CHANNELCOMBO: // Change WinTV Card channel
        break;
      default:
        return CBasePropertyPage::OnReceiveMessage(hwnd, uMsg, wParam, lParam);
      }
      return (LRESULT) 1;
    default:
      return CBasePropertyPage::OnReceiveMessage(hwnd, uMsg, wParam, lParam);
  }
}
```

If the message received is of type `WM_INITDIALOG`, it tells us that the property page’s controls have been initialised & it is about to be drawn. Here the list boxes are filled with items & the initial country, input source & channel are retrieved from the WinTV Card. The list boxes are then updated accordingly. If the message is of type `WM_COMMAND`, it means the user has affected a control on the property page. Depending on this control it is, the country, input source or channel is switched accordingly. Any other messages are delegated to the property page’s parent.
The TV audio property page is implemented in the same way. It provides user selection of:

- Audio input: either from the TV tuner or line in.
- Tuner mode: Either mono, stereo, language A, B or C.
- The capabilities of the TV Card are detected & controls are disabled accordingly (see figure).

The IAMTVAudio & IAMCrossbar interfaces of the host filter are used to perform this functionality.

DirectShow requires that all filters be registered with the system registry before they can be used in filter graphs. This is so all the filters on the system can be automatically enumerated, queried & selected when DirectShow builds the filter graph. In the Win32 API, registration of a dll is performed with the 2 methods in the global namespace DllRegisterServer & DllUnregisterServer. Luckily, DirectShow provides implementations for these 2 methods in the AMovieDllRegisterServer2 Class.

For input into the registry, the filter & the 2 property pages require globally unique identifiers (GUID's). The identifier is a 128bit number, generated using an algorithm published by the Object Management Group [23]. For example, the GUID for the filter is generated by running the uuidgen.exe utility, & defined in the CFilter class with:

```c
DEFINE_GUID(CLSID_HCWTVCardFilter, 0x11a25a3, 0x51f2, 0x4819,
0xb4, 0xf9, 0xfb, 0x96, 0x23, 0xb8, 0xfe, 0x59);
```

The 3 entries are then published via the COM utility class CFactoryTemplate with a pointer to their CreateInstance method & a pointer to a structure with information about the component. In the case of a the TV Card filter, the AMOVIESETUP_FILTER structure contains the following information:

- The filter’s name ‘awd97 Hauppauge TV Card Filter’.
- A ‘do not use’ merit. The merit tells the filter graph manager what precedence the filter has in being automatically chosen for filter graph insertion. In this case the filter is never chosen.
- The number of pins. Is this case 2: 1 input; 1 output. For each pin the following information is recorded:
  - Its name.
  - Whether it is an input or output pin.
  - The media types allowed. In our case the major media type must be video, but we are not too picky about the subtype.
- It’s GUID - CLSID_HCWTVCardFilter
Intel Pentium & some 486’s have an instruction for detecting the CPU type & instruction set extensions supported. This CPUID instruction is encapsulated into the \texttt{CMachineTypeSingleton} class. Here is its public interface:

```cpp
class CMachineTypeSingleton {
public:
    enum EXTENSIONS {
        EXTENSIONS_NONE = 0x00000000,
        EXTENSIONS_MMX = 0x00000001,
        EXTENSIONS_3DNOW = 0x00000002,
        EXTENSIONS_SSE = 0x00000004,
        EXTENSIONS_SSE2 = 0x00000008
    };
    typedef ULONG INSTRUCTION_SET;
    static CMachineTypeSingleton* Instance();
    INSTRUCTION_SET GetInstructionSet();
};
```

Note the constructor is not public. The singleton object is only available through the \texttt{Instance} method. \textit{Instance} uses lazy initialisation to create the object on the first request for it. The constructor calls the method that detects the instruction set. First we must detect whether the CPUID instruction is supported, which is a case of trapping an illegal instruction exception when we try to perform a CPUID:

```cpp
__try {
    __asm cpuid
}
__except (EXCEPTION_EXECUTE_HANDLER) {
    return (0);
}
```

If this occurs, the \texttt{CMachineTypeSingleton} is set to no MMX support.

The CPUID instruction is executed a further 3 times:

1. To obtain the vendor ID string.
2. To check the standard feature flags. These include support for:
   a. Time Stamp Counter (reports how many cycles the computer has performed since being switched on).
   b. CMOV instruction (conditional move supported by Pentium CPU’s).
   c. MMX
   d. SSE & SSE2
3. To check the extended feature flags. These include support for:
   a. AMD 3DNow! & Extended 3DNow!

Since we are only interested in integer extensions, Extended 3DNow! & SSE are reported both as SSE. The \textit{INSTRUCTION_SET} returned by \texttt{GetInstructionSet} is a combination of items from the \texttt{EXTENSIONS} enumeration.
Since MMX is a major subset of integer SSE, the majority of code for both implementations of a strategy is identical. Therefore the different sections of code were selected at compilation time depending on the definition of macros (in RoutinesMMX.h, Routines3DNow.h & RoutinesSSE.h).

The YUY2 format allows either parallel MMX computation of 8 bytes (4 Y & 2 U & V values) or 4 words (4 pixels of YU or YV data). Since the human eye is less sensitive to colour than luma values, to increase performance the UV data in most of the deinterlacing strategies were extracted from the input & written to the output with a simple bobbing strategy. This has the effect of reducing the subjective sampling space from 4:2:2 to 4:2:0. However, it also has the pleasant side effect that we can use 16 bits rather than 8 bits to transform each luma value.

For an example of MMX vectoring, combining the YUV values into the output vector afterwards is performed with 3 logical operations. The COMBINE macro below was also used frequently after a compare operation for combining 2 lines of video with a mask.

```
#define COMBINE(a,b,mask) __asm {
   __asm pand a, mask
   __asm pandn mask, b
   __asm por a, mask
}
```

The mask is intended to be either all 0, or all 1, for each byte in the MMX register. Therefore the output byte is taken from a if the mask is 0xFF at that point, or b if the mask is 0x00. The pand instruction (packed AND) performs a 64 bit bitwise AND on its arguments. The pandn (packed NOT & AND, not NAND), inverts the mask & then AND’s the mask & b. por (packed OR) combines the results of the 2 previous instructions.

The AVERAGE_BYTE & AVERAGE_WORD macros are the most used in the strategies. They perform the bytewise or wordwise average of their 2 operands respectively. Contrasting the MMX, 3DNow! & SSE versions illustrates the performance gains of SSE over MMX:

```
# define AVERAGE_BYTE(a,b,temp) – temp is an MMX register used for intermediate values

<table>
<thead>
<tr>
<th>MMX</th>
<th>3DNow!</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mask = 0x7E7E7E7E7E7E7E7E; movq temp, b</td>
<td>pavgusb a, b</td>
<td>pavgb a, b</td>
</tr>
<tr>
<td>pslw a, 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pslw temp, 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pand a, Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pand temp, Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>paddb a, temp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

SSE & 3DNow! processors already have an average instruction (though they look different they perform identically). The MMX version, however, has to first logically shift both operands right one bit (division by 2). Since MMX does have a byte-wise shift instruction, only a word-wise one (pslw), we must then mask off the least significant bit, which half the time would be shifted into the next byte! The two arguments are then byte-wise added to complete the (a+b)/2 calculation.
The latency of the SSE AVERAGE_BYTE macro on a Pentium 3 is 1. For the MMX version the latency rockets to 6.

The other main macros are for calculating the packed absolute difference between 2 operands, & the byte wise parallel sorting of 2 operands (for median calculation).

```
#define ABSDIFF_BYTE(a, b, temp) __asm {
  __asm movq temp, a
  __asm psubusb a, b
  __asm psubusb b, temp
  __asm psr a, b
}
```

```
#define SORT(a, b, temp) __asm {
  __asm movq temp, a
  __asm pminub a, b
  __asm pmaxub b, temp
}
```

It is a shame that there is no MMX or SSE absolute difference instruction, as the latency of 4 for this macro is disappointing. First b is subtracted from a, packed with saturation (the psubusb instruction). This in effect returns the maximum between a-b & 0. Likewise, max(b-a, 0) is found. The absolute difference, |a-b|, is then the maximum of these two results, & since one of them must be 0, |a-b| is found by OR’ing both results.

The MMX implementation of SORT is too slow. What you see is the SSE implementation with SSE specific pminub & pmaxub instructions. These find the byte wise minimum & maximum of the 2 operands respectively.

**OTHER OPTIMISATIONS**

The other optimisations that were taken into account include:

- Pipelining memory accesses: To avoid a massive speed decrease, all loops must only write one piece of data to memory, to space out memory writes effectively.
- Instruction pairing: MMX instructions may be paired & run simultaneously if they satisfy some conditions. Namely they must not both use the same execution unit. There are 2 MMX ALU’s, but only 1 multiplier, 1 barrel shifter & 1 memory access/integer access unit.
- Cache consistency: The frame buffers are traversed, one line at a time, to keep the benefit of spatial caching.
- Memory alignment: Frame buffers, in all possible circumstances, are aligned to 8 byte boundaries for the benefit of the quadword MMX instructions, as a misaligned store or load operation suffers a minimum one-cycle penalty in most processor’s load/store pipeline.
- Lack of memory alignment: The edge-orientated deinterlacing algorithm purposely fetches unaligned data so that it can perform MMX vectoring on the diagonal candidates.
For the video-processing filter a CTransformFilter Base Class was extended. The CTransformFilter Base Class is actually the part of CTransInPlaceFilter, without the negotiation for passing on the upstream allocator instead of copying upstream samples into empty buffers created by our own allocator.

The CTransformFilter requires us to implement one more method: DecideBufferSize. This lets us specify the size of the empty buffer our allocator takes, in case the output media type is different from the input media type. The CheckInputType method is also pickier than the Hauppauge TV Card filter’s version, as we can only accept YUY2 video data.

The deinterlacing algorithms require sample rate 2x up-conversion. This is achieved by overriding the Receive method with a version that calls the parent’s Receive twice. This way two output samples are allocated & delivered to the input pin connected to our output pin for every input sample. The first receive is preceded by setting m_bFlipField to true, & the second by setting it to false, so the image buffer knows how to calculate references to the buffer.

The image buffer holds past frames & buffers incoming frames, for spatio-temporal use in the strategies. Currently it is set to keep 2 future & 2 past fields.

The CImageBuffer class derives from CFILOBuffer. CFILOBuffer implements a circular queue of pointers to IMediaSample’s. To make sure DirectShow’s garbage collection does not reuse or destroy the media samples they are AddRef’ed on insertion. Before insertion the old sample that will be overwritten is Release’ed. CImageBuffer adds each input media sample twice to the CFILOBuffer. This is because we work on fields & not frames.

<table>
<thead>
<tr>
<th>Image Buffer</th>
<th>Past Fields</th>
<th>Current Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetWeavePointer(0, x)</td>
<td>Line x=0</td>
<td>Line x=0</td>
</tr>
<tr>
<td>Line x=1</td>
<td>Line x=1</td>
<td></td>
</tr>
<tr>
<td>Line x=2</td>
<td>Line x=2</td>
<td></td>
</tr>
<tr>
<td>Line x=3</td>
<td>Line x=3</td>
<td></td>
</tr>
<tr>
<td>Line x=4</td>
<td>Line x=4</td>
<td></td>
</tr>
<tr>
<td>GetWeavePointer(1, y)</td>
<td>Line y=0</td>
<td>Line y=0</td>
</tr>
<tr>
<td>Line y=1</td>
<td>Line y=1</td>
<td></td>
</tr>
<tr>
<td>Line y=2</td>
<td>Line y=2</td>
<td></td>
</tr>
<tr>
<td>Line y=3</td>
<td>Line y=3</td>
<td></td>
</tr>
<tr>
<td>Line y=4</td>
<td>Line y=4</td>
<td></td>
</tr>
</tbody>
</table>

Figure 23: GetWeavePointer assuring the ordering of lines in fields for uniform strategy implementation.
It was a challenge to devise a system of referencing lines in the image buffer, which could keep the special cases to a minimum. In the interlaced format every 2\textsuperscript{nd} line is missing. However, if we referenced fields separately the relative position of a field to the last changes every field. e.g. even line 0 is above odd line 0, but odd line 0 is below the next even line 0. So that we did not need separate strategies for even & odd fields, the developer must access the buffer through the \texttt{GetWeavePointer} methods of \texttt{CImageBuffer}. \texttt{GetWeavePointer} ‘weaves’ the current field with the last so that all references are to frames rather than fields. The order of lines is implicitly assured. This is illustrated in figure 23.

\texttt{CImageBuffer} also inherits from \texttt{CMediaHeader}. This was a performance choice; it could equally well have included one as an attribute. \texttt{CMediaHeader} extracts important statistics from the media type to present them in a uniform way for use by the strategies.

---

**STRATEGIES**

If the media sample is dynamically changed during the course of the streaming, \texttt{CStrategy} detects it & \texttt{CMediaHeader} recalculates it. The strategies’ \texttt{Update} method is also called at this time.

The \texttt{CDeinterlaceStrategy} class recalculates the start & stop time of each output sample so the extra output samples needed for the 2x up-conversion are spaced evenly between the original output samples.

The \texttt{CLineStrategy} class extends the \texttt{GetWeavePointer} principle for deinterlacing algorithms that work on each missing line. It implements \texttt{GetRelativeWeavePointer} methods which retrieve the line specified by the weaved row \textit{relative to the current line}.

What follows are notes on the individual video processing strategies.

**Flip Field Detection**

The flip field detection strategy was implemented in the Intel Performance Primitives. It was seen to be unnecessary to fully optimise it to assembly language, due to its minimal processor overhead. The implementation decisions were as follows:

- A motion difference tolerance of 0.1 intensity values per byte of YUY2 was deemed to be robust enough to noise whilst sensitive enough to detect changes in flip field.
- To avoid incorrect flip field changes due to global changes in motion, the flip field was only changed if 2 consecutive frames reported the same flip field.
- It was found to be unnecessary to calculate the 2 field differences for all the lines in the 2 frames. A sample of 128 lines was taken at regular intervals in the frames. This increased performance of the algorithm without limiting detection ability.
- A short circuit was added. If a frame difference became greater than the total tolerance value for the whole sample, the algorithm exited with its result straight away. Since output frames are buffered by the image buffer, this will show an (average) reduction in the overhead of the algorithm.

Note that if the flip field detector finds a \texttt{VIDEOINFOHEADER2} media type with flip field information, it will not run the above algorithm.
Noise Reduction Strategy 0: Spatial Gaussian

To contrast with the upcoming noise reduction strategies, a spatial Gaussian filter was implemented using the Intel Performance Primitives. Since the IPP does not have native YUY2 support, the chroma values were extracted & written unchanged to the output buffer. This meant no noise reduction was performed on the chroma values. The benefit was that the luma values left each occupied 16 bit words. Performing a 3x3 Gaussian filter on the frame buffer then consisted of calling the method:

```
static IppSize r16Size = {m_pBuf->iWidth, m_pBuf->iHeight};
ippiFilterGauss_16s_CLR(InputBuffer, m_pBuf->iByteWidth,
                        TempBuffer, m_pBuf->iByteWidth,
                        r16Size, ippMskSize3x3);
```

The luma values were then interleaved back with the chroma values in the output buffer.

Noise Reduction Strategy 1: Weighted Recursive Temporal Averaging

This strategy was assembly optimised. It was extended from the CStrategy class, so that the algorithm works on the full frame in one pass.

- The Luminescence & the Chromaticity values were independently measured for motion with different thresholds. It was found through subjective experimentation that a difference greater than 8 in luma value between this & the last frame represented motion. Likewise, a difference greater than 7 in chroma represented motion.
- The algorithm was converted to assembly as follows:
  1. The next 64bit vectors of the current & last frame are placed in the MMX registers mm0 & mm1.
  2. mm0 & mm1 are copied to mm2 & mm3.
  3. $\frac{1}{4}$ (OldPixel) + $\frac{3}{4}$ (NewPixel) is calculated by calling the AVERAGE_BYTE macro twice:

```
AVERAGE_BYTE(mm2, mm3, temp) // mm2 = (NewPixel + OutPixel)/2
AVERAGE_BYTE(mm2, mm3, temp) // mm2 = (3/4)NewPixel + (1/4)OldPixel
```

  4. The byte wise absolute difference between the new pixel & the old pixel is calculated using the ABSDIFF_BYTE macro.
  5. Comparing the absolute difference to the luma & chroma thresholds creates a mask.
  6. The interpolated pixels from step 3 are combined with the pixels with the current frame using the mask & the COMBINE macro.
  7. The result is written to the output frame buffer.
  8. The above steps are repeated until the end of the frame.

Noise Reduction Algorithm 2: Adaptive Temporal Averaging

This strategy was assembly optimised. It also was extended from the CStrategy class, so that the algorithm works on the full frame in one pass.

- A block size of 4x2 was found to give a good balance between robustness against noise & vectoring convenience.
- The implementation had to be split into 3 loops to allow adequate memory write pipelining. A 1 pass implementation was over 300% slower.
Loop 1 acts as the motion detector. It calculates the SAD for each 4x2 block for the luma & chroma values separately. The SAD values are then thresholded to the maximum motion detect value $\beta$. Finally the luma & chroma SAD’s are packed into bytes & interleaved to form a multi-level mask. This mask is stored in a temporary line buffer.

Loop 2 operates on the line that formed the top of the 4x2 blocks in the previous loop. It weights the interpolation between the current & last frame by the mask’s value. It then writes this to the output frame.

Loop 3 performs identically to Loop 2 for the bottom line of the 4x2 blocks.

Due to the lack of a SAD instruction in MMX, this algorithm works under SSE only.

Deinterlacing Algorithm 0: Weave & Bob

A weave deinterlacing strategy was implemented. The difference between this weave & the original weaved input from the capture card is that the video is up-converted to 50Hz. Therefore half the time the output frame will be identical to the input frame, but the other half of the time the output frame will be the first field of the input weaved with the last field of the previous input. For efficiency purposes then, the weave deinterlacer extends CDeinterlaceStrategy & works on a field at a time. Either the whole input frame is copied using the IPP, or lines are copied alternating between this input frame & the last.

A simple bob deinterlacer strategy was implemented. This extends CLineStrategy & therefore works on a line at the time. It simply uses the IPP to interpolate between its 2 vertical neighbouring lines.

Deinterlacing Algorithm 1: Vertical-Temporal

To illustrate the process of MMX & SSE vectoring, the VT deinterlacer strategy is reproduced & commented in its entirety. It extends CLineStrategy. A, B, C, D & E are illustrated on figure 7:

```assembly
mov ecx, iCycles // iCycles = iByteWidth / 8 [64 bits]
mov eax, dword ptr [psA] // eax = A
mov ebx, dword ptr [psB] // ebx = B
mov edx, dword ptr [psC] // edx = C
mov esi, dword ptr [psD] // esi = D
mov esp, dword ptr [psE] // esp = E
mov edi, dword ptr [psOut] // edi = Out
align 8
Loop:

movq mm0, qword ptr[eax] // mm0 = A
movq mm1, qword ptr[ebx] // mm1 = B
movq mm2, qword ptr[edx] // mm2 = C
movq mm3, qword ptr[esi] // mm3 = D
movq mm4, qword ptr[esp] // mm4 = E
movq mm5, mm0 // mm5 = A
AVERAGE_BYTE(mm5,mm1,mm6) // mm5 = (A + B) / 2
pand mm0, YMask // mm0 = A[Y]
pand mm1, YMask // mm1 = B[Y]
pand mm2, YMask // mm2 = C[Y]
pand mm3, YMask // mm3 = D[Y]
pand mm4, YMask // mm4 = E[Y]
psslw mm0, 4 // mm0 = 16A[Y]
psslw mm1, 4 // mm1 = 16B[Y]
pmov mm6, mm3 // mm6 = D[Y]
psslw mm6, 1 // mm6 = 2D[Y]
psslw mm3, 4 // mm3 = 16D[Y]
```
Deinterlacing Algorithm 2: VT with 5 Point Median Protection

The extension of VT with a 5-point median means we needed a sorting algorithm. Whilst sorts such as quicksort & mergesort have better O at large n, at n=5 bubble sort is optimal. Or rather, this pruning of bubble sort:

```
SORT(mm1, mm2, temp1, temp2)
SORT(mm1, mm4, temp1, temp2)
SORT(mm2, mm5, temp1, temp2)
SORT(mm3, mm4, temp1, temp2)
SORT(mm2, mm3, temp1, temp2)
```

Where the 5 items to be sorted are in MMX registers mm1, mm2, mm3, mm4 & mm5. The median item is then located in mm3.

The lack of an optimized sort for MMX precludes the use of this strategy on non-SSE based machines.

Deinterlacing Algorithm 3: 3D Edge Orientated

The 3D Edge Orientated strategy extends CLineStrategy. Keeping this algorithm simple was a challenge, & in the end an iterative strategy was devised, concentrating on 1 of the 6 edge orientations at a time, & building up the interpolation mask.

1. The algorithm only works on the luma values. The chroma values are separated at this point.
2. The edge gradient of the 1st edge orientation is calculated (the absolute difference) & placed in mm2.
3. The linear interpolation (average) along the 1st edge orientation is calculated & placed in mm1.
4. For edge orientations 2 to 6:
   a. The edge gradient is calculated.
   b. The linear interpolation is calculated.
c. A mask is constructed of those edge gradients greater than those in mm2.

d. Where the edge gradients are smaller, the linear interpolation replaces those pixels in mm1 (using the mask).

e. Where the edge gradients are smaller, the edge gradients are replaces those pixels in mm2 (using the mask).

5. mm1 now contains the best interpolation. These are combined with the bobbed chroma values & placed in a temporary line buffer.

6. Finally the 5 point median is calculated as before, with the median values entering the output line buffer.

**PROPERTY PAGES**

The video processing property pages work on the same principles as the Hauppauge TV Card filter’s. There are 2 property pages, 1 to choose the noise reduction method & 1 to choose the deinterlacing method. If the strategy supports a property page the Properties button enables & can launch it.

The IDeinterlace & INoiseReduction interfaces of the host filter are used to perform this functionality.

![Video Processor Deinterlacing property page](image)
From the Use Case testing it was decided to implement the dialog boxes as shown in figures 25 & 26.

- The 3 radio buttons in the input dialog form a radio group.
- Selecting Video for Windows Capture or Hauppauge TV Card updates the list box with all the VfW capture drivers available to the system. This is necessary even for the Hauppauge TV Card as a limitation of VfW is that we cannot tell if a driver’s vendor is Hauppauge.
- Selecting WDM Capture updates the list box with the WDM Capture devices on the system.
- If the user selects a VfW capture driver & selects OK but the capture driver filter is unable to connect to the filter graph as it is not set to YUY2, the user is requested to change the video source’s format to YUY2. This is another limitation of VfW.

The Output dialog chooses between renderer only, or renderer with AVI file capture. If the renderer only radio button is selected, the compression codec list box is dimmed. Otherwise the user must select a compression codec.

Once the user confirms his choice by pressing OK, a Save As dialog box is invoked to gather the user’s intended destination file.

The main GUI is a simple menuing system that allows access to the Hauppauge TV Card & Video Processing filters through a Settings menu. Video preview is displayed in the main client area.

The decision to use the MFC for the application was probably overkill, however it is useful for future expansion. The decision was made to keep the application’s state in the CDocument class & to keep the command patterns in the CView class. This is because CDocument keeps a list of views, but CView points to one document. Commands only operate on one filter graph, whereas the state controls all filter graphs. The finished structure is illustration in figure 27.

The diagram does not show all the commands, for example CHauppuageCaptureCommand, CAVIWriteCommand.
The `CInitCommand` initialises the filter graphs & adds the video processing filter to. The `CRenderCommand` attaches the renderer to the client area. The `CResizeCommand` resizes the video renderer when the client area is resized. For example, to create a filter graph with manager & obtain a specific interface to it we use the COM `CoCreateInstance` & `QueryInterface` methods:

```c++
// Create the filter graph
CoCreateInstance (CLSID_FilterGraph, NULL, CLSCTX_INPROC,
    IID_IGraphBuilder, (void **) &(m_pDoc->g_pGraph));

// Obtain interfaces for media control & Video Window
m_pDoc->g_pGraph->QueryInterface(IID_IMediaControl,(LPVOID *) &m_pDoc->g_pMC);
```

The other commands either generate the start of the filter graph (input source), or the end of it (output sink). Creating a Video for Windows source filter follows the same process as above, except we also need to enumerate the system’s devices & select one. The devices are found with a call to the System Enumerator with `CLSID_VideoInputDeviceCategory` as the search criterion:

Connecting filters together requires you to find the relevant pins on both filters & invoking the `ConnectDirect` method of `ICaptureBuilder2` – an interface the filter graph manager supports. `ICaptureBuilder2` supports intelligent searching of pins with various criteria such as output type & direction.

Application execution operates as follows:

- The Operating System passes control to `CProjectTVApp`.
- `CProjectTV` app creates the `CMainFrame`. `CProjectTVView` & `CProjectTVDocument` objects.
- `CProjectTV` app creates & executes an instance of `CInitCommand`.
- `CProjectTV` app creates singleton versions of the other commands.
- Control passes to the GUI, which executes command objects on demand.
EVALUATION

DirectShow comes with a utility that allows the user to interactively construct & run filter graphs. This program can be used to test harness individual & group of filters without relying on the correctness of the project video processor application. The video processing filters can be composed with an AVI source filter & video renderer. Testing can then be performed through inspection of the video display, & the quality control statistics output by the renderer without the need for an encompassing application.

Testing was an iterative part of the project, which was performed in parallel with development:

- Each function was tested after implementation whilst unimplemented functions had stubs.
- Tests included:
  - Boundary Analysis: Inputting values near & over the limits expected by the function.
  - Stress tests: Response time of functions & the GUI under high processor usage conditions.
  - Monkey tests: Entering gibberish data to test the robustness & error reporting features. Connecting the Hauppauge TV Card & Video Processing filters to unexpected filters to test connection refusal.
  - Interaction tests: Ensuring the DirectShow components communicated & interopreated with 3rd party components in a uniform & stable manner.
- Each filter was tested in the Graph Edit debug environment.
- The filters were incorporated into the application & the application was thoroughly tested for consistency.

Testing the correctness of the video processing algorithms could only be carried out subjectively. Most problems were easy to spot, exhibiting for example, null pixels & other extreme artifacts. The video processing algorithms were also inspected by 3rd parties to reduce the probability of an oversight.
DirectShow includes a performance profiler with its Base Classes. By including a preprocessor directive `PERF` at the start of your code, the time taken for each call to the `Transform` is recorded to a large circular buffer, before being dumped to a specified file at program end.

Or at least, this was the idea. I could not get it to work & I was left with the distinct impression that it hadn’t actually been implemented yet. I can only speculate that it was not complete before the DirectX 8 deadline, & all the documentation on it was removed (except for the part I was reading in the first place). If somebody knows different, I would be most grateful for you to get into contact with me. I am especially interested in the `measure.dll` library that is referred to but does not appear to exist, even in the MSDN or the Internet at large.

Luckily, I was to find a better way. All Pentium CPU’s (& it was retrofitted for some 486’s) include a RDTSC instruction, which returns the processor cycles completed since the computer was started, as a 64bit number. The high 32bits are placed in the `edx` register, & the low 32bits are place in the `eax` register. If we find the difference in the Time Stamp Counter before & after a frame of video-processing, we have found the number of processor cycles required for that video processing algorithm, assuming that no other Windows program is timesliced into that segment. This allows us to get a more accurate measurement of the performance of the video processing strategies, & it also makes the analysis processor independent to a certain degree.

The processor megacycles (millions of cycles) needed for one frame of video to be processed is illustrated in figure 29 (The test is an average of 1000 frames). The test was conducted on an Athlon 700Mhz, with SSE disabled for the MMX tests. The test is almost processor independent, notably:

- For all but one instruction used in the project, the latencies are equivalent on the AMD Athlon & the Intel Pentium 3. Athlon latencies are exactly double – 2 for a common MMX instruction instead of 1 for a Pentium 3. However, pipelining negates any relevance to the doubling. We should only consider relative latencies.
- The `psadbw` SSE instruction, which calculates the SAD in the Adaptive Recursive Temporal Averager Noise Reduction algorithm, has a latency of 5 on the Pentium3 & 3 on an AMD Athlon. Since this instruction is used minimally in only 1 strategy, I have ignored the differences as inconsequential.

Other notables:

- The test is conducted on full digitised PAL resolution YUY2 video (768 x 576).
- The flip field & noise reduction algorithms are called once per frame. Therefore the result represents one `FieldProcess` call on the strategy.
- The deinterlacer algorithms are called once per field. Therefore the result represents two `FieldProcess` calls on the strategy.
- The MMX versions of the median type deinterlacers are non-functional simulations. This will be explained further presently.
Figure 29: Performance analysis

<table>
<thead>
<tr>
<th>Video processing strategy</th>
<th>Cycles per frame × 25fps</th>
<th>Processor speed in Hz</th>
<th>Processor Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFD</td>
<td></td>
<td>100%</td>
<td>MMX: 1.2%</td>
</tr>
<tr>
<td>RTA NR</td>
<td></td>
<td>100%</td>
<td>SSE: 1.2%</td>
</tr>
<tr>
<td>ARTA NR</td>
<td></td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>W DI</td>
<td></td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>B DI</td>
<td></td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>VT DI</td>
<td></td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>MVT DI</td>
<td></td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>EO DI</td>
<td></td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>MEO DI</td>
<td></td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>

We can convert the megacycles measurement to the percentage of CPU time the strategy uses given an output media rate of PAL video (50Hz) using the following equation:

\[
\text{Percentage} = \left( \frac{\text{Cycles per frame} \times 25\text{fps}}{\text{Processor speed in Hz}} \right) \times 100\% \quad (24)
\]

For an Athlon 700Mhz system the results are shown in the table to the right.

Confirmation of the correctness of the results was obtained by inspecting the Windows NT Task Manager’s processor usage graph during normal (non-test) operation of the strategies.

Figure 30: Table of CPU usage for the various strategies.
Comments on the results:

- The flip field detector turns out to impact negligibly on the overall CPU usage. This proves one argument against those who thought retrofitting a Video for Windows capture source with interlaced video support was a waste of time.
- The difference in performance between MMX & SSE was not as wide as I expected at the start of the project. This can attributed to the lack of a great deal of extra integer based instructions in SSE that are applicable to this project.
- An MMX sorting routine involves:
  1. Subtracting with saturation one vector from the other (a – b).
  2. Creating a mask of those values that are now 0 (a < b).
  3. The lowest values from the two vectors are found by combing a & b with the mask.
  4. The highest values are found by combing a & b from the negated mask.
This routine requires 4 temporary MMX register, which proved a headache when we have 5 items to select the median from, & only 8 MMX registers. The MMX routines for the median type deinterlacers are therefore non-functional simulations. Functional versions would have been even slower.
- Since the last machine without SSE was a 450Mhz Pentium 2, which could not have run the median type deinterlacers in real-time, it was decided to avoid implementing functional MMX versions of them.
- Despite the algorithm being the same, adding an MMX 5-point median to the 3D EO was much slower than adding one to the VT deinterlacer. This shows how much effect memory pipelining has on the speed of the algorithm (VT implements the median in a separate loop).
- As expected, the adaptive version of the noise reduction algorithm was much slower. It is still a very acceptable real-time solution though.
- The VT deinterlacer had a 4 megacycle penalty over the simple deinterlacers (bob & weave), & the 3D Edge Orientated deinterlacer had a 4 megacycle penalty over the VT deinterlacer. This corresponds well with their increasing visual performance (see next section).
- There was an approximate 4.5 megacycle penalty for protecting the algorithms with a 5-point median. This shows that median protection is an unsatisfactory method for improving the robustness of the deinterlacing algorithm. A better deinterlacing strategy would have 4.5 megacycles to play with to beat a median protection. It would be interesting to see if a VT with median strategy fared worse than a 3D EO without, since the algorithms are of equal expense...

Subjective Analysis

Video quality is still a subjective matter, as it proves difficult to design a reliable objective measure reflecting the subjective impression. The additional problems with specific objective measures are:

- An MSE criterion between a noise reduced frame & a noiseless original requires that we have the noiseless origin.
- The MSE criterion calculates the MSE between the restored interlaced frame, & the progressive original. But this requires an original progression video sequence, something that is not easy to acquire, & which is of course not available from a Hauppauge WinTV card.
- The Motion Trajectory Inconsistency criterion for evaluating deinterlacing algorithms is defined as:
\[
MTI(n) = \frac{1}{\Pi_w(I(x,y,t) - I(x - d_x, y - d_y, t - 1))}
\]

Where \( \Pi_w \) is the number of pixels in the measurement window, \( W \) is the measurement window & \((d_x, d_y)\) is the motion vector for each particular pixel. The implicit assumption is that two images \( I(x,y,t) \) & \( I(x - d_x, y - d_y, t-1) \) from a perfect de-interlacer are identical, even though one is derived from an odd input field & the other from an even field. It is the mean-square deviation from this ideal that is measured by the MTI. But not only do we not have an adequate motion estimator, but the outcome of this criterion is always effected by the quality of the motion vectors as well as the deinterlacer.

Because of the above, & as evaluations for the algorithms devised in my project are well researched by academics, I decided to concentrate on subjective evaluation in this report. Figures identify shortcomings in the implemented algorithms.

The evaluation was on 3 video sequences:

- Horse racing sequence.
- Cartoon sequence.
- Quiz sequence.

The horse racing sequence tests the algorithm’s robustness to fast motion. The cartoon sequence tests the algorithm’s robustness to sharp edges. The Quiz sequence provides an expected everyday use for the TV viewer.
Figure 31: Artifacts of the Vertical-Temporal Deinterlacer

Figure 32: VT with 5-point Median deinterlacer
Figure 33: 3D Edge Orientation gives the best results, even on noisy sequences.

Figure 34: No noise reduction
Figure 35: Spatial Gaussian noise reduction also blurs fine details.

Figure 36: Adaptive Temporal Recursive Averager noise reduction
Comments on the results:

- Figure 30 shows that weave artifacts are quite visible and horrendous on anything above low motion.
- The Vertical-Temporal deinterlacer looked quite reasonable, but in closer inspection ‘ghosting’ artifacts were found (figure 31) trailing motion in horizontally moving sequences with sharp edges. This is believed to be caused by the linearity of the interpolation, as the temporal element of the VT interpolator still affects the result when its vertical neighbours do not cancel it out (because they do not include the edge information).
- Adding the 5-point median only removed some of these artifacts. This is because the incorrect interpolation was not wildly out from its correct value. However, when spatial neighbours of the incorrect interpolation are also incorrect by the same degree, the human mind recognizes the object formed, by the Gestalt principles of similarity and closure.
- The 5-point median had more disturbing artifacts, however, which were clearly visible in the horse racing sequence. These artifacts cannot be shown static; they require the viewer to see the moving sequence. When the 5-point median springs into action & protects an ‘incorrect’ interpolation, it only replaces it with a nearest-neighbour. Therefore there will now be a 2x1 block of uniform intensity. This shows up as a blocking of the area involved. This would not be so problematic, but when large areas share the same properties & switch from smooth interpolation to nearest neighbour & back again in a coordinated way, it is clearly visible. This occurs on the railings in the horse racing sequence.
- The 3D Edge Orientated deinterlacer had the least artifacts, see figure 33. Some median blocking was observed in noisy sequences, such as the cartoon sequence. On the whole this deinterlacer coped very well with the cartoon sequence. Unlike the VT deinterlacer, the edges remained free from artifacts. This is no doubt due to the diagonal information incorporated into the algorithm. Sometimes the noise would generate an incorrect gradient, which would lead to an incorrect interpolation, at which point the median would protect the image.
- Figure 34 illustrates the quiz scene without noise reduction. The spatial Gaussian noise reducer (figure 35) is quite effective at reducing the noise from the background, however it has the expected side effect of eliminating all high frequencies from the sequence. The presenter looks blurred as his fine details are not present.
- Figure 36 shows off the Adaptive Recursive Temporal Averager noise reduction strategy. Here the background is noise reducer well (though not as well as the Gaussian approach), & the presenter has kept himself together quite well.
- The Recursive Temporal Averager could not cope with the amount of noise in the cartoon or quiz sequences. Its effect is subtle & I would only recommend it for use eradicating EMI noise. Broadcast noise is beyond its capabilities.
- Both the temporal noise reduction algorithms suffered the expected limitation of objects under motion being noisy. The effect is most noticeable when the object is in your tracking view rather than your periphery view, as when you are tracking it its relative velocity is effectively 0. The noise is barely discernable in otherwise.
- None of the noise reduction techniques could eliminate the vertical banding noise in the quiz sequence. This is because its frequency is too low. There is a point at which it becomes too difficult to discern noise from real scene elements, & in other sequences this banding could be intended, so it was expected that without a specialist algorithm this noise would remain.
- Very noisy sequences does not benefit from any of the noise reducers on test.
I compared my best-performing deinterlace algorithm, 3D Edge Orientation with 5-point median, to dScaler’s best-performing, the aptly named Video Deinterlace.

This evaluation was difficult, as dScaler does not support recording to disk. This meant I could not compare the same exact frames in the same video sequence for relative artifacts. Also, sadly time constraints stopped me from hooking up a video recorder & viewing the same video sequence on both TV viewers. It was, however, quite easy to stop dScaler’s weaknesses (figures 37 & 38).

dScaler’s algorithm is as follows. \( x \) represents the pixel to be interpolated. For each \( x \):

1. Weave \( x \) ‘s last temporal neighbour if it is similar to either of \( x \) ‘s current vertical neighbours. ‘Similar’ means within a set spatial tolerance.
2. Also weave \( x \) ‘s last temporal neighbour if there has not been motion between the last frame & this frame. This is calculated by comparing the difference between \( x \) ‘s vertical neighbours & last temporal neighbour with \( y \) ‘s vertical neighbours & last temporal neighbour. \( y \) is the location of \( x \) in the last frame \( I(x, y, t-2) \).
3. If neither of the two above conditions is satisfied, the \( x \) is bobbed.

The algorithm is hard to analyse mathematically, it has clearly be devised by subjective means.

Comments on the comparison:

- dScaler’s algorithm works similar to a 3-point median deinterlacer, but the eventual alias around notice due to dScaler’s protection algorithm:

![Figure 37: dScaler artifacts](image-url)
output is a linear interpolation rather than a nearest neighbour. This is certainly preferential. An illustration would be the difference between the nearest neighbour & bi-linear interpolation image processing techniques for resizing images.

- In the dScaler algorithm, the spatial & temporal tolerances are fine tuned by the subjective analysis of balancing high frequency artifacts (weave like ‘jaggies’), with low frequency artifacts (bob like blurring). Figure 38 shows a scene under medium horizontal global motion. dScaler’s tolerances see this movement & bob accordingly. The notice with the green edges exhibits gaps in the green edge as dScaler interpolates between the white wall above & the white notice below. The 3D edge orientated algorithm would be more likely to interpolate diagonally along the green line, thus retaining more of the detail of the green edges of the notice.

- However, the 3D edge orientated algorithm’s 5-point median would kick in to make the pixel white unless both the spatial & temporal neighbours exhibited motion (with a strong gradient, as you cannot find local motion on a uniform object). This is another argument against using a median protection, as it can clobber a correct interpolation.

- A 5-point median is superior to dScaler’s protection though, because under motion dScaler’s protection will always produce a value that does not increase the vertical frequency (except by laxing the spatial tolerance). So a horizontal line pattern exhibiting vertical motion will exhibit disappearing lines (as the lines are decimated by the interlacing process). A 5-point median will not decimate the line as long as one is present in the previous & last temporal neighbours.

- The dScaler algorithm always blurs items under local motion (see figure ). The 3D edge orientated algorithm can detect the edge orientation to ensure the edges are blurred the minimum possible.

Figure 38: dScaler: Fast horizontal motion blurring, as diagonal information not taken into account.
I am quite pleased that I managed to prove some commentators wrong & showed that real-time
deinterlacing & noise reduction was possible with the DirectShow architecture on current PC’s.
The Vertical-Temporal deinterlacer can operate on 300Mhz computers whilst combined noise
reduction & 3D edge orientated deinterlacing is possible on 1Gz+. CPU usage is of little
consequence when the majority of use of the TV viewer will not coincide with the computer
performing other activities. This achievement now facilitates a whole host of additional
functionality that was not possible using previous programs:

- Support for a wider range of current & future video input sources, including most
  notably digital video through the IEEE 1394 Firewire protocol & the new crop of DTV &
  HDTV TV cards.
- Real—time processing of AVI files.
- Implicit support for real-time AVI & Mpeg processing streamed from the Internet.
- Simultaneous preview of processed video & recording to an AVI in any format supported
  by DirectShow (slow computers need not apply!).
- Easy integration of other features such as time-shifted video recording & live instant
  replay.
- Component architecture allows straightforward integration of the video-processing filter
  in 3rd party products such as DVD players & video editing suites.

The implementation of the Hauppauge WinTV Card filter finally gives European Hauppauge
users the ability to use a program other than the simple TV application bundled with their cards
(there does exist a WDM driver for NTSC cards).

I am glad to have made the first non-proprietary step at integrating academic research papers on
deinterlacing into a PC software based system. Comparison tests showed a quality enhancement
over the best deinterlacing techniques found in previous software. The discovery of the 3D Edge
Orientated algorithm being well suited, if CPU intensive, to an SSE implementation was
particularly pleasing. The Adaptive Recursive Temporal Averager noise reduction method is
robust enough to suppress light noise without adding artifacts to the video output. This is ideally
suited to alleviate the problems of poor TV tuner shielding on some modern TV cards.

On a more personal note, there were several things I learnt from implementation difficulties with
this project:

- My previous experience had been exclusively with Java, & some work with the Java
  Media Framework. I assumed incorrectly that Microsoft DirectShow was equivalent in
  implementation to the JMF. In reality, DirectShow almost gives no help to the filter
developer, except for a few constraining Base Classes. Even the most routine tasks you
have to implement yourself, as there is a lack of a standardised API in Visual C++ in
general.
- This lead me to overestimate the amount I could achieve with the system, which you can
  attest to if you read my outsourcing document.
- There are too many special cases & workarounds you have to provide in the DirectShow
  framework. Many 3rd party filters & hardware devices have not been written fully to the
  specification.
- There is a lack of good support in Microsoft Visual C++ 6 for writing inline assembly in
  an architecturally sound way. It is missing macro support, & avoiding duplication of
  code meant implementing a novel procedure of preprocessor directives.
To sum up, I am pleased to achieve the objective of a usable (as in compatible with today’s TV Cards) TV Display application with plenty of future scope for enhancement.

**Limitations**

It is only natural in a project such as this that the limitations & aspirations far out way the achievements. For instance, the majority of the implementation was concentrated on the video-processing filter. Learning & programming filter’s in the DirectShow framework was no easy task. Less than a week was spent on the application, which shows just how easy it is for the end developer to add a large amount of DirectShow functionality into a working product, but it also means that so much more could have been done if the time were available. At this point, the viewer does its intended job of demonstrating the Video Processing & Hauppauge TV Card filters, but it requires considerable enhancement to be attractive to customers:

- Time shifted recording
- VCR like interface
- Channel auto-tuning & a convenient selection mechanism.
- Teletext & subtitle support.

One notable limitation the video processing filter does not handle is stream quality control. DirectShow includes quality control messages sent upstream from the video renderer to the video source. This allows frames to be skipped at the source if the renderer is having trouble keeping up. The messages should be processed by the video-processing filter to step up & step down quality in two ways:

1. Output of the deinterlacer could be reduced to 25Hz.
2. The strategy employed could be dynamically configured to relate to system load.

This would be an improvement to the jitter that currently occurs as whole frames are thrown away by the Video for Windows source filter. The factory classes include a mechanism to report the speed & subjective quality of the strategies. This could be used by the quality control implementation.

With such a vast array of deinterlacing algorithms available in electrical engineering research, it was challenging to find the most ideal algorithm that would trade off quality against speed on a PC MMX/SSE implementation. I am confident the 3D Edge Orientated algorithm is an improvement on what was previously available, but it would take more research to validate that it could not be bettered for the processor overhead price. The future lies with motion compensated algorithms for only they can include more than token motion information (the edge orientated algorithms are unusual in that they implicitly account for motion of 1 pixel per field). A motion estimator based on the 3D recursive search block estimator (see proposed extensions) was prototyped but I could not get the motion field accurate enough for worthwhile implementation – deinterlacing requires sub-pixel accurate motion vectors to avoid artifacts more damaging than non-motion compensated deinterlacing, I attribute this failure to incomplete information on the algorithm due to its patented nature. Much more research would have to be carried out to validate this point, however.

In the short term, a faster & better performing alternative for the 5-point median protection would enhance the deinterlacing algorithms by a fair amount. Available research has not brought
any forward, however, but I expect a technique based on spatial interpolation would reduce the noticeable artifacts.

Some improvement of the performance of the SSE algorithms could be achieved by incorporating the SSE cache control instructions. However, the algorithms are all virtually spatially linear in memory reading (there are no table lookups), so the improvement may not be great considering the default cache behavior is already optimised for this for the SSE instructions.

The noise reduction strategies were perhaps less effective than I had hoped, but my expectations were probably too high. Whilst any improvement is some improvement, the current algorithms are good for EMI caused noise, but cannot really be effective on high levels of broadcast noise. A motion compensated algorithm would improve the temporal averages in that they could also noise reduce moving objects. But the really effective techniques, such as Wiener filtering [10], are cost minimisation problems so definitely don’t lend themselves to real-time implementation even in the foreseeable future.

I would have also liked to have had the time to create other video-processing filters like gamma filters, missing data reconstruction & broadcast ghost removal (for which I had a couple of interesting ideas), but time constraints & DirectShow constraints relating to implementation time meant I had to limit the scope of the project.

Finally, the Hauppauge TV Card filter serves its purpose well, but it was found half way through development that it would not have the scope I had originally intended. I based the filter around a DirectShow concept that turned out had not been implemented by Microsoft – hierarchical filter graphs. My original intention was for the TV Card filter to ‘pull’ in a Hauppauge VFW video source filter on insertion into a filter graph. If this could happen seamlessly then the filter could be designated a video capture source itself, & other TV applications such as PowerVCR could use the filter without modification.

CONCLUSION

In the proposed extensions section following, I illustrate a recent technique to improve the accuracy of the 3D Edge Orientated deinterlacer. This may pave the way for an algorithm that does not require a 5-point median protection. I also discuss a motion estimator developed by Philips that is currently used in their 100Hz televisions, together with how it could be applied to motion compensated deinterlacing & noise reduction.

On reflection, I have made some progress into the emerging field of real-time video processing on PC’s. I have no doubt that more will be made by specialist companies in the following years, as new technologies such as Intel’s Pentium 4 with SSE2, PC HDTV cards & home cinema integration flourish. This is a technology led sector where consumers have to be educated as to the possibilities offered, but once they are educated they need no convincing of the benefits. It is real that PC’s are beginning to threaten the domain of high-end consumer audio & video equipment, & Hifi / AV companies will have to ponder their continuing strategies.

In the short term, this project has given the ordinary consumer the ability to watch television or another video source on his PC with quality improved from previous programs. Also new is the ability to deinterlace & record the video for later viewing simultaneously. And with the application using the DirectShow framework, he can be assured that it will continue to function with future TV cards.
The technique for improving the edge orientated deinterlacing discussed in [21] concentrates on reducing the number of candidates used in finding the edge direction. The theory being that noise could adversely affect the outcome of the edge detector if too many orientations are used in its calculation.

[21] only discusses a spatial edge orientation algorithm, & I will soon extend it to the temporal dimension too. Concentrating on the spatial plane for the time being, we only calculate the edge gradients at 60 degrees to the horizontal. See figure 39:

The 2 candidates are calculated by averaging the 60° gradients at either side of the pixel to be restored:

\[
W_i = \frac{|a - e| + |b - f|}{2} \\
W_2 = \frac{|b - d| + |c - e|}{2}
\]  

(25)

\[
F_{edge}(x, y, n) = \begin{cases} 
X_a & U_a = \text{MIN}(U_a, U_b) \land W_i < W_2 \\
X_b & U_b = \text{MIN}(U_a, U_b) \land W_i < W_2 \lor U_b = \text{MIN}(U_a, U_b) \land W_i \geq W_2 \\
X_c & U_c = \text{MIN}(U_a, U_b) \land W_i \geq W_2
\end{cases}
\]  

(26)

Where U & X are defined as before. So we use the values of W for the gradients {60°, -60°} to restrict our search of the minimum gradient of the U candidates, the {45°, 0°, 45°} gradients. The least likely edge direction given the minimum W is discarded.

Extending this algorithm to include the temporal direction involves recalculating it for a,b,c,d,e,f in the temporal-horizontal plane. We are then left with finding the minimum of 4 U candidates to decide the X interpolator to use.
Papers [28] & [29] describe a fast block-based motion estimation algorithm patented by Philips, & implemented in their 100Hz Televisions. The major properties of this deinterlacer are:

- Designed for implementation on IC’s – low memory overheads, simple block matching architecture.
- Sub-pixel accurate motion vectors, which are demanded by motion compensated deinterlacing techniques.
- A very smooth motion field to reduce the possibility of harmful artifacts caused by incorrect motion vectors.
- Objects are assumed to be large than a block.
- Objects are assumed to have inertia (an objects motion vector does not change much between frames).
- Most motion can be found only by considering neighbouring blocks.

Consider diagram 40:

![Diagram 40](image)

Figure 40: Aperature of the 3D Recursive Search Block Matcher

Assuming blocks are processed from the top-left, the motion vector for the current block is one of the motion vectors from its 3 spatial neighbours shown in diagram 4, or its 5 temporal neighbours, which are its spatial neighbours in the previous field. A standard block-matching algorithm (such as the SAD or MSE) is then applied between the candidate block & the candidate block shifted by the 9 candidate vectors. The best fit is the current block’s assigned motion vector.

For the motion field to converge to a true representation of the sequence (at time 0 all the motion vectors would be 0), the assigned vector must be randomly shifted slightly by an update set. Including non-integer shift in the update set allow convergence to a sub-pixel estimation.

\[
U_{sc} = \begin{bmatrix} 0 & 0 & 0 \\
0 & 1 & -1 \\
-1 & 2 & -2 \\
0 & 0 & 0 \\
1 & 0 & 3 \\
0 & 0 & -3 \\
0 & 0 & 0 \\
0 & 0 & 0.25 \\
0 & 0 & -0.25 \\
0 & 0 & 0 \
\end{bmatrix}
\]  \hspace{1cm} (26)

A post-processing erosion step smoothes the motion field to remove blocking artifacts.

All non Motion Compensated algorithms can be upgraded with this motion estimator by replacing references to the last field with references to the last field brought forward in time by the motion vectors. This should alleviate the need for adaptive algorithms that switch off the temporal techniques under motion.
For example, the MC Time Recursive deinterlacer [30] uses the previously deinterlaced frame instead of the previous field in a MC Weave algorithm. Sampling rate conversion theory can be used to interpolate the samples required to de-interlace the current field:

\[
I(x, y, t) = \begin{cases} 
I(x, y, t), & (y \mod 2 = n \mod 2) \\
I(x - d_{(x, y, t), y - d_{(x, y, t), t - 1}}) & \text{(otherwise)}
\end{cases}
\] (27)

However, this depends on accurate motion vectors & an original perfectly de-interlaced frame to start from.
APPENDIX A - BIBLIOGRAPHY


[23] E. Gamma, R. Helm, R. Johnson & J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, (1994)


APPENDIX B - LINKS

[1]  Ian Harries, Teaching Associate, Department of Computing, Imperial College, ih@doc.ic.ac.uk

[2]  Dr Guang-Zhong Yang, Director Royal Society/Wolson MIC Laboratory, Department of Computing, Imperial College, gzy@doc.ic.ac.uk


[5]  Pete Harrison, Professor, Department of Computing, Imperial College, pgh@doc.ic.ac.uk

[6]  Gerard de Haan, Research Fellow, Video Processing & Visual Perception Group, Philips Research, Eindhoven g.de.haan@philips.com


DirectShow TV Display with Real-time Video Processing
