UNIVERSIDAD POLITECNICA DE VALENCIA

ESCUELA POLITECNICA SUPERIOR DE GANDIA

UNIVERSIDAD
POLITECNICA
DE VALENCIA

"AJUSTE DE FILTROS DIRECTSHOW DEL SISTEMA DE EMISIÓN DE VÍDEO DE LA UPV TV"

TRABAJO FINAL DE CARRERA

Autor/es:
EFRÉN MIÑANA MARTÍNEZ

Director/es:
ANTONI JOSEP CANÓS MARÍN
IGNACIO DESPUJOL ZABALA

GANDIA, 2010

Agradezco a toda mi familia el apoyo recibido durante todos mis estudios universitarios y muy especialmente a mi querida Esther Pérez Prada.

También a todos los componentes de la UPV TV el trato recibido durante la realización del proyecto en sus instalaciones.

Y agradecer inmensamente la ayuda y conocimientos recibidos de mi cotutor Ignacio Despujol Zabala y mi tutor Antoni Josep Canós Marín.

ÍNDICE

0. Objetivos y plan de trabajo 5
1. Estado inicial del sistema de continuidad automática de la UPV TV y su problemática 7
2. Filtros DirectShow 11
    2.1. Filtros y grafos 11
    2.2. Tipos de filtros 14
    2.3. Conexión entre filtros y negociación 15
    2.4. Funcionamiento interno de un grafo 16
3. El reproductor Zoomplayer 19
4. El editor GraphEdit 23
5. Microsoft Visual Studio 27
6. Filtros Decklink desarrollados 31
    6.1. Filtro Decklink Composite 31
    6.2. Filtro Decklink PAL Field Swap 52
7. Conclusiones y aplicaciones futuras 59
Bibliografía 61
Anexo 1: El problema de la dominancia de campos en el vídeo entrelazado 63
Anexo 2: Contenido del CD 71

0. OBJETIVOS Y PLAN DE TRABAJO

El objetivo principal del presente Trabajo Final de Carrera se enmarca en la sustitución del sistema de continuidad actual de la UPV TV por uno que consiga mayor calidad. Para ello, se pretende ajustar un sistema que proporcione calidad *broadcast* a 720x576 entrelazado.

Asimismo se quiere conseguir un sistema de reproducción estable y que represente en pantalla los vídeos de una forma correcta según la norma PAL utilizada en España.

Básicamente los problemas principales que se tienen en el sistema de emisión de la UPV TV, son con respecto a una serie de vídeos que se encuentran con una resolución de 704x576, así como otros vídeos que tienen la secuencia de campos de imagen invertidos, es decir, que presentan el campo lower (impar) primero y a continuación el upper (par), pese a estar etiquetados en su cabecera como upper.

Debido al mal etiquetado de la dominancia en los vídeos en la cabecera del paquete MPEG2, el espectador puede observar vibraciones en la imagen debido a la mala representación de los campos en pantalla. Al estar mal etiquetas, el reproductor de vídeo representa en pantalla los campos invertidos sin darse cuenta de ello, es decir, primero representa el campo lower y a continuación el upper. La dominancia de campos en vídeo entrelazado está explicada en el Anexo 1 de este texto.

Encontrar la solución a estos problemas es la enmienda de este Trabajo Final de Carrera.

Para ello se creará un sistema basado en filtros DirectShow, descritos en el capítulo 6, que detecte la inversión de campos PAL en el sistema de reproducción de vídeo y corrija, representando en primer lugar el campo par y a continuación el impar, y evitando el parpadeo de las imágenes en movimiento. También se realizará el paso de los vídeos 704 x 576 a otros con calidad *broadcast*, 720 x 576.

Para poder implementar todas estas soluciones se ha optado por cambiar las tarjetas de vídeo empleadas por una tarjeta Blackmagic que permite la edición y creación de diferentes filtros DirectShow específicamente para ella.

Para ello ha sido necesario familiarizarse con el manejo de diferentes herramientas como ZoomPlayer, descrito en el capítulo 3; el editor GraphEdit, descrito en el capítulo 4; el entorno de desarrollo integrado Visual Studio 2005, descrito en el capítulo 5; y el Software Development Kit (SDK) de las tarjetas Blackmagic.

1. ESTADO INICIAL DEL SISTEMA DE CONTINUIDAD AUTOMÁTICA DE LA UPV TV Y SU PROBLEMÁTICA.

Comenzaré describiendo el sistema de continuidad automática que se utiliza en la radiotelevisión de la Universidad Politécnica de Valencia, para obtener así una visión general de su estado antes de la realización de este trabajo.

El sistema de continuidad automática consta de un ordenador con el reproductor multimedia ZoomPlayer versión 5.0 y una tarjeta de vídeo con salida especial de vídeo compuesto: la Matrox Parhelia. La tarjeta Matrox Parhelia es la única cuya salida de vídeo que proporciona una señal que se ha considerado lo suficientemente buena para la emisión. Todas las tarjetas de ATI y NVIDIA que se habían probado hasta la fecha desentralazan el vídeo y lo vuelven a entrelazar, dando problemas de estelas, además de modificar ligeramente el aspect ratio y otras características. ZoomPlayer es el reproductor seleccionado ya que es un programa con licencia shareware que permite un control exhaustivo de los filtros DirectShow empleados para la reproducción, algo crucial para obtener una reproducción correcta. Otra de las ventajas que nos proporciona este reproductor es la utilización de listas (playlists) de texto plano de fácil configuración y que pueden ser cargadas desde la línea de comandos, lo que permite la automatización de las tareas de emisión con ficheros .bat y .txt.

El formato empleado para la automatización de vídeo es el MPEG2 IBP por varios motivos:

a) La mayor parte de los contenidos externos que llegan a la UPV TV procedentes de diferentes entidades públicas como ATEI, Diputació, o Cultura, lo hacen en formato DVD (MPEG2) y realizar la conversión del formato supondría un tiempo adicional del personal técnico del que no se dispone.

b) Un archivo MPEG2 ocupa entre un tercio y un cuarto del tamaño que ocuparía el mismo archivo en formato AVI DV, lo que, dada la disponibilidad de almacenamiento, supone una ventaja adicional.

El problema en la reproducción aparece dada la amplitud e indefinición del formato MPEG2, que permite múltiples variantes en formato que no siempre son entendidas y/o reproducidas correctamente por los programas reproductores. Además, también hay que tener en cuenta los problemas de edición inherentes a la compresión utilizada. Esto ha causado repetidos percances en la reproducción de los distintos ficheros, los cuales se

han ido solventando a medida que aparecían, no pudiendo descartar más en el futuro, ya que pueden aparecer en una combinación no probada.

El sistema operativo Windows utiliza para reproducir contenidos audiovisuales el subsistema DirectShow. Directshow se basa en el uso de diferentes programas específicos para tareas determinadas. Dichos programas se denominan "filtros" y son conectados entre sí formando "gráficos" o "grafos", como puede verse en la figura 1. Para reproducir un archivo MPEG2 en DirectShow deberemos conectar a la salida del fichero un filtro, que se denomina demultiplexor o separador (splitter), que nos separe el audio y el video que contiene el archivo, permitiéndonos a partir de éstas dos salidas tratar el audio y el vídeo por separado. El vídeo pasa a un descompresor de vídeo que decodifica MPEG2, y a cuya salida se conecta el sistema de renderizado de Windows (WMR) que es capaz de representar el vídeo en la pantalla del ordenador. El audio, por su parte, se conecta a un filtro descompresor de audio que lo suministra al sistema de reproducción de audio de Windows. La peculiaridad está en que entre cada uno de estos filtros se pueden introducir otros con funciones específicas de tratamiento de la señal. Algunos serán desarrollados en éste TFC.

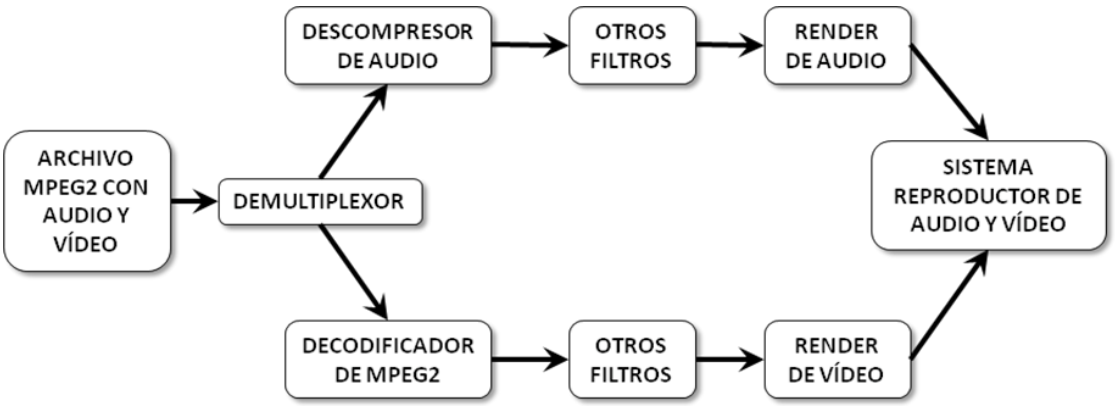

Figura 1. Grafo de filtros DirectShow

Directshow tiene un sistema de méritos o prioridades, mediante numeración, encargado de discernir, en caso de tener más de un filtro con la misma capacidad de reproducción de contenidos, cual de ellos será utilizado (el que tenga el mérito más alto). Asimismo dispone de un sistema de negociación automática entre filtros que permite seleccionar los filtros más adecuados para la reproducción de cada tipo de contenido.

Cuando un usuario instala un pack de códecs, como K-LITE, se realiza automáticamente la asignación de este sistema de méritos, sin que el usuario sepa exactamente qué se está haciendo. Casi todos los códecs necesarios están incluidos en el K-LITE codec pack, que permite instalarlos fácilmente y sin alterar los méritos de los filtros ya instalados. Si por cualquier motivo aparece algún problema, podemos realizar la instalación de cada códec de forma individual e incluso manualmente. Para poder registrar los filtros en el sistema (Windows) se ha de utilizar la herramienta regsvr32.exe como en cualquier archivo .dll (Dynamic Link Library)

ZoomPlayer también permite cambiar el mérito de los filtros registrados en el sistema, con la necesidad de reiniciar el sistema cada vez que realizamos un cambio para que éste tenga efecto. Además, desde las últimas versiones aparecidas, dispone de la herramienta SMARTPLAY que permite al usuario saltarse el sistema de reproducción de Windows y elegir los filtros que se deseen utilizar para cada tipo de contenidos.

Hasta el momento se estaba utilizando, como se ha indicado anteriormente en este documento, una tarjeta Matrox Parhelia. Para evitar problemas de imágenes en negro al realizar el cambio de vídeo a vídeo en la lista de reproducción del sistema de continuidad, hay que utilizar el sistema de reproducción de vídeo (video render) más antiguo de los que ofrece Microsoft (video renderer, overlay mixer, WMR 7 y WMR 9), que es posible seleccionar desde ZoomPlayer.

Como filtro de reproducción para el audio se emplea el AC-3 Filter que es gratuito y de gran potencia para el tratamiento de un gran número de tipos de audio. Es necesario utilizar la última versión disponible (posteriores a 1.45) ya que hay algunas versiones anteriores que no son capaces de manejar audio a 32 kHz proveniente de la conversión de ficheros AVI DV y provocan problemas con la sincronización entre el vídeo y el audio.

Se ha constatado, en referencia este tema, el problema de reloj en algunos ficheros de la AETEI que ha obligado a utilizar el formato de audio de MPEG1. Hay que tener en cuenta que podemos configurar lo anterior a través de la herramienta SMARTPLAY según el tipo de audio a tratar.

El demultiplexor utilizado es Mainconcept MPEG Splitter incluido con Windows, ya que proporciona una reproducción con menos saltos y funciona con mayor número de ficheros.

El filtro de reproducción de vídeo utilizado es Mainconcept Vídeo Decoder ya que es el único que permite extraer el contenido del vídeo sin realizar ningún proceso de desentrelazado, y produce una gran calidad de imagen.

Existe un problema al reproducir vídeo entrelazado debido a que en MPEG2 el vídeo puede tener distinta dominancia de campo (upper o lower), siendo esta característica marcada en un flag de cada frame de la trama MPEG2, concretamente en la cabecera TS del paquete MPEG2. Los reproductores de vídeo de sobremesa no tienen ningún problema para reproducir vídeos con cualquiera de las dominancias ya que comprueban que el flag contenido en la cabecera TS del paquete MPEG2 coincida con el que aparece en el VIDEOINFOHEADER del vídeo codificado en la parte de datos del paquete MPEG2. La mayoría de vídeos con los que se trabaja en la UPV TV tienen dominancia upper, ya que son los que produce el grabador de disco duro que hay en el estudio, así como los que envía ATEI a la UPV TV. Asimismo, todos los vídeos que se capturan y convierten de DV a MPEG2 partiendo de uno original con dominancia lower, seguirán teniendo dicha dominancia una vez convertidos, pese a que el grabador siempre los etiqueta como upper, lo que provoca que su representación en pantalla sea con los campos invertidos, provocando el parpadeo de la imagen que observa el espectador.

Debido a este problema específico se plantea la solución de utilizar una tarjeta Blackmagic Decklink Pro, sustituyendo a la Matrox Parhelia, y desarrollando un filtro que sea capaz de realizar el cambio de la dominancia del vídeo reproducido en función del bit de la trama MPEG2.

La solución que adopto para corregir el problema es desplazar una línea hacia arriba la imagen que se está tratando en ese momento, aprovechando que las últimas líneas en PAL no son visibles en pantalla, y de esta forma corregir en tiempo real la mala representación de la imagen en pantalla.

2. FILTROS DIRECTSHOW.

2.1 FILTROS Y GRAFOS

Los filtros DirectShow tienen su origen con la creación del proyecto Quartz que Microsoft inició al lanzar al mercado Windows 95. La idea inicial de este proyecto era suministrar la funcionalidad de vídeo para Windows (VFW) con soporte MPEG a 32 bits. Microsoft creó un set de interfaces de programas (API's) que proporcionaban funciones de captura de vídeo y audio que podían ser utilizados con los nuevos dispositivos digitales de vídeo de la época.

El fuerte crecimiento y desarrollo de dichos dispositivos hizo que Microsoft cambiara el proyecto Quartz a otro basado en una arquitectura "Framework" que permite colocar varios elementos juntos, como si aplilláramos una serie de fichas de dominó que casaran unas con otras. Quartz proporcionaba una serie de componentes de construcción llamados filtros para proporcionar funciones básicas como leer de un fichero de datos, reproducirlos por un altavoz o renderizarlos para ser reproducidos en una pantalla. Además, utilizando los por entonces nuevos COM (Microsoft Component Object Model), Quartz juntó los filtros formando los llamados grafos (filter graph).

Los filtros Microsoft DirectShow, desde el punto de vista del programador, se componen de dos tipos de clases de objetos:

- Filtros: Código fuente que al compilar formará un filtro
- Grafos (filter graph): Colección de filtros conectados entre sí para realizar una función determinada.

Los filtros están compuestos de pins, o lo que es lo mismo, conectores de entrada/salida que reciben un stream de vídeo por una entrada y lo envían por uno de salida hacia otro filtro.

Si nos referimos al código de programación puramente dicho, podemos entender los filtros como llamadas a una función que tiene una serie de variables de entrada y devuelve una salida, mientras que los grafos son programas que contienen las llamadas a esas funciones.

Una estructura a modo de ejemplo sería:

```
Grafo(){
    Función Filtro 1();
    Función Filtro 2();
    ….
}
```

Así pues, los grafos DirectShow ejecutan de manera secuencial desde el primer filtro hasta el último. La diferencia con respecto a un programa desarrollado en el lenguaje común de C++ es que se ejecuta de forma continua. Cuando un grafo inicia su ejecución los datos comienzan a fluir de un filtro a otro, es lo que se conoce como stream, dentro del grafo.

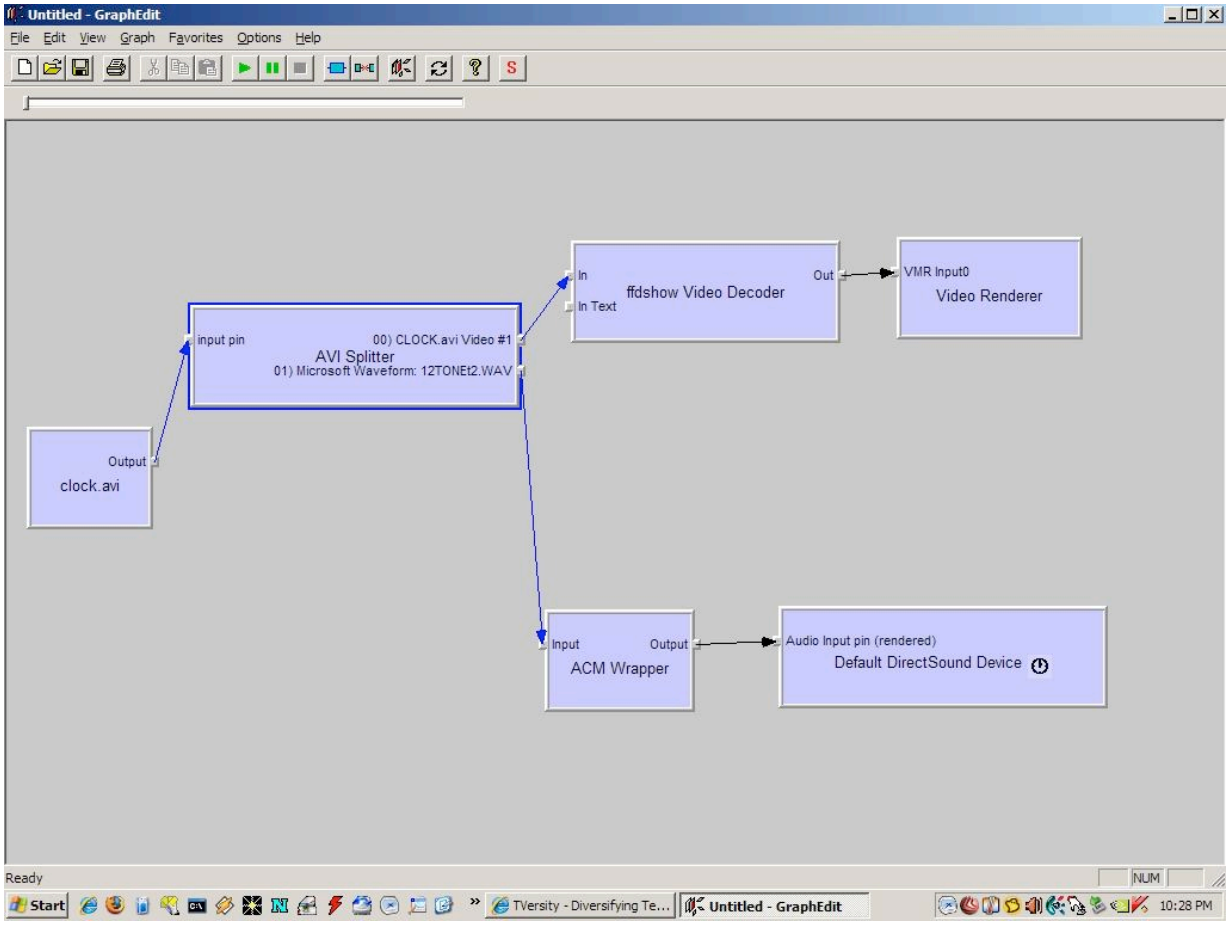

Figura 2. Grafo de ejemplo en GraphEdit

Otro detalle importante a destacar de los grafos es que admiten varios streams al mismo tiempo (depende de cómo sean programados), así como encaminamientos por distintos filtros para cada uno de ellos en el grafo. De hecho, podríamos realizar una captura de vídeo al mismo tiempo que capturamos el audio con un mismo grafo, que de no tener este funcionamiento, deberíamos utilizar un filtro para capturar el audio y otro diferente para capturar el vídeo. Asimismo, podríamos enviar un stream de audio y vídeo a un filtro de render para mostrarlo por pantalla, mientras que almacenamos otro en el disco duro. En definitiva, las combinaciones pueden ser infinitas.

Los filtros DirectShow tienen la capacidad de modificar el valor de los bits del stream que los atraviesa, pero son incapaces de tomar decisiones que afecten a la estructura del grafo, es decir, no van a ser capaces de modificar un determinado grafo sino que simplemente el stream de datos atravesará un filtro dentro del grafo y éste se lo pasará a otro filtro al cual esté conectado.

DirectShow define una serie estándar de objetos de programación COM (Component Object Model) que permiten a cualquier programador utilizar una serie de funciones ya predefinidas y que no tiene porqué comprender. Por ejemplo, cuando estemos programando una aplicación para capturar vídeo en formato AVI (Audio Vídeo Interleaved) no va a ser necesario conocer el proceso de codificación para este formato, sino simplemente utilizar una secuencia de filtros correcta en un grafo.

Los filtros DirectShow son “inteligentes” ya que son capaces de reconocer con qué otros filtros se pueden conectar y con cuáles no, así como la forma en la que pueden conectarse con ellos. Esto es debido a que comprueban los pins de entrada y salida de cada uno de ellos para saberlo. De esta forma, si se conectan 2 filtros entre sí, el conjunto formado por ambos, va a ser capaz de detectar si pueden conectarse a un tercer filtro o no.

A este proceso se le conoce como negociación, y todos los filtros lo realizan a la hora de intentar conectarse con otros.

Los filtros tienen una serie de propiedades definidas para poder establecer conexiones con otros, así como negociar con ellos los tipos de conexiones que pueden aceptar.

Esta estructura modular hace que programar filtros DirectShow se convierta en una tarea mucho más sencilla ya que se van a poder utilizar objetos COM ya creados para formar otro COM que sea del interés del programador. Además esta estructura modular se extiende hasta los grafos por lo que podemos crear grafos que sean capaces de elegir

qué filtros de los que lo componen ha de conectar entre sí dependiendo de los datos que tenga a la entrada. Incluso podríamos crear un COM de un grafo en el que utilicemos un filtro para renderizar.

El problema de esta cierta inteligencia que tienen los grafos es que el programador no sabrá realmente por qué filtros pasa el stream de vídeo ya que, si existe más de un filtro que pueda realizar la misma tarea, será el grafo automáticamente quien decida qué filtro utilizar.

2.2 TIPOS DE FILTROS

Existen tres tipos de filtros, los llamados Source Filters, los Transform Filters y los Render Filters.

Cualquier filtro que produzca un stream de datos es un Source Filter. El stream se puede almacenar en un archivo en el disco duro, o se envía a un dispositivo capaz de emitirlo en directo como un monitor, o a un grabador de vídeo.

Los más interesantes, y de hecho en los que se basa este Trabajo Final de Carrera, son los Transform Filters o filtros de transformación. Un filtro de este tipo recibe a la entrada un stream de datos desde otro filtro y realiza la transformación, tanto en el vídeo como en el audio del stream para la que esté programado y, finalmente lo envía a otro filtro. Por poner un ejemplo, si creamos un filtro (o grafo) que permita realizar la conversión entre audio codificado en WAV y audio codificado en MP3, este sería del tipo Transform Filter. Estos filtros también permiten crear varias copias exactas del stream que los atraviesa a tiempo real, es decir, podemos crear un filtro para tener varias salidas simultáneas del mismo stream que está procesando dicho filtro. Esta utilidad es muy interesante ya que mediante un filtro de éste tipo podríamos estar emitiendo un stream al mismo tiempo que lo almacenamos en un archivo en el disco duro y lo grabamos con un magnetoscopio grabador. Además, este tipo de filtros también puede realizar el proceso inverso, es decir, tomar varios streams de datos a la entrada y ofrecer un único stream a su salida.

En este Trabajo Final de Carrera se van a utilizar este tipo de filtros para crear un filtro (DecklinkComposite.cpp) para solucionar el problema de reescalado de los streams de vídeo que no tienen una calidad broadcast (720x576) y que se emplean en la UPV TV.

El último tipo de filtros existentes son los llamados Render Filters que traducen el stream de datos al formato de cualquier dispositivo de salida. El más básico de todos los de este tipo escribe el stream de datos en un archivo en el disco duro. Existen filtros de este tipo que envían un stream de audio a unos altavoces directamente o uno de vídeo a una ventana en el escritorio del ordenador. Los Render Filters utilizan DirectDraw y DirectSound que podemos decir que son tecnologías que permiten a los filtros DirectShow pasar un stream de vídeo, o audio, a la tarjeta de vídeo o de sonido del equipo.

Un grafo puede tener múltiples Render Filters para poder enviar diferentes tipos de formatos del stream de salida a elementos que utilicen diferentes formatos como por ejemplo una tarjeta de vídeo, un archivo en el disco duro o un magnetoscopio grabador.

En definitiva, cualquier grafo consta de la combinación de estos 3 tipos de filtros y ha de estar formado por un Source Filter, un Transform Filter, y un Render Filter.

2.3 CONEXIÓN ENTRE FILTROS Y NEGOCIACIÓN

La conexión entre filtros se realiza mediante los pines de entrada y salida de los filtros. Como ya he comentado, no siempre va a ser posible realizar la conexión entre dos filtros, ya que va a depender de la negociación que realicen sobre el tipo de stream que pueden soportar. Cada filtro ha de publicar los tipos de stream que puede enviar y recibir, así como una serie de técnicas para enviar dicho stream desde un pin de salida hasta uno de entrada. Cuando un grafo intenta conectar un pin de salida de uno de sus filtros con uno de entrada de otro comenzará este proceso de negociación, es decir, el grafo examinará los tipos de media (tipos de streams) que puede transmitir el pin de salida y los tipos de media que el pin de entrada del otro filtro puede aceptar. Si no existiera ningún tipo de media igual en ambos pines, la conexión no se podría realizar y nos aparecería un error.

Además, los pines han de aceptar el mecanismo de transporte, y si esto no sucede, nos aparecería otro error en la conexión.

Si ambos pines tienen un tipo de media en común y utilizan el mismo mecanismo de transporte, los pines crearán lo que se llama un allocator, que no es más que un objeto que crea y dirige el buffer por donde circulará el stream desde el pin de salida hasta el pin de entrada. Dicho allocator puede ser de cualquiera de los dos pines ya que realmente da igual porque ambos pines han de estar de acuerdo con el tipo de media y con el mecanismo de transporte.

Cuando se cumpla todo lo anterior, los pines estarán conectados.

Para conectar otros pines se seguirá el mismo proceso que el que he descrito anteriormente.

Como he comentado antes, la inteligencia de los filtros reside precisamente en ser capaces ellos mismos de saber con qué filtros se pueden conectar y, más concretamente, con qué pines de dichos filtros.

2.4 FUNCIONAMIENTO INTERNO DE UN GRAFO

Respecto a los grafos, podríamos decir que organizan un grupo de filtros, como ya se ha comentado, para conseguir una determinada función. Han de indicar a los filtros que contienen cuando comenzar a realizar sus operaciones, cuando parar y cuando esperar. Además, los filtros han de estar sincronizados entre sí ya que el tipo de datos que utilizan (streams) necesitan dicha sincronización. Cuando un programador utiliza cualquiera de estos 3 comandos básicos en un grafo (play, pause y stop), éste envía un mensaje a cada filtro que forma el gráfo para que acate la orden dada. Esta información se conoce como stream data, es decir, se enviará una ráfaga de datos de control para el filtro. Dicho stream se eliminará una vez finalizada su tarea. Para que quede más claro utilizaré un ejemplo concreto. Si le indicamos que comience la ejecución a un grafo que hayamos creado, éste enviará un stream data a los filtros que lo forman. Una vez dichos filtros han recibido dicha información, dicho stream data se elimina y el filtro (o filtros) comienzan a ejecutarse. Lo mismo sucedería para el caso en que ordenáramos que el grafo parase, pero si lo que queremos es que se pause el proceso, no se eliminará dicha orden (stream data) hasta que le indiquemos si queremos que pare definitivamente o que comience de nuevo la ejecución.

Para que se pueda entender un poco mejor el funcionamiento de los filtros DirectShow, voy a explicar el ciclo que sigue un stream dentro de un grafo.

Los grafos organizan los filtros para conseguir una funcionalidad general. Cuando se conecta un grafo, los filtros presentan un camino a seguir por los streams de datos a través de un filtro de transformación (Transform Filters) hacia un filtro de renderización (Render Filter). Los filtros que diseñan el camino a través de los filtros, son los llamados Source Filters, o filtros que contienen la fuente de datos (como por ejemplo una cámara de vídeo). Los grafos han de indicar a los filtros cuando comenzar a funcionar, cuando

para y cuando ponerse en pausa. Además, los filtros han de estar sincronizados ya que van a tratar un tipo de datos (streams) que necesitan mantener una sincronización para poder ser reproducidos correctamente. Debido a ello, el grafo genera una señal de reloj que está disponible para todos los grafos y, de ésta forma, conseguir que los filtros transmitan el stream de datos ordenados cuando pasen la información de un filtro a otro.

Cuando indicamos al grafo que arranque, pare o se ponga en pause, éste manda un mensaje a todos los filtros con dicha orden, por lo que los filtros han de ser capaces de interpretar cualquiera de las 3 órdenes.

Si pensamos en la situación de tener que capturar vídeo desde una cámara que trabaja en DV (Digital Vídeo) y almacenarlo en el disco duro, los filtros que van a intervenir son:

- Source Filter ---------- Cámara de vídeo DV.
- Transform Filter ---------- Decodificador de vídeo DV.
- Render Filter ---------- Vídeo (archivo de vídeo).

El proceso que se va a seguir para realizar la tarea será el siguiente:

La cámara de vídeo generará una muestra de vídeo (stream) que se captura con un “Capture Source Filter” ya que se está utilizando una fuente de vídeo en vivo. La cámara está conectada directamente a un PC (mediante Fireware), por lo que es una situación bastante común a la hora de realizar ingestas de vídeo en un servidor de vídeo. El Capture Source Filter, manda un comando a la cámara indicándole que está preparado para recibir datos, y en ese instante la cámara comienza a mandar el stream. Asimismo, se encarga de recoger el stream de datos y convertirlo en muestras, marcadas con un tiempo para mantener el sincronismo, que se volverán a convertir en un stream para ser enviados como un stream de datos formateado ahora para DirectShow.

De esta forma ya puede enviar, a través del pin de salida un stream de datos DirectShow con sincronismo hacia el filtro de transformación (Transform Filter) que en este caso es un decodificador de vídeo DV. El filtro de transformación trata de dos formas al stream de datos que recibe, por un lado tratará el espacio de color convirtiéndolo a otro si fuera necesario para poder realizar alguna función determinada (por ejemplo convirtiendo de YUV a RGB), y por otro lado desentrelazará el vídeo entrelazado procedente del Capture Source Filter. Al utilizar vídeo entrelazado tenemos que un fotograma se descompone en dos campos. El primero de ellos lleva toda la información de las líneas impares del

fotograma del que proceden, mientras que el segundo campo contiene toda la imformación de las líneas pares. El proceso de desentrelazar consiste precisamente en combinar los dos campos de cada fotograma para obtener un stream de vídeo desentrelazado, que es con el que trabajan todos los monitores de vídeo. Así pues, estos procesos de conversión de color y desentrelazado que realiza el filtro de transformación, consiguen crear un stream que va a poder ser tratado por un Render Filter y ser reproducidos en un monitor de vídeo. Cuando el filtro de transformación finaliza estos procesos, sitúa el stream de vídeo en el pin de salida.

Finalmente el stream llega a su destino, es decir al Render Filter, que se va a encargar de dibujar el stream de datos en un monitor de vídeo. Conforme le va llegando cada muestra del stream de vídeo (fotograma) lo va dibujando en pantalla. Todos son dibujados en orden ya que el filtro se encarga de comprobar el tiempo que tienen marcado cada uno de ellos. Para almacenar las muestras que no son recibidas en orden, el filtro crea un buffer en memoria y de esta manera almacenarlos hasta el momento de su reproducción.

Mediante este proceso, hemos visto como actúa un grafo en una situación bastante común como la de reproducir el vídeo que estamos capturando con una cámara en la pantalla de un PC. Se ha de tener en cuenta que el flujo del stream de datos a través del grafo no terminará hasta que le indiquemos una orden de parada o puesta en pausa.

A continuación voy a explicar la herramienta que nos permite ver este proceso, GraphEdit.

3. EL REPRODUCTOR ZOOMPLAYER

Tal y como he comentado en el punto anterior, el reproductor multimedia que se utiliza en la UPV TV es 'ZoomPlayer', concretamente su versión 5.0. Después de este trabajo se va a seguir utilizando este reproductor básicamente porque.

- La UPV TV posee una licencia shareware
- Permite un control total de los filtros Directshow
- Tiene un sistema de listas de reproducción de texto plano fáciles de cargar desde la línea de comandos para la automatización de las tareas de emisión con ficheros .bat y .txt.

El reproductor ZoomPlayer presenta, una vez arrancado, la pantalla que se muestra en la figura 3. Como puede apreciarse, es un interfaz muy sencillo, en el que básicamente podemos obsevar dos ventaras: una de ellas en la que tenemos la lista de reproducción que se está reproduciendo; y otra en la que se reproduce el vídeo. Mediante la combinación de teclas CTRL + O conseguimos acceder a las opciones del reproductor, desde donde podemos configurar desde los botones que queremos que aparezcan en la ventana de reproducción, hasta el orden de los filtros DirectShow a utilizar en la reproducción de los vídeos de una lista de reproducción . En la figura 4 se muestra una imagen de dicha ventana de opciones avanzadas del reproductor.

Figura 3. ZoomPlayer

La configuración del reproductor utilizado para el desarrollo de este Trabajo Final de Carrera, en cuanto a interfaz gráfica, no es muy importante para el objetivo del mismo, ya que simplemente se deciden en esta sección cuales son los botones que queremos ver en la ventana de reproducción, cuales son los atajos de teclado a utilizar para diferentes acciones, cuánto tiempo tarda en desaparecer el puntero del ratón mediante la reproducción de un vídeo y un sin fin de distintas posibilidades en cuanto a la utilización de la interfaz del reproductor y los distintos periféricos de los que se dispone en un PC.

A continuación tenemos las opciones de reproducción “Playback” donde se incluye la utilidad más importante para el control total de los filtros de reproducción, SmartPlay.

SmartPlay nos permite tener un control total de cómo se deben leer y decodificar los archivos de vídeo con los que trabaje ZoomPlayer. Cuando ejecutamos ZoomPlayer por primera vez en un ordenador realiza una búsqueda en el sistema de cuales son los componentes que tenemos instalados para reproducir vídeo. Hay veces que se tienen instalados varios componentes, cada uno para un tipo de vídeo, e indicando la forma de decodificarlos y reproducirlos, así como los conflictos que podrían causar. Esta característica permite a ZoomPlayer conocer de antemano cuales son los componentes más adecuados según el tipo de vídeo que queramos reproducir y crear un perfil avanzado para ayudar a reproducir los vídeos de una forma rápida y cómoda para el usuario.

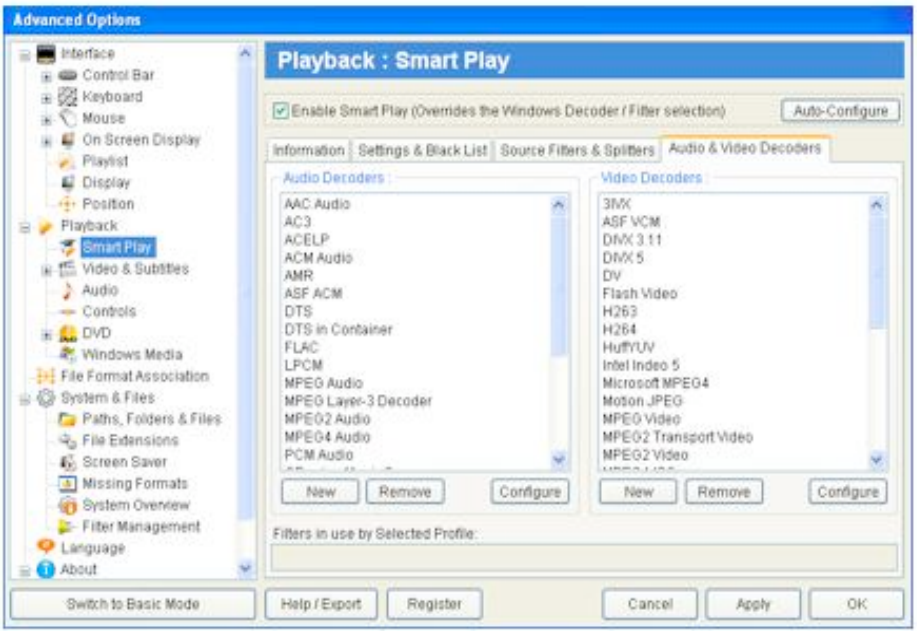

Figura 4. SmartPlay

Vemos en la figura 4 que para utilizar esta herramienta basta con marcar el check que aparece junto a “Enable SmartPlay”. También disponemos de un botón “Auto-Configure” cuya ejecución provoca el reescaneo de nuestro sistema en busca de componentes que tengamos instalados para reproducir vídeos. En definitiva, lo que realiza este botón es la búsqueda automática de los filtros que tengamos instalados en nuestro equipo.

La utilización de ésta herramienta es parte esencial del correcto desarrollo de este trabajo ya que nos permite seleccionar, según la compresión de vídeo utilizada, los filtros que nos interesen. Para poder seleccionar un filtro en concreto, basta con que éste esté registrado en el sistema, mediante el comando regsvr32 de Windows que nos permite registrar los archivos .dll generados de la compilación del filtro.

Una vez tengamos los filtros registrados, desde esta utilidad seleccionando la opción MPEG2 vídeo, de la columna de Vídeo Decoders y pulsando el botón Configure, aparecerá una pantalla como la mostrada en la figura 5.

En la parte superior de la ventana vemos una lista de perfiles predefinidos para decodificar este tipo de vídeos. Estos filtros predefinidos son los que la opción autoconfigure de ZoomPlayer ha encontrado instalados en nuestro equipo y ha considerado utilizables para realizar la codificación de éste formato de compresión de vídeo (códec). Zoom Player crea un perfil para cada uno de ellos automáticamente.

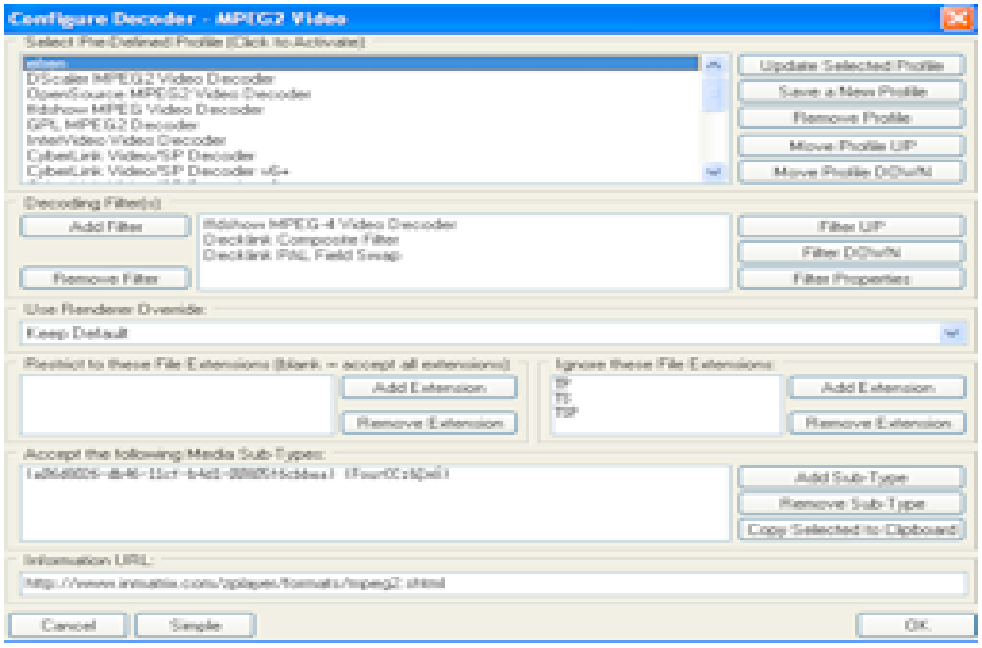

Figura 5. Configuración del Decodificador

Desde esta sección de la ventana y mediante la utilización de la botonera existente a la derecha de la lista de perfiles, vamos a ser capaces de eliminar perfiles, así como crear uno personalizado para nuestro uso, tal y como se puede apreciar en la figura 5.

He creado un perfil con el nombre Efrén que será el que contenga los filtros a utilizar para la decodificación de este tipo de vídeos. Los filtros que utiliza este perfil, así como el orden de aplicación de los mismos a los streams de vídeo, se pueden ver en la parte central de la ventana de la figura 5 en forma de lista.

La configuración utilizada para el correcto funcionamiento de este trabajo es, por orden de aplicación:

- Mainconcept MPEG decoder
- Decklink Composite Filter
- Decklink Pal Field Swap Filter

Mainconcept MPEG decoder es un decodificador de vídeo MPEG2 que no realiza un desentrelazado del vídeo, con lo que se consigue una mejor calidad de imagen una vez finalizado el proceso que con otros decodificadores como ffdshow MPEG-2 decoder, que sí realiza un desentrelazado del vídeo para realizar este proceso.

A continuación se aplica el filtro Decklink Composite Filter que se explicará ampliamente en el siguiente apartado de esta memoria, ya que es un filtro creado expresamente para el correcto reescalado de los vídeos que no cumplan con la resolución estándar para broadcast, 720 x 576.

Finalmente se aplica el filtro Decklink Pal Field Swap para corregir los vídeos que tengan la secuencia de campos PAL incorrecta, es decir, que pese a estar etiquetados como upper, realmente tengan el campo lower primero.

Mediante esta configuración de Zoom Player se pueden aplicar los filtros que he desarrollado para la correcta reproducción, ya que éstos quedarán registrados en el registro de Windows y podré seleccionarlos desde la ventana que aparece en la figura 5, para utilizarlos en la decodificación del vídeo.

Hay que tener en cuenta que Zoom Player tiene muchísimas características más que se pueden configurar a medida del usuario, pero que no son esenciales para el correcto desarrollo del objetivo de este proyecto.

4. EL EDITOR GRAPHEDIT

Este programa se entiende como un papel en blanco en el que vamos a poder ir introduciendo los filtros que tengamos registrados en nuestro PC para ir probando combinaciones de los mismos hasta que consigamos la función que estemos persiguiendo.

En mi caso ha sido de gran utilidad en la prueba de los filtros DecklinkComposite y PalFieldSwap, ya que de forma sencilla se puede ver cual es el encaminamiento que se utiliza dentro del grafo para su utilización, así como comprobar si las conexiones entre los filtros se realizan de forma correcta.

La interfaz es la que se muestra en la figura 6. Es una utilidad bastante sencilla de manejar si se tienen claros los conceptos que he descrito en el punto anterior.

Para entender el tipo de grafos que vamos a poder crear tenemos que listar todos los filtros DirectShow que tenemos en nuestro equipo.

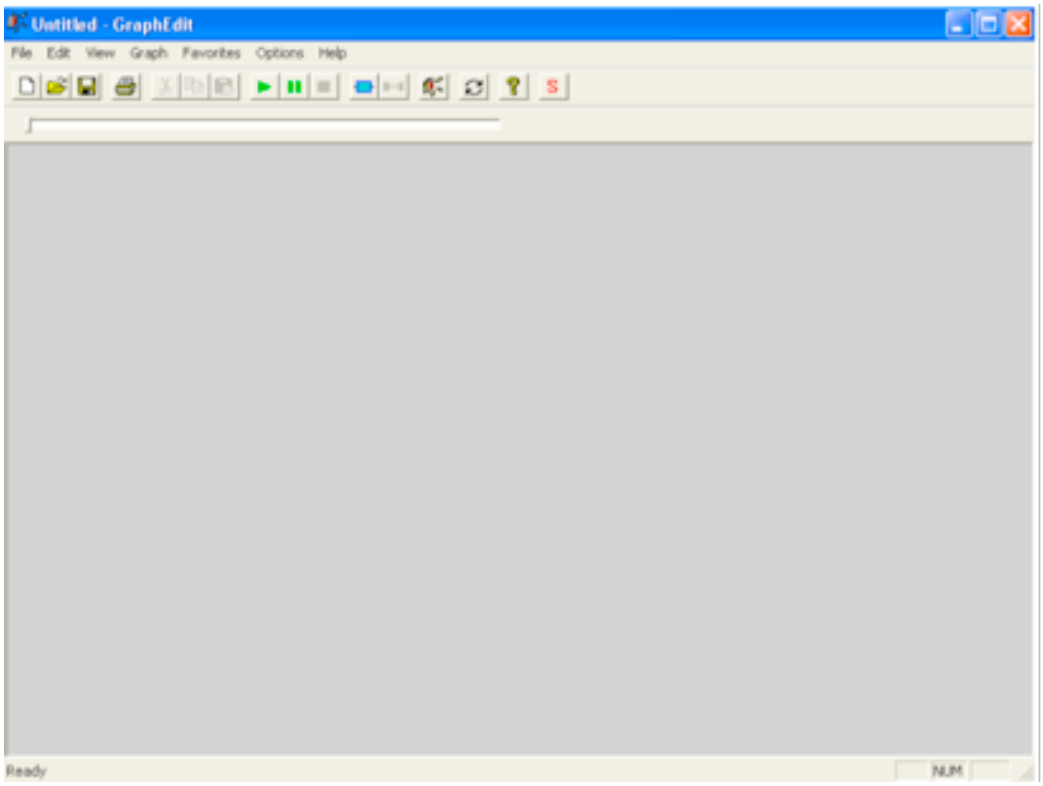

Figura 6. Interfaz de GraphEdit

Figura 7. Ventana de selección de Filtros

Esta opción la tenemos en GRAPH / INSERT FILTER, y desde ésta ventana, que aparece en la Figura 7, vamos a seleccionar que filtro vamos a insertar en el grafo. No es necesario abrir la ventana para insertar uno por uno cada uno de los filtros, sino que se pueden ir seleccionando desde esta venta e ir pulsando el botón INSERT para que se vayan insertando en el grafo.

Cada uno de los filtros que insertamos aparecen como una caja como se puede ver en la Figura 8, donde simplemente vemos un vídeo titulado Sunset.avi que tiene un pin de salida etiquetado como “output”, y otro filtro, “AVI Splitter” que posee un pin de entrada etiquetado como “input pin” y otros dos de salida etiquetados como “Stream 00” y “Stream 01” respectivamente.

Tal y como se ve la conexión entre la salida del primer filtro con la entrada del segundo, se representa con una flecha que nos indica cual es la dirección que siguen las muestras de vídeo.

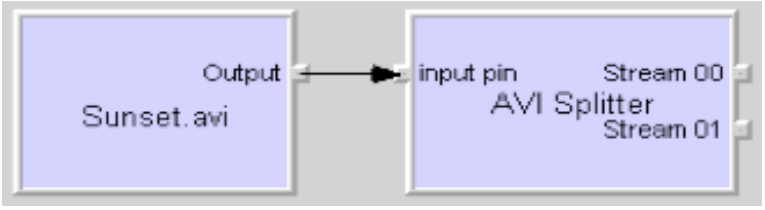

Figura 8. Grafo simple

Esta va a ser la estructura de todos los filtros, los cuales tendrán más o menos pins de entrada o salida, dependiendo de cuales sean sus funciones.

El filtro “AVI Splitter” realiza la separación del audio y el vídeo, y por ello tiene dos pins de salida.

Si en este esquema introducimos un decodificador de vídeo (DV) y un vídeo render conectados a la salida “Stream 00” conseguiremos obtener una imagen en la ventana de salida de vídeo de la aplicación, al pulsar la tecla de reproducir o “play”.

Pero si además queremos ser capaces de escuchar el sonido, debemos conectar al segundo pin de salida el dispositivo de reproducción de audio que tengamos por defecto.

Si realizamos ambas inserciones, y conectamos los filtros entre sí, obtendremos la estructura de la Figura 9.

En el filtro “Default DirectSound Device” se ve que aparece un reloj, lo cual nos indica que todo el sincronismo entre los filtros va a estar sincronizado por este filtro.

Así pues, ya hemos creado un grafo que nos permite reproducir por pantalla el vídeo que contiene el filtro fuente (Source Filter), que en este caso no es más que el archivo de vídeo sweet.avi que tenemos en el disco duro de nuestro PC.

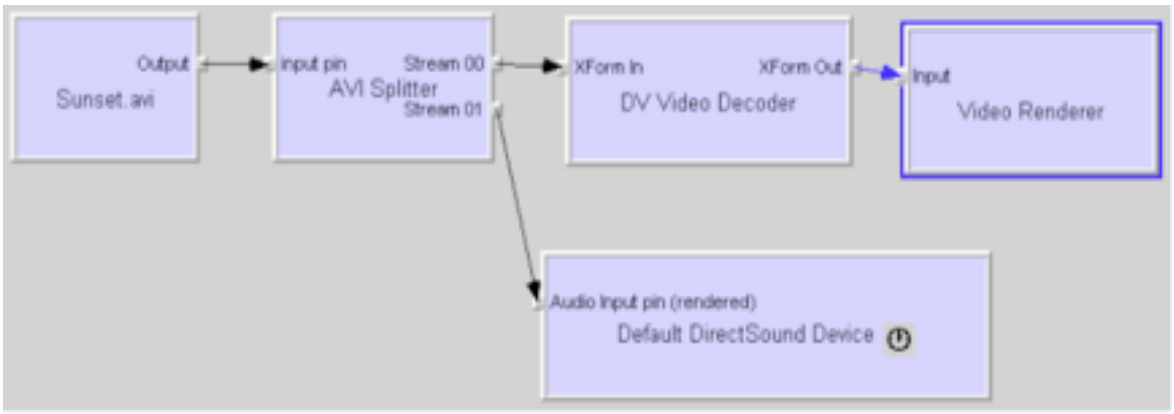

Figura 9. Grafo con reproducción de vídeo y audio

Si presionamos play en este momento, el resultado va a ser el mostrado en la Figura 10:

Se abrirá ante nosotros una ventana en la que comenzará a reproducirse el vídeo, al mismo tiempo que podremos escuchar el audio que contenga el archivo.

Esta es la forma en la que tenemos que utilizar este software para facilitar el correcto desarrollo tanto de nuevos filtros como de aplicaciones para el tratamiento de imágenes.

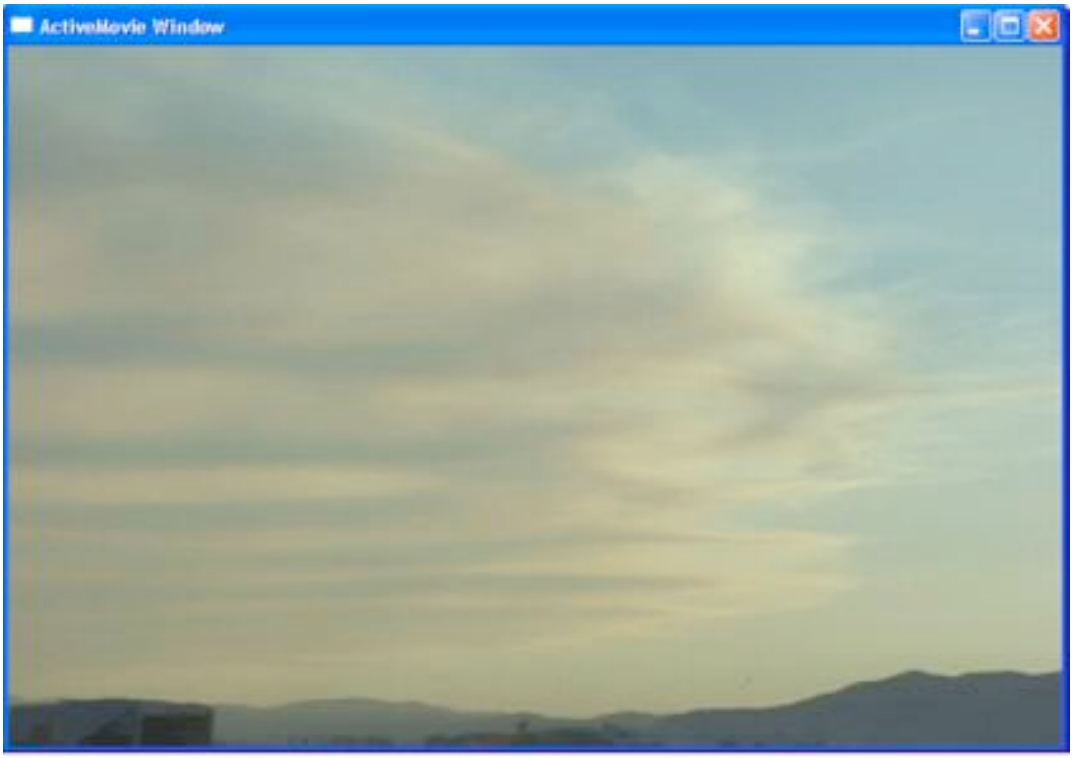

Figura 10. Ventana de reproducción de GraphEdit

5. MICROSOFT VISUAL STUDIO

Es un entorno de desarrollo integrado (IDE, por sus siglas en inglés) para sistemas Windows. Soporta varios lenguajes de programación tales como Visual C++ (que es el empleado para el desarrollo de este trabajo), Visual C#, Visual J#, ASP.NET y Visual Basic .NET, aunque actualmente se han desarrollado las extensiones necesarias para muchos otros.

En este trabajo final de carrera se emplea para desarrollar los filtros DirectShow con los que trabajará el nuevo sistema de continuidad de la UPV TV, debido a que el SDK de Blackmagic trabaja sobre él.

Visual Studio permite a los desarrolladores crear aplicaciones, sitios y aplicaciones web, así como servicios web en cualquier entorno que soporte la plataforma .NET (a partir de la versión net 2002). Así se pueden crear aplicaciones que se intercomuniquen entre estaciones de trabajo, páginas web y dispositivos móviles.

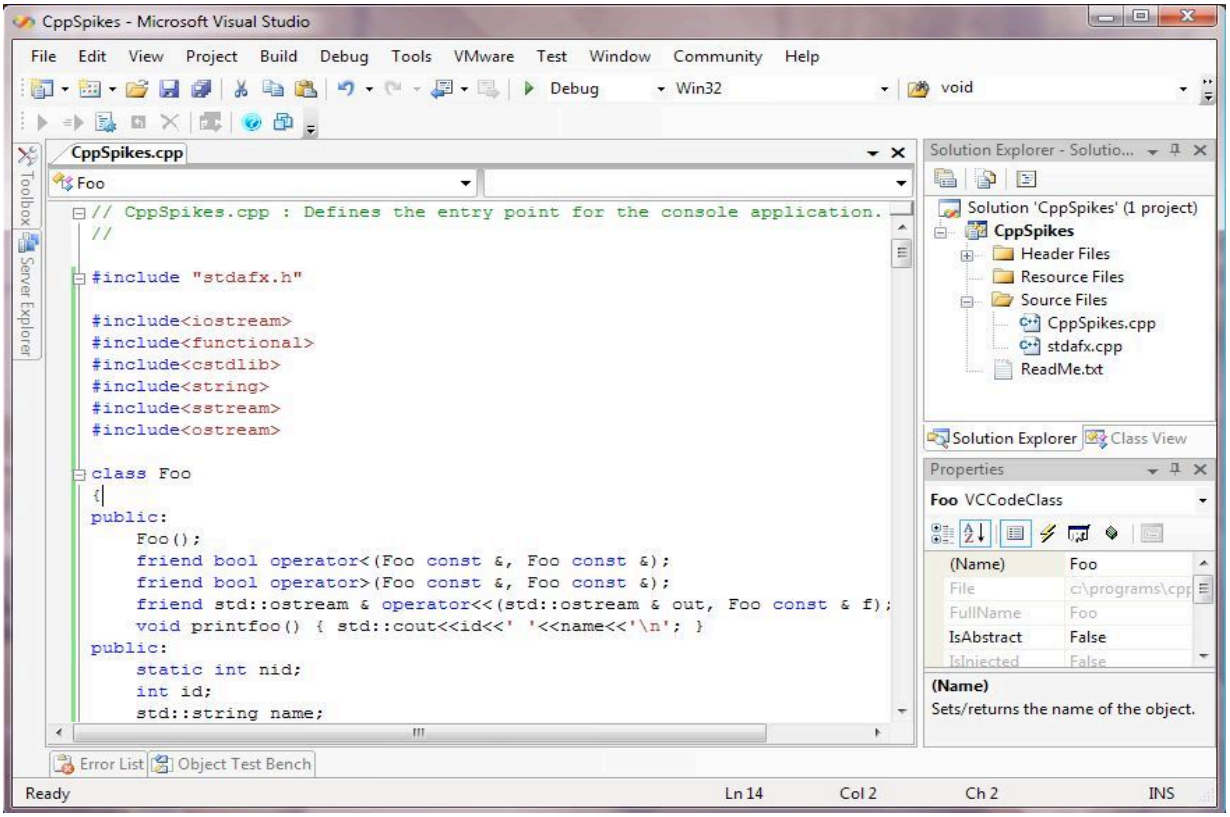

Figura 11. Ventana de Visual Studio 2005

A partir de la versión 2005 Microsoft ofrece gratuitamente las Express Editions. Estas son varias ediciones básicas separadas por lenguajes de programación o plataformas enfocadas para novatos y entusiastas. Estas ediciones son iguales al entorno de desarrollo comercial pero sin características avanzadas. Las ediciones que hay son:

- Visual Basic Express Edition
- Visual C# Express Edition
- Visual C++ Express Edition
- Visual J# Express Edition (Desapareció en Visual Studio 2008)
- Visual Web Developer Express Edition (para programar en ASP.NET)

La versión utilizada para el desarrollo de éste trabajo es Visual Studio 2005.

Visual Studio 2005 se empezó a comercializar a través de Internet a partir del 4 de Octubre de 2005 y llegó a los comercios a finales del mes de Octubre en inglés. En castellano no salió hasta el 4 de Febrero de 2006. Microsoft eliminó .NET, pero eso no indica que se alejara de la plataforma .NET, de la cual se incluyó la versión 2.0.

La actualización más importante que recibieron los lenguajes de programación fue la inclusión de tipos genéricos, similares en muchos aspectos a las plantillas de C#. Con esto se consigue encontrar muchos más errores en la compilación en vez de en tiempo de ejecución, incitando a usar comprobaciones estrictas en áreas donde antes no era posible. C++ tiene una actualización similar con la adición de C++/CLI como sustituto de C# manejado.
Se incluye un diseñador de implantación, que permite que el diseño de la aplicación sea validado antes de su implantación. También se incluye un entorno para publicación web y pruebas de carga para comprobar el rendimiento de los programas bajo varias condiciones de carga.
Visual Studio 2005 también añade soporte de 64-bit. Aunque el entorno de desarrollo sigue siendo una aplicación de 32 bits Visual C++ 2005 soporta compilación para x86-64 (AMD64 e Intel 64) e IA-64 (Itanium). El SDK incluye compiladores de 64 bits así como versiones de 64 bits de las librerías.

Visual Studio 2005 tiene varias ediciones radicalmente distintas entre sí: Express, Standard, Professional, Tools for Office, y 5 ediciones Visual Studio Team System. Éstas últimas se proporcionaban conjuntamente con suscripciones a MSDN cubriendo los 4 principales cometidos de la programación: Architects, Software Developers, Testers, y

Database Professionals. La funcionalidad combinada de las 4 ediciones Team System se ofrecía como la edición Team Suite.

Tools for the Microsoft Office System está diseñada para extender la funcionalidad a Microsoft Office.

Las ediciones Express se han diseñado para principiantes, aficionados y pequeños negocios, todas disponibles gratuitamente a través de la página de Microsoft[2] se incluye una edición independiente para cada lenguaje: Visual Basic, Visual C++, Visual C#, Visual J# para programación .NET en Windows, y Visual Web Developer para la creación de sitios web ASP.NET. Las ediciones express carecen de algunas herramientas avanzadas de programación así cómo de opciones de extensibilidad.

Se lanzó el service Pack 1 para Visual Studio 2005 el 14 de Diciembre de 2006.

La versión interna de Visual Studio 2005 es la 8.0, mientras que el formato del archivo es la 9.0.

Empleando esta plataforma se puede desarrollar cualquier aplicación para Windows, tal y como se ha hecho en este trabajo.

6. FILTROS DECKLINK DESARROLLADOS

Se han desarrollado dos filtros Decklink para conseguir que el sistema de emisión corrija automáticamente los fallos de los que es objeto este proyecto final de carrera.

El primero de ellos, Decklink Composite, se emplea para transformar los vídeos de 704x576 a calidad broadcast (720x576), mientras que el segundo, Decklink Pal Field Swap, se emplea para detectar cuando un stream tiene como primer campo el campo lower de la imagen y lo transforma por el campo upper mediante la eliminación de una de las líneas del mismo, ya que siempre ha de ir primero el campo upper.

Comenzaré explicando el filtro Decklink Composite, para el que se han realizado una serie de modificaciones respecto al que está disponible en la versión 7.1 del SDK.

6.1. FILTRO DECKLINK Composite.

Mediante este filtro se realiza la adecuación de vídeos en 704x576 a 720x576, es decir, a calidad broadcast. Lo que realmente se va a hacer es centrar la imagen en pantalla para que la imagen final tenga un aspecto de 720x576 ya que las barras laterales no van a ser apreciadas por el usuario final.

La realización de este filtro, se basa en un ejemplo que existe en el SDK de Blackmagic (como se puede ver en los datos adjuntos de este trabajo), y sobre el que he realizado una serie de modificaciones, resaltadas en color azul y cursiva, para que se realice la transformación de la imagen deseada.

El contenido de dicho filtro es el que se muestra a continuación:

```
//------------------------------------------------------------------------------
// $Id: DecklinkComposite.cpp,v 1.2 2007/05/04 04:56:25 ivanr Exp $
//
// Desc: DirectShow sample code - Illustrates a basic frame composition by taking
//                                a non-SDI source frame and compositing onto
an
//                                SDI frame size for rendering with the Decklink
//                                video renderer.
//------------------------------------------------------------------------------

#include "stdafx.h"
#include "Dshow.h"
#include "DecklinkFilters_h.h"
```

```
#include "DecklinkComposite.h"
#include "DecklinkCompositeProp.h"
#include "..\resource.h"
//------------------------------------------------------------------------
// CreateInstance
// Provide the way for COM to create a CDecklinkComposite object
CUnknown* WINAPI CDecklinkComposite::CreateInstance(LPUNKNOWN punk, HRESULT* phr)
{
    HRESULT hr = S_OK;
    CDecklinkComposite* pFilter = static_cast<CDecklinkComposite*>(new CDecklinkComposite(NAME("Decklink
Composite filter"), punk, &hr));

    if (NULL == pFilter)
    {
        *phr = E_OUTOFMEMORY;
    }

            if (FAILED(hr) && phr)
            {
                    *phr = hr;
            }

    return pFilter;
}

//------------------------------------------------------------------------
// Constructor
//
CDecklinkComposite::CDecklinkComposite(TCHAR* tszName, LPUNKNOWN punk, HRESULT* phr)
    : CTransformFilter(tszName, punk, CLSID_DecklinkComposite)
{
            m_mtOutput.InitMediaType();


}

//------------------------------------------------------------------------
// Destructor
//
CDecklinkComposite::~CDecklinkComposite()
{
}


//------------------------------------------------------------------------
```

```
// NonDelegatingQueryInterface
//
STDMETHODIMP CDecklinkComposite::NonDelegatingQueryInterface(REFIID riid, void** ppv)
{
	if (riid == IID_ISpecifyPropertyPages)
	{
		return GetInterface(static_cast<ISpecifyPropertyPages*>(this), ppv);
	}

	if (riid == IID_IAMStreamConfig)
	{
		return GetInterface(static_cast<IAMStreamConfig*>(this), ppv);
	}

   return CTransformFilter::NonDelegatingQueryInterface(riid, ppv);
}

//----------------------------------------------------------------------------
// Transform
//
HRESULT CDecklinkComposite::Transform(IMediaSample* pIn, IMediaSample* pOut)
{
   CheckPointer(pIn, E_POINTER);
   CheckPointer(pOut, E_POINTER);
   HRESULT hr = S_OK;

   // transform the sample data
   BYTE* pSrc, *pDst;
   long cbSrc = pIn->GetActualDataLength();
   long cbDst = pOut->GetActualDataLength();

   pIn->GetPointer(&pSrc);
   pOut->GetPointer(&pDst);


	if (pSrc && pDst)
	{
		BITMAPINFOHEADER* pbmihSrc = CUtils::GetBMIHeader(m_pInput->CurrentMediaType());
		BITMAPINFOHEADER* pbmihDst = CUtils::GetBMIHeader(m_pOutput->CurrentMediaType());

		if (pbmihSrc && pbmihDst)
		{
			//--------------------------------------------------------------------
			// TODO: for RGB formats honour -ve heights
```

```
                        LONG srcHeight = (pbmihSrc->biHeight);
                        LONG dstHeight = (pbmihDst->biHeight);



                        unsigned long srcRowBytes = cbSrc / srcHeight;
                        unsigned long dstRowBytes = cbDst / dstHeight;
                        unsigned long cbPixel = CUtils::GetImageSize(pbmihSrc) / pbmihSrc->biWidth / pbmihSrc-
>biHeight;
                        unsigned long cLines = 0, cbLine = 0;
                        if (srcRowBytes && dstRowBytes)
                        {
                                if (pbmihSrc->biWidth < pbmihDst->biWidth)
                                {


                //------------------------------------------------------------------------
                // source frame is smaller than the destination frame,
                // center the source frame in the destination frame
                // provide horizontal offset into destination buffer
                //          pDst += dstRowBytes/2;
pDst += (((pbmihDst->biWidth - pbmihSrc->biWidth)/2 * cbPixel) >> 1);
                                        cbLine = cbPixel * pbmihSrc->biWidth;
                                }
                                else
                                {
        // source frame is larger than than the destination frame,
        // show a small portion of the source in the destination
        pSrc += (((pbmihSrc->biWidth - pbmihDst->biWidth) * cbPixel) >> 1);
// provide horizontal offset into source buffer
                                        cbLine = cbPixel * pbmihDst->biWidth;
                                }


                                LONG offsetToPreventFieldSwap = 0;          // modifier to ensure a field swap does
not take place when copying interlaced media
                                if (srcHeight < dstHeight)
                                {
                // source frame is smaller than the destination frame,
                // center the source frame in the destination frame
                offsetToPreventFieldSwap = (dstHeight - srcHeight) % 2;
                pDst += (((dstHeight - srcHeight - offsetToPreventFieldSwap) >> 1) * dstRowBytes);          // provide
vertical offset into destination buffer
                                        cLines = srcHeight;
                                }
                                else
                                {
        // source frame is larger than than the destination frame,
        // show a small portion of the source in the destination
```

```
                offsetToPreventFieldSwap = (srcHeight - dstHeight) % 2;
                pSrc += (((srcHeight - dstHeight - offsetToPreventFieldSwap) >> 1) * srcRowBytes);          // provide
vertical offset into source buffer
                                cLines = dstHeight;
                        }

                        if (BI_RGB == pbmihSrc->biCompression)
                        {


        // honour -ve height RGB formats by inverting the frame, start at the top of the buffer
        // and move down through it, note that RGB formats are bottom up for +ve heights
                                if (pbmihSrc->biHeight < 0)
                                {
                                        pSrc = pSrc + (srcRowBytes * (srcHeight - 1));
                                        srcRowBytes = ~srcRowBytes + 1;
                                }

                                if (pbmihDst->biHeight < 0)
                                {
                                        pDst = pDst + (dstRowBytes * (dstHeight - 1));
                                        dstRowBytes = ~dstRowBytes + 1;
                                }
                        }

                        for (unsigned long line=0; line<cLines; ++line)
                        {
                                memcpy(pDst, pSrc, cbLine);
                                pSrc += srcRowBytes;
                                pDst += dstRowBytes;

                        }

                }
        }
        else
        {
                hr = E_POINTER;
        }

        if (SUCCEEDED(hr))
        {
                pOut->SetActualDataLength(pbmihDst->biSizeImage);

           // Copy the sample times
```

```
                REFERENCE_TIME rtStart, rtEnd;
                    if (SUCCEEDED(pIn->GetTime(&rtStart, &rtEnd)))
                    {
                            CRefTime rtStreamOut;
                            StreamTime(rtStreamOut);
                            TCHAR buf[128];
                            StringCchPrintf(buf, 128, TEXT("VID: %10I64d [%10I64d  %10I64d]  %10I64d"),
rtStreamOut, rtStart, rtEnd);
                            OutputDebugString(buf);
                            pOut->SetTime(&rtStart, &rtEnd);
                    }

                    LONGLONG rtMediaStart, rtMediaEnd;
                    if (SUCCEEDED(pIn->GetMediaTime(&rtMediaStart, &rtMediaEnd)))
                    {
                            pOut->SetMediaTime(&rtMediaStart, &rtMediaEnd);
                    }

                    // copy the sync property
                    if (S_OK == pIn->IsSyncPoint())
                    {
                            pOut->SetSyncPoint(TRUE);
                    }
                    else if (S_FALSE == pIn->IsSyncPoint())
                    {
                            pOut->SetSyncPoint(FALSE);
                    }

                    // copy the preroll property
                    if (S_OK == pIn->IsPreroll())
                    {
                            pOut->SetPreroll(TRUE);
                    }
                    else if (S_FALSE == pIn->IsPreroll())
                    {
                            pOut->SetPreroll(FALSE);
                    }

                    // copy the discontinuity property
                    if (S_OK == pIn->IsDiscontinuity())
                    {
                            pOut->SetDiscontinuity(TRUE);
                    }
                    else if (S_FALSE == pIn->IsDiscontinuity())
                    {
```

```
                                        pOut->SetDiscontinuity(FALSE);
                                }
                        }
                }
                else
                {
                        hr = E_POINTER;
                }

    return hr;
}
//------------------------------------------------------------------------
// CheckInputType
// Some basic checks on preferred formats.
HRESULT CDecklinkComposite::CheckInputType(const CMediaType* mtIn)
{
        CheckPointer(mtIn, E_POINTER);
        HRESULT hr = S_FALSE;
        // Make some very rudimentary checks on the media type
        if (MEDIATYPE_Video == mtIn->majortype)
        {
                if ((MEDIASUBTYPE_UYVY == mtIn->subtype) || (IID_MEDIASUBTYPE_HDYC == mtIn->subtype) ||
(MEDIASUBTYPE_YUY2 == mtIn->subtype) ||
                        (MEDIASUBTYPE_RGB24 == mtIn->subtype) || (MEDIASUBTYPE_RGB32 == mtIn-
>subtype) || (MEDIASUBTYPE_ARGB32 == mtIn->subtype))
                {
                        if ((FORMAT_VideoInfo == mtIn->formattype) || (FORMAT_VideoInfo2 == mtIn->formattype))
                        {
                                BITMAPINFOHEADER* pbmih = CUtils::GetBMIHeader(*mtIn);
                                if (pbmih)
                                {
                                        if (('YVYU' == pbmih->biCompression) || ('2YUY' == pbmih-
>biCompression) ||
                                                (((24 == pbmih->biBitCount) || (32 == pbmih->biBitCount)) &&
(BI_RGB == pbmih->biCompression)))
                                        {
                                                // acceptable media type
                                                hr = S_OK;
                                        }
                                }
                        }
                }
        }

        return hr;
}
```

```
//------------------------------------------------------------------------
// CheckTransform
//
HRESULT CDecklinkComposite::CheckTransform(const CMediaType* mtIn, const CMediaType* mtOut)
{
    CheckPointer(mtIn, E_POINTER);
    CheckPointer(mtOut, E_POINTER);

    HRESULT hr = CheckInputType(mtIn);
    if (SUCCEEDED(hr))
    {
                    if (mtOut->majortype == mtIn->majortype)
                    {
                            if (mtOut->subtype == mtIn->subtype)
                            {
                                    if ((FORMAT_VideoInfo == mtOut->formattype) || (FORMAT_VideoInfo2 == mtOut-
>formattype))
                                    {
                                            BITMAPINFOHEADER* pbmihIn = CUtils::GetBMIHeader(*mtIn);
                                            BITMAPINFOHEADER* pbmihOut = CUtils::GetBMIHeader(*mtOut);
                                            if (pbmihIn && pbmihOut)
                                            {
                                                    if (pbmihOut->biCompression == pbmihIn->biCompression)
                                                    {
                                                            // acceptable media type
                                                            hr = S_OK;
                                                    }
                                            }
                                    }
                            }
                    }
    }
            return hr;
}

//------------------------------------------------------------------------
// DecideBufferSize
//
HRESULT CDecklinkComposite::DecideBufferSize(IMemAllocator* pAlloc, ALLOCATOR_PROPERTIES* pProperties)
{
    CheckPointer(pAlloc, E_POINTER);
    CheckPointer(pProperties, E_POINTER);
            HRESULT hr = S_OK;
```

```
    // Is the input pin connected
    if (m_pInput->IsConnected())
            {
            //          BITMAPINFOHEADER* pbmih = CUtils::GetBMIHeader(m_pOutput->CurrentMediaType());
                        BITMAPINFOHEADER* pbmih = CUtils::GetBMIHeader(&m_mtOutput);

                        if (pbmih)
                        {
                                    pProperties->cBuffers = 1;
                                    pProperties->cbBuffer = CUtils::GetImageSize(pbmih);
                                    ASSERT(pProperties->cbBuffer);

            //pProperties->cbAlign = 16;          // SSE2 alignment requirement ESTE ES EL QUE HE DESCOMENTADO

            // Ask the allocator to reserve us some sample memory, NOTE the function
            // can succeed (that is return NOERROR) but still not have allocated the
            // memory that we requested, so we must check we got whatever we wanted
                                    ALLOCATOR_PROPERTIES Actual;
                                    hr = pAlloc->SetProperties(pProperties, &Actual);
                                    if (SUCCEEDED(hr))
                                    {
                                                ASSERT(Actual.cBuffers >= 1);
                                                if ((pProperties->cBuffers > Actual.cBuffers) || (pProperties->cbBuffer >
Actual.cbBuffer))
                                                {
                                                            hr = E_FAIL;
                                                }
                                    }
                        }
                        else
                        {
                                    hr = E_POINTER;
                        }
            }
            else
    {
       hr = E_UNEXPECTED;
    }
            return hr;

}

//------------------------------------------------------------------------
// GetMediaType
//
```

```
HRESULT CDecklinkComposite::GetMediaType(int iPosition, CMediaType* pMediaType)
{
	CheckPointer(pMediaType, E_POINTER);
	HRESULT hr = S_OK;

	// Is the input pin connected
	if (m_pInput->IsConnected())
	{
		// This should never happen
		if (iPosition >= 0)
		{
			// Do we have more items to offer
			if (iPosition == 0)
			{
				hr = pMediaType->Set(m_mtOutput);
			}
			else
			{
				hr  = VFW_S_NO_MORE_ITEMS;
			}
		}
		else
		{
			hr = E_INVALIDARG;
		}
	}
	else
	{
		hr = E_UNEXPECTED;
	}

	return hr;
}

//------------------------------------------------------------------------
// SetMediaType
//
HRESULT CDecklinkComposite::SetMediaType(PIN_DIRECTION direction, const CMediaType* pmt)
{
	HRESULT hr = S_OK;

	if (pmt)
	{
		if (PINDIR_INPUT == direction)
```

```
{
        // once the input pin has been connected, set the default
        // output media type to match the input media type

        m_mediaType2=*pmt;

                if (FORMAT_VideoInfo == m_mediaType2.formattype)
                {
                        VIDEOINFOHEADER* pv =
(VIDEOINFOHEADER*)m_mediaType2.pbFormat;
                        m_mediaType2.SetSampleSize(829440);
                        m_pbm = &pv->bmiHeader;
                        m_pbm->biHeight = 576;
                        m_pbm->biWidth = 720;
                        m_pbm->biSizeImage = 829440;
                        pv->rcSource.right = 720;
                        pv->rcSource.bottom = 576;
                        pv->rcSource.left = 0;
                        pv->rcSource.top = 0;
                        pv->rcTarget.right = 0;
                        pv->rcTarget.bottom = 0;
                        pv->rcTarget.left = 0;
                        pv->rcTarget.right = 0;
                }
                else if (FORMAT_VideoInfo2 == m_mediaType2.formattype)
                {
                        VIDEOINFOHEADER2* pv =
(VIDEOINFOHEADER2*)m_mediaType2.pbFormat;
                        m_pbm = &pv->bmiHeader;
                        m_pbm->biHeight = 576;
                        m_pbm->biWidth = 720;
                        m_pbm->biSizeImage = 829440;
                        pv->rcSource.right = 0;
                        pv->rcSource.bottom = 0;
                        pv->rcSource.left = 0;
                        pv->rcSource.top = 0;
                        pv->rcTarget.right = 0;
                        pv->rcTarget.bottom = 0;
                        pv->rcTarget.left = 0;
                        pv->rcTarget.right = 0;
                }

        //      m_mtOutput.Set(*pmt);
                m_mtOutput = m_mediaType2;
```

```
                }
        }
        else
        {
                hr = E_POINTER;
        }

        return hr;
}

//------------------------------------------------------------------------
// ISpecifyPropertyPages interface
//------------------------------------------------------------------------
//------------------------------------------------------------------------
// GetPages
//
STDMETHODIMP CDecklinkComposite::GetPages(CAUUID* pPages)
{
        if (pPages == NULL)
                return E_POINTER;

        pPages->cElems = 1;
        pPages->pElems = (GUID*)CoTaskMemAlloc(sizeof(GUID));

        if (pPages->pElems == NULL)
                return E_OUTOFMEMORY;

        pPages->pElems[0] = CLSID_DecklinkCompositeProperties;
        return S_OK;
}

// ------------------------------------------------------------------------
// IAMStreamConfig interface
// ------------------------------------------------------------------------
// GetFormat
// If connected, return the media type of the connection, otherwise return the
// 'preferred' format.  As DS states that the formats should be listed in descending
// order of quality, our preferred format is the best one which is the first item
// in the list.
//
STDMETHODIMP CDecklinkComposite::GetFormat(AM_MEDIA_TYPE** ppamt)
{
```

```
	HRESULT hr = S_OK;
// --------------------------------------------------------------------
//DESDE AQUI SE PUEDE COMENTAR Y NO HAY NINGÚN PROBLEMA CON EL FUNCIONAMIENTO DEL FILTRO
	CheckPointer(ppamt, E_POINTER);

	if (m_pInput->IsConnected())
	{
		*ppamt = (AM_MEDIA_TYPE*)CoTaskMemAlloc(sizeof(AM_MEDIA_TYPE));
		if (*ppamt)
		{
			AM_MEDIA_TYPE *pamt = CreateMediaType((const AM_MEDIA_TYPE*)&m_mtOutput);
			*ppamt = pamt;
		}
		else
		{
			hr = E_OUTOFMEMORY;
		}
	}
	else
	{
		hr = VFW_E_NOT_CONNECTED;
	}
//HASTA AQUI
// --------------------------------------------------------------------
	return hr;
}



//------------------------------------------------------------
// SetFormat
//
STDMETHODIMP CDecklinkComposite::SetFormat(AM_MEDIA_TYPE* pamt)
{
	HRESULT hr = S_OK;
	CheckPointer(pamt, E_POINTER);

	if (m_pInput->IsConnected())
	{
		if (IsStopped())
		{
			// TODO: Check that this is an acceptable media type
```

```
//                      m_mtOutput.Set(*pamt);
                }
                else
                {
                        hr = VFW_E_NOT_STOPPED;
                }
        }
        else
        {
                hr = VFW_E_NOT_CONNECTED;
        }

        return hr;
}

//------------------------------------------------------------------
// GetNumberOfCapabilities
// Returns the number of video formats supported by the underlying
// HW object.
//
STDMETHODIMP CDecklinkComposite::GetNumberOfCapabilities(int* piCount, int* piSize)
{
        CheckPointer(piCount, E_POINTER);
        CheckPointer(piSize, E_POINTER);
        HRESULT hr = S_OK;

        if (m_pInput->IsConnected())
        {
//                      *piCount = 1;
//                      *piSize = sizeof(VIDEO_STREAM_CONFIG_CAPS);
        }
        else
        {
                hr = VFW_E_NOT_CONNECTED;
        }

        return hr;
}

//------------------------------------------------------------------
// GetStreamCaps
// Returns the media format at the specified index.
//
STDMETHODIMP CDecklinkComposite::GetStreamCaps(int iIndex, AM_MEDIA_TYPE** ppamt, BYTE* pSCC)
```

```
{
	HRESULT hr = S_OK;
	CheckPointer(ppamt, E_POINTER);
	CheckPointer(pSCC, E_POINTER);
// ------------------------------------------------------------------------
//DESDE AQUI SE PUEDE COMENTAR Y NO HAY NINGÚN PROBLEMA CON EL FUNCIONAMIENTO DEL FILTRO

	if (m_pInput->IsConnected())
	{
		if (iIndex < 1)
		{
			// Allocate a block of memory for the media type including a separate allocation
			// for the format block at the end of the structure
			*ppamt = (AM_MEDIA_TYPE*)CoTaskMemAlloc(sizeof(AM_MEDIA_TYPE));
			if (*ppamt == NULL)
			{
				return E_OUTOFMEMORY;
			}

			ZeroMemory(*ppamt, sizeof(AM_MEDIA_TYPE));

			AM_MEDIA_TYPE *pamt = CreateMediaType((const AM_MEDIA_TYPE*)&m_mtOutput);
			*ppamt = pamt;

			if (pamt)
			{
				VIDEO_STREAM_CONFIG_CAPS* pscc = (VIDEO_STREAM_CONFIG_CAPS
*)pSCC;
				VIDEOINFOHEADER* pvih = (VIDEOINFOHEADER *)pamt->pbFormat;

				ZeroMemory(pSCC, sizeof(VIDEO_STREAM_CONFIG_CAPS));
				pscc->guid = pamt->formattype;
				pscc->InputSize.cx = pvih->bmiHeader.biWidth;
				pscc->InputSize.cy = pvih->bmiHeader.biHeight;
				pscc->MinCroppingSize = pscc->MaxCroppingSize = pscc->InputSize;
				pscc->MinOutputSize = pscc->MaxOutputSize = pscc->InputSize;
				pscc->MinBitsPerSecond = pscc->MaxBitsPerSecond = pvih->dwBitRate;
			pscc->MinFrameInterval = pscc->MaxFrameInterval = pvih->AvgTimePerFrame;
			}
			else
			{
				hr = E_OUTOFMEMORY;
			}
		}
		else
```

```
                {
                        hr = S_FALSE;
                }
        }
        else
        {
                hr = VFW_E_NOT_CONNECTED;
        }

        if (FAILED(hr) && (*ppamt))
        {
                CoTaskMemFree(*ppamt);
                *ppamt = NULL;
        }
//HASTA AQUI
// --------------------------------------------------------------------
        return hr;
}
```

Como se ve a simple vista, este filtro consta de las siguientes secciones:

- La sección “Create Instance” crea el Objeto COM, es decir, el filtro.
- Las secciones “Constructor” y “Destructor” realizan la construcción y destrucción de dicho filtro, respectivamente. Se construirá en el momento de su ejecución, y se destruirá en el momento de la finalización de la ejecución.
- La sección “Non Delegating Query Interface” se utiliza, como su propio nombre indica, para que no se delegue a otro filtro la consulta de la página de propiedades del filtro.
- La sección “Transform” es en la que se realizan las tareas necesarias para realizar el reescalado deseado.
- La sección “Check Input Type” realiza la comprobación del formato del sample de entrada, para ver si es o no admisible por el filtro.
- La sección “Check Transform” comprueba si el tipo de vídeo que tenemos a la entrada acepta o no la transformación que se realiza en la sección “Transform”.
- La sección “Decide Buffer Size” comprueba si el tamaño del buffer de memoria que se va a emplear para realizar la transformación es o no adecuado.
- La sección “Get Media Type” obtiene el tipo de vídeo que hay a la entrada del filtro una vez realizada la conexión de su pin de entrada de vídeo.

- La sección “Set Media Type” se utiliza para indicar las características del tipo de vídeo que habrá en el stream de vídeo que el filtro ofrezca a su salida, dependiendo de si se utiliza Vídeo info header o Vídeo info header 2.
- La sección “ISpecify Property Pages” se utiliza para definir la página de propiedades del filtro, a la que podremos acceder desde GraphEdit.
- La sección “IAMStream Config” nos sirve para devolver el tipo de formato del vídeo que se realiza con el filtro una vez tengamos la entrada del mismo conectada a un stream. Si no está conectado, nos devuelve cual es el formato de vídeo ideal para el filtro.
- La sección “Set Format” nos va permitir especificar el formato de vídeo con el que vamos a trabajar cuando el filtro esté conectado.
- La sección “Get Number of Capabilities” nos va a indicar el número de formatos de vídeo soportados por la tarjeta que estemos utilizando.
- La sección “Get Stream Caps” nos devuelve los formatos de vídeo soportados de forma ordenada.

De todas estas secciones, las más importantes a la hora de la realización del reescalado de la imagen de entrada son la sección “Transform” y la sección “Set Media Type”, ya que con ellas indicaremos cuales son las operaciones a realizar con la ristra de bits de entrada y cuál es el formato de salida que vamos a obtener.

Comenzaré explicando la sección Set Media Type, ya que en ella es donde vamos a indicar cuál es el formato del vídeo que vamos a obtener a la salida del filtro.

Como se puede ver en dicha sección, lo primero que realizamos es comprobar si utilizamos VIDEOINFOHEADER o VIDEOINFOHEADER2, y definimos las características del vídeo de salida para ambos casos, ya que así vamos a poder trabajar con los dos tipos de formatos sin ningún problema.

La diferencia entre VIDEOINFOHEADER o VIDEOINFOHEADER2 es simplemente la cantidad de información que nos suministra a cerca del mapa de color y los bits que utiliza el vídeo que estemos tratando en ese momento. Ambos simplemente son la estructura de los vídeos tratados. A continuación vuelvo a mostrar el contenido para ambos casos:

```
VIDEOINFOHEADER* pv = (VIDEOINFOHEADER*)m_mediaType2.pbFormat;
VIDEOINFOHEADER2* pv = (VIDEOINFOHEADER2*)m_mediaType2.pbFormat;
```

Este comando va a variar, dependiendo de si utilizamos VIDEODEOINFOHEADER o VIDEOINFOHEADER2, y básicamente se copia el formato de vídeo a una variable,pv, en la que almacenaremos el nuevo formato de vídeo que vamos a utilizar.

```
m_mediaType2.SetSampleSize(829440);
```

Con el comando anterior indico que el tamaño total de la imagen será de 829440 bytes (2 bytes x 720 x 576).

```
m_pbm = &pv->bmiHeader;
m_pbm->biHeight = 576;
m_pbm->biWidth = 720;
m_pbm->biSizeImage = 829440;
```

Con estos comandos indico el alto y ancho de la imagen, así como el tamaño total, en el formato de vídeo de salida del filtro. Cabe destacar que se utilizan 2 bytes para la definición de color y de ahí su utilización en la fórmula.

De esta forma consigo indicar cuál es el formato de salida del vídeo que deseo, y como lo indico de forma fija, este siempre será 720x576, salvando de esta manera el problema con los vídeos cuya resolución era 704 x 576.

Una curiosidad de esta versión del SDK que me he encontrado desarrollando estos filtros es que si se realiza la anterior multiplicación utilizando las variables locales biHeight, biWidth no se calculaba correctamente el tamaño de la imagen. Es decir, si se emplea el siguiente comando

```
m_pbm->biSizeImage = 2 x m_pbm->biHeight x m_pbm->biWidth;
```

el resultado de la multiplicación no es 829440. Según he podido indagar en la lista de desarrolladores de Blackmagic, este problema se ha solucionado en la versión 7.3 del SDK.

Ahora bien, aquí simplemente indico cual es el formato que quiero obtener a la salida del filtro, pero no he realizado ningún cambio en el stream de vídeo que tengo a la entrada del filtro, por lo que si dejara así el filtro, pese que el tamaño de salida sería el correcto, el tamaño de la imagen con información seguiría siendo de 704x576 y el resto se rellenaría de negro, ya que no hay información para la parte derecha de la imagen (16 píxels) de cada una de las líneas. Así pues, el resultado sería la imagen de 704x576 y a la derecha de ésta una columna negra.

Para solucionar este problema utilizo la sección “Transform” para centrar el vídeo en pantalla y conseguir así una mayor calidad de imagen.

Lo primero que realizo en esta sección es crear unos punteros que apunten a las posiciones de memoria del vídeo origen y destino en memoria. Así como comprobar que existe un vídeo en la entrada del filtro.

```
CheckPointer(pIn, E_POINTER);
CheckPointer(pOut, E_POINTER);
HRESULT hr = S_OK;

BYTE* pSrc, *pDst;
long cbSrc = pIn->GetActualDataLength();
long cbDst = pOut->GetActualDataLength();

pIn->GetPointer(&pSrc);
pOut->GetPointer(&pDst);
```

A continuación obtengo el tipo de formato de vídeo con el que voy a trabajar mediante los comandos

```
BITMAPINFOHEADER* pbmihSrc = CUtils::GetBMIHeader(m_pInput->CurrentMediaType());
BITMAPINFOHEADER* pbmihDst = CUtils::GetBMIHeader(m_pOutput->CurrentMediaType());
```

Una vez obtenidos estos datos completamente necesarios, almaceno cuál es el ancho de la imagen tanto en el vídeo de origen como en el de destino

```
LONG srcHeight = (pbmihSrc->biHeight);
LONG dstHeight = (pbmihDst->biHeight);

unsigned long srcRowBytes = cbSrc / srcHeight;
unsigned long dstRowBytes = cbDst / dstHeight;
```

Además, me calculo cual es el tamaño en bits de un pixel de la imagen e inicializo dos contadores para las líneas del vídeo origen y destino.

```
unsigned long cbPixel = CUtils::GetImageSize(pbmihSrc) /
pbmihSrc->biWidth / pbmihSrc->biHeight;

unsigned long cLines = 0, cbLine = 0;
```

A continuación se tratan las cuatro posibles opciones que se pueden dar:

- La primera es que el ancho de la imagen de vídeo origen sea menor que la de destino, caso que me interesa especialmente, ya que es precisamente para este caso para el que se desarrolla este filtro.

```
if (pbmihSrc->biWidth < pbmihDst->biWidth)
```

  Vemos con este IF que se compara el ancho de la imagen fuente (pbmihSrc->biWidth=704) con el de la imagen que se obtendrá representada en pantalla (pbmihDst->biWidth)

  Mediante un *else* contemplo el caso contrario, es decir, cuando tengan el mismo ancho o que el ancho de la imagen destino sea menor que el de la imagen origen.

- Otro caso que se puede dar, en cuanto a la altura de la imagen de vídeo, es que la imagen origen sea más pequeña que la imagen destino.

```
if (srcHeight < dstHeight)
```

  Así vemos que SrcHeight (el de la imagen origen) sea menor que dstHeight (el de la imagen final)

  Como en el caso anterior, mediante un *else* contemplo el caso contrario, es decir, cuando el alto de la imagen destino es menor o igual a la imagen origen.

El caso que comprende este proyecto final de carrera es el primero, es decir, el caso en que la imagen origen tiene un ancho menor que la imagen destino, (pbmihSrc->biWidth < pbmihDst->biWidth).

Como se puede ver en el código del filtro arriba indicado, esta sección realiza las siguientes tareas.

1. Sitúo el puntero del vídeo de destino (pDst) un número de píxels determinado a la derecha del vídeo origen mediante el comando:

   pDst += (((pbmihDst->biWidth - pbmihSrc->biWidth)/2 * cbPixel) >> 1);

   donde pbmihDst->biWidth = 720, pbmihSrc->biWidth = 704 y cbPixel = el número de bits que tenemos en un píxel de la imagen destino, como se explica a continuación.

   Para calcular cbPixel, como se ve en el código, hago el siguiente cálculo:

   cbPixel = 704 x 576 / 720 / 576 = 0.977777

   Es decir, me calculo el tamaño en bits de un píxel de la imagen destino referido a un píxel de la imagen origen, para poder pintar la imagen destino en pantalla de una forma acorde a la imagen origen que esté tratando en cada momento.

   Con ello se ve que 1 píxel de la imagen origen se corresponde con 0.977777 píxels de la imagen destino. Con ello se indica que simplemente el número de bits A que forman un píxel de la imagen origen, sólo formarían 0.977777 píxels de la imagen destino. Como el tamaño de la imagen es mayor, se ha de realizar ésta correspondencia a nivel de bit (o byte) para que la imagen origen y destino concuerden.

   De esta forma consigo una relación en cuanto al tamaño de los píxeles de la imagen origen y destino que será constante, sea cual sea el tamaño de la imagen origen.

   Es decir, he de mover el primer píxel de la imagen origen, 7.82 píxeles hacia la derecha para obtener una imagen centrada en pantalla en relación al nuevo tamaño de la imagen, como se ve en el código siguiente

   pDst = ((720-704)/2)*0.9777= 7,822 bytes

2. Indico también cual es el tamaño en bits de una línea del vídeo destino, empleando para ello el tamaño en bits de un píxel y el ancho de la imagen destino.

   cbLine = cbPixel * pbmihSrc->biWidth;

   donde cbLine = número de píxels que tenemos en una línea de la imagen destino.

Así se consigue centrar cada una de las líneas de la imagen.

Finalmente utilizo un bucle FOR para recorrer todas las líneas y conseguir que la transformación descrita anteriormente se realice para toda la imagen.

```
for (unsigned long line=0; line<cLines; ++line)
        {
            memcpy(pDst, pSrc, cbLine);
            pSrc += srcRowBytes;
            pDst += dstRowBytes;

        }
```

Como se puede observar, se copia el contenido de una línea, de tamaño igual a la variable cbLine (tamaño en bits de una línea de la imagen), desde la posición de memoria que contiene el vídeo origen (pSrc) a la del vídeo destino (pDst, ya modificado). A continuación se incrementa en 1 el número de línea que se está tratando hasta que se llega a la última línea de la imagen.

Con ello se termina la transformación de la imagen.

6.2. FILTRO DECKLINK PalFieldSwap

Mediante este filtro pretendo conseguir detectar los vídeos que tengan los campos PAL invertidos, es decir, que tengan marcado el primer campo como “lower”, y representarlos en pantalla con los campos en el orden correcto. Para ello me baso en el filtro del mismo nombre que viene incluido en el Software Developement Kit (SDK) de Blackmagic, cuyo contenido es el siguiente.

```
//------------------------------------------------------------------------------
// DecklinkFieldSwap.cpp
//
// Desc: DirectShow sample code - Illustrates a very basic field swap filter implementation.
// Based entirely upon the NullNull filter sample in the
// DirectShow SDK samples.
//------------------------------------------------------------------------------
```

```
#include "stdafx.h"

#include "DecklinkFilters_h.h"
#include "DecklinkFieldSwap.h"


//------------------------------------------------------------------------
// CreateInstance
// Provide the way for COM to create a CDecklinkFieldSwap object




CUnknown* WINAPI CDecklinkFieldSwap::CreateInstance(LPUNKNOWN punk, HRESULT* phr)
{
    CheckPointer(phr, NULL);

    CDecklinkFieldSwap* pNewObject = new CDecklinkFieldSwap(NAME("Decklink field swap filter"), punk,
phr);

    if (NULL == pNewObject)
    {
        *phr = E_OUTOFMEMORY;
    }

    return pNewObject;
}

//------------------------------------------------------------------------
// Transform
// This is where the field swap actually takes place.  Its a very simple trans-in-place operation,
// if we've flagged field swapping, move the buffer contents down by one line.  We lose one active
// line in this process and have to black out the first line.  The MoveMemory operation is fast but
// still not ideal if there is a lot of other processing in the filter graph.


HRESULT CDecklinkFieldSwap::Transform(IMediaSample* pSample)
{
        HRESULT hr = S_OK;
```

```
        if (pSample)
        {
                AM_MEDIA_TYPE amt = {0};
                if (SUCCEEDED(m_pInput->ConnectionMediaType(&amt)))
                {
                        BITMAPINFOHEADER* pbmih = CUtils::GetBMIHeader(&amt);

                        if (576 == pbmih->biHeight)
                        {
                                // field swap the buffer contents
                                long cbData = pSample->GetActualDataLength();
                                BYTE* pFrame = NULL;
                                hr = pSample->GetPointer(&pFrame);
                                if (SUCCEEDED(hr))
                                {
                                        long rowBytes = cbData / pbmih->biHeight;


// move the buffer contents down by one line to achieve field swap
                        MoveMemory(pFrame + rowBytes, pFrame, cbData - rowBytes);


// black out the first line
                        unsigned long Black[2];

// TODO: Add more complete range of FCCs, e.g. Apple equivalents
                        if ((FOURCC('UYVY') == pbmih->biCompression) || (FOURCC('HDYC') == pbmih-
>biCompression))
                                        {
                                                // 8-bit YUV black
                                                Black[0] = Black[1] = 0x10801080;
                                        }
                                        else if (FOURCC('YUY2') == pbmih->biCompression)
                                        {
                                                // 8-bit YUV black
                                                Black[0] = Black[1] = 0x80108010;
                                        }
                                        else if (FOURCC('v210') == pbmih->biCompression)
                                        {
                                                // 10-bit YUV
                                                Black[0] = 0x20010200;
                                                Black[1] = 0x04080040;
```

```
                                        }
                                        else if (FOURCC('r210') == pbmih->biCompression)
                                        {
                                                // 10-bit RGB
                                                Black[0] = Black[1] = 0x40000104;
                                        }
                                        else
                                        {
                                                // just assume BI_RGB for all others
                                                Black[0] = Black[1] = 0x00000000;
                                        }



                        unsigned long* pData = reinterpret_cast<unsigned long*>(pFrame);
                                        int iCount = rowBytes / sizeof(Black);
                                        for (int i=0; i<iCount; ++i, ++pData)
                                        {
                                                CopyMemory(pData, &Black, sizeof(Black));
                                        }
                                }
                        }
                        else
                        {
                                // do nothing
                        }

                        FreeMediaType(amt);
                }
                else
                {
                        // not connected
                }
        }
        else
        {
                hr = E_POINTER;
        }

        return hr;
}
```

```
//-------------------------------------------------------------------------
// CheckInputType
// Some basic checks on preferred formats.  In this sample enable field swapping for PAL sized frames.

HRESULT CDecklinkFieldSwap::CheckInputType(const CMediaType* mtIn)
{
        // Make some very rudimentary checks on the media type
        // This would be improved by examining the VIDEOINFOHEADER2 interlace flags
        // and processing accordingly.

        if (MEDIATYPE_Video != mtIn->majortype)
                return S_FALSE;

        if (FORMAT_VideoInfo != mtIn->formattype)
                return S_FALSE;

        if ((MEDIASUBTYPE_UYVY != mtIn->subtype) && (IID_MEDIASUBTYPE_HDYC != mtIn->subtype)
&& (MEDIASUBTYPE_YUY2 != mtIn->subtype) &&(IID_MEDIASUBTYPE_v210a != mtIn->subtype) &&
(IID_MEDIASUBTYPE_r210 != mtIn->subtype) &&(MEDIASUBTYPE_RGB24 != mtIn->subtype) &&
(MEDIASUBTYPE_RGB32 != mtIn->subtype) && (MEDIASUBTYPE_ARGB32 != mtIn->subtype))
                return S_FALSE;

        return S_OK;
}
```

Como se puede ver, el filtro consta de 3 partes claramente diferenciadas:

- La primera de ellas es “Create Instance”, en la que como su propio nombre indica, se crea la instancia del filtro, es decir, se crea el propio filtro.
- En la segunda parte “Transform”, es donde se realizan las transformaciones, como su propio nombre indica, al stream de vídeo de entrada. Es precisamente en esta sección donde realizaré los cambios adecuados para adecuarlo al sistema que a mi me interesa.
- Finalmente, vemos una tercera parte “Check Input Type” en la que se definen los tipos de stream de vídeo que serán aceptados por este filtro. Vemos en el código cómo se comprueba el tipo de media que tiene el vídeo, y si el tipo de media no concuerda con el *majortype* del proyecto, en el cual el VIDEOINFOHEADER2 tiene el flag de campo etiquetado como upper, se activa la ejecución del filtro ya que el vídeo estará etiquetado como lower.

Así pues, la parte más importante del filtro es la sección “Transform”. Como se ve en su contenido, lo primero que se hace es comprobar si existe un sample de vídeo a la entrada, si el tipo de vídeo es aceptado por el filtro, y si la resolución del mismo corresponde con 576.

```
if (pSample)
if (SUCCEEDED(m_pInput->ConnectionMediaType(&amt)))
if (576 == pbmih->biHeight)
```

Una vez concuerdan todas estas condiciones, se comprueba el tamaño del sample de vídeo y se crea un puntero a una posición de memoria con dicho tamaño.

Si alguna de estas condiciones, no se cumple, el filtro no tiene que ser aplicado y se saldrá de su ejecución.

Si, por otro lado, sí que se cumplen dichas condiciones y se ha reservado correctamente el espacio en memoria, se pasa a realizar las tareas de transformación dependiendo del sistema de color utilizado.

La primera tarea es conocer cuál es el tamaño de una línea en bytes mediante la orden

```
long rowBytes = cbData / pbmih->biHeight;
```

Así conseguimos almacenar en la variable rowBytes el valor correspondiente a dicho tamaño.

A continuación se realiza un movimiento en el buffer de memoria una línea hacia abajo, mediante la orden

```
MoveMemory(pFrame + rowBytes, pFrame, cbData - rowBytes);
```

A continuación se crea una variable que contiene una línea en negro para introducirla en el espacio dejado en memoria, debido al movimiento anterior. A partir de este momento, se chequea cual es el mapa de color utilizado, para realizar la inserción de una línea negra de una forma correcta. Así pues, comprobamos si utilizamos los mapas de color 8-bits YUV, YUY2, 10-bits YUV, 10-bits RGB, y uno genérico para cualquier otro mapa de color, BI_RGB.

A continuación, se realiza un bucle for para ir copiando el contenido de las líneas del sample de vídeo y dejar definitivamente bien dibujada la imagen en pantalla.

```
unsigned long* pData = reinterpret_cast<unsigned long*>(pFrame);
        int iCount = rowBytes / sizeof(Black);
        for (int i=0; i<iCount; ++i, ++pData)
        {
                CopyMemory(pData, &Black, sizeof(Black));
        }
```

Como se ve, simplemente se van copiando líneas de la imagen mediante el contador iCount.

Mediante este sencillo filtro se realiza la transformación de la posición de los campos, añadiendo una línea más en negro al comienzo del sample de vídeo o no. Cabe destacar que no estamos perdiendo información ya que simplemente estamos añadiendo, o no, una línea negra al comienzo del sample. Así mismo, esta línea es inapreciable a la hora de realizar la reproducción del vídeo en pantalla, ya que queda fuera del marco de la imagen, pero con ello conseguimos dibujar los campos PAL de la forma correcta (upper).

7. CONCLUSIONES Y APLICACIONES FUTURAS.

Con todo lo descrito anteriormente se consigue el objetivo deseado del presente trabajo final de carrera, así como solucionar los problemas de emisión que se venían dando hasta el momento en la UPV TV.

La utilización de los filtros DirectShow para la emisión de vídeo en la UPV TV representa una gran ventaja, ya que permiten optimizar al máximo la utilización del hardware del que se dispone, así como el empleo de una gran cantidad de filtros ya realizados por terceras personas que pueden solucionar posibles problemas que pudieran aparecer en un futuro.

La versatilidad en cuanto a programación de los filtros DirectShow, va a permitir a la UPV TV desarrollar filtros de vídeo para realizar cualquier tipo de transformación de imagen en tiempo real, sin necesidad de utilizar hardware de terceros, con el consecuente ahorro económico que esto supone.

Para el correcto funcionamiento de estos filtros se han tenido que superar una serie de dificultades, algunas de ellas las enumero a continuación:

- Comprender el funcionamiento de VisualStudio 2005 y la forma de organizar un proyecto.
- Familiarizarse con las posibilidades de configuración de ZoomPlayer y su aplicación SmartPlay, para exprimir sus posibilidades al máximo.
- Comprender el funcionamiento de los filtros DirectShow y la forma de trabajar con ellos.
- Elegir un lenguaje de programación adecuado para el desarrollo de los filtros DirectShow, ya que no sólo se puede utilizar C++ para crear aplicaciones basadas en dichos filtros. En este caso, es necesario usar C++ o C# para que la interacción del código con la tarjeta sea total.
- Familiarizarse con la programación en C++ con Visual Studio.
- Errores de compilación con Visual Studio por la mala organización de las librerías a utilizar en el proyecto. He tenido bastantes problemas de dependencias en el proyecto ya que el desarrollador que utiliza el SDK de Blackmagic no tiene información de cómo insertar el proyecto proporcionado por la compañía para su correcta compilación. Además, siempre que se elimina información del proyecto genérico, aparece una gran cantidad de problemas que se han de ir depurando uno por uno, para una correcta compilación.

Los filtros que DirectShow que he desarrollado se pueden aplicar en futuras necesidades de la UPV TV como puede ser la implantación el centrado de imágenes de gran tamaño en una resolución de salida de 720 x 576, realizando un recorte de las mismas en tiempo real.

También se pueden utilizar para el cambio de resolución de una imagen de menor tamaño a calidad broadcast, teniendo en cuenta que simplemente se realizará un centrado en pantalla de la imagen origen.

BIBLIOGRAFÍA

MONOGRÁFICOS

- DIRECTSHOW FOR DIGITAL VIDEO AND TELEVISION. Mark D. Pesce. (Microsoft Press)
- APRENDA C++ COMO SI ESTUVIERA EN PRIMERO. Javier García de Jalón, José Ignacio Rodríguez, José María Sarriegui, Ruffino Goñi, Alfonso Brazález, Patxi Funes, Alberto Larzabal y Rubén Rodríguez. (Escuela Superior de Ingenieros Industriales de la Universidad de Navarra)
- MANUAL DE ZOOMPLAYER (Imatrix)
- TUTORIAL DE C++. Peter Class.
- PROFESSIONAL VISUAL STUDIO 2005. Andrew Parsons,Nick Randolph
- COMPRESSION FOR GREAT DIGITAL VIDEO: POWER TIPS, TECHNIQUES, AND COMMON SENSE. Ben Waggoner
- FUNDAMENTALS OF AUDIO AND VIDEO PROGRAMMING FOR GAMES. Peter Turcan y Mike Wasson.
- PROGRAMMING MICROSOFT DIRECTSHOW. Michael Linetsky
- DIRECTX 9 USER INTERFACE: DESING AND IMPLEMENTATION. Alan Thorn

PÁGINAS WEB

- http://www.gdcl.co.uk/dshow.htm
- http://msdn.microsoft.com
- http://msdn.microsoft.com/en-us/library/dd391015(VS.85).aspx
- http://www.leadtools.com/sdk/multimedia/directshow.htm
- http://social.msdn.microsoft.com/Forums/es-ES/windowsdirectshowdevelopment/threads
- http://www.cvc.uab.es/shared/teach/a20383/practiques/practica3/filtros.htm
- http://www.inmatrix.com/

ANEXO 1: EL PROBLEMA DE LA DOMINANCIA DE CAMPOS EN EL VÍDEO ENTRELAZADO.

La dominancia se refiere a la manera en que el sistema con el que estamos trabajando ordena los campos o líneas cuando los reproduce.

Los fotogramas entrelazados están compuestos por dos bloques de líneas pero el problema es conocer cual es la primera línea que reproducirá nuestro sistema, cual es la primera línea que ha capturado en primer lugar la cámara...

La dominancia varía según el sistema de vídeo y el hardware que usemos. Además la nomenclatura para definir la dominancia tampoco está estandarizada y se pueden escuchar términos como: Odd/Even (Par o impar); Lower/Upper (inferior o superior) ; F1/F2 (campo 1 o campo 2)....

El problema es que si nuestro proyecto está configurado con una dominancia diferente a la del material capturado, percibiremos un efecto de parpadeo al reproducir en un monitor de vídeo, que no en una televisión convencional.

Para solucionar el problema podemos reconfigurar el proyecto con la dominancia correcta o aplicar algún filtro al material de vídeo para que invierta la dominancia. Casi todos los programas de edición de vídeo tienen una opción denominada “shift fields”, “reverse fields”, o similares, para poder ejecutar ésta tarea.

La dominancia también es muy importante a la hora de elaborar imágenes 3D ya que debemos configurar el render correctamente para que coincida con la dominancia de nuestro proyecto.

No existe una dominancia propia del sistema PAL o NTSC, sino que las dos dominancias son posibles en ambos sistemas. A continuación muestro una serie de dominancias dependiendo del material (códec) con el que estamos trabajando:

D-1 PAL -Discreet edit* PAL
Frame size: 720 x 576
Frame aspect ratio: 4:3
Pixel aspect ratio: 1.067
Frame rate: 25 fps
Field order: upper field first

D-1 PAL -Discreet edit* PAL
Frame size: 720 x 576
Frame aspect ratio: 4:3
Pixel aspect ratio: 1.067
Frame rate: 25 fps
Field order: upper field first

PAL Video Frame size: 768 x 576
Frame aspect ratio: 4:3
Pixel aspect ratio: 1
Frame rate: 25 fps
Field order: lower field first

PAL-DV Frame size: 720 x 576
Frame aspect ratio: 5:4
Pixel aspect ratio: 1.067
Frame rate: 25 fps
Field order: lower field first

PAL-DV Widescreen
Frame size: 720 x 576
Frame aspect ratio: 16:9
Pixel aspect ratio: 1.422
Frame rate: 25 fps
Field order: upper field first

NTSC-DV Widescreen
Frame size: 720 x 486
Frame asp. ratio: 16:9
Pixel aspect ratio: 1.2
Frame rate: 29.97 fps
Field order: lower first

NTSC-DV Frame size: 720 x 480
Frame aspect ratio: 3:2
Pixel aspect ratio: 0.9
Frame rate: 29.97 fps
Field order: lower first

NTSC D-1 - Discreet edit* NTSC
Frame size: 720 x 486
Frame aspect ratio: 4:3
Pixel aspect ratio: 0.9
Frame rate: 29.97 fps
Field order: lower field first

NTSC Frame size: 640 x 480
Frame aspect ratio: 4:3
Pixel aspect ratio: 1
Frame rate: 29.97 fps
Field order: upper first

NTSC Full Frame size: 648 x 486
Frame aspect ratio: 4:3
Pixel aspect ratio: 1
Frame rate: 29.97 fps
Field order: upper first

HDTV 720/30p Frame size: 1280 x 720
Frame aspect ratio: 16:9
Pixel aspect ratio: 1
Frame rate: 30 fps
Field order: upper field first

HDTV 1080/24p Frame size: 1920 x 1080

Frame aspect ratio: 16:9

Pixel aspect ratio: 1

Frame rate:24 fps

Field order: upper field first

Tal y como podemos ver la dominancia del campo depende del códec con el que estamos trabajando en cada momento.

Si tomamos como ejemplo el funcionamiento en PAL, entenderemos cual es la importancia de conocer de la dominancia entre campos.

El objetivo de un sistema básico de televisión es captar una escena y reproducirla de la manera más parecida posible.

La información de imagen se capta explorando secuencialmente cada punto de la superficie de la cámara y midiendo las variaciones de tensión que produce la luz en cada punto. La exploración o barrido se realiza igual que se lee un libro: de izquierda a derecha, y de arriba abajo.

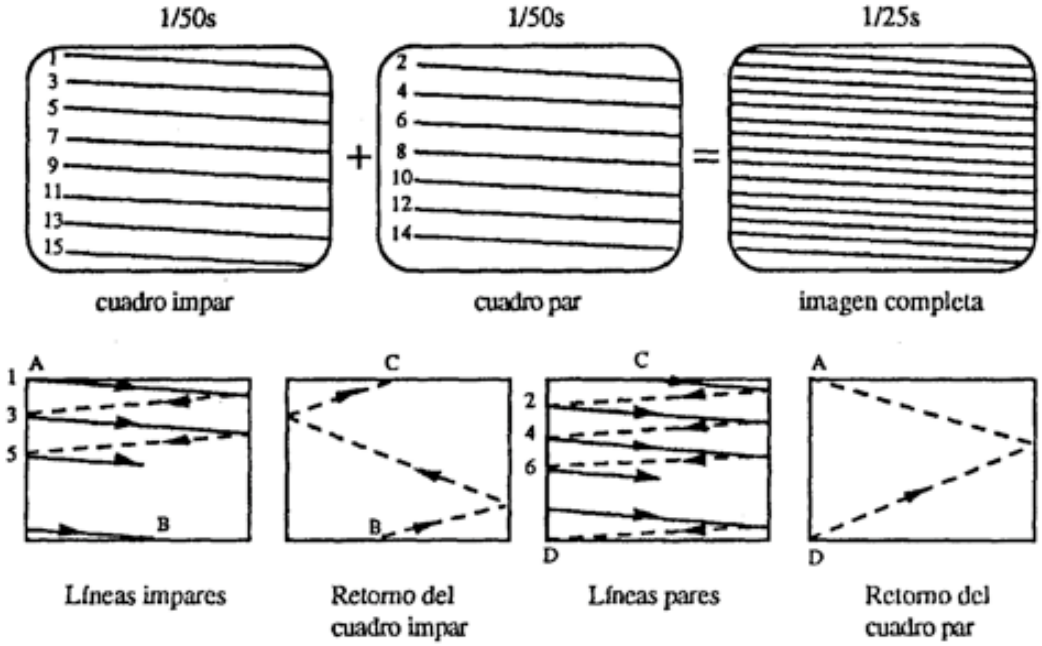

Figura A1.1. Líneas de la imagen PAL

El muestreo temporal de la imagen consiste en dividir cada imagen en dos subimágenes o campos explorando alternativamente las líneas pares y las impares. La proximidad entre líneas consecutivas hace que el espectador integre las dos subimágenes y obtenga la sensación de que éstas se están renovando a una frecuencia doble de la real. Haciendo desparecer la sensación de parpadeo.

Cuando la señal transmitida llega al receptor, éste ha de tener alguna manera de reconocer cuándo comienza una línea, y qué línea de campo es.

Habrá que añadir algunas señales para sincronizar la exploración de la imagen realizada por la cámara con el barrido en el receptor.

Se añadirán dos señales de sincronismo. Una de sincronismo horizontal o de línea al inicio de cada línea y otra de sincronismo vertical o de campo al final de cada campo.

Estas señales se pondrán en las zonas de la señal que no se utilizan para la información de imagen y que son usadas por el receptor efectuar los retornos a las posiciones iniciales de exploración.

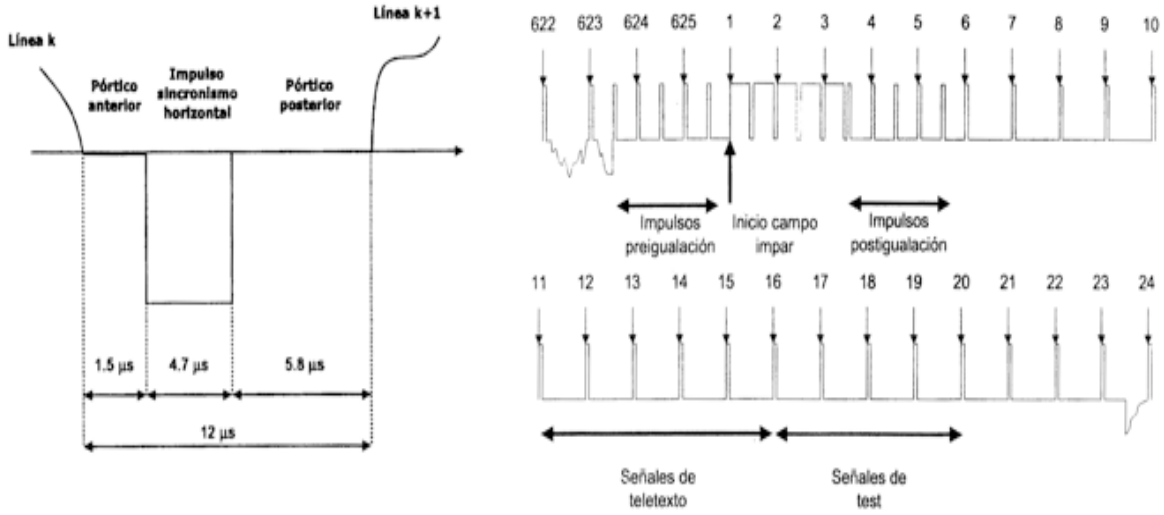

Figura A1.2. Señales de sincronismo en PAL

Una secuencia es todo aquello que se repite de una forma periódica. En el sistema PAL hay tres secuencias distintas: la secuencia de líneas, la secuencia de cuatro campos o secuencia BRUCH y la secuencia de ocho campos o secuencia PAL.

Secuencia de dos campos o secuencia de línea

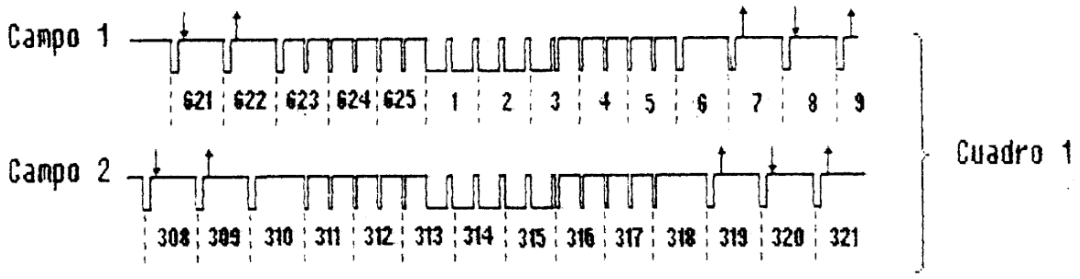

Figura A1.3. Secuencia de 2 campos

El campo 1 empieza en la línea 1 mientras que el campo 2 empieza en la línea 313, pero en la mitad de la línea, y termina en la 625. En el campo 1 el primer impulso de línea esta después de media línea, es decir entre el último impulso igualador y el primero de línea hay media línea (5).

En el campo 2, el primer impulso de línea está después de una línea, es decir entre el último impulso igualador y el primero de línea hay una línea completa (318).

Sí nos fijamos en los demás cuadros veremos que en esto son todos iguales.

Secuencia de cuatro campos o secuencia BRUCH

La secuencia de Bruch se forma con el borrado de burst. en cada campo se borra en nueve líneas el burst, representado por flechas sobre la línea, pero no se borra en las mismas líneas. en el campo 1 se borra desde la 623 a la 6 ambas inclusive, en el campo 2, desde la 310 a la 318, en el campo 3, desde la 622 a la 5, y en el campo 4, desde la 311 a la 319. Como podemos observar en ningún campo se borran en las mismas líneas.

Pero desde el campo 5 al 8 veremos que en esto si son iguales, es decir, en el campo 1 se borra el burst en las mismas líneas que en el campo 5, o sea, en cuanto a borrado de burst los campos 1 a 4 son idénticos a los campos 5 a 8.

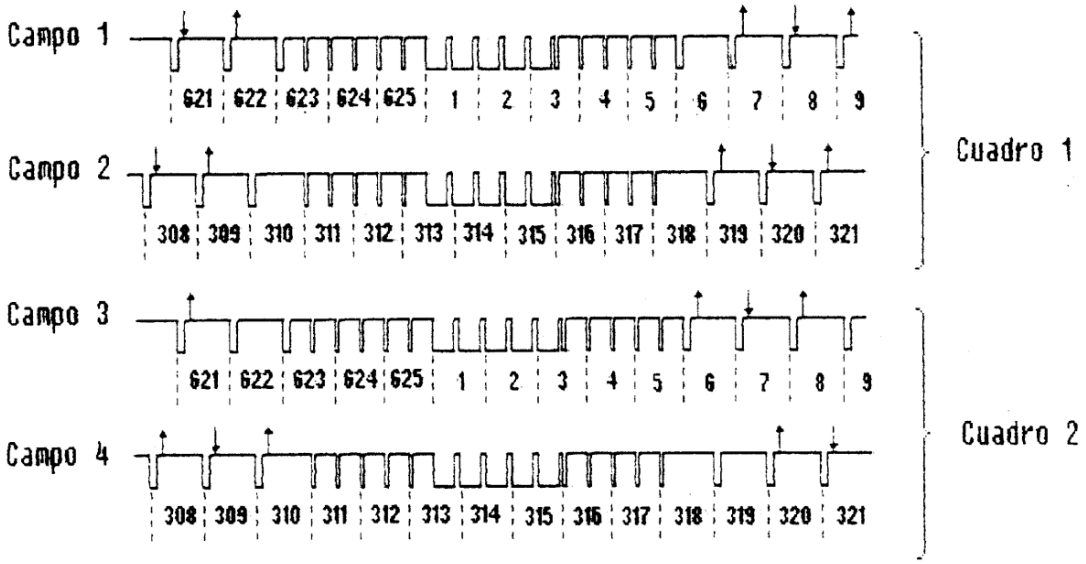

Figura A1.4. Secuencia de cuatro campos o secuencia BRUNCH

Secuencia de ocho campos o secuencia PAL

Para estudiar la secuencia de ocho campos o secuencia PAL, tenemos que basarnos exclusivamente en la fase de la subportadora.

Frecuencia subportadora: 4.433.618,75 Hz

Frecuencia de líneas: 15.625 Hz

Ciclos subportadora por línea: 283,7516 Hz

En cada línea hay 283 ciclos completos de subportadora y 0,7516. Por lo que la fase se repetirá cada 2.500 líneas.

En consecuencia los campos 5 a 8 serán idénticos a los campos 1 a 4 excepto en la fase de la subportadora que estará invertida 180º.

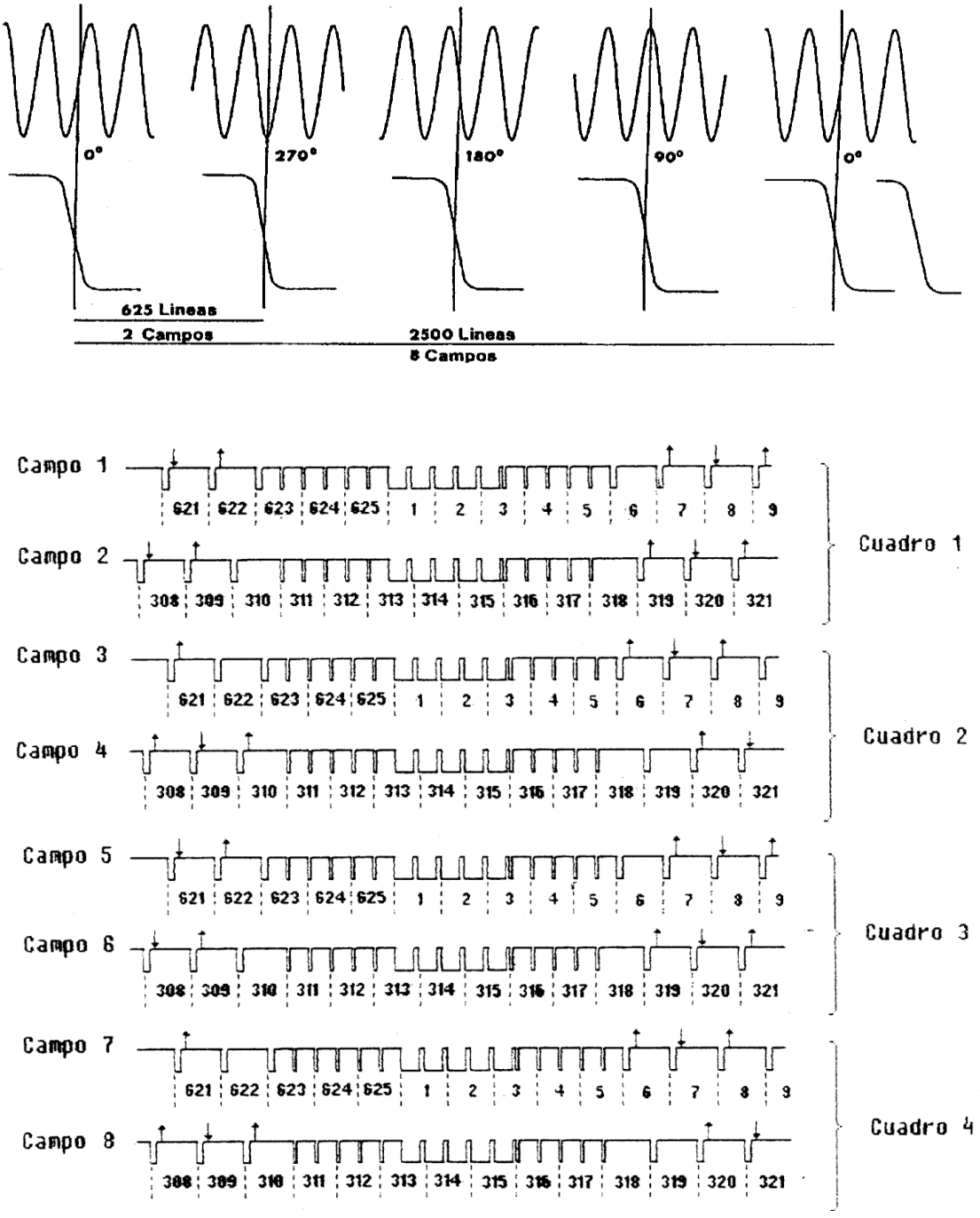

Figura A1.5. Secuencia de 8 campos o secuencia PAL

ANEXO 2: CONTENIDO DEL CD

El contenido del CD de este Trabajo Final de Carrera es:

- Este documento de texto en formato pdf.
- SDK 7.1.zip. Contiene el Software Developement Kit de Blackmagic utilizado.
- CODIGO-TFC.zip. Contiene el proyecto de Visual Studio completo realizado por mí.