Annex to DE5.1.1 of AXMEDIS project
Descrizione delle proprietà del ***reference software IM1 Player***

Technical report derived from the work of the following students:
Claverini Aldo, Casucci Marco, Iacopetti Lorenzo, Laura Carnevali, Marco Lippi;
under the strict supervision of DSI people such as: P. Nesi, D. Rogai, P. Bellini, A. Vallotti, etc.

1. Introduzione a MPEG-4

Oggetto di studio di questo elaborato è stato l'MPEG-4 Reference Software, in particolare la parte riguardante la decodifica e la riproduzione dell'audio all'interno dell'Im1-2Dplayer.

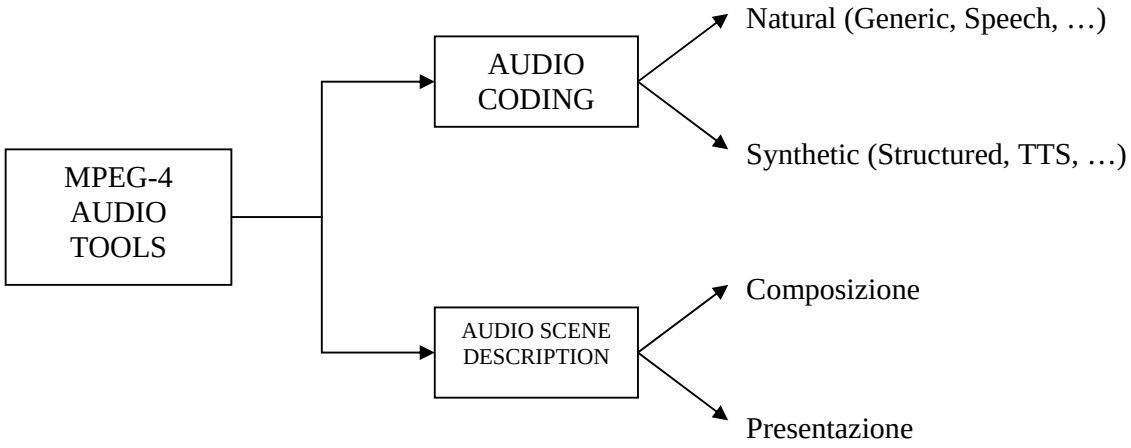
MPEG-4 è uno standard che fornisce una serie di strumenti per i contenuti multimediali, per soddisfare le esigenze di autori, sviluppatori, service providers e utenti.
Esso consente infatti la produzione, la distribuzione ed il consumo di contenuti multimediali che hanno grande riusabilità e flessibilità, come la televisione digitale, i media contenuti nelle pagine web e molti altri ancora; con MPEG-4 è inoltre possibile proteggere i contenuti multimediali, consentendo quindi transizioni sicure anche su Internet.
Un altro vantaggio fornito da questo standard è la possibilità di sfruttare anche bassi bitrate di trasmissione, così come utilizzare sistemi mobili: questo avviene per mezzo di una specifica Quality of Service (QoS) che viene richiesta per la distribuzione dei media.
I file MPEG-4, inoltre, possono essere interattivi, il che significa che l'utente finale può fruire del contenuto multimediale in modo del tutto personalizzato.

Per soddisfare tutte queste caratteristiche, lo standard MPEG-4 utilizza una ben precisa tecnica per la descrizione di un contenuto multimediale: una scena audiovisiva, infatti, è composta da un insieme di oggetti audiovisivi (*audiovisual objects*) le cui proprietà devono essere definite mediante un linguaggio di descrizione della scena.
Questo meccanismo offre molte potenzialità, in quanto consente l'interazione con elementi individuali del media (gli *objects* di cui sopra), adatta gli schemi di codifica al contenuto stesso, e consente inoltre molto facilmente il riutilizzo del contenuto audiovisivo.

Per quanto riguarda più specificatamente la codifica dell'audio, MPEG-4 supporta una grande varietà di bitrate e un elevato numero di canali di trasmissione, in numerosissimi ambiti applicativi:

- General audio signals – da bitrate molto bassi (anche solo 6 kbit/s per canale) in formato mono, fino a elevati rate trasmissivi multicanale
- Speech signals – tools specifici per la codifica/decodifica del linguaggio parlato
- Synthetic audio – strumenti per la sintesi elettronica di file audio, ad esempio mediante l'interfacciamento con programmi che generano suoni con timbri diversi, ed anche per la creazione e l'utilizzo di spartiti musicali veri e propri

- Text-To-Speech audio – i TTS audio tools consentono di tradurre il testo in parlato



2. MPEG-4 Reference Software

2.1 Binary Format for Scene (Bifs) e Multiplexing

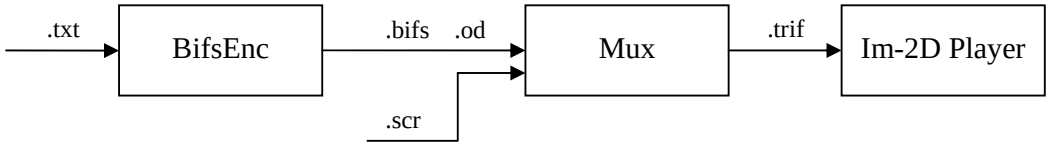
Il Reference Software per MPEG-4 si propone di fornire una serie di strumenti per la creazione e la riproduzione dei file MPEG-4: nel contesto di questo elaborato si è utilizzato Im1-2Dplayer, un player MPEG-4 implementato all'interno del Reference Software, che supporta e riproduce files con estensione .mp4 o .trif (trivial format).

I files utilizzati in questo elaborato per testare l'applicazione sono stati ottenuti mediante le utilities BifsEnc e Mux fornite con il Reference Software.

- BifsEnc è un codificatore Bifs (Binary Format for Scenes), che effettua il parsing di files di testo (.txt) che descrivono la scena da rappresentare (indicando anche le url dei file che saranno utilizzati per comporre il file finale), generando un file .bifs ed un file .od (Object Descriptor).

- Mux è un multiplexer che può essere utilizzato per costruire un file mpeg-4 (in formato .trif) a partire dai file generati dal BifsEncoder, con l'appoggio di un file .scr (script file) che descrive gli streams che si intendono multiplexare.

Il processo di costruzione di un file .trif è riassunto nella seguente figura:



2.2 Audio

Assieme al Reference Software sono stati forniti alcuni script di esempio per costruire files .trif che potessero essere letti dal Player-2D: questi files sono composti da uno stream audio, codificato in formato G723, e da uno stream video, codificato nel formato H263. Non sono stati forniti files audio/video funzionanti per nessun altro formato, nonostante all'interno del Reference Software siano presenti anche altri decoder, quali il decoder audio AAC o il decoder per JPEG.

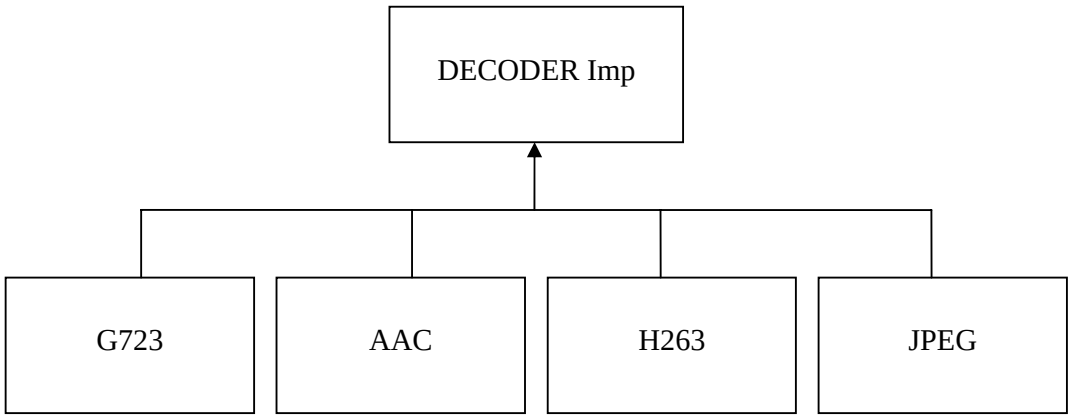
Le prove effettuate per l'audio testing nel corso di questo elaborato sono perciò state eseguite esclusivamente su files audio G723.

Anche utilizzando i files G723 forniti con il Software, si sono tuttavia riscontrate alcune difficoltà nella riproduzione, più precisamente nella sincronizzazione dell'audio con il video.
Quando i campioni giungono all'Audio Renderer, questi effettua un controllo sul loro timestamp, verificando che esso sia maggiore dell'Object Clock attuale (orologio che misura il tempo di riproduzione del brano): se il timestamp risulta invece minore del tempo segnalato dal clock, significa che i dati sono arrivati in ritardo, per cui il software semplicemente ne effettua uno "skip", di fatto perdendoli, in quanto non verranno mai riprodotti.

Eliminando il controllo sul timestamp, è stato possibile riprodurre i files audio, testando quindi il corretto funzionamento del decoder audio G723; tuttavia ciò ha provocato un non corretto sincronismo tra audio e video in quanto i campioni audio venivano comunque riprodotti, nonostante il loro timestamp indicasse che essi erano giunti in ritardo.

2.3 Implementazione e gestione dei decoder

All'interno del Reference Software sono implementati alcuni decoder audio (G723, AAC) e video (JPEG, H263), secondo una struttura gererachica ben precisa che deve essere seguita anche per la creazione di eventuali nuovi decoder.



Ciascun decoder esistente, e ciascun nuovo decoder che si intende creare, deve ereditare dalla classe DecoderImp presente all'interno del progetto: si tratta di una classe virtuale che fornisce una implementazione di default dei metodi del decoder; si richiede ovviamente al programmatore di

effettuare l'overload dei metodi caratteristici del decoder che si sta costruendo, in modo da renderlo funzionale per il tipo di stream che si intende decodificare e riprodurre.

Una volta costruito il decoder, ed una volta effettuato l'overload dei metodi necessari, è necessario registrare il nuovo decoder all'interno del progetto: questo è fatto mediante una factory; la riga di codice seguente mostra la registrazione del decoder G723 mediante la suddetta factory.

```
static DecoderFactory<G723Decoder> g723Decoder (6, "G723");
```

Ciascuno dei decoder che sono implementati nel progetto (sia per la parte audio, sia per la parte video) è associato ad una corrispondente DLL (Dynamic-Link Library), che viene caricata a run-time ogni volta che è necessario effettuare la decodifica del tipo di stream multimediale cui essa si riferisce.

Affinché le varie DLL – corrispondenti ai vari decoder – possano essere utilizzate in modo corretto, è necessario effettuarne la registrazione all'interno del Registro di Sistema. Per ogni tipo di stream supportato da una DLL, si deve inserire una nuova chiave nel registro, specificando la corrispondente DLL che deve essere caricata a run-time, come è illustrato nel seguente esempio:

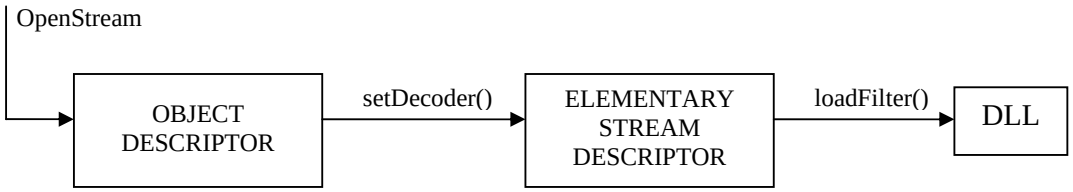
```
Nome                Tipo                Dati
Audio-c1            reg_sz              G723.dll
Visual-c2           reg_sz              H263.dll
```

La chiave del corretto funzionamento dei decoder all'interno del progetto sta proprio nell'analisi dello stream audio che si deve riprodurre, e nel conseguente caricamento della DLL giusta: il procedimento è descritto dettagliatamente nel seguito.
All'interno del Player, il processo che porta al caricamento della corretta DLL avviene per mezzo dei cosiddetti Object Descriptors (ODs), e più precisamente, degli Elementary Stream (ES) Descriptors.

Un Object Descriptor è un descrittore dei cosiddetti *media objects*, ed esso contiene quindi i riferimenti a tutti gli streams multimediali che sono associati ad una specifica locazione della scena. In caso di stream audio, ad esempio, un OD può essere rappresentato da un AudioSource node.
Ciascun Object Descriptor contiene al suo interno una serie di sottodescrittori che descrivono le caratteristiche specifiche di ciascuno stream che fa parte dell'OD.

Più precisamente, sono gli Elementary Streams (ES) Descriptors a descrivere dettagliatamente ciascuno stream elementare, e sarà mediante l'utilizzo di essi che verranno settati i decoder corrispondenti agli streams utilizzati.

La figura seguente illustra nel dettaglio il processo di settaggio del decoder:



Al momento dell'apertura dello stream, vengono analizzati i vari Object Descriptor che lo compongono; si richiede che ciascun ES Descriptor fornisca le informazioni necessarie a caricare il decoder giusto per il file da decodificare e riprodurre. Sempre per mezzo dell'Elementary Stream Descriptor viene effettuata l'operazione di loadFilter() che provvede al caricamento della DLL che verrà utilizzata a run-time.

Ecco le istruzioni mediante le quali vengono settati i decoder : `esDescr` è un array contenente i descrittori degli Elementary Streams; per ciascuno di essi viene settato il decoder corrispondente, sulla base del tipo di stream corrente.

```
for (int i=0; i<esDescr.GetLength(); i++)
{
     MediaStreamImp *pStream =
      ((ES_DescriptorImp *) esDescr [i]) -> SetDecoder(this,pNode);
     if (pResult == NULL)
          pResult = pStream;
}
```

Di seguito si riportano invece le righe di codice mediante le quali avviene il caricamento della libreria DLL desiderata: si può notare che il caricamento a runtime avviene proprio grazie all'interazione con il registro di sistema (chiamata alla funzione `registry.CreateIstance`), che carica la libreria in base alle informazioni sul tipo di stream (contenute in szFilterType).

```
Filter *pFilter =
     (Filter *) registry.CreateInstance(szFilterType,hDLL);
if (pFilter != NULL)
     pFilter -> SetLibrary (hDLL);
return pFilter;
```

Una volta che sono stati settati tutti i decoder (in corripondenza di tutti gli stream elementari che compongono la scena che si intende riprodurre), per ciascuno di essi viene eseguito un thread specifico, che si occuperà di effettuare la decodifica dei campioni al momento opportuno.

```
for (int i = 0; i < esDescr.GetLength (); i++)
     ((ES_DescriptorImp *) esDescr [i]) -> Start ();
```

La chiamata alla funzione Start() dell'ESDescr determina anche la successiva chiamata alla funzione Start() del decoder corrispondente a tale stream elementare:

```
m_pDecoder -> Start ();
```

al cui interno c'è la chiamata esplicita al thread del decoder che viene lanciato:

```
StartDecoderThread ();
```

Da questo momento l'attività del decoder si svolge per così dire *parallelamente* a quella dell'Audio Renderer, in quanto il decoder effettua il `fetch` di ciascun frame che fa parte dello stream, lo decodifica (mediante la funzione `decode` propria di ciascun decoder) e ne effettua quindi il `dispatch`, andando in pratica a scrivere i dati (ora decodificati) sul buffer che verrà letto dall'Audio Renderer, il quale si occuperà a questo punto di dialogare con le DirectSound in modo da riprodurre il file.

La sequenza delle chiamate all'inerno della funzione Run() del decoder è dunque la seguente:

```
m_pInStream -> FetchNext(…);
[…]
Decode (…);
[…]
m_pOutStream -> Dispatch (…);
```

Queste istruzioni si ripetono all'interno di un ciclo, che non si interrompe fino a che non si raggiunge la `END_OF_STREAM`, a meno che lo stream in questione non venga disattivato per qualche motivo esterno (indipendente dal decoder).

Bisogna sottolineare che la funzione `decode()` si differenzia da decoder a decoder: si tratta infatti di un metodo virtuale della classe DecoderImp descritta in precedenza, di cui ciascun decoder effettua l'overload. Vedremo che la funzione `decode()` del decoder G723 effettua soltanto funzioni di controllo, delegando alla libreria DirectSound la decodifica vera e propria dei campioni audio; nel caso del decoder audio AAC, invece, tale funzione implementa l'intera decodifica della forma d'onda, grazie anche alla libreria dinamica aactools.dll, anch'essa parte integrante del Reference Software.

2.4 Rendering Audio

Dopo aver settato i decoder ed eseguito i thread, è necessario stabilire una comunicazione con il Renderer Audio, in modo da predisporre la scheda audio alla riproduzione del media[1]. La gestione di questo processo avviene mediante un AudioSourceProxy.

L'AudioSourceProxy deve svolgere i seguenti compiti:

- Carica i decoder corrispondenti a ciascuno stream audio e ne estrae i parametri caratteristici (numero di canali, numero di campioni per secondo, numero di bit per campione).
- Aggiunge al Renderer Audio i nodi corrispondenti ai media che devono essere riprodotti, comunicandogli i parametri estratti dai vari decoder.
- Funge da tramite tra il Renderer Audio e gli streams, fornendo al Renderer i dati che dovranno essere inviati alla scheda audio.

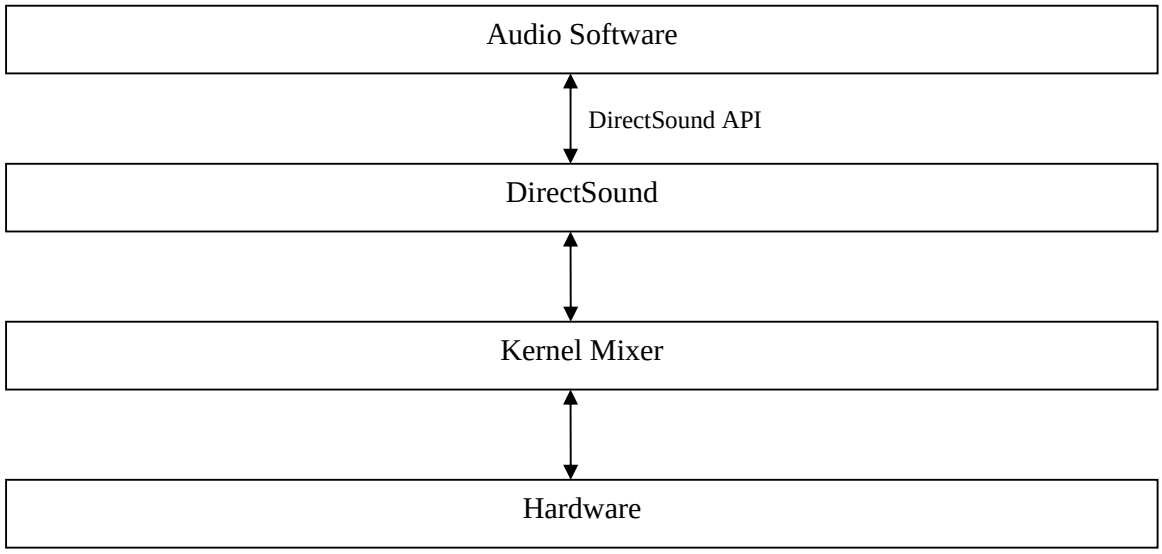
All'interno del Reference Software, la comunicazione con la scheda audio avviene per mezzo delle librerie DirectSound.

Le DirectSound sono una delle componenti principali di DirectX e si occupano della registrazione, modifica, posizione e mixaggio dei suoni.

[1] Ovviamente, un procedimento analogo avviene per gli stream video, che verranno gestiti da un Renderer Video.

Sotto il Windows Driver Model (WDM), utilizzato da Windows NT, Windows 2000 e Windows XP, le DirectSound non hanno comunque l'accesso vero e proprio all'hardware della scheda sonora: questo compito è invece svolto dal cosiddetto *kernel mixer*, che riceve gli streams audio nei diversi formati, li converte in un formato comune, ne effettua il mixaggio e li invia quindi all'hardware.

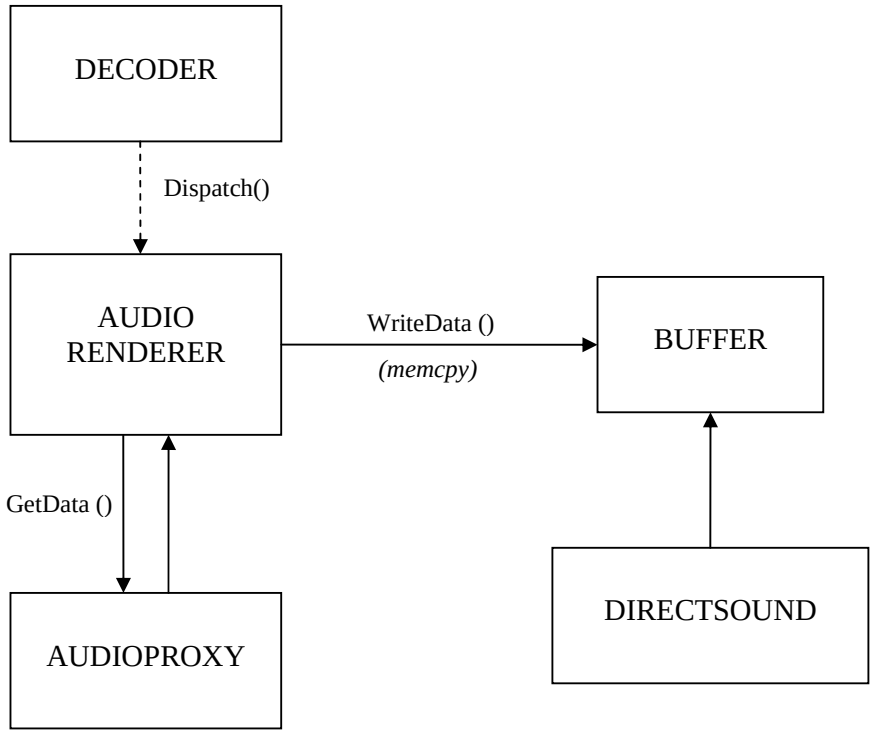
Lo schema di funzionamento è mostrato nella figura seguente:



Nel modello VxD (Virtual "something" Device), supportato dai sistemi operativi precedenti (a partire da Windows 95), le operazioni di basso livello (interazione con l'hardware) erano invece svolte dal driver virtuale Dsound.vxd.

Nel Reference Software MPEG-4 è l'Audio Renderer che comunica con le DirectSound e che invia loro i dati da riprodurre: l'Audio Renderer interagisce con l'Audio Proxy descritto in precedenza, richiedendogli i dati che devono essere inviati alla scheda audio, e va quindi a scrivere i dati (per mezzo della funzione *memcpy*) su di un buffer che verrà quindi letto dalle DirectSound.

Il funzionamento di questo procedimento è descritto dal seguente diagramma a blocchi:



Un'altra funzione dell'Audio Renderer è quella di settare i parametri della struttura dati WAVEFORMATEX, presente all'interno delle DirectSound. Tale struttura dati descrive i parametri dei file audio (in formato .wav) che dovranno essere riprodotti dalla scheda audio: essi vengono settati in conseguenza dei dati letti dai vari decoder.

Per quanto riguarda il decoder G723, che è stato quello più utilizzato nel corso di questo elaborato, il Reference Software non riporta nel dettaglio gli algoritmi di decodifica dei campioni audio da inviare al Renderer: ciò è dovuto al fatto che le librerie DirectSound contengono al loro interno un decoder G723. Il Reference Software demanda quindi alle DirectSound il compito di effettuare la codifica vera e propria dei dati che devono essere riprodotti.

Più precisamente, all'interno della funzione `Decode()` del decoder G723 viene effettuata una chiamata ad una funzione `EXTERN "C"` di `CALLBACK` che effettua la decodifica vera e propria:

```
DecSoundFrame (pInput, pOutput);
```

Il decoder presente all'interno del Reference Software svolge invece la funzione – comunque indispensabile – di far interagire l'Audio Renderer con le DirectSound: il decoder contiene infatti al suo interno i valori dei parametri che dovranno essere utilizzati per riprodurre i file audio (frequenza di campionamento, bit per campione e numero di canali). Questi parametri sono *fissati* all'interno del decoder (questo avviene probabilmente perché si tratta di un software dimostrativo) e vengono richiesti al decoder sia nel momento in cui l'AudioSourceProxy deve istanziare un nuovo nodo sull'Audio Renderer, sia quando l'Audio Renderer setta i dati del WAVEFORMATEX.
In questo modo si comunicano alla scheda audio le informazioni necessarie per riprodurre i campioni audio.

3. Testing del Software

Come già detto, tra i parametri presenti all'interno del decoder per la gestione dello stream audio, vi è il numero di canali che dovranno essere aperti sull'Audio Renderer.
Per come era stato costruito il Reference Software, tutti i decoder audio esistenti hanno il numero di canali fissato a 1 (ovvero supportano solo audio mono).

Le prove iniziali effettuate sull'audio (che si sono rivelate necessarie per riprodurre i file audio, a causa dell'errore di sincronizzazione che è già stato descritto in precedenza) sono state dunque effettuate sui files audio in formato G723 mono, forniti assieme al Reference Software.

Mediante le utilities BifsEnc e Mux, sono stati costruiti alcuni file .trif contenenti ciascuno una traccia video (in formato H263) ed una traccia audio (in formato G723); in alcuni casi si è utilizzata inoltre anche una terza componente, ovvero una semplice linea di testo che veniva stampata a video parallelamente alla riproduzione degli altri due streams.

Vediamo un po' più nel dettaglio l'organizzazione di questi file di testo (.txt e .scr) che hanno portato alla creazione dei file .trif riprodotti dal Player2D.

Il file .txt deve contenere un riferimento alla presenza di una traccia audio all'interno del file che si intende creare; questo riferimento viene effettuato mediante i tags Sound2D e AudioSource, come mostra il codice sottostante.
La riga `url 13` indica invece che questo stream audio sarà descritto (all'interno del file .scr) dall'ObjectDescriptor che ha ID pari a 13.
Il fatto che lo `stopTime` sia settato a `-1.0` indica che si deve riprodurre l'audio fino alla fine dello stream.

```
Sound2D
{
     source AudioSource
     {
          url 13
          startTime 0.0
          stopTime -1.0
     }
}
```

Nella pagina seguente si riporta invece la descrizione dell'oggetto 13 all'interno del file .scr che verrà processato dal multiplexer Mux.

Ovviamente bisogna tenere conto del fatto che, all'interno dei file .txt e .scr non ci sono soltanto le definizioni dei nodi audio che saranno inseriti nel file .trif finale, bensì c'è una descrizione completa della scena che si intende riprodurre, compresi dunque i nodi video (con i loro appositi descrittori) ed eventuali nodi di grafica o di testo.

```
ObjectDescriptorID 13
```

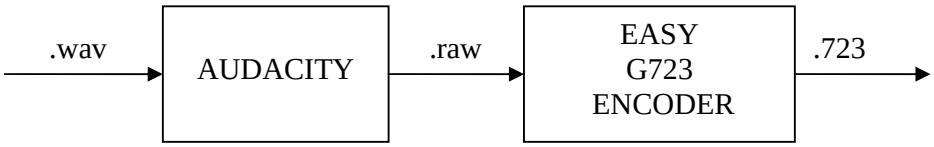
```
esDescr
[
      {
            ES_ID 2113
            muxInfo
            {
                  fileName linda.723
                  streamFormat G723
            }
            streamPriority 4
            decConfigDescr
            {
                  streamType 5                  // AudioStream
                  objectTypeIndication 0xc1     // G723
                  bufferSizeDB 300
            }
            slConfigDescr
            {
                  predefined 3
            }
      }
]
```

I parametri che si trovano all'interno del tag `decConfigDescr` servono ad identificare il tipo di stream che si deve inserire nel media (`streamType 5` → `AudioStream`) ed il formato dello stream stesso: in questo caso, trattandosi di un file G723, il codice utilizzato deve essere `0xc1`.

Utilizzando i file di esempio così creati, si è potuto effettuare il debug dell'applicazione, e si è così potuto comprendere a fondo il funzionamento dei decoder audio e l'interazione con le DirectSound, mediante l'AudioSourceProxy e l'Audio Renderer.

Una volta messa a punto questa parte, si è cercato di costruire un file .trif che contenesse come traccia audio uno stream stereo anziché mono (oltre ad una traccia video che è stata lasciata per verificare la correttezza dell'interazione con il Video Renderer); per fare questo è stato necessario utilizzare altri programmi shareware scaricabili da Internet, per la conversione di formato dei file audio e la generazione di un file G723 compatibile con il nostro Im1-2DPlayer.

Il procedimento per la creazione del file G723 è mostrato nel seguente diagramma a blocchi:



Audacity e EasyG723Encoder sono i due software che sono stati utilizzati per generare, a partire da un file .wav stereo, e passando per il formato .raw, un file audio stereo nel formato G723.

Utilizzando tali software si è potuto procedere al seguente esperimento: si è creato un nuovo file audio stereo mixando due tracce ben distinguibili tra loro (la voce di una donna ed il suono delle campane), e si è provato a riprodurre il file con il Player2D sia con un solo canale aperto sul decoder, sia con due.

Nonostante la bassa qualità del formato audio G723 (predisposto per 8000 Hz e 16 bit per parola di codice) ed i problemi di sincronizzazione con il video già citati in precedenza, si è potuto riscontrare un notevole miglioramento nella riproduzione audio quando il numero di canali era settato a due, proprio perché si stava tentando di riprodurre un file stereo.
Ciò significa che la funzione svolta dal decoder G723 è proprio quella di aprire la comunicazione con i driver della scheda audio (le DirectSound), fornendo le caratteristiche del formato dello stream che si intende riprodurre.

Analisi dettagliata IM1

In questo documento sarà spiegato, dettagliatamente, sia il funzionamento che le proprietà del reference software *Im1 player*, player MPEG-4 di riferimento per chiunque deve elaborare documenti di tipo MPEG-4. Questo nuovo formato standard è stato introdotto per permettere di realizzare una vasta quantità di funzionalità. Ad esempio, è possibile:

- Interagire col contenuto attraverso hyperlink (come accade spesso sul web);
- Dividere i flussi audio-video in sotto sistemi indipendenti, così da poterli riusare, modificare o renderizzare ogni sotto-sistema in modo personalizzato (per esempio possiamo sottocampionare lo sfondo rispetto agli oggetti in primo piano);
- Creare oggetti artificiali, sia 2D che 3D: si può creare effetti a partire da forme geometriche quali rettangoli, cerchi, linee, punti e così via, per realizzare cornici, grafici, o altro; mentre nel 3D queste potenzialità sono ancora più grandi: si può creare forme come sfere, cubi,… per poi applicarci varie texture;
- Si può decidere come costruire la scena a partire dai flussi base dei dati, cioè quando far partire un flusso, come combinarlo con gli altri, etc.

Queste proprietà sono state realizzate dal gruppo MPEG nel reference software “IM1 player” per esemplificare lo sviluppo dei player MPEG-4. La realizzazione di un reference software obbliga gli altri sviluppatori a costruire i loro software in modo compatibile con il progetto del gruppo MPEG. La diretta conseguenza è il fatto che tutti i player sul mercato saranno compatibili con lo standard MPEG-4.

L’obiettivo che ci siamo posti in questo elaborato è quello di costruire il reference software, i cui sorgenti sono disponibili in rete, e testare tutte le funzionalità che sono state dichiarate. In particolare interessa la realizzazione di figure geometriche all’interno della scena e le loro possibili

composizioni. Per questo occorre una buona conoscenza del linguaggio BIFS con il quale viene descritta ogni scena di un filmato MPEG-4.

In seguito l'obiettivo sarà realizzare un altro decoder per il reference software e di integrarlo al suo interno. Per questo obiettivo sarà necessario studiare le interfacce che il reference software mette a disposizione e, se necessario, dovrà anche essere creato un nuovo nodo BIFS per gestire il nuovo tipo di flusso dati.

Costruzione del reference software:

Il reference software viene fornito in C++ ed è possibile costruirlo attraverso il programma Microsoft Visual C++ 6.0. Insieme al reference software, è necessario ottenere anche alcune particolari librerie Java, necessarie per la parte Mpeg-j del player: è, infatti, possibile eseguire anche script java all'interno di un filmato per le funzioni più complesse.
Le librerie richieste sono: Java Communication API e JavaNativeInterface.Registry.

La costruzione del player si sviluppa in diversi passi:

- Settaggio delle variabili d'ambiente, seguendo le istruzioni fornite dalla documentazione di supporto;
- Costruzione della parte mpeg-j: è necessario, quindi, avere installata la Java Development Kit, liberamente scaricabile dal sito della Sun;
- Costruire i progetti del reference software IM1 Player, tra cui i più importanti per i nostri fini sono 3: BifsEnc. Mux, Im1Player 2D.

BifsEnc:

Questo progetto rientra nell'encoder BIFS. Permette di tradurre una scena in linguaggio BIFS, scritto semplicemente in un file di testo, nella corrispondente codifica binaria da integrare nel file MPEG-4. Inoltre crea, per ogni flusso elementare, come possono essere audio, video, immagine, etc.. , un file con il suo Object Descriptor. Il tool che viene costruito è una console, che può essere mandata in esecuzione con una linea di comando, ovvero:

`BifsEnc` *`pathname`* `[-{`**`v`**`|`**`n`**`}]`

L'estensione del file di input è il txt. Il secondo argomento, ovvero quello tra parentesi quadre è opzionale. Se lancio l'applicazione con il **v,** allora lo stream in output possiede in ogni linea un campo che mostra la grandezza, il nome ed il valore del campo. Se utilizzo l'opzione ***n***, allora codifico la scena con una particolare opzione che riporta l'Username. Una osservazione interessante è che il file di input potrebbe contenere anche dei commenti, rappresentati da linee speciali precedute da caratteri particolari, quali # e //. Abbiamo già accennato al fatto che l'applicazione produca un file .**bif** ed uno .**od,** ma crea anche un file .**lst.** Come suggerisce il nome, questo file riporta una lista di tutte le linee di ingresso, ognuna seguita da una descrizione degli eventuali errori. Questa traduzione consente di risparmiare spazio, generalmente con un rapporto di compressione di 10:1.

Mux:

È il progetto del programma Mux.exe che permette di multiplexare tutti i flussi elementari creati dal BifsEnc.exe in un unico file nel formato MPEG-4. Esistono due possibili formati: .**trif** e .**mp4**. Il formato standard è quello .mp4, mentre quello .trif è per le prove (trivial format). Anche il Mux è una console.

Im1Player2D:

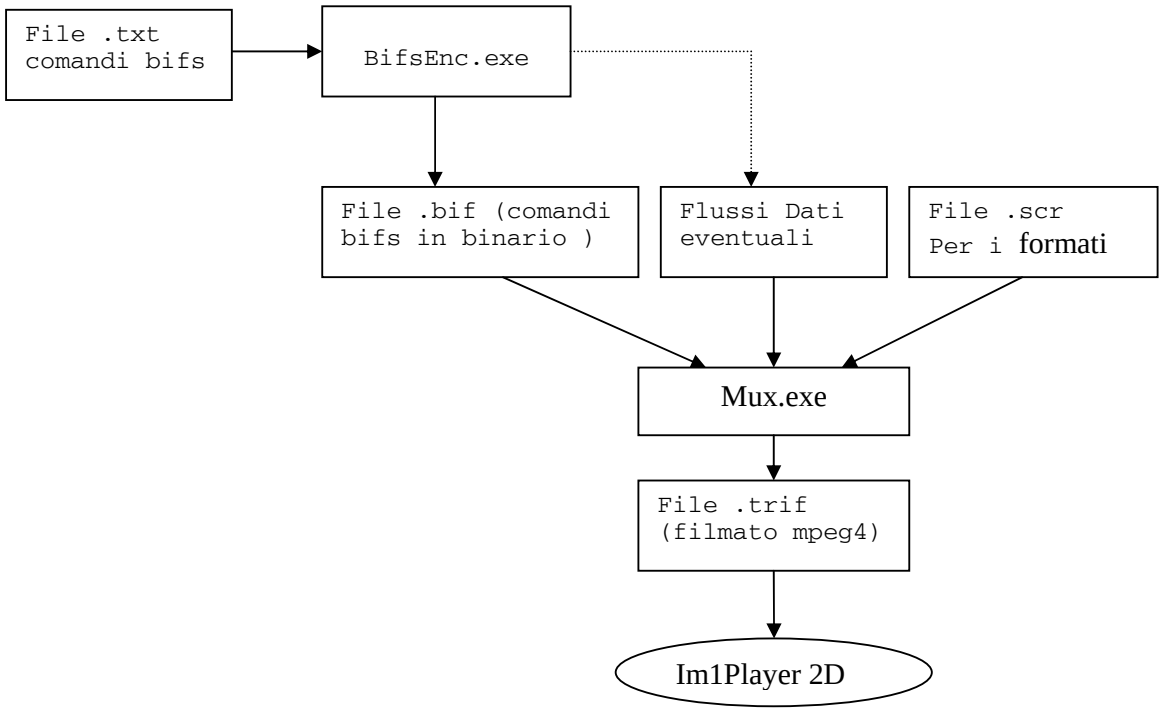
Questo è il progetto del Player, grazie al quale è possibile eseguire tutti i filmati creati con le due precedenti applicazioni.

Una volta costruiti tutti i progetti, sono disponibili i tre programmi eseguibili. Adesso possiamo provare a scoprire le funzionalità che questo player mette a disposizione.

Funzionalità del IM1 Player

Per analizzare le funzionalità che il reference software mette a disposizione è necessario capire la struttura dei comandi BIFS. Di seguito riportiamo i listati dei files di prova che abbiamo creato per verificare le principali funzioni che vengono realizzate.

Le fasi di costruzione del test sono schematizzate di seguito:



La parte che ci riguarda è quella dei comandi Bifs, perciò riportiamo i listati delle scene di test che abbiamo creato per comprendere le potenzialità del reference software.
Alleghiamo tutti i codici sorgenti e tutti i file creati per poter provare personalmente i risultati. Ovviamente se non introduciamo il Bifs, nulla potrà essere presentato in maniera esaustiva. Quindi:

Bifs:

È un formato binario, il che implica la forzata presenza di un compilatore che traduca la scena da un file di testo ad un file binario. Il formato testuale utilizzato è il cosiddetto XMT. È stato

standardizzato dal MPEG. Per prima cosa, esponiamo la struttura della scena MPEG-4 e come creare semplici scene composte da figure geometriche, quali, per esempio, rettangoli, cerchi, quadrati ecc..
Cominciamo con la rappresentazione di un rettangolo colorato.

Rect & Color:

Questo è il file testo che poniamo in input al BifsEnc.exe per produrre dei rettangoli colorati:

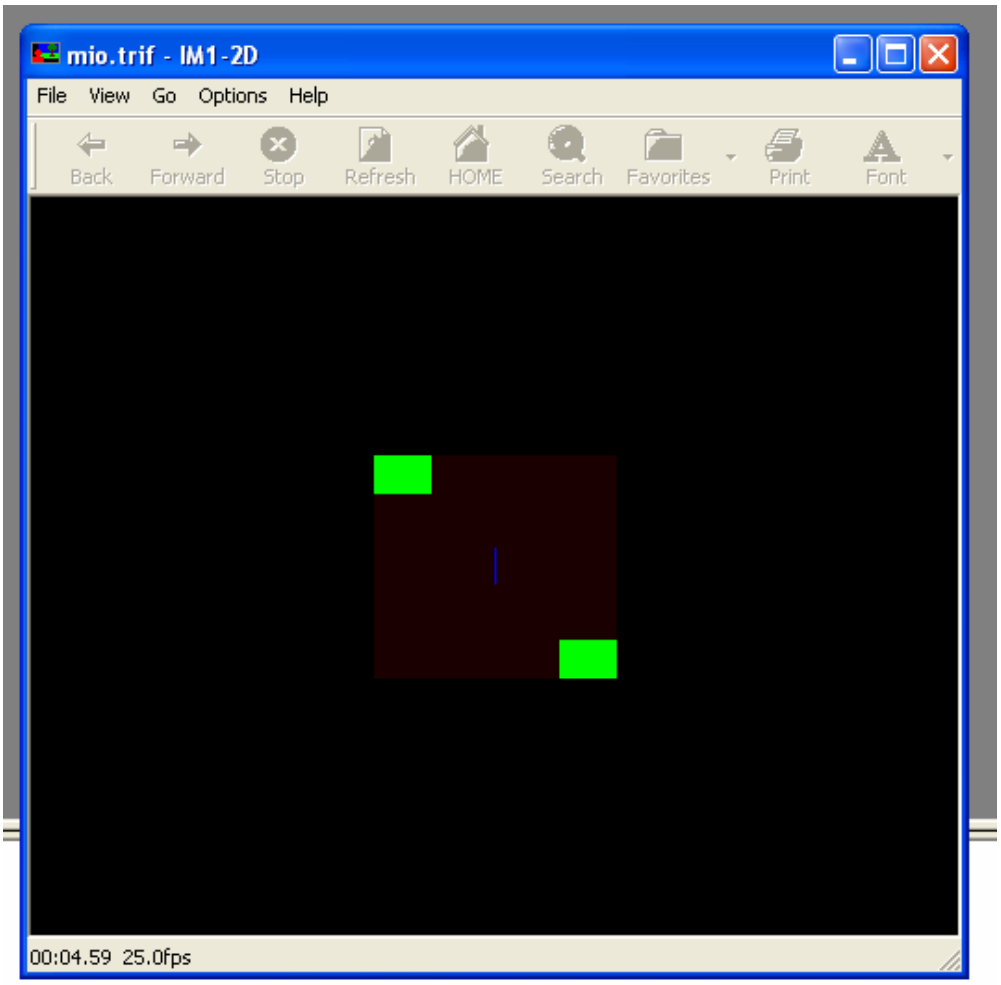
```
OrderedGroup {
      children [
            Background2D {
                        backColor 0.1 0.0 0.0   #sfondo
                          }

            Transform2D {
            translation 50.0 -50.0  #posizionarlo;
            children [
                  Shape {
                              appearance Appearance {
                              material Material2D {
                                    lineProps LineProperties {
                                          lineColor 1.0 1.0 1.0
                                          width 20.0
                                    }
                                    emissiveColor 0.0 1 0.0
                                    filled false
                                    transparency 0
                                    }
                        }
                              geometry Rectangle {
                              size 30.0 20.0
                              }
                  }
            ]
      }
#secondo rettangolo

      Transform2D {
            translation -50.0 50.0
            children [
                  Shape {
                              appearance Appearance {
                              material Material2D {
                                    lineProps LineProperties {
                                          lineColor 1.0 1.0 1.0
                                          width 20.0




                                    }
                                    emissiveColor 0.0 1 0.0
                                    filled false
                                    transparency 0
                                    }
                        }
```

```
                              geometry Rectangle {
                              size 30.0 20.0
                              }
                        }
                  ]
            }

  #terzo rettangolo

Transform2D {

            translation 0.0 0.0
            children [
                  Shape {
                              appearance Appearance {
                              material Material2D {
                                    lineProps LineProperties {
                                          lineColor 1.0 1.0 1.0
                                          width 20.0
                                    }
                                    emissiveColor 0.0 0.0 1.0
                                    filled false
                                    transparency 0
                                    }
                        }
                              geometry Rectangle {
                              size 0.0 20.0
                                    #per fare la linea, creo un rettangolo.
                                    #NB: dimensioni: n° di pixel che vuoi - 1
                              }     #cioè 0 per una linea ( 1 pixel )!
                        }
                  ]
            }
            ]
}
```

Affrontiamo la comprensione del linguaggio Bifs utilizzando questo esempio. Notiamo, innanzitutto, la presenza di una parola chiave quale **`OrderedGroup.`** La prima cosa da notare, infatti, è che il linguaggio Bifs si basa su una struttura ad albero, le cui foglie contengono i media object, come possono essere il video, le immagini o l'audio, mentre i nodi rappresentano gruppi di media object. La radice di questo albero, invece, definisce il contesto grafico di questa rappresentazione, ovvero il 2D oppure il 3D. Per ora, considereremo solo esempi in 2D, la cui radice può essere chiamata sia `Layer2D` che `OrderedGroup.` Abbiamo, quindi, capito cosa significa `OrderedGroup.` Questo nodo radice raggruppa una serie di figli, che vengono rappresentati in ordine con la loro dichiarazione nel file, ovvero il primo dichiarato è il primo ad essere "disegnato" sullo sfondo. Il nodo **`Background2D`** fornisce le caratteristiche dello sfondo della scena. Per rappresentare la forma geometrica particolare, invece, utilizziamo il nodo **`Shape.`** Si compone di due campi: il campo geometry definisce la figura stessa, mentre il campo appearance fornisce le caratteristiche visive della figura specifica, come possono essere la luminosità oppure il colore oppure la texture, etc.. Come è semplice vedere, il campo appareance contiene a sua volta altri 4 campi:

1. **Filled:** indica se la specifica figura è "riempita";
2. **EmissiveColor:** indica il colore della figura;
3. **Transparency:** indica il livello di trasparenza della figura;
4. **LineProps:** indica le proprietà del bordo.

Il risultato a cui siamo giunti, una volta salvato il .trif e dato in ingresso al player, è questo:



Si nota che la linea centrale è fatta con un rettangolo di dimensioni 1x20, perché i comandi per disegnare le linee non funzionano. Siamo, quindi, arrivati alla conclusione che il player riesce a visualizzare sia rettangoli che linee, anche se per quel che riguarda le linee dobbiamo usufruire di un piccolo stratagemma.

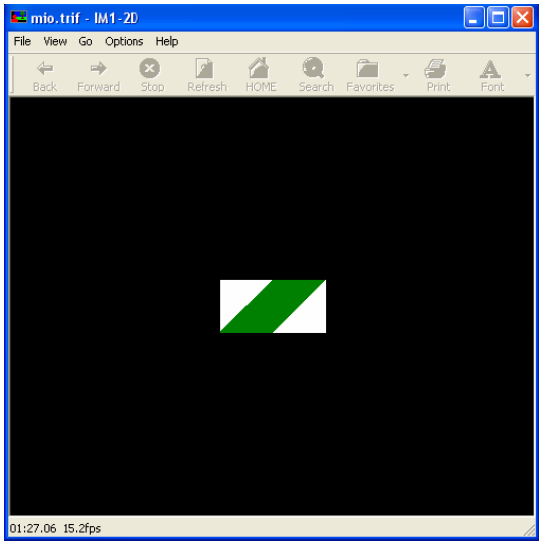
Rotazione:

Questo è il file testo che poniamo in input al BifsEnc.exe per produrre una rotazione di una determinata figura. La figura, come si può appurare dal campo geometry, è un rettangolo. Tramite il parametro **rotationAngle**, sottoponiamo a rotazione la figura. Riportiamo di seguito il file sopra nominato:

```
OrderedGroup {
  children [
    Background2D {
      backColor 1.0 1.0 1.0
    }
    Transform2D {
      rotationAngle -1.1780972          #comandi per la rotazione
      scale 1.0 2.4142137
      translation 0.0 25.0
```

```
        children [
      Transform2D {
        rotationAngle 0.7853982
        scale 0.5411961 0.76536685        #e per il fattore di scala
        children [
          Transform2D {
            translation 25.0 -25.0
            children [
            Shape {
              appearance Appearance {
                material Material2D {
                  emissiveColor 0.0 0.5019608 0.0
                  filled true
                  lineProps LineProperties {
                  lineColor 0.0 0.0 0.0
                  }
                }
              }
              geometry Rectangle {
                size 50.0 50.0
              }
            }
            ]
          }
        ]
      }
        ]
      }
    ]
  }
```

Si nota immediatamente che abbiamo un figlio, ovvero children, nominato, quindi possiamo già immaginarci di avere un rettangolo sovrapposto ad uno sfondo di colore bianco come riportato dal campo **backcolor** di background2D. Il risultato a cui arriviamo è visibile nella figura riportata qui sotto:



Come è visibile dalla foto, siamo riusciti a riprodurre sul player IM1 anche l'effetto della rotazione di una specifica figura, ovvero, in questo caso, del rettangolo. Per questo tipo di effetto non è stato riscontrato alcun tipo di problemi durante il nostro lavoro di testing.

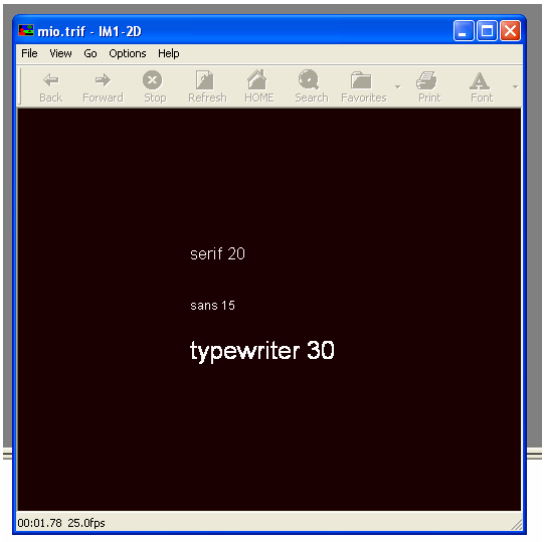
Testo:

Questo, invece, è il file testo che poniamo in input al BifsEnc.exe per produrre una determinata scritta sul player. Si può vedere come vi sia il bisogno di utilizzare nel campo geometry di Shape la parola chiave **Text.** In questo caso dobbiamo specificare i vari attributi che appartengono a **fontstyle.** Per esempio, dobbiamo specificare la grandezza del carattere, la famiglia e lo stile del carattere e così via. Il file è:

```
OrderedGroup {
      children [
            Background2D {
                  backColor 0.1 0.0 0.0
            }
Transform2D {
            translation -80.0 50.0
            children [
                  Shape {
                        appearance Appearance {
                        material Material2D {
                              lineProps LineProperties {
                                    lineColor 1.0 1.0 1.0
                                    width 1.0
                              }
                              emissiveColor 1.0 1.0 1.0
                              filled false
                              transparency 0
                              }
                        }
                              geometry Text {
                              string "serif 20"
                              fontStyle FontStyle {
                                    family serif
                                   #esiste anche Typewriter, Sans, Justify
                                    size 20.0      #da 8 a 72 di sicuro
                                    #style PLAIN
                                    }

                              }
                  }
            ]
      }

Transform2D {
            translation -80.0 0.0
            children [
                  Shape {
                        appearance Appearance {
                        material Material2D {
                              lineProps LineProperties {
                                    #lineColor 1.0 1.0 1.0
                                    width 1.0
                              }
                              emissiveColor 1.0 1.0 1.0
                              filled false
                              transparency 0
                              }
```

```
                        }
                              geometry Text {
                              string "sans 15"
                              fontStyle FontStyle {
                                    family sans
                                    size 15.0
                                          }

                              }
                  }
            ]
      }

Transform2D { translation -80.0 -50.0
            children [  Shape {
                              appearance Appearance {
                              material Material2D {
                                    lineProps LineProperties {
                                    lineColor 1.0 1.0 1.0  width 1.0
                                    }
                                    emissiveColor 1.0 1.0 1.0
                                    #filled false
                                    #transparency 0
                                    }
                        }
                        geometry Text {
                              string "typewriter 30"
                        fontStyle FontStyle { family typewriter
                                    size 30.0
                                    }
                        }
                  }
            ]
      }
      ]
}
```

Come si vede è possibile realizzare scritte di stile differente, dimensione a piacere e allineamenti voluti. Combinando con le altre funzioni già viste è possibile ruotarle, colorarle etc…



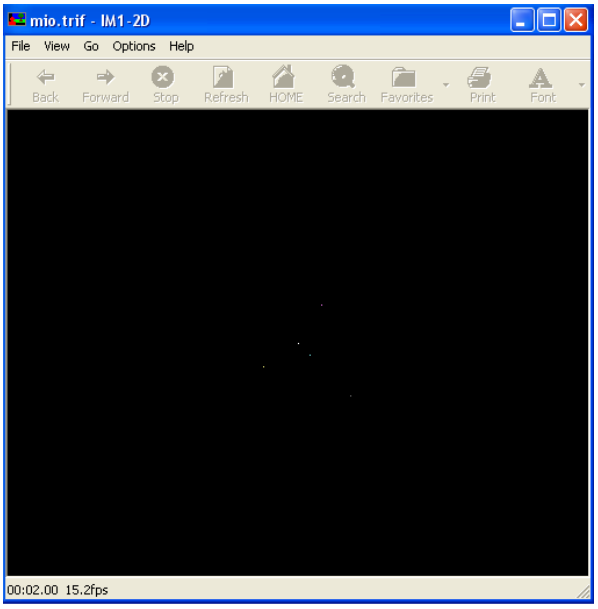
Anche nel caso delle scritte a video, non abbiamo riscontrato alcun tipo di problema.

Punti:

Questo è il file testo che poniamo in input al BifsEnc.exe per produrre dei semplici punti sul video. Per realizzare questo intento basta utilizzare **`PointSet2D`** come geometry e specificare I vari colori da attribuire a semplici punti che vengono individuate tramite coordinate, ovvero fornendo ascissa ed ordinate. Il file è:

```
OrderedGroup {
  children [
    Background2D {
      backColor 0.0 0.0 0.0
    }
    Shape {
      geometry PointSet2D {
   color Color {                  #colori in RGB
     color [ 1.0 1.0 1.0 0.5 1.0 1.0 1.0 0.5 1.0 1.0 1.0 0.5 0.5 0.5 0.5 ]
   }
   coord Coordinate2D {
     point [ 0.0 0.0 10.0 -10.0 20.0 33.0 -30.0 -20.0 45.0 -45.0 ]
   }
      }                           #Coordinate in [x1 y1 x2 y2 … ]
    }
  ]
}
```

Questo è un semplice esempio per accendere un set di pixel, di colore arbitrario, sullo schermo.



Traslazione:

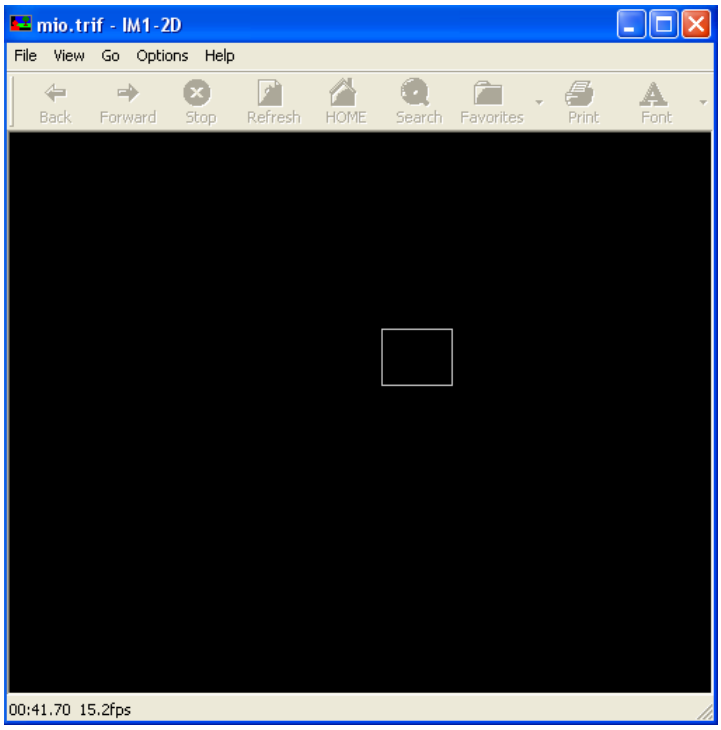
Questo esempio mostra come usare i comandi bifs per animare un oggetto (in tal caso è un rettangolo, ma niente vieta di muovere una scritta, o una combinazione di più oggetti …). In questo caso dobbiamo definire un cammino tra due campi di due nodi ed il meccanismo utilizzato è il costrutto **DEF/USE**, che permette di assegnare un nome ad un nodo (**DEF**) e riutilizzare quello stesso nodo in qualsiasi parte della scena (**USE**). La sintassi per il cammino è:

```
<Route fromNode="Node1" fromField="field1" toNode="Node2" toField="field2"/>
```

Come è facile intuire, possiamo utilizzare qualsiasi nome per denominare i vari nodi. In questo esempio, inoltre, vi è la presenza di un nodo mai incontrato, ovvero **TimeSensor.** È un nodo utilizzato per generare eventi di tipo periodico. Il periodo è dato con il campo **cycleInterval** ed è in secondi. Andando avanti con l'esposizione del file testo, incontriamo i cosiddetti Interpolatori, che permettono una interpolazione su un intervallo. La funzione di interpolazione è definita da N valori presi dal campo **key** ed N valori presi dal campo **keyValue.** Questi valori non sono ristretti ad alcun intervallo, ma devono essere disposti in ordine sempre crescente.

```
OrderedGroup {
  children [
    DEF N0 Transform2D {
      children [
      Shape {
        geometry Rectangle {
          size 50.0 40.0
        }
      }
      ]
    }
    DEF N2 TimeSensor {
      cycleInterval 10.0
    }
    DEF N1 PositionInterpolator2D {
      key [ 0.0 1.0 ]
      keyValue [ 0.0 0.0 40.0 40.0 ]
    }
  ]
}

ROUTE N2.fraction_changed to N1.set_fraction
ROUTE N1.value_changed to N0.translation
```

A questo punto abbiamo tutto quello che occorre per creare una semplice animazione. Il risultato a cui siamo arrivati è quello in figura:



Anche in questo caso non abbiamo riscontrato alcuna forma di problema.

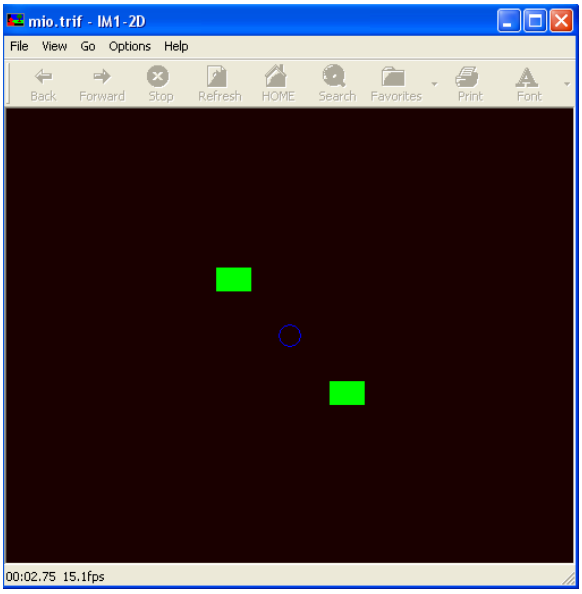
Combinazione di più oggetti:

Qui si introduce anche l'uso di cerchi. Si può giocare su parametri come colore o riempimento semplicemente indicandoli attraverso gli attributi. In questo caso abbiamo il seguente listato:

```
OrderedGroup {
      children [
            Background2D {
                  backColor 0.1 0.0 0.0
            }

            Transform2D {
            translation 50.0 -50.0
            children [
                  Shape {
                        appearance Appearance {
                        material Material2D {
                              lineProps LineProperties {
                                    lineColor 1.0 1.0 1.0
                                    width 20.0
                              }
```

```
                                    emissiveColor 0.0 1 0.0
                                    filled false
                                    transparency 0
                                    }
                        }
                              geometry Rectangle {
                              size 30.0 20.0
                              }
                        }
                  ]
            }


            Transform2D {
                  translation -50.0 50.0
                  children [
                        Shape {
                              appearance Appearance {
                                    material Material2D {
                                          lineProps LineProperties {
                                                lineColor 1.0 1.0 1.0
                                                width 20.0
                                          }
                                          emissiveColor 0.0 1 0.0
                                          filled false
                                          transparency 0
                                          }
                              }
                                    geometry Rectangle {
                                    size 30.0 20.0
                                    }
                        }
                  ]
            }
      Transform2D {
                  translation 0.0 0.0
                  children [
                        Shape {
                                    appearance Appearance {
                                    material Material2D {
                                          lineProps LineProperties {
                                                lineColor 1.0 1.0 1.0
                                                width 1.0
                                          }
                                          emissiveColor 0.0 0.0 1.0
                                          filled false
                                          transparency 0
                                          }
                              }
                                    geometry Circle {
                                    radius 10
                                    }
                        }
                  ]                                    }



            ]
      }
```

Si nota immediatamente come il seguente file testo definisca tre forme geometriche, ovvero due rettangoli ed un cerchio. Nel caso del cerchio abbiamo un nuovo parametro che indica le caratteristiche del raggio (**radius**). Il risultato a cui siamo arrivati è in figura:



Non si sono verificati problemi.
Dopo queste prime semplice prove, tentiamo di realizzare uno schema riassuntivo dei limiti del player IM1. Il comando apposito per le linee (IndexedLineSet) non viene interpretato, perciò ogni linea deve essere realizzata con un rettangolo di cui un lato ha dimensione 1.
Non siamo riusciti, per ora, a trovare il comando per creare ellissi, ma sicuramente esiste e si dovrebbe comportare come gli altri: avrà due attributi per i fuochi, uno per il raggio, uno per il colore e via dicendo. Purtroppo tante altre funzioni provate non hanno effetto:

Curve 2D: per creare questo tipo di immagini, si fissano dei punti di controllo che possono variare nel loro numero da un minimo di 3 ad un massimo non definito, stabilito dall'utente. Abbiamo scelto 10, un numero sufficiente di punti per creare curve in 2D abbastanza complesse. Infatti si richiede alla curva di passare per tutti questi punti, o almeno rappresentarne l'andamento. Nell'esempio, i punti sono rappresentati con piccoli rettangoli rossi, mentre le linee, che dovrebbero apparire con un colore azzurrino, sono disegnate tramite la funzione **IndexedLineSet2D**.

Il codice di questo esempio è:

```
OrderedGroup
{
  children
  [
    Background2D
    {
      backcolor 1.0 0.8 0.5
    }


    Transform2D
    {
      translation -385 0
      children
      [
        # 3 ---------------------------

        Transform2D
        {
```

```
translation 0 0
children
[
  DEF CPtC3 OrderedGroup
  {
    children
    [
      Transform2D
      {
        translation 0 0
        children
        [
          DEF CPRect Shape
          {
            appearance
            {
              material Material2D
              {
                emissiveColor 1.0 0.0 0.0
                filled FALSE
              }
            }
            geometry Rectangle
            {
              size 3 3
            }
          }
        ]
      }
      Transform2D { translation   0 150 children [ USE CPRect ] }
      Transform2D { translation  50 150 children [ USE CPRect ] }
    ]
  }

  Shape
  {
    appearance DEF AppILS Appearance
    {
       material Material2D
      {
        lineProps
        {
          lineColor 0.7 0.7 0.7
          lineStyle 1
        }
      }
    }
    geometry IndexedLineSet2D
    {
      coord DEF C3CP Coordinate2D
      {
        point[0 0, 0 150, 50 150]
      }
    }
  }

  Shape
  {
    appearance DEF App Appearance
    {
       material Material2D
      {
        emissiveColor 0.2 0.8 1.0
        filled FALSE
      }
    }
    geometry DEF C3 Curve2D
    {
      point USE C3CP
      fineness 1.0
    }
  }

  Transform2D
  {
    translation -10 10
    children
    [
      Shape
      {
        appearance DEF AppText Appearance
        {
          material Material2D
```

```
            {
              emissiveColor 0.0 0.0 0.0
            }
          }
          geometry Text
          {
            string ["3"]
            fontStyle DEF MyFont FontStyle
            {
              family "SANS"
              justify ["MIDDLE", "MIDDLE"]
              size 16
            }
          }
        }
      ]
    }
  ]
}

# 4 ---------------------------

Transform2D
{
  translation 70 0
  children
  [
    DEF CPtC4 OrderedGroup
    {
      children
      [
        USE CPtC3
        Transform2D { translation  50 100 children [ USE CPRect ] }
      ]
    }

    Shape
    {
      appearance USE AppILS
      geometry IndexedLineSet2D
      {
        coord DEF C4CP Coordinate2D
        {
          point[0 0, 0 150, 50 150, 50 100]
        }
      }
    }

    Shape
    {
      appearance USE App
      geometry DEF C4 Curve2D
      {
        point USE C4CP
        fineness 1.0
      }
    }

    Transform2D
    {
      translation -10 10
      children
      [
        Shape
        {
          appearance USE AppText
          geometry Text
          {
            string ["4"]
            fontStyle USE MyFont
          }
        }
      ]
    }
  ]
}

# 5 ---------------------------

Transform2D
{
  translation 180 0
  children
```

```
    [
      DEF CPtC5 OrderedGroup
      {
        children
        [
          USE CPtC4
          Transform2D { translation -40 100 children [ USE CPRect ] }
        ]
      }

      Shape
      {
        appearance USE AppILS
        geometry IndexedLineSet2D
        {
          coord DEF C5CP Coordinate2D
          {
            point[0 0, 0 150, 50 150, 50 100, -40 100]
          }
        }
      }

      Shape
      {
        appearance USE App
        geometry DEF C5 Curve2D
        {
          point USE C5CP
          fineness 1.0
        }
      }

      Transform2D
      {
        translation -10 10
        children
        [
          Shape
          {
            appearance USE AppText
            geometry Text
            {
              string ["5"]
              fontStyle USE MyFont
            }
          }
        ]
      }
    ]
}

# 6 --------------------------

Transform2D
{
  translation 290 0
  children
  [
    DEF CPtC6 OrderedGroup
    {
      children
      [
        USE CPtC5
        Transform2D { translation -50 125 children [ USE CPRect ] }
      ]
    }

    Shape
    {
      appearance USE AppILS
      geometry IndexedLineSet2D
      {
        coord DEF C6CP Coordinate2D
        {
          point[0 0, 0 150, 50 150, 50 100, -40 100, -50 125]
        }
      }
    }

    Shape
    {
      appearance USE App
      geometry DEF C6 Curve2D
```

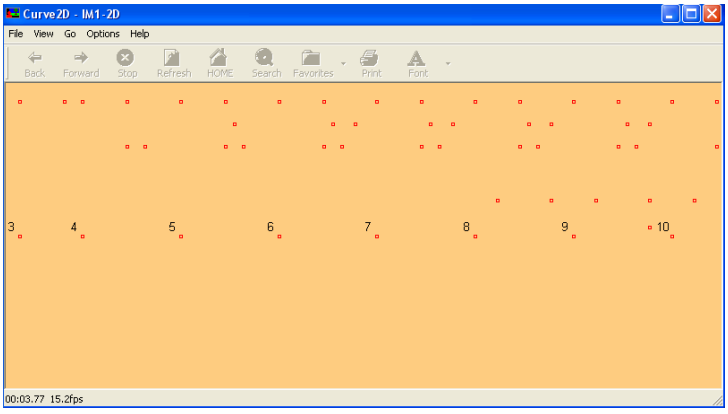
```
      {
        point USE C6CP
        fineness 1.0
      }
    }

    Transform2D
    {
      translation -10 10
      children
      [
        Shape
        {
          appearance USE AppText
          geometry Text
          {
            string ["6"]
            fontStyle USE MyFont
          }
        }
      ]
    }
  ]
}

# 7 ---------------------------

Transform2D
{
  translation 400 0
  children
  [
    DEF CPtC7 OrderedGroup
    {
      children
      [
        USE CPtC6
        Transform2D { translation -25 125 children [ USE CPRect ] }
      ]
    }

    Shape
    {
      appearance USE AppILS
      geometry IndexedLineSet2D
      {
        coord DEF C7CP Coordinate2D
        {
          point[0 0, 0 150, 50 150, 50 100, -40 100, -50 125, -25 125]
        }
      }
    }

    Shape
    {
      appearance USE App
      geometry DEf C7 Curve2D
      {
        point USE C7CP
        fineness 1.0
      }
    }

    Transform2D
    {
      translation -10 10
      children
      [
        Shape
        {
          appearance USE AppText
          geometry Text
          {
            string ["7"]
            fontStyle USE MyFont
          }
        }
      ]
    }
  ]
}

# 8 ---------------------------
```

```
Transform2D
{
  translation 510 0
  children
  [
    DEF CPtC8 OrderedGroup
    {
      children
      [
        USE CPtC7
        Transform2D { translation  25  40 children [ USE CPRect ] }
      ]
    }

    Shape
    {
      appearance USE AppILS
      geometry IndexedLineSet2D
      {
        coord DEF C8CP Coordinate2D
        {
          point[0 0, 0 150, 50 150, 50 100, -40 100, -50 125, -25 125, 25 40]
        }
      }
    }

    Shape
    {
      appearance USE App
      geometry DEF C8 Curve2D
      {
        point USE C8CP
        fineness 1.0
      }
    }

    Transform2D
    {
      translation -10 10
      children
      [
        Shape
        {
          appearance USE AppText
          geometry Text
          {
            string ["8"]
            fontStyle USE MyFont
          }
        }
      ]
    }
  ]
}

# 9 ---------------------------

Transform2D
{
  translation 620 0
  children
  [
    DEF CPtC9 OrderedGroup
    {
      children
      [
        USE CPtC8
        Transform2D { translation -25  40 children [ USE CPRect ] }
      ]
    }

    Shape
    {
      appearance USE AppILS
      geometry IndexedLineSet2D
      {
        coord DEF C9CP Coordinate2D
        {
          point[0 0, 0 150, 50 150, 50 100, -40 100, -50 125, -25 125, 25 40, -25 40]
        }
      }
    }
```

```
    Shape
    {
      appearance USE App
      geometry DEF C9 Curve2D
      {
        point USE C9CP
        fineness 1.0
      }
    }

    Transform2D
    {
      translation -10 10
      children
      [
        Shape
        {
          appearance USE AppText
          geometry Text
          {
            string ["9"]
            fontStyle USE MyFont
          }
        }
      ]
    }
  ]
}

# 10 -------------------------

Transform2D
{
  translation 730 0
  children
  [
    DEF CPtC10 OrderedGroup
    {
      children
      [
        USE CPtC9
        Transform2D { translation -25  10 children [ USE CPRect ] }
      ]
    }

    Shape
    {
      appearance USE AppILS
      geometry IndexedLineSet2D
      {
        coord DEF C10CP Coordinate2D
        {
          point[0 0, 0 150, 50 150, 50 100, -40 100, -50 125, -25 125, 25 40, -25 40, -25 10]
        }
      }
    }

    Shape
    {
      appearance USE App
      geometry DEF C10 Curve2D
      {
        point USE C10CP
        fineness 1.0
      }
    }

    Transform2D
    {
      translation -10 10
      children
      [
        Shape
        {
          appearance USE AppText
          geometry Text
          {
            string ["10"]
            fontStyle USE MyFont
          }
        }
      ]
```

```
            }
          ]
        }

      ]
    }

    # ---- The filled shapes

    Transform2D
    {
      translation -385 -160
      children
      [
        Transform2D
        {
          translation 0 0
          children
          [
            Shape
            {
              appearance DEF AppF Appearance
              {
                 material Material2D
                {
                  emissiveColor 0.2 0.8 1.0
                  filled TRUE
                }
              }
              geometry USE C3
            }
          ]
        }

        Transform2D { translation   70 0 children [ Shape { appearance USE AppF geometry USE C4  } ] }
        Transform2D { translation  180 0 children [ Shape { appearance USE AppF geometry USE C5  } ] }
        Transform2D { translation  290 0 children [ Shape { appearance USE AppF geometry USE C6  } ] }
        Transform2D { translation  400 0 children [ Shape { appearance USE AppF geometry USE C7  } ] }
        Transform2D { translation  510 0 children [ Shape { appearance USE AppF geometry USE C8  } ] }
        Transform2D { translation  620 0 children [ Shape { appearance USE AppF geometry USE C9  } ] }
        Transform2D { translation  730 0 children [ Shape { appearance USE AppF geometry USE C10 } ] }
      ]
    }

  ]
}
```

Purtroppo siamo arrivati alla conclusione che questo tipo particolare di funzione non è supportata, poiché il player ha letto l'esempio in questo modo:



Come si può vedere nella figura, le curve non vengono rappresentate. Non è possibile, quindi rendere le curve 2D tramite questa funzione. Conseguentemente è praticamente impossibile riuscire a visualizzare qualsiasi tipo di curva 2D.

Poligoni tramite Face_set: è una particolare funzione utilizzata per visualizzare poligoni regolari in successione senza il bisogno di definire più children (figli) nel file .bif, ma è utilissima pure per creare poligoni irregolari. Questa funzione è dichiarata con il nome **IndexedFaceSet2D.** Ancora una volta il player IM1 non è in grado di rappresentare l'informazione messa in input, anzi addirittura si comporta come se nulla venisse dato in ingresso. Non riproduce nulla. È inutile, quindi, riportare la figura di questo esempio.

Il codice del particolare esempio preso in esame è:

```
OrderedGroup
{
  children
  [
    Background2D
    {
      backcolor 0.3 0.3 0.3
    }

    # Color per Vertex is FALSE ------------------------------------------------------------

    # Has colorIndex
    #
    Transform2D
    {
      translation -200 150
      children
      [
        Shape
        {
          appearance DEF AppearF Appearance
          {
             material Material2D
            {
              emissiveColor 0.0 1.0 1.0
              filled TRUE
            }
          }
          geometry DEF IFS1 IndexedFaceSet2D
          {
            color DEF Colors Color
            {
              color [0 0 1, 1 0 0]
            }
            coord DEF TwoTriangles Coordinate2D
            {
              point[-30 2, 30 2, 0 44, -30 -2, 30 -2, 0 -44]
            }
            colorIndex [1 0]
            colorPerVertex FALSE
            coordIndex [0 2 1 -1 3 5 4]
          }
        }
      ]
    }

    Transform2D
    {
      translation -200 50
      children
      [
        Shape
        {
          appearance DEF AppearFL Appearance
          {
             material Material2D
            {
              emissiveColor 0.0 1.0 1.0
              filled TRUE
              lineProps lineProperties
```

```
            {
              lineColor 1.0 1.0 1.0
            }
          }
        }
        geometry USE IFS1
      }
    ]
  }

  Transform2D
  {
    translation -200 -50
    children
    [
      Shape
      {
        appearance DEF Appear Appearance
        {
           material Material2D
          {
            emissiveColor 0.0 1.0 1.0
            filled FALSE
          }
        }
        geometry USE IFS1
      }
    ]
  }

  Transform2D
  {
    translation -200 -150
    children
    [
      Shape
      {
        appearance DEF AppearL Appearance
        {
           material Material2D
          {
            emissiveColor 0.0 1.0 1.0
            filled FALSE
            lineProps lineProperties
            {
              lineColor 1.0 1.0 1.0
            }
          }
        }
        geometry USE IFS1
      }
    ]
  }


  # No colorIndex
  #
  Transform2D
  {
    translation -120 150
    children
    [
      Shape
      {
        appearance USE AppearF
        geometry DEF IFS2 IndexedFaceSet2D
        {
          color USE Colors
          coord USE TwoTriangles
          colorPerVertex FALSE
          coordIndex [0 2 1 -1 3 5 4]
        }
      }
    ]
  }

  Transform2D
  {
    translation -120 50
    children
    [
      Shape
      {
```

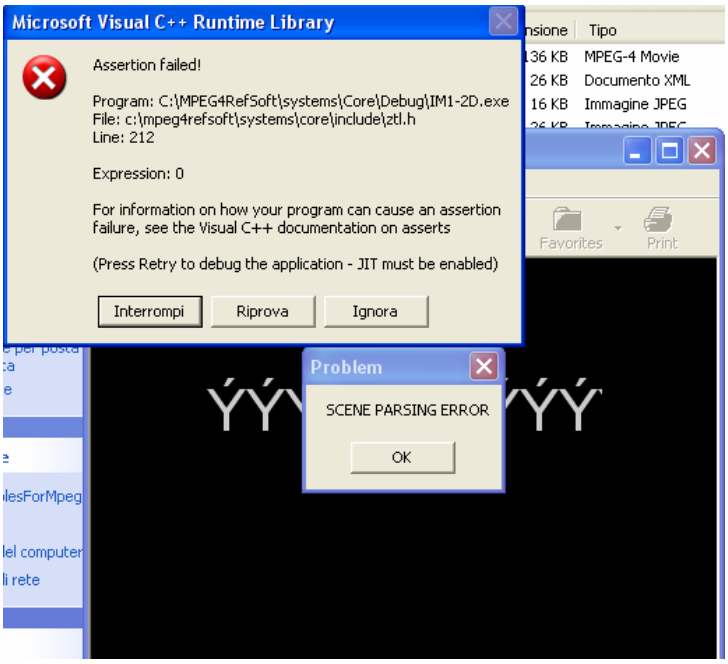
```
      appearance USE AppearFL
      geometry USE IFS2
    }
  ]
}

Transform2D
{
  translation -120 -50
  children
  [
    Shape
    {
      appearance USE Appear
      geometry USE IFS2
    }
  ]
}

Transform2D
{
  translation -120 -150
  children
  [
    Shape
    {
      appearance USE AppearL
      geometry USE IFS2
    }
  ]
}


# No colors (and no colorIndex)
#
Transform2D
{
  translation -40 150
  children
  [
    Shape
    {
      appearance USE AppearF
      geometry DEF IFS3 IndexedFaceSet2D
      {
        coord USE TwoTriangles
        colorPerVertex FALSE
        coordIndex [0 2 1 -1 3 5 4]
      }
    }
  ]
}

Transform2D
{
  translation  -40 50
  children
  [
    Shape
    {
      appearance USE AppearFL
      geometry USE IFS3
    }
  ]
}

Transform2D
{
  translation -40 -50
  children
  [
    Shape
    {
      appearance USE Appear
      geometry USE IFS3
    }
  ]
}

Transform2D
{
  translation -40 -150
  children
```

```
    [
      Shape
      {
        appearance USE AppearL
        geometry USE IFS3
      }
    ]
  }

  # Color per Vertex is TRUE ------------------------------------------------------

  Transform2D
  {
    translation 40 150
    children
    [
      Shape
      {
        appearance USE AppearF
        geometry DEF IFS4 IndexedFaceSet2D
        {
          color DEF VColors Color
          {
            color [1 1 0, 0 1 0, 0 0 1, 1 1 0, 1 0 1, 0 0.5 0.8]
          }
          coord USE TwoTriangles
          colorIndex [1 3 2 -1 5 1 4]
          colorPerVertex TRUE
          coordIndex [0 2 1 -1 3 5 4]
        }
      }
    ]
  }

  Transform2D
  {
    translation  40 50
    children
    [
      Shape
      {
        appearance USE AppearFL
        geometry USE IFS4
      }
    ]
  }

  Transform2D
  {
    translation 40 -50
    children
    [
      Shape
      {
        appearance USE Appear
        geometry USE IFS4
      }
    ]
  }

  Transform2D
  {
    translation 40 -150
    children
    [
      Shape
      {
        appearance USE AppearL
        geometry USE IFS4
      }
    ]
  }

  Transform2D
  {
    translation 120 150
    children
    [
      Shape
      {
        appearance USE AppearF
        geometry DEF IFS5 IndexedFaceSet2D
        {
```

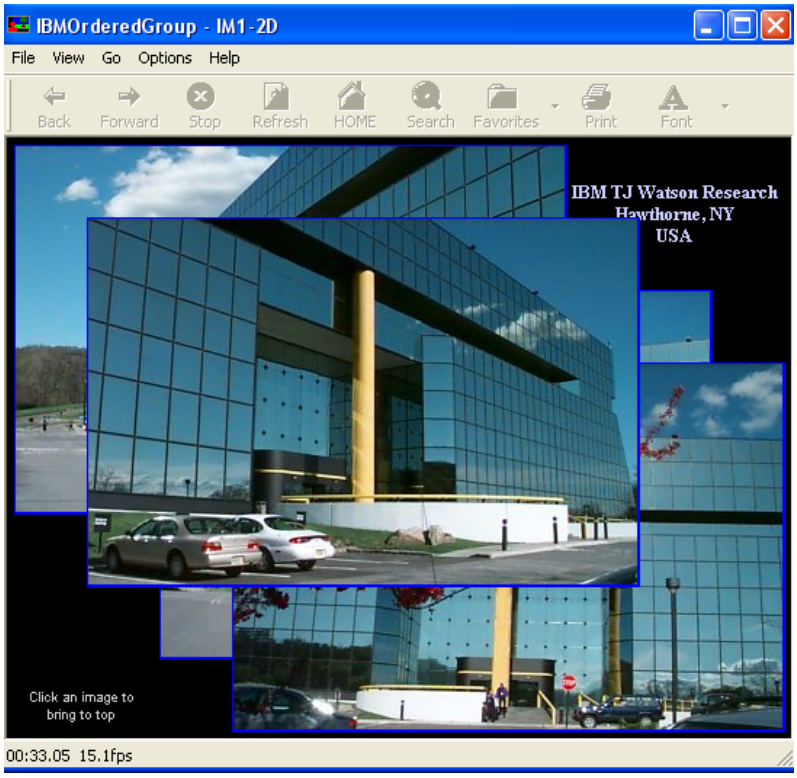
```
        color USE VColors
        coord USE TwoTriangles
        colorPerVertex TRUE
        coordIndex [0 2 1 -1 3 5 4]
      }
    }
  ]
}

Transform2D
{
  translation 120 50
  children
  [
    Shape
    {
      appearance USE AppearFL
      geometry USE IFS5
    }
  ]
}

Transform2D
{
  translation 120 -50
  children
  [
    Shape
    {
      appearance USE Appear
      geometry USE IFS5
    }
  ]
}

Transform2D
{
  translation 120 -150
  children
  [
    Shape
    {
      appearance USE AppearL
      geometry USE IFS5
    }
  ]
}

Transform2D
{
  translation 200 150
  children
  [
    Shape
    {
      appearance USE AppearF
      geometry DEF IFS6 IndexedFaceSet2D
      {
        coord USE TwoTriangles
        colorPerVertex TRUE
        coordIndex [0 2 1 -1 3 5 4]
      }
    }
  ]
}

Transform2D
{
  translation 200 50
  children
  [
    Shape
    {
      appearance USE AppearFL
      geometry USE IFS6
    }
  ]
}

Transform2D
{
  translation 200 -50
  children
```

```
      [
        Shape
        {
          appearance USE Appear
          geometry USE IFS6
        }
      ]
    }

    Transform2D
    {
      translation 200 -150
      children
      [
        Shape
        {
          appearance USE AppearL
          geometry USE IFS6
        }
      ]
    }

  ]
}
```

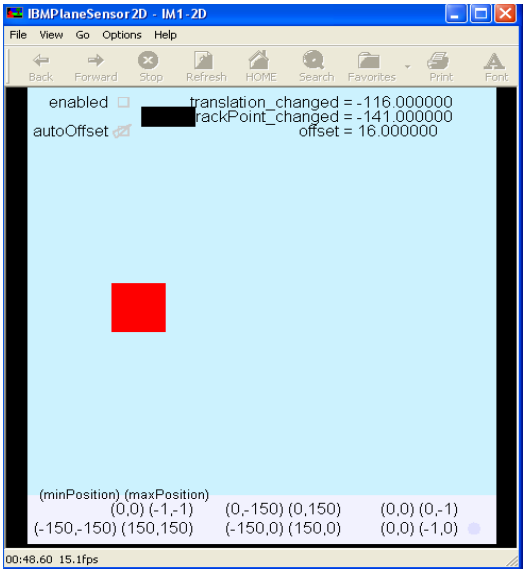
Interazione con l'utente: l'IM1 non riesce nemmeno ad interpretare e realizzare una eventuale interazione con l'utente. Abbiamo dato in ingresso una sequenza di immagini, che al massimo rimangono visualizzate per 10 secondi, ma tramite un click del mouse, dopo i 3 secondi, si dovrebbe poter passare immediatamente alla successiva immagine. Purtroppo il player non solo non interpreta assolutamente il click del mouse, ma nel momento in cui avviene (il click), da un errore che deriva da una particolare libreria, ovvero la **ztl.** Il risultato è quello della figura sottostante:



In definitiva, nell'IM1, l'interazione sembrerebbe una funzionalità inesistente. In realtà, non è del tutto vero. Ovviamente, per asserire che un determinata proprietà non sia realizzata, bisogna verificarla con più file che appartengono alla stesa categoria di esempi. Tra i tanti elementi a nostra disposizione, abbiamo provato ad aprire un file di prova della IBM ed il risultato è stato confortante:



Questo esempio, dapprima consiste nel far susseguire ad intervalli regolari di 2 secondi le quattro immagini, che si dispongono una sopra l'altra. A partire dagli 8 secondi, ciccando con il mouse su una immagine qualsiasi, quest'ultima viene portata in primo piano. E con soddisfazione, abbiamo appurato che questa funzionalità è coperta dal player IM1. Quindi, possiamo concludere, che l'interazione con l'utente talvolta è soddisfatta. Un problema possibile potrebbe essere la sincronizzazione che ci deve essere con il click del mouse e lo scorrere delle immagini. Infatti, in questo secondo esempio, il click viene correttamente interpretato solo se tutte le immagini sono apparse sul display. Altro esempio di interazione con l'utente soddisfatto può essere questo:



In questo caso, clickando con il mouse sul rettangolo rosso e tenendo premuto, è possibile trascinare il rettangolo rosso in qualsiasi parte del display. Possiamo affermare, quindi, gli esempi che richiedono l'interazione con l'utente vengono soddisfatti in gran parte, tranne quelli con i famosi problemi di sincronizzazione, come il primo esposto in questo paragrafo.

Video & Audio: Qualsiasi file di video o audio o, ancora, di video & audio, non viene visualizzato. Gli errori dati sono soprattutto 3:

1. Buffer too small;
2. Long access unit;
3. Scene parsing error.

Abbiamo provato tutti i file di esempio possibili ed immaginabili, sia quelli in dotazione con il reference software, sia quelli rimediati in rete, ma il risultato è sempre lo stesso: il player IM1 sembra non essere in grado di riprodurre alcun tipo di video o di audio.

Aspetti positivi: Oltre a quelli già esposti dell'interazione con l'utente, vi sono altri aspetti positivi del player. Non ha assolutamente problemi nel visualizzare forme geometriche e esempi semplici come quelli che abbiamo scritto ed esposti nelle prime pagine dell'elaborato. Inoltre, non vi alcun tipo di intoppo nel momento in cui vogliamo rappresentare una sequenza di immagini. Abbiamo aperto un file, che consisteva in un ciclo ripetuto di 10 immagini, le quali restavano rappresentate sul display per 0.1 secondi cadauna. Il risultato ottenuto era proprio quello cercato. Quindi sia Immagini che sequenze di immagini vengono rappresentate correttamente.

Problema principale: Dentro un singolo file .txt non è possibile scrivere tanti comandi BIFS, perché quando si multiplexa con gli altri flussi il Mux.exe dà un errore di *Long Access Unit*, oppure quando si apre con il player si ha un errore di *buffer too small*. Sembra che non sappia gestire i file lunghi. Questi due problemi sono molto simili ma si verificano in due fasi distinte del nostro lavoro.
Dopo diversi tentativi, siamo arrivati alla conclusione che il progetto Mux, non funziona in maniera corretta. Infatti, nel momento in cui si linka un file esterno all'interno del file testuale, il BifEnc realizza correttamente il file binario, ma il Mux non riesce mai a produrre un file .trif leggibile dal player, anzi, talvolta, si verifica un "*parsing error*" (non producendo, quindi, alcun file .trif).
È un problema molto grave, soprattutto per chi debba sviluppare moduli da inserire all'interno del progetto Im1. La seconda fase del nostro elaborato richiede di sviluppare un nuovo decoder per il player in esame. Il "*parsing error*" del Mux.exe, tuttavia, non permette di sviluppare esempi specifici per il testing del nuovo decoder. Il primo obiettivo era quello di capire che passi seguiva il flusso di informazioni nella rappresentazione di una immagine Jpeg interamente colorata in maniera uniforme, come può essere un rettangolo bianco. Purtroppo, il cattivo funzionamento dell'applicazione Mux, non permetteva di crearci esempi concreti per questa analisi. In seguito, abbiamo provato a linkare altri tipi di oggetti, come audio e video, ma il risultato è stato lo stesso. Di seguito riportiamo un esempio, non correttamente interpretato. In questo caso, l'obiettivo è il linkaggio di una immagine Jpeg rappresentante un fiore. Poiché è presente un linkaggio esterno, il file.txt deve far riferimento ad un particolare file, di estensione scr. Il file testuale è il seguente:

```
DEF N32 Group {
       children [
              Transform2D {
                     translation 0.0 0.0
```

```
                    children [
                          Shape {
                                geometry Bitmap {}
                                appearance Appearance {
                                      texture ImageTexture {
                                            url 12
                                      }
                                }
                          }
                    ]
              }
              Transform2D {
                    translation -100.0 -116.0
                    children [
                          Shape {
                                geometry DEF TextBar Text {
                                      string [ "This movie was created by TDK" ]
                                      fontStyle FontStyle {
                                            family [ "SansSerif" ]
                                            size 20.0
                                            style "BOLD"
                                      }
                                }
                                appearance Appearance {
                                      material Material2D {
                                            emissiveColor 0.0 1.0 0.0
                                      }
                                }
                          }
                    ]
              }
        ]
}

UPDATE OD [
        {
              objectDescriptorID  12
              muxScript av.scr
        }
]
```

La texture in esame è *"ImageTexture"*, proprio perché il file esterno è un'immagine. All'interno viene data un url di riferimento denotata dal numero 12. Il corrispondente `objectDescriptorID` denota la prossima azione da eseguire: vi è un riferimento al file av.scr. Il codice riportato all'interno è:

```
InitialObjectDescriptor {
        ObjectDescriptorID  1
        ODProfileLevelIndication 1
        sceneProfileLevelIndication 1
        audioProfileLevelIndication 2
        visualProfileLevelIndication 1
        graphicsProfileLevelIndication 1
        esDescr [
              ES_Descriptor {
                    ES_ID 101
                    muxInfo MuxInfo {
                          fileName av.od
                          streamFormat BIFS
                    }
                    decConfigDescr DecoderConfigDescriptor {
                          streamType 1        // OD Stream
                          bufferSizeDB 500
                    }
                    slConfigDescr SLConfigDescriptor {
                          useAccessUnitStartFlag TRUE
```

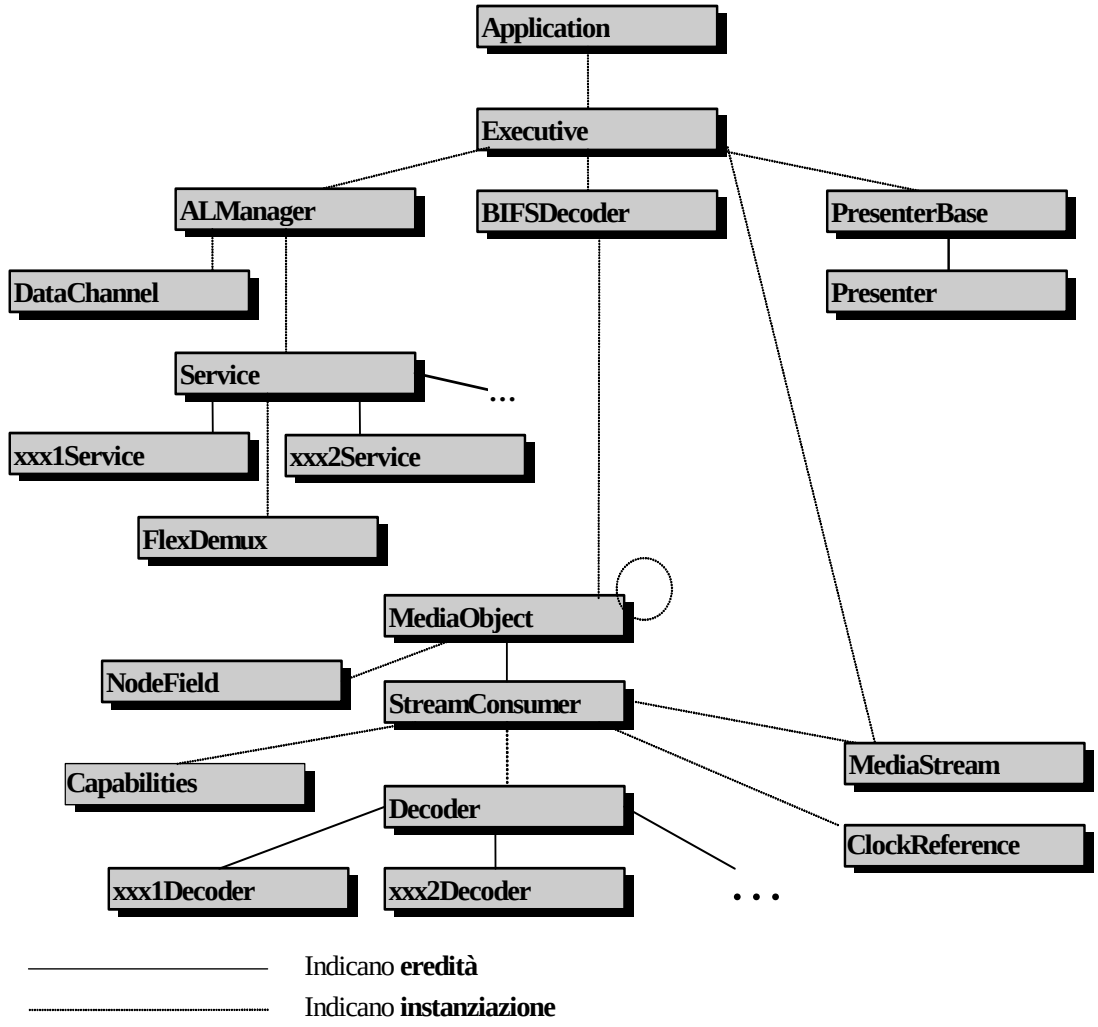
```
                        useAccessUnitEndFlag TRUE
                        useTimeStampsFlag TRUE
                        timeStampResolution 1000
                        timeStampLength 14
                }
        }
        ES_Descriptor {
                ES_ID 201
                OCR_ES_Id 101
                muxInfo MuxInfo {
                        fileName av.bif
                        streamFormat BIFS
                }
                decConfigDescr DecoderConfigDescriptor {
                        streamType 3        // BIFS Stream
                        objectTypeIndication      2      // this is a version 2 stream
                        bufferSizeDB 2000
                }
                slConfigDescr SLConfigDescriptor {
                        useAccessUnitStartFlag TRUE
                        useAccessUnitEndFlag TRUE
                        useTimeStampsFlag TRUE
                        timeStampResolution 100
                        timeStampLength 14
                }
        }
    ]
}


{
    ObjectDescriptorID 12
    esDescr {
           ES_ID 2115
           OCR_ES_Id 2113
           muxInfo
           {      fileName fiore.jpg
          streamFormat JPEG
           }
           decConfigDescr {
                  streamType 4        // VisualStream
                  objectTypeIndication      0x6C   // JPEG
                  bufferSizeDB 16000
           }
           slConfigDescr {
                  predefined 4
                  useAccessUnitEndFlag 1
           }
    }
}
```

L' `ObjectDescriptorID` è il numero 12. All'interno di `esDescr` vi è un riferimento tramite la parola chiave *muxInfo*, al cui interno vi è il nome del file da linkare e il formato del particolare stream. Se andiamo a *muxare* il file.bif, otteniamo un errore di *parsing*. Abbiamo anche provato a *muxare* singolarmente il file.scr. In questo modo, viene creato il file.trif, che, aperto con il player Im1, non viene correttamente interpretato: viene visualizzata solo la parte descrittiva riportata direttamente nel codice testuale, come colore dello sfondo o eventuale presenza di scritte sul display, ma non viene visualizzata la Jpeg allegata.

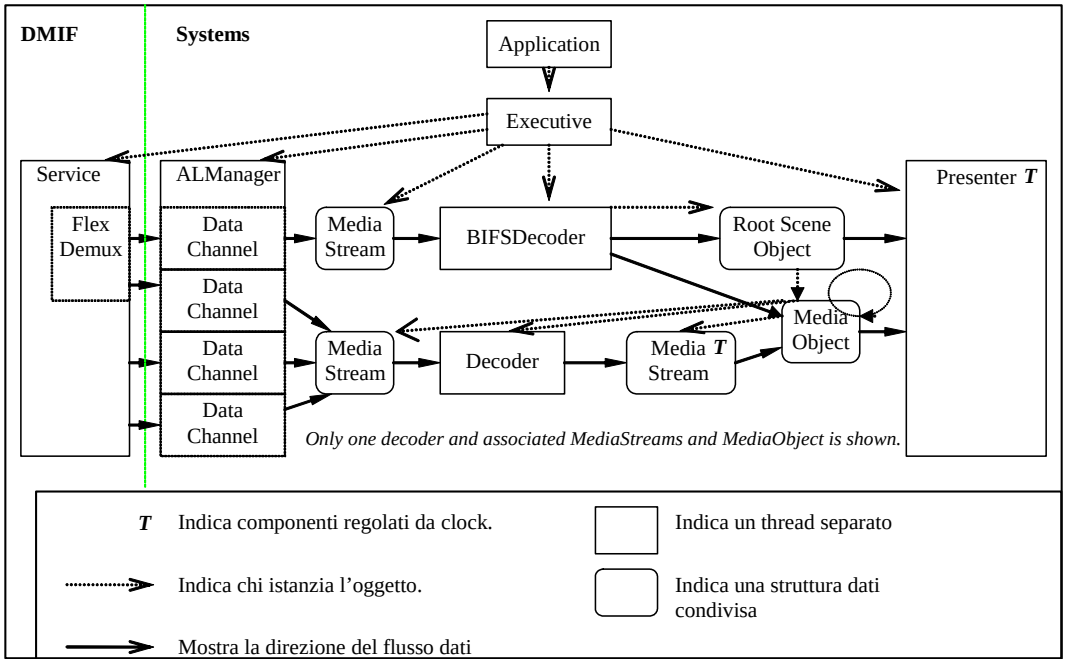
Conclusioni: Questo reference riesce a visualizzare solo le cose più semplici.

Aggiunta di un nuovo decoder:

Per capire esattamente come aggiungere un nuovo decoder nel Reference Software, dobbiamo fornire una descrizione dell'organizzazione delle classi, seppur in maniera generale, ovvero senza focalizzare eccessivamente l-attenzione sui vari metodi. Il diagramma delle classi è riportato in figura:



Prima di addentrarci nella discussione, riportiamo anche il **flusso delle informazioni,** che può essere molto utile per comprendere particolari aspetti della trattazione che verrà riportata di seguito:



Si nota immediatamente come la classe principale sia **Application.** Viene instanziata nel momento in cui viene lanciata l'applicazione e ogni sua classe derivata deve prevedere le seguenti proprietà:

1. Crea l'interfaccia grafica;
2. Instanzia l'oggetto Executive;
3. Acquisisce il nome di un file locale, oppure un indirizzo IP per una sessione remota;
4. Da l'avvio ad una nuova sessione Executive;
5. Passa dei messaggi di controllo ad Executive;
6. Chiude la sessione;
7. Elimina Executive;
8. Termina la sua esecuzione stessa.

Service:

Implementa il layer DMIF. Sviluppa generici servizi di controllo e funge da classe madre per tutte quelle che manipolano specifici protocolli. Interessante è enumerare le varie funzioni che si possono incontrare:

1. **ALManager ::** *Connect()* instanzia un oggetto **xxxService** e chiama **Service ::** *Connect()*;
2. **Service ::** *Connect()* è una funzione virtuale implementata dalle classi derivate. Crea uno o più thread e da inizio alla procedura di stabilimento del servizio;

3. I canali di input sono richiesti dalle chiamate a *Service :: OpenChannels()*, un'altra funzione virtuale implementata nelle classi derivate;
4. I thread creati durante la chiamata a *Connect()* completano la procedura di stabilimento del servizio, creano il canale, maneggiano i pacchetti dati e realizzano l'abbattimento di questo servizio;
5. I vari "eventi" sono riportati alla classe madre oppure a **ALManager** attraverso la funzione *OnXXX()*.

ALManager:

Implementa il cosiddetto "adaptation layer" dell'IM1. Riceve stream elementari di pacchetti da un **Service,** analizza le varie intestazioni e passa l'unità di accesso all'appropriato componente attraverso i **Mediastream.**

DataChannel:

Implementa l'adaptation layer dell'IM1. Per esempio,riceve flussi elementari di pacchetti, analizza le intestazioni dei cosiddetti AU e divide nei rispettivi **MediaStream** l'informazione contenuta nell'intestazione e quella contenuta nei dati veri e propri. Lo stabilimento di un DataChannel include quattro passi:

1. L'applicazione riceve informazioni su uno stream elementare, con inclusi i parametri di configurazione, in un **object descriptor.** Si chiama, poi, **ALManager** :: *AttachStream()*. Questa funzione instanzia un proprio oggetto DataChannel, lo lega al rispettivo Mediastream, setta i parametri di configurazione, e salva un puntatore all'oggetto in una tabella interna;
2. Quando l'applicazione desidera ottenere dati da uno o più DataChannel, chiama **ALManager** :: *RequestChannels()*. Questa funzione ha bisogno di **Service** per aprire i canali che sono stati richiesti ma non ancora aperti.
3. Quando l'apertura dei canali è stata confermata, allora Service chiama *ALManager:: OnChannelOpen()*, che ritorna un puntatore al puntatore associato all'oggetto DataChannel;
4. Quando l'applicazione desidera ricevere dei dati sul canale, chiama *ChannelReady()*.

Flex Demux:

Implementa, come dice il nome del resto, il Flex Demux layer. Riceve dei pacchetti FlexMux, li demultiplexa in stream elementari e li passa all'oggetto DataChannel appropriato.

Decoder:

Questo è l'oggetto base per tutti i decoder specifici. Ogni decoder esegue un thread. Nello schema riportato nel flusso di informazioni, è situato tra due MediaStreams, rispettivamente quelli di entrata e quelli di uscita. Ma quali operazioni esegue nello specifico un decoder? Sono le seguenti:

1. Il decoder ottiene un AU dallo stream di input. Se non ci sono dati disponibili, il thread del decoder è sospeso finché dei dati non saranno presenti in input;
2. Il decoder chiama una funzione virtuale pura, *Decode()*. Verrà poi ridefinita nelle varie classi derivate che implementeranno decoder specifici;
3. L'uscita di *Decode()* è memorizzata nel MediaStream di uscita. Questa operazioni include l'attacco di un pacchetto di presentazione assieme all'informazione vera e propria.

Questo non è altro che il comportamento di default del Decoder. Le classi derivate possono utilizzare la funzione *Run()* nel caso in cui si voglia implementare un diverso flusso dati. Il nostro scopo, dopo aver presentato in parte le funzionalità dell'IM1, è appunto riuscire ad aggiungere un nuovo Decoder NEL Reference Software. È un'azione possibile, anche senza toccare i file sorgenti già presenti. Per aggiungere un nuovo Decoder, la procedura è esattamente questa (anche se in seguito la vedremo più nello specifico):

1. Derivare una classe da Decoder ed implementare le relative funzioni virtuali;
2. Includere nell'implementazione una linea al di fuori di qualunque corpo di qualsiasi funzione. La linea deve avere, necessariamente, questo formato:

```
static DecoderFactory <your-class-name> any-name (streamType, "specificInfo");
```

Quando i valori di streamType e specificInfo sono uguali a quelli nei campi con i medesimi nomi del **DecoderConfigDescriptor,** allora conosciamo esattamente quale decoder dobbiamo utilizzare.

PresenterBase:

Ha un collegamento diretto con **Presenter,** che controlla la presentazione della scena. Esegue le seguenti funzioni:

1. Excutive instanzia Presenter;
2. Nel momento in cui BIFSDecoder finisce di costruire la scena, la funzione *Init()* di Presenter viene chiamata. Questa funzione deve chiamare *PresenterBase::Init()*, che da il via al thread del Presenter.
3. Il thread del Presenter gira in un loop, chiamando, ogni tot millisecondi, la funzione *Render()* della radice della scena;
4. Ogni MediaObject viene realizzato, così come ogni suo nodo figlio;
5. Alla fine viene chiamata **Presenter**::*Terminate()*. Questa cancella le varie informazioni che oramai sono inutili e chiama *PresenterBase::Terminate()*, che termina il thread del Presenter. Segue una chiamata a *Terminate()*, e l'Excutive elimina la scena.

ClockReference:

Un MediaStream consiste in delle unità a cui è associato un Clock. Il clock è specifico per ogni MediaStream. Il suo comportamento è regolato dalle classi ClockReference. Un oggetto ClockReference è assegnato ad ogni MediaStream, il quale maneggia il flusso di informazioni, dello stream, per l'appunto. Lo stesso ClockReference è assegnato a due MediaStream che costituiscono lo stesso cammino per lo stream. Un esempio può essere costituito dallo stream prima di incontrare il decoder e da quello tra il decoder ed il presenter.

MediaStream:

Questo è l'oggetto responsabile del buffering e dello stream dei dati. Consiste di un meccanismo Buffer con politica FIFO. Incorpora anche un meccanismo che controlla il tempo. Il trasporto dei dati su un MediaStream non è semplice: si compone di diverse fasi.

1. Prima di memorizzare alcun dato all'interno del buffer, viene chiamata la funzione *Allocate()*, per allocare lo spazio necessario per la memorizzazione delle informazioni;
2. Nel caso in cui lo spazio non fosse sufficiente, l'oggetto chiama la funzione *Reallocate()*, per espandere l'eventuale spazio di buffer precedentemente allocato;
3. A questo punto, la grandezza del buffer allocato viene passata come parametro alla funzione *Dispatched()*, che libera lo spazio in eccesso, "attacca" il clock a quello stream e rende libera l'unità di accesso al buffer per eventuali seguenti stream;
4. La funzione chiama *Fetch()* oppure *FetchNext()*. Ritornano un puntatore alla prossima unità di accesso libera e la bloccano finché non è chiamata la funzione *Release()*.
5. Dopo aver preso i contenuti dell'unità, l'oggetto chiama la funzione *Release()*.

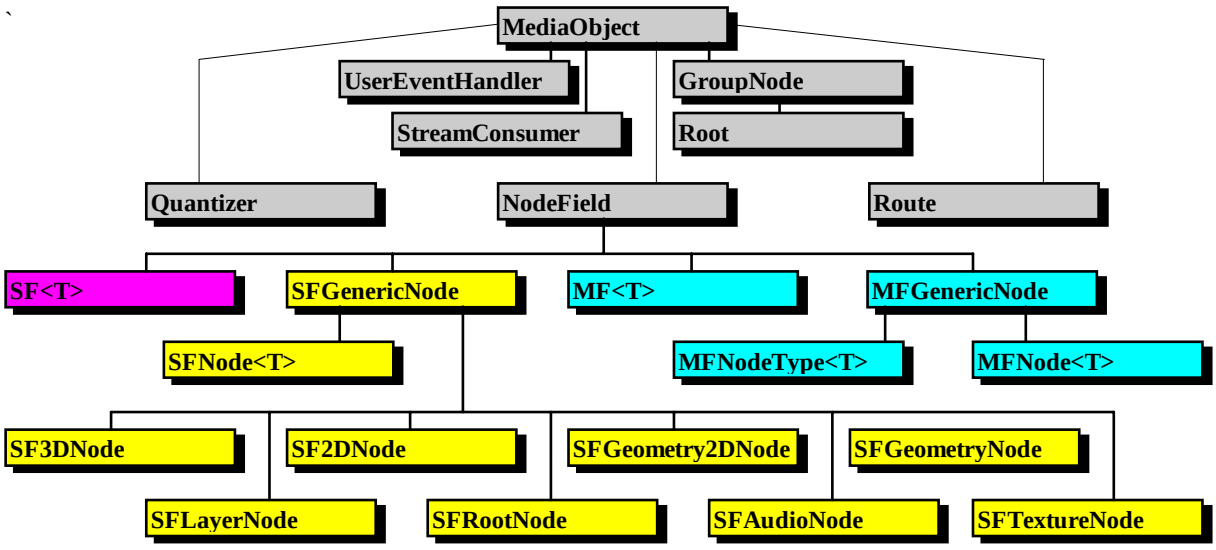
Capabilities:

Contiene informazioni riguardo la capacità del Decoder, del Presenter o di ogni altro specifico nodo.

Scene Classes: Media Object

È la classe madre di tutti gli altri nodi, di tutte le altre classi derivate, di tutti gli eventi e di tutti i cammini. Uno schema rappresentativo può essere quello della figura seguente:

`



Il MediaObject è la classe basse di tutti i nodi definiti dal BIFS. Come si può notare, un MediaObject è organizzato con una gerarchia ad albero con le seguenti proprietà:

- Ha zero o più nodi di "field", ognuno definito come oggetto di una classe derivante da **NodeField**.
- Un MediaObject avere più figli a loro volta MediaObject. Tutti i figli ereditano gli attributi dal MediaObject padre.
- Un MediaObject può "renderizzare" sia se stesso che tutti i propri figli. Per far ciò, è di estrema utilità la funzione *Render()*.
- Ogni MediaObject deve includere le proprie **macro BIFS**.
- Ad ogni MediaObject può essere attaccato un MediaStream.

Una volta enumerate queste caratteristiche, vediamo di analizzarle più attentamente. Abbiamo accennato che ogni MediaObject ha più nodi di "field". In particolare, molti field sono utilizzati per descrivere il tipo di dati, come SFxxx oppure MFxxx. Questi **field type** sono implementati come derivazioni da **NodeField**. I NodeField contengono anche informazioni sul particolare Bifs utilizzato. Nella figura appare anche una classe denominata **GroupNode,** importantissima per tutti i nodi che includano dei figli. La classe **Root** riguarda i nodi che fungono da radice. I nodi che "consumano" un media stream, come può essere quando contengono un campo *url*, derivano tutti dalla classe **StreamConsumer**. Interessante è, poi, la presenza della classe **UserEventHandler**. Riguarda i nodi che hanno bisogno di essere informati di eventuali azioni dell'utente, come, per esempio, un movimento o un click del mouse. Questo tipo di nodi devono derivare da questa classe.
La classe **Route** implementa i cosiddetti Routes, ovvero oggetti che linkano un NodeField ad un altro oppure un NodeField ad una funzione "event-handler", nel caso in cui il campo di arrivo sia un *EventOut.* Rimane il **Quantizer:** non è altro che un oggetto utilizzato per il campo *quantizzazione.* Come notiamo anche dalla figura riportata sopra, il campo più importante e su cui vale la pena focalizzare l'attenzione è il **NodeField.**

NodeField:

Questa classe può realizzare le seguenti azioni:

- Può memorizzare e restituire il valore di un campo di un nodo;
- Compararsi con uno stream BIFS;
- Tracciare un cammino nel momento in cui il valore è stato aggiornato;
- Copiare il valore di un altro NodeField, come risultato di un cammino.

Si nota che vi è un collegamento diretto tra la classe NodeField e la classe SF<T>. Questo è un "template" per tutti i campi SF. T può essere le classi **int**, **float**, **Bool, Color**, **Time**, **Rotation**, **ZString**, **Vec2f** e **Vec3f** che sono definite in ZTL. Si nota, inoltre, che la classe **SFGenericNode** mette a disposizione metodi pubblici per tutti quei nodi che si caratterizzano per tipo di dati specifici, quali **2D, 3D, Layer, Root, Geometry, Geometry2D, Texture, Audio.**
Un'altra classe interessante è la MF<T>, dove T può essere **int**, **float**, **bool** o addirittura può essere sinonimo di ulteriori classi quali **Color**, **Time**, **Rotation**, **ZString**, **Vec2f** e **Vec3f.** La definizione di T è, quindi, analoga a quella data per le classi SF. Anche in questo caso è presente una classe generica, denominata **MFGenericNode.**

Dopo questa breve presentazione delle varie classi dell'IM1, ovvero del nostro Reference Software, possiamo esporre in maniera dettagliata come si possa aggiungere un nuovo decoder alla struttura

preesistente senza dover modificare alcun file sorgente. Nella parte seguente, quindi, si inizierà ad analizzare il codice sorgente di un decoder già inserito nel progetto per capirne struttura e funzionamento, in modo da essere successivamente in grado di aggiungerne altri personalmente. I cosiddetti "media decoders" sono implementati tramite la derivazione della classe **Decoder.** Non è altro che un'interfaccia per un oggetto che riceve unità di accesso decodificate dal *buffer di decoding*, le decodifica e le separa per mandarle nel *composition buffer*. Questi due buffer sono gestiti dalla classe MediaStream. Il primo concetto che dobbiamo esporre in maniera esauriente è sicuramente il **DecoderImp**. Estende l'interfaccia di Decoder ed è utilizzato come classe base per tutti i decoder. Ogni decoder ha un suo thread specifico ed è, generalmente, limitato da due oggetti MediaStream, ovvero lo stream di input e lo stream di output. Tipicamente, quindi, un decoder lavora in questo modo:

1. Un oggetto che deriva da DecoderImp è instanziato in un modulo specifico di decoder. Se prendiamo come riferimento il decoder H263, andando ad osservare l'implementazione, troviamo questo frammento di codice:

```
/*------------------------------------------------------------
The exported function that instantiates the proper decoder
------------------------------------------------------------*/
void *CreateInstance ()
{
        return new H263Decoder;
}
```

Mentre nell'header troviamo:

```
#include "DecoderImp.h"
#include "DecInstW.h"
#include "declib.h"

class H263Decoder : public DecoderImp
{…… };
```

2. L'applicazione, inoltre, chiama *DecoderImp::Setup()* per settare i parametri del decoder ed inizializzarlo. Nell'implementazione infatti troviamo:

```
BOOL H263Decoder :: Setup (const BYTE *pSpecificInfo, int nSpecificInfoLength, int nESNumber)
{
        /*------------------------------------------------------------
        Check if this is QCIF or CIF. QCIF is the default.
        ------------------------------------------------------------*/
        BOOL bCIF = FALSE;
        LPBYTE pSeparator = (LPBYTE) memchr (pSpecificInfo, ';', nSpecificInfoLength);
        if (pSeparator && !memicmp (pSeparator + 1, "cif", 3))
                bCIF = TRUE;

        m_biDst.biSize = sizeof (BITMAPINFOHEADER);
        m_biDst.biWidth = 176 * (1 + bCIF);
        m_biDst.biHeight = 144 * (1 + bCIF);
```

```
        m_biDst.biPlanes = 1; //always
        m_biDst.biBitCount = 24;
        m_biDst.biCompression = BI_RGB; //RGB_FORMAT
        m_biDst.biSizeImage =
                m_biDst.biWidth * m_biDst.biHeight * m_biDst.biBitCount / 8;
        m_biSrc.biSize = sizeof (BITMAPINFOHEADER);
        m_biSrc.biWidth = m_biDst.biWidth;
        m_biSrc.biHeight = m_biDst.biHeight;
        m_biSrc.biPlanes = 1;
        m_biSrc.biBitCount = 24;
        m_biSrc.biCompression = FOURCC_H263;
        m_decodeObject.m_DecodeInst.DecFrameBuffer = NULL;
        m_decodeObject.m_DecodeInst.VDODecBuffer = NULL;
        m_decodeObject.m_RGBLow = NULL;
        m_decodeObject.m_InputFourcc = 0;
        m_decodeObject.m_OutputFourcc = 0;
        m_decodeObject.m_DoLowRes = 0;
        m_decodeObject.m_HurryUp = FALSE;
        m_decodeObject.m_pCompressedData = (LPBYTE) &m_biSrc;
        return TRUE;
}
```

Si nota come la funzione *setup* sia stata ridefinita nell'implementazione dello specifico Decoder. Ha il compito di inizializzare il Decoder. L'argomento *pSpecifiInfo* punta ai dati passati (sottoforma di array) dal campo *specificInfo* del **DecoderConfigDescriptor** (nel Bifs) dell'associato **ObjectDescriptor.** Il contenuto del campo dipende dallo specifico Decoder. Questa funzione è, solitamente, chiamata una sola volta dopo l'installazione dell'oggetto Decoder, tranne nel caso in cui il decoder in questione riceva input multipli da più stream elementari. In questo caso la funzione è chiamata tante volte quanti sono gli stream elementari. L'argomento *nSpecificInfoLength*, invece, specifica la lunghezza di questo array. L' *nESNumber*, infine, rappresenta il valore del campo *ESNumber* assegnato allo stream elementare nell' **ObjectDescriptor.** La funzione ritorna il valore TRUE nel caso in cui termini con successo. Nel caso in cui si abbia FALSE, allora l'applicazione cancellerebbe l'oggetto associato al Decoder e non eseguirebbe lo stream elementare corrispondente.

3. L'applicazione chiama *DecoderImp::SetFormat()* per comunicare quale formato abbiano i vari stream elementari, come possono essere un audio PCM e un video YUV. Nell'implementazione del Decoder H263 troviamo:

```
/*-------------------------------------------------------------
Find a compatible format and initialize the decoder
-------------------------------------------------------------*/
BOOL H263Decoder :: SetFormat (Capabilities &formatOptions,
DecoderParams &decoderParams)
{
        decoderParams.m_nOptimalAUSize = 256; //PAB_core50RC1+:
        decoderParams.m_nDBSize = 10000; //PAB_core50RC1+:
        decoderParams.m_nCBSize = GetOptimalOutputSize () * 5; //PAB_core50RC1+:
        decoderParams.m_bRequiresFullAUs = TRUE; //PAB_core50RC1+:
```

```
    /*-------------------------------------------------------------
        If the preferred format is YUV, set the precise format descriptor
        in the BITMAPINFOHEADER structure
        -------------------------------------------------------------*/
        BOOL bResult = FALSE;
        int i;
        for (i = 0; !bResult; i++)
        {       if (i >= formatOptions.GetFormatCapabilityLength ())
                        return FALSE;
                switch (formatOptions.GetFormatCapability (i).m_code) {
                case MediaFormat :: YUV:
                        m_biDst.biCompression =
                                formatOptions.GetFormatCapability (i).m_dwAdditionalInfo;
                        bResult = TRUE;
                        break;
                case MediaFormat :: RGB:
                        bResult = TRUE;
                        break;
                }
        }
        decoderParams.m_nStreamNumber = 0;
        decoderParams.m_format = formatOptions.GetFormatCapability (--i);
        if (decoderParams.m_format.m_code == MediaFormat :: RGB)
                decoderParams.m_format.m_code=MediaFormat :: BGR;

//PAB_core50RC1-:  decoderParams.m_nOptimalAUSize = 256;
//PAB_core50RC1-:  decoderParams.m_nDBSize = 10000;
//PAB_core50RC1-:  decoderParams.m_nCBSize = GetOptimalOutputSize () * 5;
//PAB_core50RC1-:  decoderParams.m_bRequiresFullAUs = TRUE;

        if(!InitDecodedObject (&m_biSrc, &m_biDst, &m_decodeObject))
                return FALSE;
        return TRUE;
}
```

Altra specifica della funzione è quella di raccogliere informazioni utili per il setup dei due buffer. Di solito, questa funzione è chiamata una sola volta dopo l'instanzazione dell'oggetto decoder. L'eccezione si ha nel momento in cui un decoder produce due stream, ognuno attaccato ad un nodo differente della scena. Si notano due argomenti in ingresso alla funzione. *FormatOptions* è un riferimento ad un oggetto di tipo **Capabilities**, passato dall'applicazione al decoder. Questo oggetto contiene una o più opzioni dei vari formati delle unità di composizione aspettati dall'applicazione, in ordine di preferenza. Per esempio, l'applicazione potrebbe avere le capacità di interpretare oggetti video sia nel formato RGB che nel formato YUV, ma si potrebbe riscontrare una efficienza maggiore in un tipo di formato rispetto ad un altro. Il decoder determina se è in grado di interpretare il particolare formato, setta i vari parametri per riprodurlo e ritorna il formato selezionato nell'argomento *decoderParams.* Alla funzione infatti viene inoltrato un riferimento ad una struttura di tipo *decoderParams*, nella quale il decoder memorizza il formato selezionato e le varie informazioni utili per il due buffer. Come si nota anche dal frammento di codice riportato la struttura di un *decoderParams* è la seguente:

```
struct DecoderParams {
                        int m_nStreamNumber;
                        MediaFormat m_format;
                        int m_nOptimalAUSize;
                        int m_nDBSize;
                        int m_nCBSize;
                        bool m_bRequiresFullAUs;
                        bool m_bBackChannel;
            };
```

Vediamo di comprendere una per una le variabili che ne fanno parte:

- ***m_nStreamNumber:*** il decoder vi immagazzina un numero che identifica il tipo di output nel caso in cui ve ne sia una molteplicità. Questo valore sarà utilizzato nel momento in cui sarà chiamata la funzione *SetStream()* per identificare lo stream di output. Il valore di default è 0.
- ***m_format:*** si memorizza il formato selezionato nell'argomento *formatOptions.* L'implementazione di default memorizza l'opzione maggiormente preferita nella lista.
- ***m_nOptimalAUSize:*** si memorizza la grandezza, in bytes, del blocco di memoria che si assuma possa gestire più unità di accesso. Se posto a zero si vuole evidenziare il fatto che il blocco di memoria non può gestire ulteriori informazioni. Il valore di default è 256.
- ***m_nDBSize:*** si memorizza la grandezza che dovrebbe essere allocata per il buffer di decoding. L'implementazione di default ritorna 1024.
- ***m_nCBSize:*** si memorizza la grandezza che dovrebbe essere allocata per il composition buffer. Se viene posto uno 0, allora si ha un NULL decoder, ovvero non è necessaria alcuna azione di decoding e lo stream di ingresso può andare direttamente al "nodo scena". Il valore di default è 1024.
- ***m_bRequiresFullAUs:*** vi si memorizza FALSE nel caso in cui il decoder possa maneggiare arbitrariamente parti di unità di accesso.
- ***m_bBackChannel:*** normalmente è posto a FALSE.

La funzione ritorna un valore vero nel caso in cui termini con successo, mentre ritorna un valore falso nel caso in cui il decoder non è capace di riprodurre alcun formato dato nella lista *formatOptions.* In quest'ultimo caso, l'applicazione cancellerebbe l'oggetto associato al Decoder e non eseguirebbe lo stream elementare corrispondente. Nella definizione di ***m_nCBSize,*** viene utilizzata una funzione particolare, ovvero:

int GetOptimalOutputSize ()

Ritorna la grandezza che dovrebbe essere allocata per il composition buffer. Il valore di default è 256. Il codice è:

```
/*------------------------------------------------------------------
Calculate the output size in bytes
------------------------------------------------------------------*/
int H263Decoder :: GetOptimalOutputSize ()
{
        return m_biDst.biWidth * m_biDst.biHeight * 3 + 1;
}
```

Nel frammento di codice, si notano funzione che non sono state analizzate, ma lo faremo in seguito tra qualche paragrafo.

4. L'applicazione chiama *DecoderImp::SetInputStream()* e *Decoder::SetOutputStream()* per far interagire il decoder con i MediaStream di input ed output.

virtual void SetInputStream (int nESNumber, MediaStream *pStream)

Questa funzione lega l'oggetto al **MediaStream** di input. Non bisogna farne *l'overload* nell'implementazione del decoder specifico senza avere la certezza che lo stesso abbia la capacità di gestire stream di input multipli. L'implementazione di default memorizza il puntatore allo stream in un campo privato, che può essere recuperato tramite una chiamata a *GetInputStream().* Gli argomenti passati alla funzione sono l'*ES_Number* dell'associato al **ES_Descriptor** ed un puntatore all'oggetto **MediaStream.**

virtual void SetOutputStream (int nStreamNumber, MediaStream *pStream)

Questa funzione, invece, lega l'oggetto al **MediaStream**s di output. Non bisogna farne *l'overload* nell'implementazione del decoder specifico senza avere la certezza che lo stesso abbia la capacità di gestire stream di output multipli. L'implementazione di default memorizza il puntatore allo stream in un campo privato, che può essere recuperato tramite una chiamata a *GetOutputStream()*. Gli argomenti passati alla funzione sono l'*nStreamNumber*, ovvero il valore ritornato nella struttura DecoderParams quando è chiamata la funzione *SetFormat()* ed un puntatore all'oggetto **MediaStream.**

5. L'applicazione chiama *DecoderImp::Start()*, che genera il thread associato al decoder. È una funzione che non deve essere presente nell'implementazione del decoder.
6. Il thread del decoder ottiene un'unità di accesso dallo stream di input. Nel caso in cui non ve ne siano di disponibili, il thread si sospende finchè non i dati non siano presenti.
7. Viene chiamata la funzione *Decode()*. Nell'implementazione dell'H263Decoder:

```
BOOL H263Decoder :: Decode (LPBYTE pInput, int nInputLength, LPBYTE pOutput,
                int &rnOutputLength)
{
                int nError;

//              fwrite(pInput, nInputLength, 1, fIn);

                if (rnOutputLength < GetOptimalOutputSize ())
                { rnOutputLength = GetOptimalOutputSize ();
                  return FALSE;
                }

                m_decodeObject.m_pCompressedData = pInput;
                m_biSrc.biSizeImage = nInputLength;
                nError = DecodeFrame (
                  &m_biSrc,
                  &m_biDst,
                  tempOutput,
                  &m_decodeObject,
                  FALSE,
```

```
                TRUE,
                FALSE,
                1
              );
              rnOutputLength = nError == 0 ? GetOptimalOutputSize () : 0;

              if (rnOutputLength > 0)
                memcpy(pOutput, tempOutput, rnOutputLength );

/*            if (nError == 0)
              {
              //fwrite ("\0\0\x81", 1, 3, fOut);
              //fwrite (pOutput, 1, rnOutputLength , fOut);
              }*/
              return TRUE;
}
```

Le unità in uscita da questa funzione dovrebbero essere composte da un'intestazione e da informazioni vere e proprie. Il formato dell'intestazione e l'offset delle informazioni variano a seconda dell'implementazione. Per accedere a queste informazioni, che non sono visibili dall'applicazione, si chiamano *GetDataOffset()* e *GetAttributes()*. Questa funzione accetta in ingresso l'indirizzo di una unità di input, la sua lunghezza, l'indirizzo dell'unità di output e la sua ipotetica grandezza. Se il valore di *rnOutputLength* è troppo piccolo, la funzione non terminerà con successo ritornando FALSE assieme alla specifica della grandezza minima di *rnOutputLength*. Ritorna TRUE altrimenti. Si intuisce come *pInput* punti ad una unità di accesso, *nInputLength* sia la grandezza dell'unità di accesso e *pOutput* punti all'unità di output.
Analizziamo le due funzioni speciali, introdotte poco prima. Introduciamo *GetDataOffset:*

```
/*-------------------------------------------------------------
In this decoder, the decoded data always strats at beginning of output
block
-------------------------------------------------------------*/
int H263Decoder :: GetDataOffset  (const LPBYTE)
{
              return 0;
}
```

viene chiamata dal compositore dopo aver sottoposto alla fase di "fetch" una unità del composition buffer. È noto che la lunghezza dell'intestazione è ignota al compositore, ma ha bisogno di analizzare l'unità e tramite questa funzione chiede al decoder dove inizino i dati veri e propri. L'implementazione di default è quella di assenza dell'intestazione e, quindi, la funzione ritorna uno 0.
Se consideriamo, invece, *GetAttributes(),* si ha il seguente frammento di codice:

```
/*-------------------------------------------------------------
In this decoder, attributes are fixed for all units
-------------------------------------------------------------*/
DWORD H263Decoder :: GetAttributes (ATTRIBUTE_CODE code, const LPBYTE)
```

```
{
        switch (code) {
        case FRAME_WIDTH:
          return m_biDst.biWidth;
        case FRAME_HEIGHT:
          return m_biDst.biHeight;
        case PITCH:
          return 3*m_biDst.biWidth;
        case OFFSETX:
          return 0;
        case OFFSETY:
          return 0;
        default:
          return 0;
        }
}
```

Dipende dal valore del code. Si nota come questa funzione riporti attributi dell'unità di composizione. ATTRIBUTE_CODE è definito in questa maniera:

```
typedef enum {
                FRAME_WIDTH,                    // la larghezza in pixel di un frame
                FRAME_HEIGHT,                   // l'altezza in pixel di un frame
                PITCH,                          // il tono dell'audio sample
                OFFSETX,
                OFFSETY,
                AUDIO_FREQUENCY,                // la frequenza dell'audio sample
                BITS_PER_SAMPLE,                // numero di bit per audio sample
                NUM_CHANNELS,                   // numero di canali audio
                FRAME_PIXMAP_WIDTH,
                FRAME_PIXMAP_HEIGHT,
                FRAME_BITS_PER_PIXEL,           // bit per pixel in un frame
                FRAME_PIXEL_FORMAT,             // il formato dei pixel in un frame
                AUDIO_CU_DURATION,              // la durata in ms di un audio sample
                IS_ALPHA
        } ATTRIBUTE_CODE;
```

8. L'uscita di Decode() è memorizzata nel MediaStream di output. Questa operazione include il settaggio di un clock per l'unità in uscita.

Gli ultimi tre passi sono ripetuti fino a quando *DecoderImp::Stop()* non viene chiamata dall'applicazione. Non abbiamo ancora presentato la funzione, forse, più importante dell'intera implementazione, ovvero *Run()*. Implementa il flusso dati del decoder. Questa funzione ha un'implementazione di default, ma deve essere implementata necessariamente nel caso in cui Decode() non sia ridefinita. Nel codice dell'H263, si trova:

```
void H263Decoder :: Run ()
{
        LPBYTE pInData, pOutData;
        int nInSize, nOutSize;
        DWORD dwTime;
        BOOL bEOU;
```

```
        while (  (pInData =
                (LPBYTE) GetInputStream () -> FetchNext (nInSize, dwTime, bEOU, TRUE)
                        ) != END_OF_STREAM )
                {
                if (IsActive ())
                {nOutSize = GetOptimalOutputSize (),
                pOutData = (LPBYTE) GetOutputStream () -> Allocate (nOutSize, TRUE, FALSE);
                if (pOutData == NULL)
                throw;
                if (pOutData == END_OF_STREAM)
                return;
                while (!Decode (pInData, nInSize, pOutData, nOutSize))
                        { pOutData = (LPBYTE) GetOutputStream () -> Reallocate (nOutSize);
                                        if (pOutData == NULL)
                                                throw;
                                        if (pOutData == END_OF_STREAM)
                                                return;
                          }
                          GetOutputStream () -> Dispatch (nOutSize, dwTime);
                          }
                          GetInputStream () -> Release (TRUE);
                        }
}
```

La funzione termina nel momento in cui lo stream di input ritorna END_OF_STREAM durante la chiamata a *Fetch()*. La funzione potrebbe essere ridefinita, anche nel caso in cui è stata definita anche Decode(), se il flusso di *Fetch()*/ *Decode()*/ *Dispatch()* non sia adeguato. La caratteristica principale è un loop che termina quando si chiude l'input stream. In questo ciclo chiama, immediatamente, *MediaStream::Allocate()* per allocare memoria. Quando si presenta la necessità di riallocare memoria, si chiama la funzione *MediaStream::Reallocate()*. Le chiamate a *MediaStream::Fetch()* o *MediaStream::FetchNext()* servono, invece, per recuperare dati dallo stream di input. Quando non si ha più bisogno di questi dati si ha *MediaStream::Release()*, mentre, se *Fetch()* o *FetchNext()* ritornano END_OF_STREAM si chiama *MediaStream::Close()* sullo stream di output e si stoppa il loop. Dopo aver completato di decodificare l'unità di composizione, viene chiamata *MediaStream::Dispatch()* e, conseguentemente, *MediaStream::Terminate()* sullo stream di input, nel caso in cui *Dispatch()* abbia dato come risultato END_OF_STREAM.
Notiamo che, in tutti questi metodi, sono state utilizzate funzioni protette:

- **MediaStream *GetInputStream ():** ritorna un puntatore allo stream di input;
- **MediaStream *GetOutputStream ():**ritorna un puntatore allo stream di output;
- **Root *GetRoot ():** ritorna la radice della scena.

CCDecoder:

CCDecoder è il nome del nostro progetto, che esporta come file di output "CCdec.dll", ovvero un file che permette al MPEG player di interpretare i file in input in un determinato modo. È stato chiesto di realizzare un decoder che rendesse a video un frame rettangolare di un determinato colore non curante dell'ingresso fornito al player. Il file .h, denominato "CCdec.h", è riportato di seguito:

```
#ifndef __CCDEC_H__
#define __CCDEC_H__

#include "defs.h"
#include "DecoderImp.h"

class CCdec : public DecoderImp
{

public:

// CCdec() : DecoderImp () {}

virtual int GetOptimalOutputSize ();

virtual BOOL SetFormat (Capabilities &formatOptions, DecoderParams &decoderParams);

virtual BOOL Decode (LPBYTE pInput,int nInputLength,LPBYTE pOutput,int &rnOutputLength);

virtual bool Quadratodecode (LPBYTE pOutput);

DWORD GetAttributes (ATTRIBUTE_CODE code, const LPBYTE);

};

#endif
```

Come è semplice notare, il nostro decoder fa uso di funzioni semplici e strettamente necessarie alla visualizzazione di un frame colorato su display. Ovviamente sono possibili varie migliorie tramite alcune aggiunte di codice, ma andiamo passo per passo. Abbiamo, quindi, utilizzato la funzione GetOptimalOutputSize, per ottenere la grandezza ottimale di uscita per regolare, di conseguenza, il buffer di uscita; SetFormat, per settare i vari parametri ed opzioni dei due buffer utilizzati da ogni decoder, ovvero quello in ingresso con l'informazione da decodificare e quello in uscita con l'informazione già codificata; Decode, funzione che restituisce un valore booleano per l'avvenuta decodifica corretta dell'informazione in ingresso puntata da pInput; QuadratoDecode, per decodificare nel modo richiesto, ovvero per visualizzare frame rettangolari di ogni colore; infine GetAttributes, fondamentale per settare le coordinate sul display.
Per un'analisi più attenta e specifica, andiamo ad analizzare il file .cpp, denominato "CCdec.cpp", dove è possibile capire in maniera concreta cosa settano e su cosa operano i metodi precedentemente nominati. Il file .cpp è riportato di seguito:

```
#include "CCdec.h"


void *CreateInstance ()
{
                        return (void *) new CCdec;
}

BOOL CCdec :: SetFormat (Capabilities &formatOptions,DecoderParams &decoderParams)
{
                        decoderParams.m_nDBSize = 10000;
                        decoderParams.m_nOptimalAUSize = 0;
                        decoderParams.m_nCBSize = 1000;
                        decoderParams.m_bRequiresFullAUs = TRUE;

                        if (formatOptions.GetFormatCapabilityLength () == 0)
                          return FALSE;
                        decoderParams.m_nStreamNumber = 0;
                        decoderParams.m_format = formatOptions.GetFormatCapability (0);
                        return TRUE;
}


int CCdec :: GetOptimalOutputSize ()
{
                        return 300*200*3;
}


bool CCdec :: Quadratodecode (LPBYTE pOutput)
{
                         BYTE temp[200*300*3];
                        for (int i = 0; i < 200*300*3; i+=3)
                          {
                                  temp[i]=255;
                                  temp[i+1]=0;
                                  temp[i+2]=0;
                          }
                        memcpy (pOutput, temp, 200*300*3);
                        return TRUE;
}


BOOL CCdec :: Decode (LPBYTE pInput, int nInputLength, LPBYTE pOutput, int
&rnOutputLength)
{
                        rnOutputLength = GetOptimalOutputSize ();
                        bool quadrato = Quadratodecode(pOutput);
                        return quadrato;
}
```

```
DWORD CCdec :: GetAttributes (ATTRIBUTE_CODE code, const LPBYTE)
{
                        switch (code) {
                        case FRAME_WIDTH:
                          return 300;
                        case FRAME_HEIGHT:
                          return 200;
                        case PITCH:
                          return 3*300;
                        case OFFSETX:
                          return 0;
                        case OFFSETY:
                          return 0;
                        default:
                          return 0;
                        }
}
```

Spieghiamo, uno per uno, i metodi utilizzati:

1. **CreateInstance:** è la funzione che instanzia fisicamente il nuovo decoder;

2. **SetFormat:** Ha in ingresso due argomenti. *FormatOptions* è un riferimento ad un oggetto di tipo **Capabilities**, passato dall'applicazione al decoder. Il decoder determina se è in grado di interpretare il particolare formato, setta i vari parametri per riprodurlo e ritorna il formato selezionato nell'argomento *decoderParams.* Alla funzione infatti viene inoltrato un riferimento ad una struttura di tipo *decoderParams*, nella quale il decoder memorizza il formato selezionato e le varie informazioni utili per il due buffer. Come si può leggere nel .cpp abbiamo settato i parametri in maniera particolare. Sappiamo che nell' ***m_nStreamNumber*** il decoder vi immagazzina un numero che identifica il tipo di output nel caso in cui ve ne sia una molteplicità. Non è il nostro caso e, quindi, lo settiamo con il valore di default è 0. In ***m_format***, invece, si memorizza il formato selezionato nell'argomento *formatOptions.* In. ***m_nOptimalAUSize*** si memorizza la grandezza, in bytes, del blocco di memoria che si assuma possa gestire più unità di accesso. La abbiamo settata a zero poichè si vuole evidenziare il fatto che il blocco di memoria non deve gestire ulteriori informazioni. In ***m_nDBSize*** si memorizza la grandezza che dovrebbe essere allocata per il buffer di decoding. In ***m_nCBSize*** si memorizza la grandezza che dovrebbe essere allocata per il composition buffer. Se viene posto uno 0, allora si ha un NULL decoder, ovvero non è necessaria alcuna azione di decoding e lo stream di ingresso può andare direttamente al "nodo scena". Inizialmente avevamo pensato di settarlo in questo modo, ma abbiamo visto, in debug, che in questo modo non avevamo allocazione dinamica del buffer di uscita, provocando il crash del sistema. In ***m_bRequiresFullAUs*** vi si memorizza FALSE nel caso in cui il decoder possa maneggiare arbitrariamente parti di unità di accesso, ma non era una specifica del decoder richiesto. Questa funzione ritorna falso nel caso in cui il decoder non è capace di riprodurre alcun formato dato nella lista *formatOptions.* Questa considerazione spiega la presenza dell'if.

3. **GetOptimalOutputSize:** Ritorna, in byte, la grandezza del nostro output. Abbiamo posto 200 * 300 * 3, in quanto lo scopo è di rappresentare a video un rettangolo di dimensioni 200 * 300, utilizzando lo spazio di colore RGB, che, come è noto, utilizza una sovrapposizione di tre strati di colore base quali il rosso, il verde ed il blu. Questa è la spiegazione del fattore 3 nella nostra funzione.

4. **QuadratoDecode:** è la funzione cardine della nostra implementazione. Restituisce un booleano, che ne nostro caso è sempre TRUE. Inizializza un vettore di BYTE di grandezza 300 * 200 * 3, che diamo in pasto ad un ciclo for, il quale setta le varie componenti dello spazio di colore RGB. Nel caso specifico rappresentiamo a video un rettangolo rosso di dimensioni 300 * 200. Tramite la funzione **memcpy** copiamo sul buffer di uscita l'informazione racchiusa nel vettore temp.

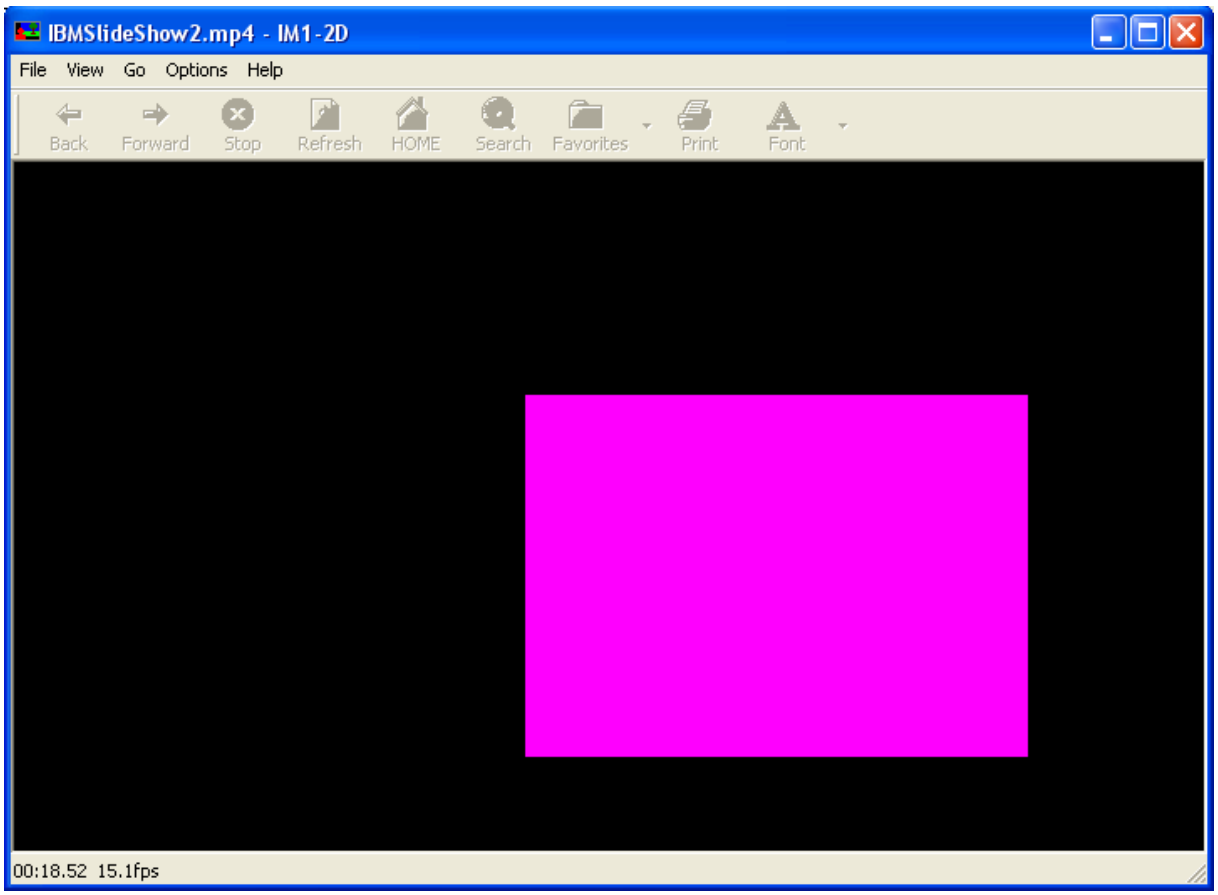
5. **Decode:** Questa funzione accetta in ingresso l'indirizzo di una unità di input, la sua lunghezza, l'indirizzo dell'unità di output e la sua ipotetica grandezza. Non corriamo il tischio che il valore di *rnOutputLength* sia troppo piccolo, perché la poniamo uguale alla grandezza in uscita. Ritornerà, sicuramente TRUE. Si intuisce come *pInput* punti ad una unità di accesso, *nInputLength* sia la grandezza dell'unità di accesso e *pOutput* punti all'unità di output.

6. **GetAttributes:** Settiamo le caratteristiche del frame da visualizzare sul display. Abbiamo settato `FRAME_WIDTH`, ovvero la larghezza in pixel di un frame, a 300 pixel; mentre `FRAME_HEIGHT`, ovvero l'altezza in pixel di un frame, a 200 pixel; ed, infine, `OFFSETX & OFFSETY`, ovvero lo spostamento, rispetto ad un punto di riferimento, sulle ordinate ed ascisse.

Questi sono stati i metodi utilizzati per creare il nostro decoder. Il risultato ottenuto è stato un file .dll da porre nel file di registro dell'IM1 in maniera che il player potesse leggerla. Abbiamo, però, incontrato un problema: una volta settato il file di registro, il player dava un errore, ovvero "tipo di decoder sconosciuto". Effettivamente andando a visualizzare la dll, si è scoperto che il file non creava alcuna istanza. Mancava il file.def presente in tutti gli altri decoder. Riportiamo il CCdec.def:

```
LIBRARY     CCDEC

DESCRIPTION 'DLL for testing object'

HEAPSIZE    4096

EXPORTS

                    CreateInstance  @1
```

Una volta creato questo file, non abbiamo incontrato alcun tipo di problema. Il risultato ottenuto sul display è:



In questo specifico caso, abbiamo variato i vari offset, ponendo OFFSETX e OFFSETY, rispettivamente, a 100 e 50. Abbiamo anche cambiato il colore settando le tre componenti, rispettivamente, a 255, a 0 ed a 250.

Miglioramenti possibili:

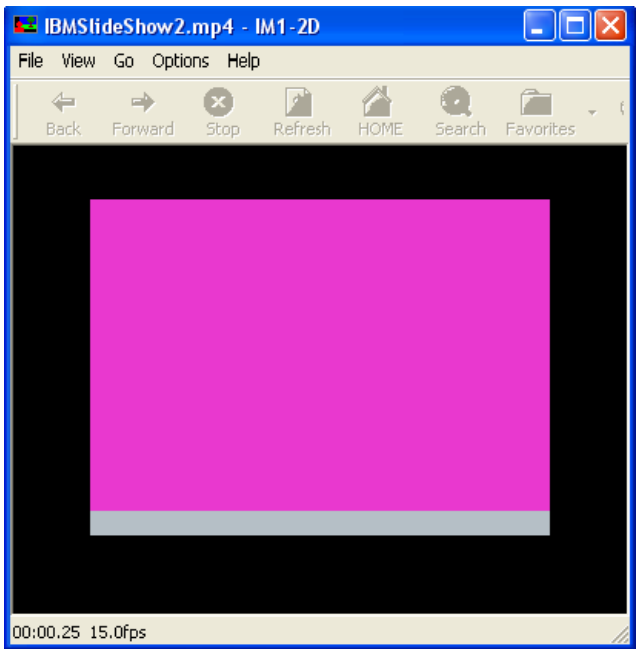
La prima cosa che salta all'occhio è l'assenza di un distruttore. Visto che allochiamo ogni volta un vettore di 18000 byte, è preferibile definire un distruttore che azzeri l'allocazione di memoria avvenuta durante la decodifica. Un altro aspetto da approfondire è l'interazione con l'input. Questo decoder, infatti, non ha alcuna interazione con l'input (testimonianza di ciò è che la funzione QuadratoDecode non ha come argomento il puntatore all'informazione in ingresso ma solo quello all'output). Si potrebbe, per esempio, ideare un decoder che cambi il colore del frame a seconda del byte in ingresso. Ovviamente si potrebbe fare in modo che non cambi aspetto ogni volta che il byte successivo sia diverso da quello precedente, poiché si avrebbe un risultato inguardabile. Una soluzione fattibile è che cambi colore ogni volta che una sequenza di tre byte sia diversa da un'altra sequenza di byte passata presa come riferimento. Un'osservazione importante è che, in questa esperienza, si è verificato un ulteriore problema, anche se, in realtà, è facilmente intuibile: dando in ingresso file derivanti da informazioni del .bif, queste vengono visualizzate sul display, in quanto vengono interpretate da un decoder a se stante, ovvero il BifsDecoder.

Interazione con l'ingresso:

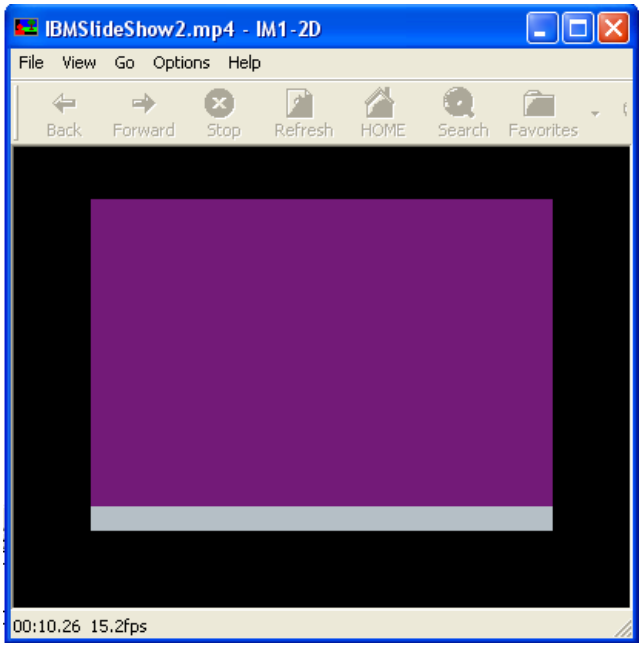
Abbiamo migliorato il nostro decoder, in maniera tale che reagisca a cambiamenti nello stream di input. Abbiamo preso una sequenza di immagini Jpeg, le abbiamo date in pasto al CCdecoder, ma applicando delle modifiche all'implementazione della funzione cardine QuadratoDecode. Il metodo è stato, così, modificato:

```
bool CCdec :: Quadratodecode (LPBYTE pInput/*,int nInputLength*/, LPBYTE pOutput)
{
                        BYTE temp[300*200*3],rgb[4];

                        memcpy(rgb, pInput+1000, 4);

                         //Modifica i parametri RGB in base allo stream in ingresso

                          for (int i = 0; i < 300*200*3; i+=3)                    //set RGB color

                                {
                                        temp[i] = rgb[0];                          //R

                                        temp[i+1] = rgb[1];                        //G

                                        temp[i+2] = rgb[2];                        //B
                                }

                                memcpy (pOutput, temp, 300*200*3);

   return TRUE;
}
```

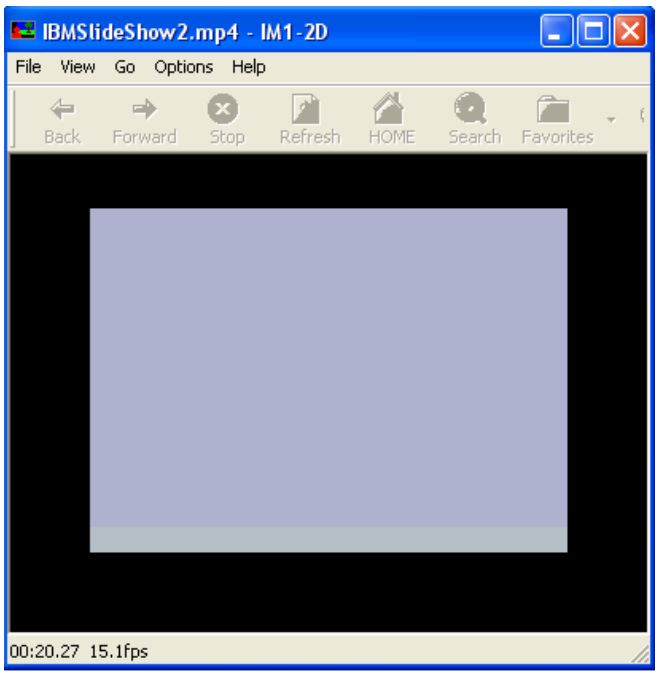
È facile notare la modifica operata. Abbiamo instanziato un ulteriore vettore di byte, di lunghezza 4. Abbiamo copiato il contenuto presente a partire dall'indirizzo ***pInput+1000*** in ***rgb*** per 4 byte. Il +1000 è giustificato dal fatto che le varie Jpeg hanno un'intestazione simile, pressoché identica, e non si sarebbero notati i cambiamenti sul display. Infatti, la nostra applicazione chiama questa funzione ogni volta che deve decodificare un'immagine: è molto probabile che le sequenza di byte presenti in ***pInput+1000*** siano diverse al variare delle immagini Jpeg. Continuando con l'analisi dell'implementazione, è possibile notare come siano variate le assegnazioni all'interno del ciclo ***for***: abbiamo posto le tre componenti uguale ai byte che si trovavano nelle prime tre posizioni del vettore ***rgb***. Infine, abbiamo copiato il tutto nel buffer di uscita ogni volta che le immagini Jpeg venivano decodificate, ottenendo un risultato soddisfacente, ovvero la variazione del colore ogni volta che una Jpeg doveva essere interpretata. Riportiamo, di seguito, cosa è apparso sul display, una volta dato in ingresso una sequenza di 4 immagini Jpeg:



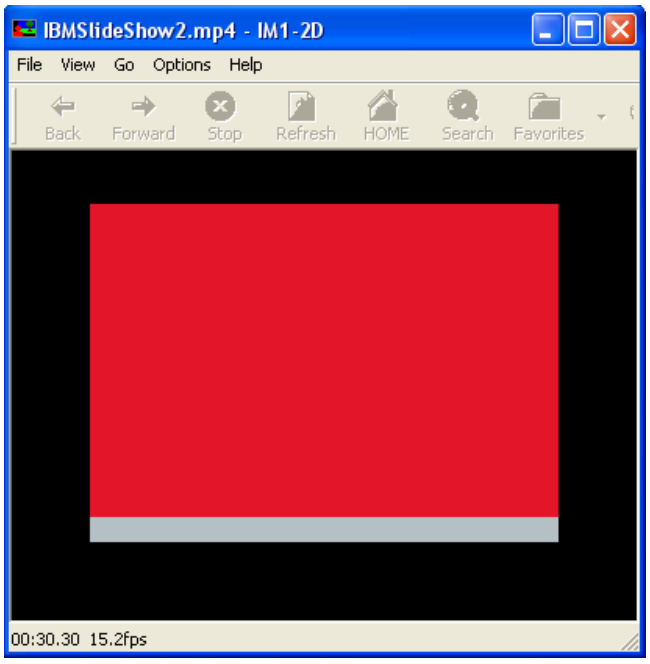
Come si nota, anche grazie al contatore di secondi, questo colore è dovuto a una sequenza di byte presente nella prima immagine Jpeg. Le immagini si succedono con un intervallo di 10 secondi, quindi, avremo il cambiamento del colore sul display proprio dopo 10 secondi, ovvero nel momento in cui si va a richiamare la funzione QuadratoDecode:



Dopo 10 secondi, infatti, il colore è cambiato. Per scrupolo di informazione mostriamo anche il colore che si ha dopo 20 secondi:



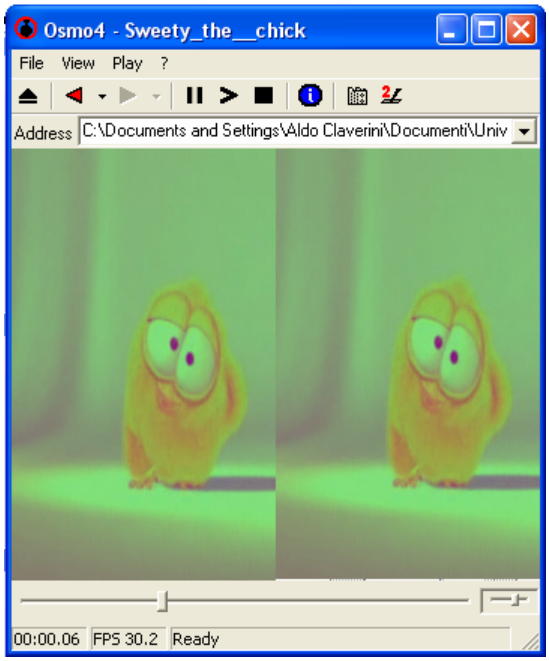
E quello che si ha dopo 30 secondi, ovvero l'ultimo cambiamento di colore:



L'unico problema incontrato, come è anche visibile dalla sequenza di immagini, è l'interpretazione da parte del BifsDecoder delle informazioni di tipo bif. In questa specifica occasione è la striscia grigia in fondo al frame, che nella sequenza video originaria era utilizzata per dare maggior risalto alle immagini Jpeg.

Confronto tra Im1 ed Osmo4:

Abbiamo testato vari tipi di file dopo aver compilato l'MPEG player Im1. Le informazioni contenute in semplici file .bif sono correttamente interpretate dal player preso in esame, mentre qualsiasi tipo di audio non è processato in maniera adeguata. Per i file video dobbiamo fare delle distinzione: quelli composti da sequenze di immagini Jpeg, anche con interazione dell'utente (come può essere il click del mouse sul display), non presentano alcuna sorta di problemi; tutti gli altri mandano in crash il programma. Questo è il motivo per il quale siamo passati ad analizzare il software Osmo4. Durante la compilazione abbiamo incontrato non pochi errori. Il primo di questi è stato causato dalla mancanza di una particolare applicazione chiamata "nasm.exe". È risolto effettuando il download di particolari libreria che, una volta compilate ed eseguite, producevano il file mancante. Altri problemi simili sono stati la mancanza di determinati file header e source, a cui abbiamo sopperito tramite ricerche accurate nella rete. È bastato copiare tali file scaricati nelle cartelle dove il compilatore ne segnalava l'assenza. Tuttavia, il problema maggiormente complesso è stato incontrato durante la compilazione delle librerie FFmpeg. Tali librerie sono state concepite per essere compilate in ambiente Linux, mentre il nostro ambiente di lavoro è Windows. Quindi, per poter procedere, abbiamo dovuto utilizzare un simulatore di ambiente Linux: è stato realizza tramite la combinazione di MinGW ed MSYS. La compilazione di queste librerie è stata molto complessa ed ha richiesto molto tempo. Alla fine, tuttavia, il software MinGW ha creato i file.lib e quelli .dll (dell'FFmpeg) senza segnalare alcuni errori. Una volta risolti tutti questi problemi esposti precedentemente, oltre a numerosi problemi di linkaggio di varia natura, il compilare ha terminato la propria esecuzione creando il file "Osmo4.exe".
Il software ottenuto ha prestazioni sicuramente migliori rispetto all'Im1 player. Infatti, riesce come ad aprire tutte le tipologie di file aperte dall'Im1, ma anche tipologie molto più complesse. Non mostra alcun problema nell'esecuzione audio dei file supportati. Per quel che riguarda i file video, Osmo4 apre varie tipologie: vanno dall' QuickTime video al video in formato DivX, passando per i formati Real Media, Windows Media Video, Advanced Streaming Format ed altri ancora. Riesce, inoltre, ad eseguire i file.bt.
Tuttavia, abbiamo riscontrato un problema nella visualizzazione dei vari video. Infatti, sul display, i video vengono duplicati ed eseguiti in maniera parallela, come rappresentato nella figura sottostante.



L'immagine mostra un ulteriore problema: i pixel vengono visualizzati con una sfumatura tendente al verde. Dopo aver riflettuto ed analizzato le varie componenti del software, siamo indotti a pensare che l'inconveniente sia dovuto alla non corretta compilazione delle librerie FFmpeg.