

**VIDEO**

**E**LABORATION

**T**RANSMISSION

**&**

**LIBRARY**

*version 1.0.2.25*  
*revision 16*

developers'  
manual

# Abstract

Il progetto VETLib si propone di creare una libreria di processing, elaborazione e trasmissione di segnali video digitali destinata all'implementazione di filtri, codificatori e decodificatori software.

La libreria è scritta in ANSI C++ ed è basata su un'architettura modulare tramite la dichiarazione di classi astratte standard (interfacce) estese con implementazioni specifiche e templates anche per applicazioni real-time.

La dipendenza da device, sistemi operativi e codifiche specifiche è ristretta a singoli moduli e trasparente a livello di applicazione secondo la logica OOP (Object Oriented Programming).

La Release corrente offre un'interfaccia base a Video4Linux ed a Microsoft DirectX (DirectShow) per l'acquisizione di dati; due interfacce di visualizzazione per ambienti Linux tramite le librerie QT/GTK (adatte per video, nei tests: massimo 33 fps) e una per ambienti Windows tramite le API di GDI (adatta a immagini statiche); inoltre sfrutta librerie esterne quali libmpeg3, xvidcore, quicktime4linux per la (de)codifica video e imageMagick per la (de)codifica di immagini (praticamente tutti i formati).

In seguito ad un progetto esterno è stato sviluppato anche un complesso modulo di acquisizione e preview live (tramite DirectX) per dispositivi ad alta fedeltà sia in formato non compresso (RAW o DV) che con codifica hardware MPEG2.

Gli strumenti forniti da VETLib consentono lo sviluppo di applicazioni in ambienti RAD (Rapid Application Development) e l'estensione dei componenti inclusi per soluzioni proprietarie (principalmente tramite l'ereditarietà), l'utilizzo pratico di ciascun oggetto è ampiamente dimostrato con i progetti di test per i casi più banali (ogni componente ha un rispettivo progetto che ne evidenzia le caratteristiche e l'utilizzo), inoltre il software VETLib WorkShop è un'applicazione completa molto strutturata basata interamente sulla libreria.

Lo sviluppo di nuovi moduli (basati su interfacce standard) è stato semplificato anche grazie allo strumento Package Studio (sviluppato in .NET C++), in grado di automatizzare la creazione di progetti personalizzati per i sistemi di sviluppo più diffusi (Microsoft Visual Studio, Borland C++ Builder, Make), il software è altamente strutturato ed elastico, si basa su una serie di moduli predefiniti e su files di configurazione in formato XML, quindi l'aggiornamento delle classi base e dei progetti predefiniti non implica la ricompilazione del programma.

Durante la fase finale del progetto è stata sviluppata un'applicazione (WorkShop) per sistemi Windows in Managed C++ (Framework .NET) in grado di testare i componenti esistenti e di gestire dinamicamente nuovi plugins tramite DLL standard.

WorkShop dimostra le potenzialità della libreria integrando in modo statico e dinamico tutti i moduli progettati per sistemi Windows, è possibile modificare in modo visuale i parametri di lavoro e costruire più catene di filtri.

# Contents

Abstract .....	IV
Acknowledgments .....	VIII
Summary .....	IX
VETLib FRAMEWORK .....	11
1.1 - Overview .....	11
1.2 - Modular Software Architecture .....	15
1.3 - Framework Design .....	16
1.4 - vetFrame .....	21
1.4.1 - vetFrameYUV420 .....	25
1.4.2 - vetFrameRGB24 .....	26
1.4.3 - vetFrameT<class T> .....	27
1.4.4 - vetFrameRGBA32 .....	29
1.4.5 - vetFrameGrey .....	30
1.4.6 - vetFrameRGB96 .....	30
1.4.7 - vetFrameHSV .....	31
1.4.8 - libETI support .....	31
1.5 - vetInput .....	32
1.6 - vetOutput .....	35
1.7 - vetFilter .....	36
1.8 - vetCodec .....	41
1.9 - vetVision .....	43
1.10 - vetBuffer .....	44
1.11 - vetObject .....	46
1.12 - vetException .....	46
1.13 - Directory structure .....	47
1.14 - Builts .....	49
1.15 - Documentation .....	50
1.16 - VETLib Online .....	51
VETLib IMPLEMENTATION .....	53
2.1 - Inputs .....	53
vetNoiseGenerator .....	53
vetPlainFrameGenerator .....	53
vetVideo4Linux .....	54
vetDirectXInput .....	54
vetDirectXInput2 .....	54
2.2 - Outputs .....	55
vetWindowQT .....	55
vetWindowGTK .....	55
vetWindow32 .....	55
vetDoctor .....	55
vetOutputVoid .....	55
2.3 - Codecs .....	56
vetCodec_BMP .....	56
vetCodec_IMG .....	56
vetCodec_MPEG .....	56

vetCodec_MOV .....	56
vetCodec_XVID.....	56
2.4 - Filters .....	57
vetFilterGeometric .....	57
vetFilterColor .....	57
vetMultiplexer .....	57
vetFilterNoiseChannel .....	57
2.5 - Buffers .....	58
vetBufferArray .....	58
vetBufferLink.....	58
2.6 - Other Modules .....	59
vetHist .....	59
vetThread .....	59
vetUtility .....	59
vetMatrix.....	59
vetDFMatrix.....	59
2.7 - Vision.....	60
vetMotionLame .....	60
Using and Extending VETLib.....	62
3.1 - Overview.....	62
3.2 - Samples.....	65
3.3 - Tools of the Trade.....	68
3.4 - VETLib Component Conventions .....	69
3.5 - Package Development .....	70
3.5.1 - Working with Frames .....	71
3.5.2 - Internal buffering.....	73
3.5.3 - Parameters for Filters and Codecs .....	76
3.5.4 - Platform specific.....	79
3.5.5 - Templates.....	80
3.5.6 - Threading.....	81
3.6 - VETLib Package Starter Kit.....	83
3.7 - Releasing VETLib .....	89
VETLib WorkShop & PlugIns.....	93
4.1 - Overview.....	93
4.2 - How It Works .....	95
4.3 - Dynamic PlugIn System .....	100
4.4 - WorkShop PlugIn Development.....	104
ReadMes.....	107
./README.....	107
./USE .....	109
./COMPILE.....	111
./FAQS .....	115
./EXTEND.....	116
./ChangeLog.....	118
./TODO .....	119
./BUGS.....	120
./AUTHORS.....	120
./lib/README .....	121
Headers.....	124

vetDefs.h .....	126
vetException.h.....	126
vetFrame.h.....	127
vetFrameRGB24.h .....	128
vetFrameYUV420.h.....	129
vetFrameT.h.....	130
vetFrameRGBA32.h .....	131
vetFrameRGB96.h .....	132
vetFrameHSV.h.....	133
vetFrameGrey.h.....	134
vetOutput.h.....	135
vetInput.h .....	136
vetFilter.h .....	137
vetBuffer.h .....	138
vetCodec.h.....	139
vetVision.h.....	140
vetObject.h.....	141
Sample Applications .....	143
License .....	147
Bibliography.....	152

# Acknowledgments

R O A D E C H U W A O R  
V I . E W G A T E I H V  
G T T N E F E G B N B L  
O S C A R L O Y K T C I  
U O P T F U M P H E S M  
U Y L A N C U Q E R O A  
G N U L B A T O R N U R  
S A S E R T E R C E R I  
T O P S U P A L A T C A  
Q D L I N U X J C K E N  
S R U G Q U 8 P I P P O  
I W S N A Z 6 G S A X P

Ringrazio quindi il mio relatore Prof. Francesco De Natale per l'attenzione e la costante disponibilità.

Ritengo inoltre necessario sottolineare la fondamentale importanza di Internet, la più grande fonte di informazioni del nostro secolo, grazie alla quale l'accesso e la distribuzione di conoscenza sono a portata di mouse.

Difficilmente avrei potuto portare a termine questo lavoro senza il contributo materiale e affettivo della mia famiglia e dei miei amici a cui dedico questa tesi.

# Summary

Questo documento vuole presentare il progetto VETLib agli utenti e soprattutto agli sviluppatori, segue una breve descrizione dei contenuti:

Il capitolo primo riassume le caratteristiche di VETLib, analizza la struttura della libreria in modo astratto e con esempi pratici, presenta le interfacce, gli oggetti principali e il sistema di interazione tra i componenti. L'ultima parte elenca la struttura delle cartelle e della documentazione integrata.

Il capitolo secondo descrive brevemente i moduli inclusi in questa release, mentre gli headers e una breve descrizione dei programmi dimostrativi sono disponibili in appendice.

Il capitolo terzo è focalizzato sull'utilizzo pratico e sull'estensione di VETLib, l'implementazione di filtri è analizzata nel dettaglio anche con esempi pratici, sono inclusi vari suggerimenti agli sviluppatori neofiti, la parte finale del capitolo presenta lo strumento VETLib Package Studio (utile per lo sviluppo di componenti) e i passi da seguire per rilasciare una nuova versione della libreria (anche con il software VETLib Distribution Manager).

Il capitolo quarto analizza il software VETLib Workshop e il sistema di gestione dei plugins (componenti dinamici), l'ultima sezione spiega i passi necessari per convertire un modulo VETLib in un componente di WorkShop (creazione di una DLL con Visual Studio), lo strumento Package Studio è in grado di generare il progetto e i principali files necessari automaticamente.

Le appendici includono i file readmes, gli headers delle classi base e una breve descrizione dei programmi dimostrativi più significativi.

La Licenza di distribuzione è la classica General Public License, riportata per intero dal testo ufficiale.

Il supporto multimediale allegato (CD o DVD) contiene l'ultima distribuzione rilasciata, la classificazione delle cartelle è riportata alla fine del capitolo primo, è consigliabile iniziare la consultazione dal sito web (directory `./Website` oppure online <http://Inx.ewgate.net/vetlib>).

Una versione aggiornata di questo documento è disponibile online all'indirizzo <http://Inx.ewgate.net/vetlib/distr/docs/pdf/VETLib-1.0.2-Manual.it.pdf>

# FRAME

W

*Tutti sanno che una cosa è impossibile da realizzare,  
finché uno sprovveduto che non lo sa la inventa.*

*Albert Einstein*

O

R

K





### 1.1 - Overview

Una libreria è l'unione di risorse, classi e metodi in un unico *object* molto simile all'output di un compilatore C++ generico (classicamente *.obj*), in ambienti Windows le librerie sono contraddistinte dall'estensione *.lib*, mentre in ambienti \*NIX lo standard prevede estensione *.a*.

Le librerie sono ampiamente diffuse nella programmazione odierna che è quasi totalmente orientata agli oggetti (*Object Oriented Programming*), quando una libreria offre un set di oggetti e strumenti dedicato allo sviluppo di software si parla di *Application Programming Interface*.

I vantaggi sono molteplici e variano dalla chiarezza nello stile del sorgente alla facilità di distribuire aggiornamenti, in VETLib si sfrutta soprattutto la possibilità intrinseca di interazione tra le estensioni e le applicazioni della libreria.

Ci si potrebbe chiedere perché creare una libreria di video processing, in effetti esistono poche librerie open source dedicate a questo campo, molte delle quali sono finalizzate a scopi relativamente ristretti e difficilmente estensibili verso altre frontiere.

L'elaborazione video, come molti altri contesti, coinvolge numerosi sotto-processi (spesso modulabili) e necessita di molti oggetti e metodi standard che spesso lo sviluppatore deve implementare una tantum (ad esempio: oggetti frame, funzioni matematiche, accesso ai dati e visualizzazione), questa implementazione preliminare è costosa (per complessità e tempo) e soprattutto propedeutica al “vero” cuore del sistema che si vuole sviluppare (sul quale ci si vorrebbe concentrare), il risultato è che spesso algoritmi e filtri efficienti devono essere analizzati, estratti dal contesto e modificati per adattarsi alle esigenze di ogni singola applicazione, ovviamente ciò è tremendamente inefficiente.

Il problema principale è il compromesso tra prestazione e flessibilità, ottimizzare il codice per uno scopo o una piattaforma (device) specifica paga moltissimo in prestazioni, ma limita l'applicazione a quel contesto e allunga i tempi di sviluppo, un'evidente esempio è la diffusione negli ultimi anni di software ludici sulla piattaforma Windows (32bit), queste applicazioni non interagiscono direttamente con i dispositivi grafici (*video card*), ma accedono ad un framework (*DirectX/OpenGL*) che fornisce un'interfaccia standard di accesso. Un altro esempio di quanto sia importante la portabilità anche a discapito delle prestazioni è la diffusione del linguaggio Java.

Il video processing mette in crisi le capacità hardware anche dei sistemi più costosi, non si tratta solo di gestire grandi volumi di dati in un tempo limitato ma di elaborare lo stream con operazioni matematiche più o meno complesse, che spesso devono essere approssimate per ottenere prestazioni accettabili, è evidente che in un contesto simile sia un algoritmo che un accesso dati migliore incrementa sensibilmente la performance del processo.

Inoltre se in questo complesso contesto si include anche la presentazione del video si aggiunge il problema della continuità temporale accettabile dall'utente, il tempo di elaborazione di uno o più frame, tipicamente variabile, deve essere normalizzato per garantire la fluidità delle immagini; inoltre e nel caso dello streaming "reale" (trasmissione su canali non ideali) attraverso reti comunemente utilizzate (IP) il sistema di trasmissione applica ritardi variabili anche ai sotto-blocchi dei frames (lo stream è diviso in pacchetti).

L'obiettivo che il progetto VETLib persegue è sviluppare un "laboratorio software" di sviluppo per filtri e (de)codificatori in grado di interagire col maggior numero di device e codifiche in modo trasparente ed astratto.

La mia prima analisi ha considerato quali strumenti e quali componenti una simile libreria dovrebbe fornire agli sviluppatori:

- ✓ Oggetti di base: frames, matrici, vettori;
- ✓ Sistema(i) per caricare, acquisire e salvare video e immagini;
- ✓ Sistema(i) per visualizzare video e immagini;
- ✓ Sistema(i) per trasmettere video e immagini tramite classiche reti informatiche;
- ✓ Funzioni matematiche (DCT, statistiche);
- ✓ Un set di filtri più comuni;
- ✓ Utilità varie (conversione di colore, ecc);

La definizione di una serie di interfacce standard consente l'interazione tra oggetti e moduli ma garantisce un alto fattore di personalizzazione (ereditarietà e composizione di classi), la versione corrente (1.0.2.25) non è ancora matura per una distribuzione ufficiale, esclusivamente per la scarsa presenza di filtri e supporti matematici necessari, il lavoro si è infatti focalizzato sulla standardizzazione degli oggetti (e la loro interazione) e sull'acquisizione/visualizzazione dei dati, lo sviluppo del progetto è stato analizzato accuratamente più volte per aggiornare l'ordine di priorità dei componenti, gradualmente nuovi moduli e metodi saranno integrati nella libreria e disponibili agli sviluppatori.

Il linguaggio di programmazione scelto è ANSI C++ (ISO/IEC 14882:2003), di cui si sfrutta pesantemente l'astrazione e la modulazione del codice tramite classi e templates che si accordano con l'architettura concettuale del framework, la semplicità con cui lo sviluppatore cliente può accedere ai metodi e alle strutture della libreria a livello di applicazione è indiscutibilmente maggiore rispetto alla soluzione in linguaggio C (meno modulare, maggiore complessità sintattica), inoltre i compilatori più recenti hanno assottigliato il divario prestazionale tra i due linguaggi.

Gli oggetti base e la maggior parte dei filtri sono completamente portabili poiché basati esclusivamente sulla Standard C++ Library, mentre i componenti legati a particolari hardware o librerie (acquisizione, visualizzazione, ecc) sono stati implementati in modo simmetrico per sistemi Windows e sistemi \*NIX (Linux, Unix).

In ambienti \*NIX (sviluppo su Linux 2.4.29 SlackWare 10.1) la libreria è compilata con il classico GNU C++ Compiler (GCC 3.3.4) ed è presente nella directory radice di VETLib il file di configurazione dell'utility Make (*Makefile*), le attuali funzionalità di VETLib sono:

- ✓ Acquisire da device compatibili con video4linux (usb cams, tv tuners, ..);
- ✓ Decodificare video in formato MPEG1-2, MPEG4 (XVID);
- ✓ (de)Codificare video in formato Quicktime (MOV);
- ✓ (de)Codificare quasi tutti i formati immagine esistenti;
- ✓ Visualizzare immagini e video in qualunque Desktop Environment (es. *KDE*, *GNOME*);

In ambienti Windows (sviluppo su Windows 2000 SP4), VETLib è costruita con il software Microsoft Visual C++ 6.0 (con Borland C++ Builder 6.0 non sono disponibili tutti i componenti rilasciati), i file di progetto sono situati in *./lib/mvc* (e *./lib/bcb/*), la libreria completa è in grado di:

- ✓ Acquisire immagini e video da device compatibili con DirectX (low fps);
- ✓ Acquisire e salvare video in formato non compresso da device compatibili con DirectX con preview in overlay (high fps) (anche IEEE1394 OHCI e device MPEG2);
- ✓ Caricare video tramite DirectX (qualunque codifica supportata dal sistema);
- ✓ Decodificare stream MPEG4 (XVID);
- ✓ (de)Codificare quasi tutti i formati immagine esistenti;
- ✓ Visualizzare immagini (GDI, librerie esterne: QT, GTK);
- ✓ VETLib WorkShop (VETLib front-end);
- ✓ VETLib Package Studio (Developers' Tool).

Le differenze sono dovute alla portabilità delle librerie esterne su cui sono basati alcuni moduli, le due applicazioni *VETLib WorkShop* e *VETLib Package Studio* sono sviluppate in Managed C++ .NET e quindi disponibili solo per ambienti Windows, il primo è uno strumento per il test, la verifica e la dimostrazione didattica di componenti (es. filtri), il secondo è un utilissimo software che automatizza la fase (iniziale) dello sviluppo di nuovi componenti, consente di personalizzare il progetto e si basa su moduli template e file di configurazione XML facilmente aggiornabili; non si prevede lo sviluppo degli analoghi softwares per Linux.

Entrambe le distribuzioni (\*NIX e Windows) sono suddivise in *special builds* ed una versione completa (*full*), è consigliato fare riferimento sempre alla versione completa ma questa classificazione può essere utile agli addetti ai lavori ed agli sviluppatori che intendono compilare VETLib sul proprio sistema, ad esempio si può evitare di compilare e gestire librerie esterne non coinvolte nel proprio progetto.

Gli strumenti e gli oggetti forniti da VETLib sono integrabili direttamente in applicazioni finali, ma lo sviluppatore può anche estenderli con implementazioni specifiche ed ottimizzate, le interfacce a contesti *OS / device dependent* sono ristrette a singoli moduli e quando possibile sono disponibili diverse implementazioni della medesimo modulo (es. *Threading*). Alcune possibili applicazioni sono:

- ✓ Strumenti e algoritmi di video e image processing a scopo didattico e commerciale;
- ✓ Applicazioni di video sorveglianza con la possibilità di creare moduli "leggeri" per software distribuiti (ad esempio su Linux Embedded su FPGA);
- ✓ Applicazioni inerenti la Computer Vision per ambienti domotici casalinghi e industriali.

Lo sviluppo della libreria in futuro dovrà concentrarsi sui seguenti aspetti:

- Interfaccia a *Video4Linux2*;
- Interfaccia a *DirectX 10* (attualmente non ancora rilasciato);
- Interfaccia diretta a dispositivi FireWire su linux (*libraw1394*, *libdv*, *libavc1394*);
- Interfaccia alla libreria *MPEG4IP*;
- Interfaccia per la codifica MPEG2-4 (*FFMPEG*);
- Implementare dei moduli driver di sorgente e rendering per *DirectX* e *VideoForWindow*;
- Sviluppo di un nuovo sistema di interfacce *vetInput2*, *vetOutput2*, *vetFilter2*, *vetProcess* in grado di gestire catene di moduli con collegamenti intelligenti (analogo a *DirectShow*), il nuovo sistema è analogo all'hardware, i moduli si interfacciano tramite una serie di *Pin* e i collegamenti (*Link*) minimizzano le operazioni di *color-space conversion* e di buffering;
- Implementare un bridge (eventualmente in *vetFilter2*) per condividere filtri con *DirectShow*;
- Ottimizzare il modulo *vetThread* ed integrarlo nei moduli esistenti;
- Implementare i filtri più comuni, relativi plugins ed applicazioni dimostrative;
- Implementare moduli dedicati alla trasmissione e ricezione di streams su reti;
- Implementazione di un sub-framework per contenuti multimediali su PDA (Java e .NET);
- Implementare moduli per il controllo/acquisizione dati di basso livello (seriale e usb);
- Implementazione di un sistema di *scripting* per il test e dimostrazioni didattiche;
- Implementazione di moduli per accesso/controllo da remoto (HTTP, TCP telnet like);
- Fornire un accesso alla libreria simile a Java e .NET, tramite la gerarchia e la composizione di *namespaces* ed una serie di metodi statici;
- Proseguire lo sviluppo dei software *Package Studio* e *Distribution Manager*.

## 1.2 - Modular Software Architecture

Una libreria modulare ed estensibile permette di inglobare standards e formati specifici, servizi di basso livello e librerie esterne in modo trasparente a livello di applicazione. I moduli intrinsecamente legati a fattori hardware o che interagiscono con servizi di basso livello possono avere più implementazioni specifiche (ad esempio rispetto al sistema operativo) che forniscono la stessa interfaccia garantendo una ottima portabilità.

Il modello è diviso in 3 livelli:

### Application Layer

Il livello di applicazione si interfaccia al layer inferiore che fornisce metodi e strutture standard e indipendenti dai livelli inferiori.

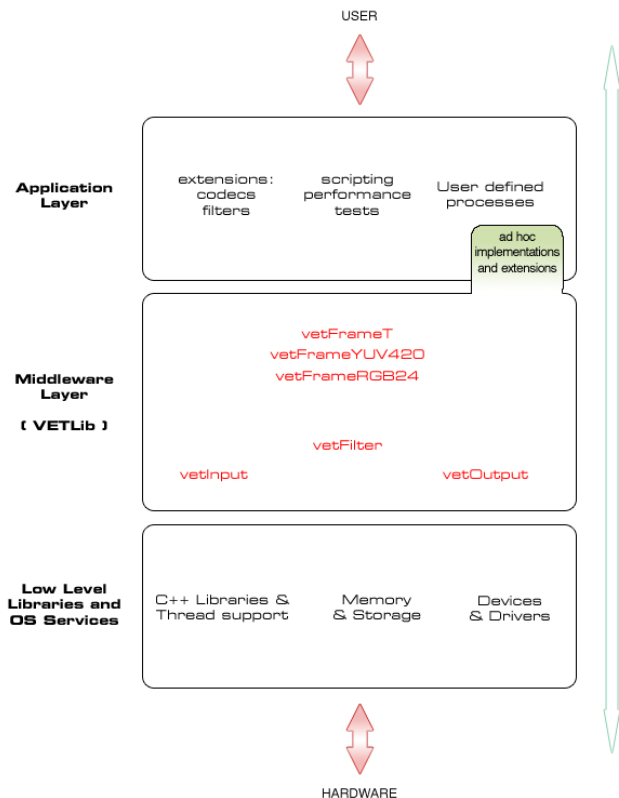
Lo sviluppatore può utilizzare supporti già esistenti o estendere gli oggetti con implementazioni specifiche tramite l'ereditarietà e la composizione di classi.

### Middleware Layer (VETLib)

Questo livello si occupa di rendere scalabili e portabili metodi e servizi specifici, di offrire strutture e funzioni comuni al livello superiore.

### Kernel Services

Il livello più basso è costituito dai servizi che offre il kernel del sistema operativo, le librerie (anche la C++ Standard Library ovviamente) e le interfacce (drivers) a device e standard specifici (codecs).



La libreria (non tutti i moduli) è compatibile con ambienti Windows e \*NIX, un applicazione sviluppata in ANSI C++ e che si basa solo sui moduli multi-piattaforma di VETLib, mantiene la portabilità del codice, ad esempio il modulo *vetThread* nasconde allo sviluppatore la complessità e la grande differenza nella gestione dei processi dei sistemi operativi Windows e \*NIX, l'implementazione di *vetThread* contenuta in *VETLib.a* (libreria statica per linux) è ovviamente diversa alle *\*.lib* (libreria statica per windows), ma fornisce gli stessi metodi e segue un eguale comportamento.

## 1.3 - Framework Design

Una libreria di video processing deve fornire un set di strumenti e oggetti generici e specifici per gli utilizzi più comuni, in modo puramente astratto possiamo sintetizzare un processo di elaborazione:



Questa analisi distingue 3 oggetti principali:

### Decoder

Il decodificatore si occupa di tradurre una codifica specifica (ex. MPEG) in uno standard riconosciuto da sistema di filtri, i dati possono essere caricati da uno stream live (VideoIn o Network) oppure da una fonte statica (memoria), in uscita si ha uno *stream on demand* (cioè è il modulo successivo o l'applicazione a richiedere i dati).

### Filtering Process

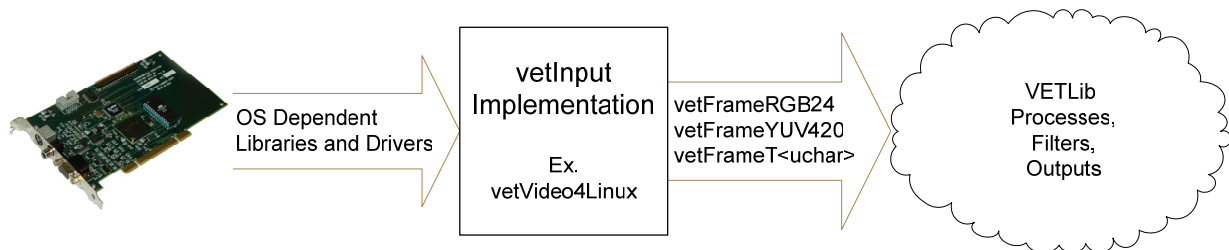
È il cuore del sistema, non conosce le caratteristiche dello stream originale in ingresso ed in uscita, ma interagisce con l'input (decoder) e l'output (encoder) tramite uno standard definito, analogamente gli altri due oggetti non conoscono il sistema di filtri.

### Encoder

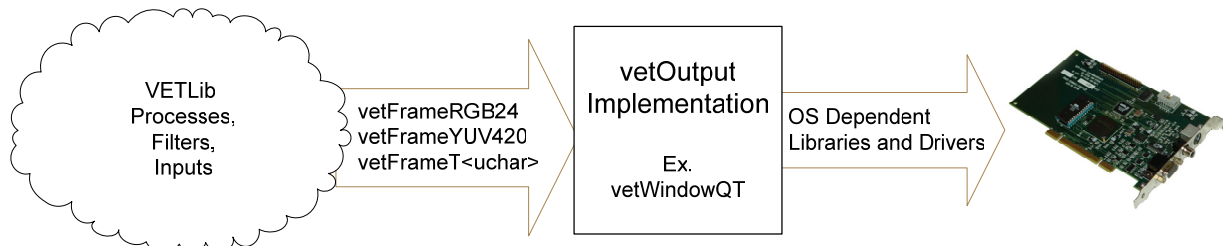
Il codificatore converte lo stream in uscita dal sistema di filtri in una codifica specifica, un successivo oggetto si occuperà di memorizzare i dati o inviarli ad un dispositivo (VideoOut, Network).

Questa visione astratta è realizzata in VETLib nelle due super classi principali: *vetInput* e *vetOutput*, sono classi astratte che definiscono l'interfaccia ad uno standard di comunicazione interno definito, questo standard è basato su tre classi: *vetFrameRGB24*, *vetFrameYUV420* ed il template *vetFrameT<unsigned char>*, le caratteristiche dei singoli oggetti saranno affrontate più avanti in questo capitolo, lo scopo di questa sezione è un'analisi di alto livello.

Segue uno schema del data-flow in ingresso:

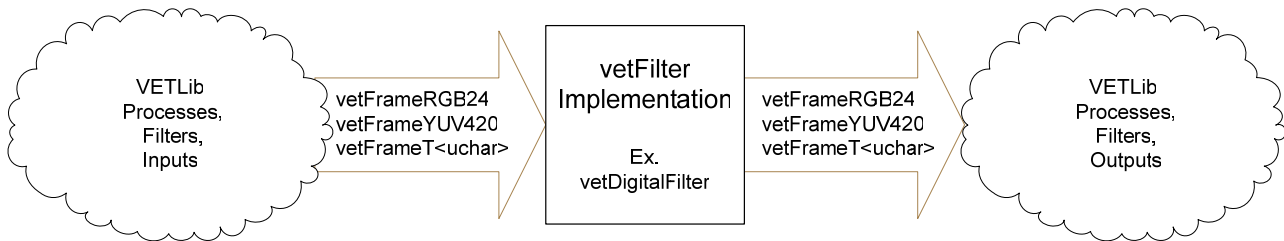


L'oggetto *vetInput* corrisponde al decodificatore astratto presentato nello schema precedente, in uscita ovviamente abbiamo un oggetto *vetOutput*:



In termini pratici l'oggetto *vetInput* è una sorgente di frames e definisce un'interfaccia che li fornisce, *vetOutput* definisce un'interfaccia di acquisizione di frames poiché è un'uscita dati.

La composizione astratta (e pratica) di questi due oggetti è un buffer: i frames vengono importati da una sorgente, memorizzati e indirizzati al successivo oggetto *vetOutput* in uscita; ma un buffer può elaborare i dati e “diventare” un filtro, *vetFilter* è l’oggetto che implementa entrambe le interfacce (tramite ereditarietà) e si pone al centro del nostro data-flow:



La classificazione input/output in VETLib può inizialmente destare confusione, come metodo pratico si possono distinguere facilmente osservando il data-flow dal punto di vista del filtro:

- In ingresso al filtro (oggetto che fornisce i frames) ci sarà un implementazione dell’interfaccia *vetInput* che definisce le funzioni *extractTo(vetFrame..)*.
- In uscita al filtro (oggetto che riceve i frames) ci sarà un implementazione di *vetOutput* che definisce le funzioni *importFrom(vetFrame..)*.

Ovviamente gli oggetti *vetFilter* possono comunicare con tutte le implementazioni di *vetInput/vetOutput* e quindi anche con altri oggetti *vetFilter*.

In prima analisi si può considerare una sorgente statica di un singolo frame, lo stream che parte da un implementazione di *vetInput* è di fatto un oggetto che rappresenta un’immagine in modo digitale. Il formato immagine (la definizione dell’oggetto frame) deve essere riconosciuto da tutti i componenti ed è uno standard in VETLib (*vetFrame..*), il caso di sequenze video è una banale estensione nel tempo. Lo stream non è prettamente dinamico, ad ogni chiamata delle funzioni di interfaccia (*extractTo/importFrom* oppure operatori di streaming) segue il caricamento di un singolo frame, il controllo del flusso dati è gestito dal livello applicazione o da moduli appositi (*vetProcess*) che possono ovviamente eseguire cicli di indirizzamento/estrazione dei frames.

Questa situazione classica e semplice può apparentemente complicarsi nel caso un filtro abbia la necessità di lavorare su *n* frames (motion detection, tracking ..), anche in questo caso non vi è alcuna differenza, il filtro sarà dotato di un multiplo buffer interno (basato sui buffer integrati nella libreria o implementazioni proprietarie) e ci sarà semplicemente un ritardo tra il primo frame in ingresso al filtro e il primo frame estratto (gap di *n-1* frames), è compito del filtro gestire eventuali richieste di estrazione (da un oggetto *vetOutput*) non valide (ad esempio non si è raggiunto il numero di frame minimi per il processing).

L’astrazione teorica è proiettata nel livello di programmazione tramite l’ereditarietà e la composizione di classi, l’implementazione di una delle interfacce base (*vetInput*, *vetOutput*, *vetFilter*, ..) prevede lo sviluppo di una classe che eredita (in modo pubblico) la classe astratta di riferimento, ricordo che una classe è astratta se almeno uno dei suoi metodi è puramente virtuale:

```
class myStorageInterface {
public:

    myInterface() {};
    virtual ~myInterface() {};

    virtual bool AddItem(void*) = 0;
    virtual int getItemsCount(void) = 0;
};
```

Questa classe è puramente astratta (*pure abstract*), chiaramente il compilatore non può istanziare una classe astratta poiché i metodi sono stati dichiarati ma non vi è alcuna implementazione, la seguente classe invece è un oggetto conforme a *myStorageInterface* e che può essere istanziato:

```
class myLameStorage : public myStorageInterface {
public:
    myObject() { i_c = 0; };
    ~myObject() { };

    bool AddItem(void* newItem)
    {
        if ( i_c ==> MAX_ITEMS )
            return false;

        data[i_c++] = newItem;
        return true;
    };
    int getItemCount(void) { return i_c; };

protected:
    int i_c;
    void* data[MAX_ITEMS]; //static
};
```

Questa strategia di programmazione è l'applicazione pratica dell'analisi astratta precedentemente illustrata, in questo modo pur non conoscendo il funzionamento interno e l'obiettivo di un oggetto, è possibile interagire tramite le proprietà che necessariamente deve implementare (visto che eredita una classe base). Ric conducendosi al precedente esempio:

```
class myDynamicArrayStorage : public myStorageInterface {
public:
    myObject() {
        i_c = 0;
        dataSize = 10;
        data = new void*[dataSize];
    };
    ~myObject() { delete data; };

    // [...]

    bool AddItem(void* newItem) {
        if ( i_c ==> dataSize ) {
            // [...] realloc array [current + REALLOC_STEP]
            data[i_c++] = newItem;
        }
        else
            data[i_c++] = newItem;

        return true;
    };
    int getItemCount(void) { return i_c; };

protected:
    int i_c;
    int dataSize;
    void** data; //dynamic
};
```



Le due implementazioni sono chiaramente differenti ma entrambe rispondono ai metodi dichiarati nella classe madre *myStorageInterface*, questa interfaccia comune garantisce il livello di astrazione tramite un classico casting:

```
int main()
{
    myStorageInterface* storage;
    // [...]

    int choice = 0;
    cout << "Select storage system {0, 1}: ";
    cin >> choice;
    if ( choice )
        storage = new myDynamicArrayStorage();
    else
        storage = new myLameArrayStorage();

    // [...]
    for (int i=0; i<100; i++)
    {
        int* newItem = new int;
        *newItem = i;
        storage->AddItem( static_cast<void*>(newItem) );
    }
    // [...]

    cout << "There are " << storage->getItemsCount() << " items.";

    // delete objects (...in this way they are lost, but it's a sample)
    delete storage;
    // [...]
};
```

In questo esempio la scelta della classe che memorizza gli oggetti è effettuata dall'utente ed è creata all'interno della stessa funzione, è il caso più banale visto che si conoscevano le due classi già durante la compilazione, in generale non è così e si ha la definizione della sola interfaccia che implementano.

Oltre all'ereditarietà possiamo sfruttare anche la composizione di classi:

```
class dataSource
{
public:
    virtual int getItem(Item* newItem) = 0;
}

class dataOutput
{
public:
    virtual int setItem(Item* newItem) = 0;
}

class dataProxy : public dataSouce, public dataOutput
{
public:
    int setItem(Item* newItem) { .. };
    int getItem(Item* newItem) { .. };
}
```

La classe *dataProxy* implementa entrambe le interfacce e se si considera *dataSource* come una sorgente di oggetti *Item* e *dataOutput* come una destinazione di oggetti (si può pensare due classi che rispettivamente leggono e scrivono da file), allora possiamo pensare all'oggetto *dataProxy* come un filtro o un buffer:

```
// [...] create or load instances..
dataSource* myInput;      // .. myInput is an implementation of dataSource
dataProxy* myFilter;     // .. myFilter is an implementation of dataProxy
dataOutput* myOutput;    // .. myOutput is an implementation of dataOutput

Item* currItem = new Item();      // the buffer object
int I = 0;

while (i++ < 100 && !myInput->EOF() )
{
    myInput->getItem(currItem);    // load the new object from source
    myFilter->setItem(currItem);   // push it to data proxy
    myFilter->getItem(currItem);  // extract last item
    myOutput->setItem(currItem);  // push to final output
}
```

L'oggetto *dataProxy* può essere un semplice buffer oppure può modificare l'oggetto, ma qualunque implementazione delle interfacce base consente l'interazione tramite oggetti *Item*.

L'analogia con VETLib è semplicissima: *dataSource* è paragonabile alla classe *vetInput*, *dataOutput* è analoga all'interfaccia *vetOutput* e *dataProxy* corrisponde a *vetFilter*. L'oggetto *Item* corrisponde agli oggetti che rappresentano un frame: *vetFrameRGB24*, *vetFrameYUV420* oppure *vetFrameT<unsigned char>*.

Prima di analizzare le interfacce base e i principali oggetti è opportuno puntualizzare alcune definizioni contenute nell'header *vetDefs.h* che è incluso in tutte le classi della libreria, l'operato di un metodo è spesso comunicato alla funzione chiamante tramite il valore di ritorno (classico *bool*, *int*, *HRESULT*), in VETLib questo stile è spesso imposto e sempre suggerito, il tipo standard è:

```
typedef int VETRESULT;
```

ed i valori di ritorno standard sono i seguenti, qualora necessario definire valori superiori a 8200:

```
#define VETRET_OK                0    /* no errors found */
#define VETRET_PARAM_ERR        1    /* illegal parameter(s) */
#define VETRET_INTERNAL_ERR     2    /* internal routine error */
#define VETRET_ILLEGAL_USE     4    /* illegal use of function */

#define VETRET_DEPRECATED_ERR   8    /* old version, removed? */
#define VETRET_OK_DEPRECATED   16    /* old version, ex.bad conversion */

#define VETRET_NOT_IMPLEMENTED 666  /* not supported/implemented */
```

Inoltre sono definite una serie di *MACRO* utili durante lo sviluppo (definire il *flag* per abilitarle):

```
#ifdef __VETLIB_DEBUGMODE__
    #include <stdio.h>

    #define INFO(x) printf("_INFO: %s\n");
    #define DEBUG(x) printf("_DBG: %s = %p \n", #x, x);
    #define DEBUGMSG(msg, x) printf("_DBG: %s %s = %p \n", msg, #x, x);
#else
    #define INFO(x) ;
    #define DEBUG(x) ;
    #define DEBUGMSG(msg, x) ;
#endif //__VETLIB_DEBUGMODE__
```

## 1.4 - vetFrame

Pure Abstract Class  
vetFrame.h

VETCLASS\_TYPE\_FRAME  
Header @ pg. 127

Un video digitale (anche analogico) è campionato nel tempo ed è classicamente rappresentato da una sequenza di immagini, la rapidità con cui le immagini si susseguono (frame per second) dà allo spettatore l'illusione della continuità (*PAL: 25fps, NTSC: 30fps*).

Le principali informazioni legate a un immagine digitale sono:

- Risoluzione (width, height)
- Rappresentazione del Pixel (pel)

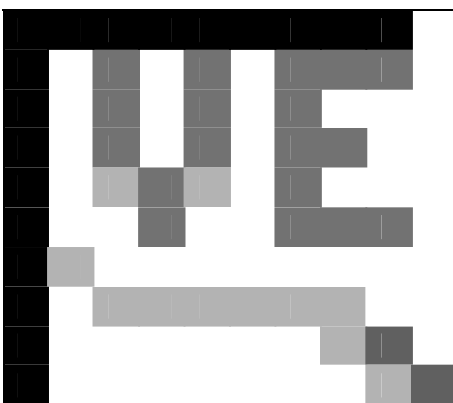
Un singolo frame è rappresentato da una matrice bidimensionale di pixels, l'oggetto pixel definisce il colore di un punto, chiaramente sono possibili molteplici rappresentazioni del colore (*Color Space*), le tecniche di compressione sono basate anche sulla correlazione spaziale oltre che sul sistema di rappresentazione, ma durante il processing i dati sono normalmente non compressi visto che l'accesso ai pixel deve essere standard e ottimizzato.

La sintesi di un colore in una memoria digitale è ovviamente legata al numero di bit a disposizione:

Bits per pixel	Numero valori (colori) rappresentabili	Calcolo
1	2	$2^1$
2	4	$2^2$
4	16	$2^4$
8	256	$2^8$
16	65,536	$2^{16}$
24	16,777,216	$2^{24}$

Esempio con pixel a 4 bit:

0	0	0	0	0	0	0	0	0	3
0	3	1	3	1	3	1	1	1	3
0	3	1	3	1	3	1	3	3	3
0	3	1	3	1	3	1	1	3	3
0	3	2	1	2	3	1	3	3	3
0	3	3	1	3	3	1	1	1	3
0	2	3	3	3	3	3	3	3	3
0	3	2	2	2	2	2	2	3	3
0	3	3	3	3	3	3	2	1	3
0	3	3	3	3	3	3	3	2	1



Dec	Bin	Color
0	00	Black
1	01	Dark Gray
2	10	Medium Gray
3	11	Light Gray

Palette

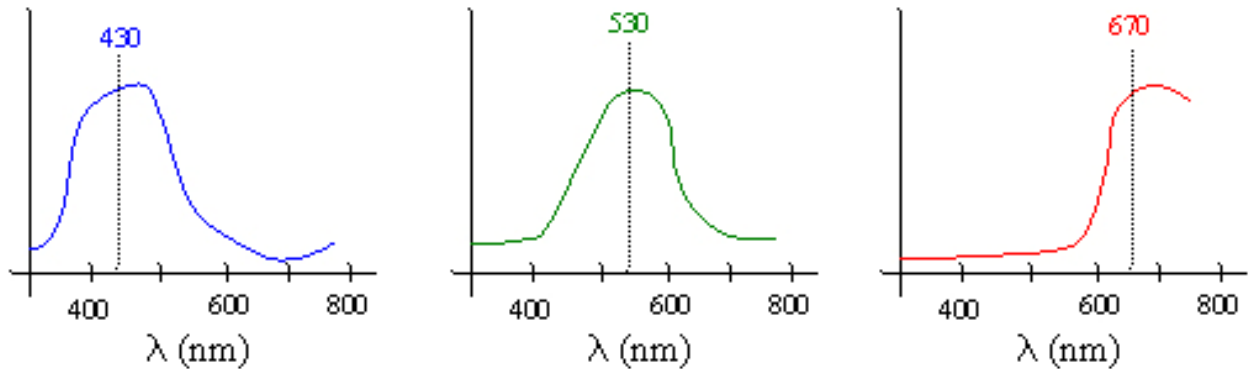
Questa matrice può essere memorizzata in un array monodimensionale in sequenza *raster*:

```
image {0, 0, 0, 0, 0, 0, 0, 0, 0, 3,
       0, 3, 1, 3, 1, 3, 1, 1, 1, 3,
       ...}
```

Cioè in bits: {00 00 00 00 00 00 00 00 11 00 11 01 11 01 11 01 01 11 ...}

Finora si è considerato il caso monocromatico, il colore è un argomento molto più ampio di quello che si crede ed è estremamente legato alla pura matematica (comunemente la conversione tra spazi colore diversi viene fatta attraverso calcoli matriciali), il colore percepito dipende dall'assorbimento e dalla rifrazione dei materiali, l'analisi più diffusa e intuitiva distingue i tre colori base: rosso, verde e blu, attraverso la sintesi additiva (composizione lineare) è possibile sintetizzare qualunque colore partendo da questo insieme base. Gli spettri sono parzialmente sovrapposti e la nostra capacità di percepire le diverse frequenze non è costante, in particolare l'occhio umano ha una buona capacità di vedere la luminosità e i contrasti (scala di grigio), lo studio del sistema visivo è importante proprio perché l'imperfezione con cui percepiamo stimoli visivi è ampiamente sfruttata nelle tecniche di compressione (il medesimo concetto è espresso in campo uditivo, ad esempio, con

il formato MP3: percepiamo suoni nel range 20÷20.000 Hz, quindi applico un filtro passa-banda e quantizzo diversamente). Seguono tre grafici dello spettro luminoso visibile che sottolineano la frequenza “centrale” per ognuno dei colori base:

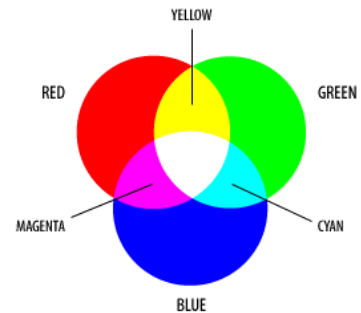


È possibile rappresentare un colore in molti modi, ma esistono essenzialmente due grandi macro-gruppi differenti: RGB e YUV.

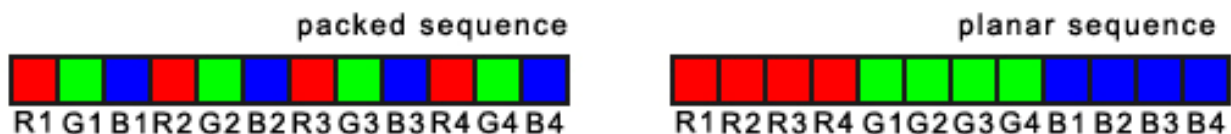
Il formato RGB (Red, Green, Blue) è stato introdotto con le considerazioni precedenti ed è diffuso nel processing digitale di immagini, il sistema di rappresentazione coincide con quello presentato all’inizio di questa sezione moltiplicato per tre canali, il risultato finale è la sovrapposizione (somma) dei tre livelli (*layers*), il formato più comune rappresenta i pixel con 24 bits, 8 bits per canale (3 *char*, cioè 3 bytes).

In seguito alle considerazioni sul sistema visivo e alla nascita del broadcasting televisivo (*PAL*, *NTSC*) si è consolidato un sistema di rappresentazione alternativo: YUV, un tempo utile per la compatibilità con i vecchi apparecchi (tv in bianco e nero), oggi utile per minimizzare la memoria, la diffusione dello spazio YUV è anche dovuta all’importanza predominante della luminosità rispetto all’informazione colore nella maggior parte degli algoritmi di processing video, lo spazio RGB infatti non prevede un canale che sintetizzi le informazioni sulla luminosità dell’immagine che dovrebbe essere estratta attraverso una composizione lineare (tempo di processing!).

Lo spazio YUV distingue tre canali: uno per la luminosità (*Luminance*) e due per il colore (*Crominance1*, *Crominance2*), vi sono molte rappresentazioni diverse, la maggior parte sotto-campiona orizzontalmente e/o verticalmente i valori delle crominanze diminuendo la memoria necessaria. Ad esempio il formato YUV 4:2:0 (*vetFrameYUV420*) memorizza due valori di crominanza (*Cb* e *Cr*) per ogni gruppo di quattro valori di luminosità.



A prescindere dallo spazio colore scelto, le informazioni (i valori) possono essere raggruppati in pixels (o nel caso YUV in macro-pixels) disposti in sequenza, in un array di lunghezza *width\*height*, oppure in modo planare (*planar*), questa disposizione prevede di disassemblare l’entità pixel negli *n* valori (tipicamente tre) e disporli in sequenza raster divisi per canale, segue la rappresentazione di quattro pixel bianchi:



Entrambe le soluzioni offrono vantaggi e svantaggi, lo spazio colore RGB è rappresentato di norma in modo packed, mentre lo spazio YUV massimizza i vantaggi con sistema planare, moltissimi filtri lavorano solo sulla luminosità dell’immagine e quindi nel caso dell’RGB, ad esempio, oltre che gestire memoria inutilmente si richiedono calcoli aggiuntivi per l’estrazione del piano luminosità dai tre canali.

VETLib propone alcune implementazioni di frame (*vetFrameYUV420*, *vetFrameRGB24*, ..) basate sugli standard di rappresentazione più comuni e un particolare oggetto *vetFrameT* in grado di adattarsi a necessità specifiche dello sviluppatore.

Tutto gli oggetti frame implementano l'interfaccia *vetFrame* (*VETCLASS\_TYPE\_FRAME*) nella quale sono definiti una serie di prototipi ed alcune variabili pubbliche:

```
unsigned int    width;
unsigned int    height;
long           timeStamp;
```

I due interi descrivono le dimensioni dell'immagine e la variabile *timeStamp* è dedicata a contesti real-time e identifica il tempo di creazione del frame (serve ai controlli di *PlayOut*), sono direttamente implementate le funzioni *getHeight* e *getWidth* (anche *const*).

Il metodo standard di (ri)allocazione della memoria è imposto dal prototipo:

```
virtual void reAllocCanvas(unsigned int w, unsigned int h) = 0;
```

La funzione *reAllocCanvas*, chiamata anche dai costruttori, sostituisce i metodi *setWidth* e *setHeight*, da notare che i dati precedenti non sono (necessariamente) mantenuti o copiati nel nuovo canvas (il piano immagine), più avanti in questo capitolo sono riportate le implementazioni di ogni oggetto immagine.

Le informazioni sul sistema di rappresentazione adottato dagli oggetti frame sono accessibili tramite i metodi imposti da *vetFrame*:

```
virtual VETFRAME_PROFILE getProfile() = 0;
virtual VETFRAME_CHANNEL_TYPE getChannelType() = 0;
```

che definisce le due enumerazioni:

```
enum VETFRAME_CHANNEL_TYPE    {

    VETFRAME_CT_NONE,
    VETFRAME_CT_PIXELPACKED,
    VETFRAME_CT_PLANAR,
    VETFRAME_CT_CUSTOM
};
```

e

```
enum VETFRAME_PROFILE    {

    VETFRAME_NONE,        //empty
    VETFRAME_BITPLANE    //bool {0, 1}
    VETFRAME_MONO,        //grayscale

    // RGB formats
    VETFRAME_RGB24,        //standard
    VETFRAME_BGR24,        //also standard
    VETFRAME_RGB96,        //standard
    VETFRAME_BGR96,        //also standard

    VETFRAME_RGB565,        //pixel 16bit!
    VETFRAME_BGR565,        //pixel 16bit!
    VETFRAME_RGB555,        //pixel 16bit!
    VETFRAME_BGR555,        //pixel 16bit!

    VETFRAME_ARGB32,        //alpha + .
    VETFRAME_ABGR32,        //alpha + .
    VETFRAME_RGBA32,        //. + alpha
    VETFRAME_BGRA32,        //. + alpha
```

```

// YUV formats
VETFRAME_I420,    //4:2:0 planar (=IYUV)
VETFRAME_YV12,   //4:2:0 planar (common in MPEG: NxM Y + N/2*M/2 V U )
VETFRAME_YUY2,   //4:2:2 packed (common in AVI and hardware devices)
VETFRAME_UYVY,   //4:2:2 packed (radius cinepack, mpeg codecs)
VETFRAME_YVYU,   //4:2:2 packed

VETFRAME_AYUV,   //alpha + 4:4:4 planar

VETFRAME_CUSTOM  //not listed here..
};

```

Sono stati definiti una serie di codici standard univoci per identificare i formati possibili, questo codice (*int*) è detto *FOURCC* e classicamente è riportato negli headers (primi 4 bytes), il sito <http://www.fourcc.org> analizza nel dettaglio la maggior parte dei formati esistenti, gli oggetti frame di VETLib identificano il proprio codice con il prototipo imposto da *vetFrame*:

```
virtual int getFOURCC() = 0;
```

La classe *vetFrame* impone anche l'implementazione dei due prototipi dedicati all'accesso diretto:

```
virtual void* dump_buffer() = 0;
virtual unsigned int getBufferSize() = 0; //in bytes
```

in tutti gli oggetti frame implementati il buffer è chiamato *data* ed il codice è banalmente un casting inline nello stile C++:

```
void* dump_buffer() { return static_cast<void*>(data); };
```

oltre all'accesso al puntatore dei dati, spesso può essere utile anche un metodo che estragga la luminosità (*brightness*) in un formato classico:

```
virtual VETRESULT extractBrightness( unsigned char* buffer,
                                     unsigned int* size = NULL ) = 0;
```

la memoria deve essere allocata dalla funzione chiamante (*unsigned char[size]*), la dimensione può essere letta con i metodi *getWidth* e *getHeight* oppure effettuando una chiamata precedente del metodo con il parametro *buffer* impostato a *NULL* (come lo stile della programmazione avanzata per Windows):

```
unsigned int size = 0;
vetFrameRGB24->extractBrightness(NULL, &size);
if ( size == 0 )
    return;
unsigned char* greyBuffer = new unsigned char[size];
vetFrameRGB24->extractBrightness(greyBuffer, NULL);
```

L'allocazione del buffer negli oggetti frame di VETLib non prevede l'inizializzazione dei pixel (spesso è solo una perdita di tempo), i seguenti prototipi aggiornano il canvas (piano immagine) rispettivamente con la minima e la massima luminosità, i valori dipendono dalla rappresentazione:

```
virtual VETRESULT setBlack() = 0;
virtual VETRESULT setWhite() = 0;
```

L'accesso alla memoria può essere ottimizzato in modo molto semplice, un *ciclo for* che azzeri i pixel è tremendamente inefficiente rispetto a metodi di sistema come *memset*, consultare la sezione "*Working with Frames*" nel capitolo terzo per dettagli ed esempi pratici.

### 1.4.1 - *vetFrameYUV420*

vetFrameYUV420.h

```
VETCLASS_TYPE_FRAME
VETFRAME_I420
VETFRAME_CT_PLANAR
0x30323449
```

Header @ pg. 129

Questo oggetto frame è uno dei tre standard I/O di VETLib, i dati (*pixels*) sono rappresentati secondo il formato YUV 4:2:0 planare, si ottimizza la memoria occupata sotto-campionando spazialmente nelle due direzioni l'informazione associata al colore (le due crominanze U e V).

I singoli valori (sia luminanza Y che U e V) sono memorizzati come *unsigned char* (8 bits) in un array di lunghezza complessiva  $width * height * 1.5$ , dove i primi  $width * height$  bytes (caratteri: un byte = 8 bits) corrispondono ai valori della luminanza in sequenza raster e i seguenti  $width / height / 2$  caratteri sono divisi in due piani consecutivi e rappresentano i valori U e V rispettivamente. Il buffer è dichiarato public:

```
unsigned char *data;
```

I seguenti puntatori sono ausiliari e devono essere usati semplicemente come scorciatoia per l'accesso ai tre canali:

```
unsigned char *Y; // = data;
unsigned char *U; // = data + width * height;
unsigned char *V; // = data + (int)( width * height * 1.25 );
```

Il metodo (imposto da *vetFrame*) designato ad (ri)allocare il buffer è riportato per intero:

```
void vetFrameYUV420::reAllocCanvas(unsigned int w, unsigned int h)
{
    if (data != NULL) {
        delete [] data;
        data = NULL;
    }

    Y = NULL;    U = NULL;    V = NULL;
    width = w;
    height = h;

    if ( width != 0 && height != 0 ) {
        data = new unsigned char[ width * height * 1.5 ];
        Y = data;
        U = data + width * height;
        V = data + (int)( width * height * 1.25 );
    }
}
```

La maggior parte dei filtri operano sulla *brightness* (luminosità) dell'immagine, sia per la quantità di informazione associata sia per ridurre il numero di operazioni, questo formato è particolarmente adatto all'accesso veloce e sequenziale a questi valori (piano Y) pur conservando le informazioni sul colore (qualità ridotta ma accettabile). Segue la banale implementazione del prototipo imposto da *vetFrame*:

```
virtual VETRESULT extractBrightness( unsigned char* buffer,
                                     unsigned int* size = NULL ) {
    if (buffer == NULL) {
        if ( size == NULL)
            return VETRET_PARAM_ERR;
        *size = width*height;
        return VETRET_OK;
    }
    memcpy (data, buffer, width*height);    // plane Y = luminance
    return VETRET_OK;
}
```

## 1.4.2 - *vetFrameRGB24*

vetFrameRGB24.h

```
VETCLASS_TYPE_FRAME
VETFRAME_RGB24
VETFRAME_CT_PACKED
0x32424752
```

Header @ pg. 128

La classica rappresentazione del colore nei canali Rosso, Verde e Blu (in questo ordine) è implementata in questo oggetto immagine con un array monodimensionale (*data*) di *PixelRGB24*, ogni istanza di *PixelRGB24* è un singolo pixel a cui corrispondono i tre valori memorizzati nel tipo standard *unsigned char*. L'array *data* contiene quindi *width \* height* pixel in sequenza raster, ogni pixel è costituito da tre caratteri (8 bits ciascuno, un byte), i valori spaziano nel range [0,256[ (è *unsigned*), secondo la convenzione classica il valore zero corrisponde al nero ed il valore 255 è la massima luminosità (sintesi additiva), la combinazione dei tre canali permette di rappresentare 16,777,216 colori (SVGA).

Questo formato corrisponde al codice *FOURCC* 0x32424752 (che non dipende dal numero di bit), il profilo (*VETFRAME\_PROFILE*) è *VETFRAME\_RGB24* e naturalmente il sistema di memorizzazione (*VETFRAME\_CHANNEL\_TYPE*) è di tipo *VETFRAME\_CT\_PACKED*.

L'allocazione del buffer è estremamente banale:

```
void vetFrameRGB24::reAllocCanvas(unsigned int w, unsigned int h)
{
    if (data != NULL)
        delete [] data;

    height = h;
    width = w;
    data = NULL;

    if ( w != 0 && h != 0)
        data = new PixelRGB24[w * h];
}
```

Si potrebbe allocare anche in modo diretto poiché in effetti l'assembly equivale a

```
data = new unsigned char[width * height * 3]
```

ed infatti la seguente conversione (*casting*) è corretta:

```
PixelRGB24    *pBuffer;
unsigned char*rawBuffer

rawBuffer = (unsigned char*)pBuffer[0];           // C style
// or better:
rawBuffer = static_cast<unsigned char*>(pBuffer[0]); // C++ style
```



### 1.4.3 - *vetFrameT*<class T>

Questa classe è la più generica e complessa struttura dati che rappresenta un frame, lo sviluppatore può configurare l'oggetto in modo da ottimizzare la memoria ed implementare strutture di pixel e metodi di accesso specifici. Analizziamo le principali variabili (dichiarate *public* e quindi accessibili):

Template Class vetFrameT.h  VETCLASS_TYPE_FRAME VETFRAME_CUSTOM VETFRAME_CT_CUSTOM variable  Header @ pg. 130
---

- unsigned int width;  
Valore intero positivo che indica la larghezza (x) dell'immagine.
- unsigned int height;  
Valore intero positivo che indica l'altezza (y) dell'immagine.
- T \*data;  
Array monodimensionale di oggetti T (template) contenente i pixel (ex. *char*, *PixelRGB24*).
- bool autoFreeData;  
Indica se l'oggetto deve liberare i dati quando viene distrutto (distruttore), è utile per ottimizzare la memoria nel caso un cui si voglia "riciclare" un buffer esistente. Il valore predefinito è *TRUE*. Ad esempio:

```
unsigned char* buffer = new unsigned char[width*height*3];
// fill the buffer...
// [...]

vetFrameT<PixelRGB24>* fakeFrame = new vetFrameT<PixelRGB24>();
// no data allocated, it's an empty object

// setup the fake vetFrameT object
fakeFrame->autoFreeData = false;
fakeFrame->width = width;
fakeFrame->height = height;
fakeFrame->data = dynamic_cast< PixelRGB24* >(buffer);
//we force the buffer

//call an external function which require a vetFrameT<PixelRGB24>.
processFunct(fakeFrame, output);
// [...]

delete fakeFrame;
// buffer has NOT been deleted
```

- enum VETFRAME\_PROFILE profile;  
Questa variabile dichiara il sistema corrente di rappresentazione del pixel, ovviamente è necessario che un algoritmo (una funzione) conosca il modo di rappresentare l'immagine per poter elaborare i dati. Le rappresentazioni possibili sono definite in *vetFrame*, la funzione *getProfile()*, restituisce il formato corrente.
- enum VETFRAME\_CHANNEL\_TYPE dataType;  
Definisce il modo si immagazzinare i pixel (vedere *vetFrame*), ovviamente questo parametro è legato al formato utilizzato (*VETFRAME\_PROFILE*), alcuni ammettono entrambe le rappresentazioni (RGB), mentre altri definiscono intrinsecamente anche la disposizione dei pixel, in tal caso il parametro prioritario è il profilo (altrimenti viene considerato prima il *profile* e poi il *dataType*).

La classe offre una serie di metodi classici per l'accesso ai pixel e la gestione della memoria, tali implementazioni sono fortemente legate al formato scelto, i profili definiti in *vetFrame* (*VETFRAME\_PROFILE*) non sono necessariamente tutti supportati (anche perché potrebbero essere aggiornati in modo asincrono) quindi il metodo:

```
bool isBuiltInSupportedProfile(VETFRAME_PROFILE pr);
```

certifica se il profilo richiesto è supportato dai metodi di base (*setPixel*, *getPixel*, *reAllocCanvas*, ..), lo sviluppatore può comunque accedere alle variabili principali e quindi implementare soluzioni proprietarie utilizzando la classe esclusivamente come struttura dati, lo stile di VETLib prevede l'uso classico dei costruttori (cioè gestione memoria affidata all'oggetto frame, classicamente con il metodo *reAllocCanvas*) e l'accesso diretto ai pixel tramite il buffer pubblico (si consiglia di evitare l'uso di *setPixel*, *getPixel*)

Il codice di identificazione del formato *FOURCC* dipende chiaramente dal profilo corrente ed è implementato in questo modo:

```
int getFOURCC() {
    switch( profile ) {
        case vetFrame::VETFRAME_MONO:
        case vetFrame::VETFRAME_RGB24:
        case vetFrame::VETFRAME_BGR24:
        case vetFrame::VETFRAME_RGB32:
        case vetFrame::VETFRAME_BGR32:
            return 0x32424752;

        // [...]

        case vetFrame::VETFRAME_I420:
            return 0x30323449;

        // [...]
    }
}
```

Lo stile *switch(profile) {...}* è altamente consigliato ed è attualmente presente in tutti i metodi che coinvolgono l'oggetto *vetFrameT*.

La configurazione predefinita (costruttore default) è la seguente:

```
width           = 0;
height          = 0;
data            = NULL;
autoFreeData    = true;
dataType        = vetFrame::VETFRAME_CT_NONE;
profile         = vetFrame::VETFRAME_NONE;
```

L'utilizzo pratico di una classe template può essere focalizzato su un tipo in particolare:

```
vetFrameRGB24* source = new vetFrameRGB24(320, 240);
// .. fill source..

vetFrameT<unsigned char> *buff;
buff = new vetFrameT<unsigned char>(); //it's: w=0; h=0; data=NULL
buff->width = source.width;
buff->height = source.height;
buff->profile = vetFrame::VETFRAME_RGB24;
buff->dataType = vetFrame::VETFRAME_CT_PACKED;
buff->autoFreeData = false;
buff->data = (unsigned char)source->data[0];

myTemplateBasedMethod(buff);

delete buff;

//source data (vetFrameRGB24) was updated through vetFrameT proxy
```

Oppure si può mantenere l'astrazione in un metodo:

```
template <class T, class S>
static int doProcessing( vetFrameT<T> &source,
                       vetFrameT<S> &dest,
                       vetDFMatrix& kernel
                       )
{
    switch( profile ) {
        // [...]
        dest.data[y * src_w + x] = (S)numb;
        // [...]
    }
}
```

In questo caso si ammette una conversione di tipo automatica, l'astrazione rischia di portare ad errori ed aberrazioni cromatiche (ex. overflow), lo sviluppatore deve assumersi il compito di eventuali conversioni dello spazio colore.

Come analizzeremo nel dettaglio più avanti in questo capitolo uno degli standard I/O tra i componenti (*vetInput*, *vetOutput*) è proprio l'oggetto *vetFrameT*:

```
VETRESULT extractTo(vetFrameT<unsigned char>& img);
```

La scelta è caduta sul tipo *unsigned char* per due ragioni: non era possibile mantenere l'astrazione template poiché il C++ non ammette metodi virtuali template (inoltre sarebbe stato abbastanza problematico gestire l'oggetto nei filtri), il tipo (*unsigned*) *char* è lo standard di accesso alla memoria per C e C++, quindi si può "ufficiosamente" utilizzare un altro tipo e accedere ai dati tramite il puntatore *char* (l'operatore *new* è basato su *malloc* che come ben sanno i programmatori C, alloca solo caratteri). Virtualmente non vi sono limitazioni al formato rappresentato, ciò rispecchia gli obiettivi di portabilità e scalabilità di VETLib, i metodi (come *extractTo*) avranno comportamenti diversi a seconda del formato (*vetFrameT::profile*).

Alcuni esempi di utilizzo dei template in C++ sono trattati nel capitolo terzo, sezione "Templates".

#### 1.4.4 - *vetFrameRGBA32*

Questo oggetto frame differisce dagli altri standard per il numero di canali gestiti, si è aggiunto l'*Alpha Channel* che indica la trasparenza del pixel, i singoli valori sono memorizzati come *unsigned char* e quindi un pixel riempie i classici registri da 32 bits, le schede grafiche di ultima generazione e il framework *DirectDraw* hanno reso questo formato molto diffuso soprattutto in campo ludico.

La disposizione dei dati è di tipo planare (quattro piani consecutivi di *width\*height* bytes).

I calcoli possono spesso essere ottimizzati con le seguenti maschere da applicare ad un pixel 32bit:

```
Red      0xFF000000
Green    0x00FF0000
Blue     0x0000FF00
Alpha    0x000000FF
```

vetFrameRGBA32.h

```
VETCLASS_TYPE_FRAME
VETFRAME_RGBA32
VETFRAME_CT_PLANAR
0x41424752
```

Header @ pg. 131

### 1.4.5 - *vetFrameGrey*

Questa classe rappresenta immagini (mono-canale) a scala di grigio (*brightness*) e fornisce metodi del tutto simili agli oggetti frame precedentemente analizzati, il pixel è *PixelGrey*, definito come *unsigned char* (8 bits = 1 byte), il profilo è *VETFRAME\_MONO*, la struttura è definita come *VETFRAME\_CT\_PACKED*, ma in realtà potrebbe essere definita anche come *VETFRAME\_CT\_PLANAR* visto che un pixel è rappresentato con un singolo valore, il codice *FOURCC* non è definito.

```
vetFrameGrey.h
VETCLASS_TYPE_FRAME
VETFRAME_MONO
VETFRAME_CT_PACKED
0
Header @ pg. 134
```

L'allocazione del buffer è banale (*data = new PixelGrey[w \* h];*), segue un frammento più interessante che converte l'immagine in formato YUV:

```
vetFrameGrey& vetFrameGrey::operator >> (vetFrameYUV420& img)
{
    if ( width == 0 || height == 0 )
        throw "Cannot do that with empty image (no size)";

    if ( width != img.width || height != img.height )
        img.reAllocCanvas(width, height);

    // valid because pixelgrey is uchar!
    memcpy(img.data, data, width * height);
    memset(img.U, '\0', width * height / 2); // u,v set to 0

    // not optimized:
    // img.setBlack();
    // const unsigned int size = width * height;
    // for (unsigned int i=0; i < size; i++)
    //     img.data[i] = (unsigned char)( data[i] );

    return *this;
}
```

### 1.4.6 - *vetFrameRGB96*

La classe è pressoché identica a *vetFrameRGB24*, differisce solo per la rappresentazione dei pixel, in questo caso si tratta di *PixelRGB96* che memorizza ogni valore (r, g, b) come *integer* (32 bits, quindi 96 bits per pixel), l'occupazione di memoria è sovradimensionata (si usano i valori *[0,256]*), ma talvolta si trovano librerie e applicazioni in cui è utile questo formato.

Nel caso di sequenze video la quantità di memoria necessaria è proibitiva (anche un frame piccolo 320\*240 richiede un'occupazione di memoria di 900Kb) e si preferisce utilizzare *vetFrameRGB24*.

La selezione del canale, richiesta in alcuni metodi, viene effettuata attraverso un parametro di tipo *enum ChannelRGB { RED, GREEN, BLUE }* (e quindi *vetFrameRGB96::RED*), il profilo (*VETFRAME\_PROFILE*) è naturalmente *VETFRAME\_RGB96* con una disposizione in memoria (*VETFRAME\_CHANNEL\_TYPE*) di tipo *VETFRAME\_CT\_PACKED*, il codice *FOURCC* è il medesimo di *vetFrameRGB24*.

```
vetFrameRGB96.h
VETCLASS_TYPE_FRAME
VETFRAME_RGB96
VETFRAME_CT_PACKED
0x32424752
Header @ pg. 132
```

### 1.4.7 - *vetFrameHSV*

L'oggetto *vetFrameHSV* rappresenta le immagini secondo lo standard HSV, diffuso nell'ambito dell'immagine processing, i tre valori che contraddistinguono il pixel sono: *Hue*, che corrisponde un angolo e varia nel range [0, 360[, *Saturation* e *Value* che indicano le due coordinate e possono assumere i valori [0,256[. La scelta del range non è univoca, spesso nei calcoli si considerano i valori normalizzati, ma per memorizzare i dati è più conveniente la scelta di VETLib:

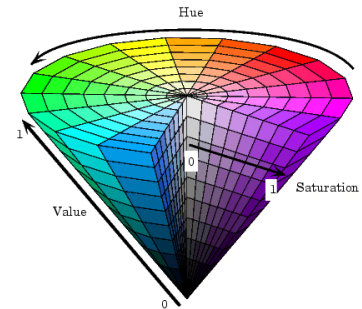
```
unsigned short int hue; //16bit |
unsigned char sat; //8bit |= 32bit
unsigned char vat; //8bit |
```

L'accesso al pixel è diverso dagli altri casi poiché l'array non è uniforme, le tre variabili sono dichiarate pubbliche e quindi il codice diventa:

```
data[offset].hue = hue;
data[offset].sat = sat;
data[offset].val = val;
```

Questo oggetto è stato creato per compatibilità ma di fatto attualmente non è usato in alcun modulo della libreria.

```
vetFrameHSV.h
VETCLASS_TYPE_FRAME
VETFRAME_HSV
VETFRAME_CT_PACKED
0
Header @ pg. 133
```



### 1.4.8 - *libETI* support

Le due classi *vetFrameRGBETI* e *vetFrameGreyETI* sono il punto di contatto tra VETLib e la vecchia libreria di elaborazione immagini (*libETI*), la gestione della memoria è affidata alle relative classi (*libETI::picture* e *libETI::image*, rispettivamente).

L'uso di questi oggetti è deprecato, l'implementazione delle estensioni di ETILib non consente prestazioni accettabili per il processing di più immagini, i pixel sono memorizzati esclusivamente in canali RGB planari di *floating point*, non accessibili direttamente, inoltre le classiche operazioni sono effettuate su una copia dell'immagine e non sull'originale.

Purtroppo la scarsa portabilità di ETILib e la diversità intrinseca rispetto a VETLib non consentono di convertire i lavori svolti in maniera semplice, è necessario modificare abbondantemente il codice a mano.

## 1.5 - vetInput

Abstract Class  
vetInput.h  
VETCLASS\_TYPE\_INPUT  
Header @ pg. 136

Questa interfaccia definisce lo standard di acquisizione dati di VETLib, le implementazioni specifiche si occupano di decodificare i dati da uno stream dinamico o statico (prelevato ad esempio da un device) e convertirli nei formati standard di VETLib: *vetFrameYUV420*, *vetFrameRGB24* e possibilmente *vetFrameT*. L'interfaccia prevede l'implementazione dei prototipi:

```
virtual VETRESULT extractTo(vetFrameYUV420& img) = 0;  
virtual VETRESULT extractTo(vetFrameRGB24& img) = 0;  
virtual VETRESULT extractTo(vetFrameT<unsigned char>& img) = 0;
```

Gli operatori di streaming sono implementati in modo da gestire (se richiesto) una frame rate costante in uscita, l'utilizzo pratico è banale (*vetNoiseGenerator* >> *bufferImg*):

```
vetInput& operator >> (vetFrameYUV420& img);  
vetInput& operator >> (vetFrameRGB24& img);  
vetInput& operator >> (vetFrameT<unsigned char>& img);  
  
vetInput& vetInput::operator >> (vetFrameRGB24& img)  
{  
    setElaborationStart();  
    extractTo(img);  
    if( v_sleeptime && v_sleeptime > (long)getElaborationTime())  
        vetSleep( v_sleeptime - (long)getElaborationTime() ); // ms  
    return *this;  
}
```

In questo modo, per i casi più elementari, è sufficiente implementare l'algoritmo di estrazione e preoccuparsi solo di minimizzare il tempo di esecuzione, se la frame rate impostata fosse minore infatti, *vetSleep(long millisec)* allinea la restituzione del processo con il tempo previsto. Le sorgenti più complesse possono sovra-scrivere gli operatori. Questa soluzione può essere utile in un contesto di testing e di debug, ovviamente è deprecabile abbassare la frame rate all'inizio di una catena di sistemi. Normalmente si può ignorare completamente la gestione della frame rate (*v\_sleeptime = 0*).

```
float getFrameRate() const { return v_framerate; };  
  
VETRESULT setFrameRate(float fps)  
{  
    // [...]  
    v_sleeptime = (long)( (float)1000 / fps ); // milliseconds  
    // [...]  
}
```

La libreria standard di I/O ci suggerisce che è molto utile disporre di una funzione che restituisca l'attuale stato dello stream, l'algoritmo che gestisce il flusso di dati potrebbe (verosimilmente) cambiare il suo comportamento quando non ci sono più nuovi frame da caricare, in alcuni casi si potrebbe addirittura incorrere in errori grossolani (ad esempio un sistema di video sorveglianza che non si accorge dello spegnimento della telecamera potrebbe valutare sempre l'ultimo frame, ovviamente uguale al precedente), il metodo da implementare è il classico acronimo di *End Of File*:

```
virtual bool EoF() = 0;
```

che restituisce *false* se l'estrazione di frame procede correttamente ed è disponibile il prossimo fotogramma, *true* se lo stream è terminato.

Un altro metodo molto comune nella programmazione a oggetti è l'azzeramento dei parametri correnti e l'impostazione della configurazione predefinita, il prototipo è

```
virtual VETRESULT reset() = 0;
```

L'interfaccia impone anche di implementare i seguenti due metodi per la lettura della risoluzione:

```
virtual unsigned int getHeight() const = 0;
virtual unsigned int getWidth() const = 0;
```

Il costruttore predefinito (pubblico) permette anche di impostare una frame rate (il valore *fps* va ad inizializzare *v\_sleeptime*), tipicamente vengono inizializzati eventuali buffer a *NULL* e si delega alla funzione *reset()* il compito di impostare i parametri predefiniti. Ad esempio:

```
vetVideo4Linux::vetVideo4Linux(float fps) : vetInput(fps)
{
    fd          = -1;
    videoBuffer = NULL; // it's a char buffer
    reset();
}
```

Dove *reset* è implementato:

```
VETRESULT reset()
{
    disconnect();

    if (videoBuffer != NULL)
        delete [] videoBuffer;

    videoBuffer = NULL;
    fd          = -1;
    win.height  = 0;
    win.width   = 0;
    win.height  = 0;
    vpic.palette = 0;

    return VETRET_OK;
}
```

Suggerisco di portare attenzione all'ordine di inizializzazione dei parametri sia in fase di creazione dell'oggetto che durante il processo, in questo caso se non si fosse impostato il puntatore *videoBuffer* a *NULL* nel costruttore si sarebbe incorso in un'eccezione di accesso alla memoria. Inoltre è buona norma impostare sempre il valore di un puntatore a *NULL* dopo aver liberato la memoria. Seguono alcuni esempi semplificati estratti da classi di tipo *vetInput*:

```
VETRESULT vetPlainFrameGenerator::extractTo(vetFrameRGB24& img) {
    memcpy(      (unsigned char*) img.data[0],
                (unsigned char*) bufferRGB->data[0],
                width * height * 3 * sizeof(unsigned char) );

    return VETRET_OK;
}

VETRESULT vetNoiseGenerator::extractTo(vetFrameRGB24& img)
{
    unsigned int size = img.width * img.height;
    unsigned int perc = (unsigned int)( (float)(size) * spread);
    unsigned int noiseCo;

    for ( unsigned int i = 0; i < perc; i++ )
    {
        noiseCo = rand() * rand() % size;
        img.data[noiseCo][0] = (unsigned char)( rand() % 255 );

        // [...] same code for the 3 channels
    }
}
```

Nella maggior parte dei casi il formato di rappresentazione della sorgente implica conversioni dello spazio colore, è utile ricordare che ad ogni conversione si degrada l'informazione originale, lo stile di VETLib prevede che lo sviluppatore ottimizzi un formato (es. YUV420 o RGB24) e non applichi conversioni automatiche per supportare gli altri, eventualmente deve occuparsene l'applicazione che utilizza il modulo (così come il flusso dati).

Il seguente esempio, estratto dal modulo *vetVideo4Linux*, estrae lo stream dal framework v4l attraverso la classica funzione *read*, la struttura *vpic* (precedentemente inizializzata) contiene le informazioni riguardo al formato nativo, e qualora sia compatibile con il formato richiesto in uscita (*img.profile*) i dati vengono estratti direttamente nel frame uscita:

```
VETRESULT vetVideo4Linux::extractTo(vetFrameT<unsigned char>& img)
{
    if (videoBuffer == NULL);
        return VETRET_INTERNAL_ERR;

    if (win.width != img.width || win.height != img.height)
        img.reAllocCanvas(win.width, win.height);

    if ( (vpic.palette == VIDEO_PALETTE_RGB24  && img.profile == vetFrame::VETFRAME_BGR24 ) ||
        (vpic.palette == VIDEO_PALETTE_RGB565 && img.profile == vetFrame::VETFRAME_BGR565) ||
        (vpic.palette == VIDEO_PALETTE_RGB555 && img.profile == vetFrame::VETFRAME_BGR555) )
    {
        read(    fd, img.data[0],
                img.getBufferSize() );

        return VETRET_OK;
    }
    return VETRET_NOT_IMPLEMENTED;
}
```

In questo caso possiamo notare che i dati non vengono bufferizzati, lo sviluppatore può implementare il sistema di estrazione che preferisce, ma ovviamente le operazioni di allocazione e di copia inutili sono da evitare.

Le attuali implementazioni sono situate in *./source/inputs/*



## 1.6 - vetOutput

Pure Abstract Class  
vetOutput.h  
VETCLASS\_TYPE\_OUTPUT  
Header @ pg. 135

*vetOutput* è una classe puramente astratta e definisce l'interfaccia standard di uscita dati di VETLib, le implementazioni ad esempio si occupano di codificare i dati e trasmetterli ad un device di visualizzazione o ad un file.

Le funzioni di lettura dei frame sono definite in modo simmetrico all'interfaccia di estrazione *vetInput*:

```
virtual VETRESULT importFrom(vetFrameYUV420& img) = 0;  
virtual VETRESULT importFrom(vetFrameRGB24& img) = 0;  
virtual VETRESULT importFrom(vetFrameT<unsigned char>& img) = 0;
```

gli operatori sono infatti definiti rispettivamente come:

```
void operator << (vetFrameYUV420& img) { importFrom(img); };  
void operator << (vetFrameRGB24& img) { importFrom(img); };  
void operator << (vetFrameT<unsigned char>& img) { importFrom(img); };
```

l'utilizzo è equivalente, ma nonostante la scrittura con gli operatori sia più leggibile, la chiamata diretta restituisce un valore di riferimento sul risultato del processo.

I seguenti prototipi sono destinati all'aggiornamento della risoluzione in uscita:

```
virtual VETRESULT setHeight(unsigned int value) = 0;  
virtual VETRESULT setWidth(unsigned int value) = 0;
```

Questa interfaccia è implementata da tutti i moduli che si occupano della visualizzazione, quali *vetWindowGTK* (visualizzatore per Linux) e *vetWindow32* (visualizzatore per Windows), anche in questo caso (come *vetInput*) sorge il problema della conversione di spazio colore tra i formati possibili in ingresso e il formato di uscita (verosimilmente solo uno oppure multiplo con selettore), visto lo scopo della classe è opportuno prediligere (ottimizzare) un formato in ingresso (quello più compatibile con l'uscita) e applicare conversioni automatiche (tramite *vetUtility*) per supportare il maggior numero di formati possibile, segue un esempio estratto dal modulo *vetWindowGTK*:

```
VETRESULT vetWindowGTK::importFrom(vetFrameRGB24& img)  
{  
    if (image == NULL) return VETRET_INTERNAL_ERR;  
  
    gdk_draw_rgb_image(        image->window,  
                               image->style->fg_gc[image->state],  
                               0,0,  
                               img.width,img.height,  
                               currDith,  
                               (guchar*)img.data[0],  
                               img.width*3  
                               );  
    return VETRET_OK;  
}  
  
VETRESULT vetWindowGTK::importFrom(vetFrameYUV420& img)  
{  
    if (image == NULL) return VETRET_INTERNAL_ERR;  
  
    vetFrameRGB24 tmp(img.width, img.height);  
    img >> tmp;  
  
    VETRESULT ret = importFrom(tmp);  
  
    if ( ret == VETRET_OK) return VETRET_OK_DEPRECATED;  
    else return ret;  
}
```

Le attuali implementazioni sono situate in *./source/outputs/*

## 1.7 - vetFilter

Abstract Class  
vetFilter.h  
VETCLASS\_TYPE\_FILTER  
Header @ pg. 137

Come visto nelle sezioni precedenti l'interazione tra componenti è possibile grazie alle interfacce *vetInput* e *vetOutput*, attraverso i tre oggetti frame *vetFrameRGB24*, *vetFrameYUV420* e *vetFrameT<unsigned char>*.

La classe *vetFilter* "implementa" entrambe le interfacce e quindi è un filtro (o un buffer) in grado di interagire in un processo complesso (catene di filtri), lo sviluppatore può ovviamente integrare le due interfacce principali in un proprio oggetto con caratteristiche specifiche, ma nella maggior parte dei casi VETLib fornisce questo oggetto per standardizzare e semplificare la creazione di filtri digitali generici.

I seguenti prototipi sono ereditati dall'interfaccia *vetInput* e devono essere implementati dallo sviluppatore, chiaramente si occupano di leggere il frame in ingresso e verosimilmente modificarlo:

```
virtual VETRESULT importFrom(vetFrameYUV420& img) = 0;  
virtual VETRESULT importFrom(vetFrameRGB24& img) = 0;  
virtual VETRESULT importFrom(vetFrameT<unsigned char>& img) = 0;
```

mentre si offre un implementazione dei metodi:

```
VETRESULT extractTo(vetFrameYUV420& img);  
VETRESULT extractTo(vetFrameRGB24& img);  
VETRESULT extractTo(vetFrameT<unsigned char>& img);  
  
bool vetFilter::EoF()  
  
VETRESULT vetFilter::setHeight(unsigned int value);  
VETRESULT vetFilter::setWidth(unsigned int value);  
unsigned int vetFilter::getHeight() const;  
unsigned int vetFilter::getWidth() const;
```

L'ingresso di dati nel filtro consiste nella lettura di un frame da un oggetto *vetInput* tramite *vetFilter::importFrom(..)*, mentre l'uscita dati è l'estrazione del frame verso un oggetto *vetOutput*, naturalmente il frame deve essere memorizzato in un buffer temporaneo all'interno del filtro in attesa che sia estratto (tramite *vetFilter::extractTo(..)*), il seguente codice chiarisce il motivo per il quale è necessario il buffering, il flusso dati è:

```
vetFrameRGB24      globalBuffer;  
vetInput           dataSource;  
vetFilter          imageFlipper;  
vetOutput          visualizationWindow;  
  
dataSource         >>   globalBuffer;  
imageFlipper       <<   globalBuffer;  
imageFlipper       >>   globalBuffer;  
visualizationWindow <<   globalBuffer;
```

L'oggetto *imageFlipper* deve bufferizzare il frame modificato (rovesciato) in attesa che sia estratto (penultima linea), la classe *vetFilter* offre un sistema integrato di gestione dei tre buffer (le interfacce base ammettono l'interazione tramite tre oggetti frame differenti), ma solo uno dei buffer può essere attivo (gli altri sono impostati a *NULL*):

```
vetFrameYUV420     *bufferYUV;  
vetFrameRGB24      *bufferRGB;  
vetFrameT<unsigned char> *bufferTuC;
```

i puntatori sono dichiarati protetti, ma sono accessibili tramite i metodi:

```
vetFrameRGB24*      dump_buffer_RGB() { return bufferRGB; };  
vetFrameYUV420*     dump_buffer_YUV() { return bufferYUV; };  
vetFrameT<unsigned char>* dump_buffer_TuC() { return bufferTuC; };
```

Il buffer in uso può essere individuato grazie a uno dei tre metodi *isBuffer\**, l'implementazione è estremamente banale e simile per i tre metodi, si riporta solo il controllo del buffer YUV420:

```
bool isBufferYUV()
{
    if ( bufferYUV != NULL ) return true;
    return false;
}
```

I buffer possono essere istanziati anche da oggetti esterni (verosimilmente durante l'inizializzazione del sistema), ma è essenziale che nella fase di acquisizione di un nuovo frame si effettui un controllo ed eventualmente un aggiornamento per allineare il buffer alla risoluzione finale, i tre metodi *useBuffer\** si occupano di selezionare il formato richiesto e ridimensionarlo se necessario, segue il codice relativo al buffer YUV420, gli altri due casi sono similari:

```
void vetFilter::useBufferYUV(unsigned int width, unsigned int height) {
    if ( bufferYUV == NULL )
        bufferYUV = new vetFrameYUV420(width, height);
    else if ( bufferYUV->width != width || bufferYUV->height != height )
        bufferYUV->reAllocCanvas(width, height);

    if ( getFilterParameters() != NULL )
        getFilterParameters()->currentBuffer = vetFilterParameters::YUV;

    if ( bufferRGB != NULL ) {
        delete bufferRGB;
        bufferRGB = NULL;
    }
    if ( bufferTuC != NULL ) {
        delete bufferTuC;
        bufferTuC = NULL;
    }
}
```

Nel caso del buffer *vetFrameT* è necessario specificare anche il formato di rappresentazione, i parametri sono passati direttamente al classico costruttore :

```
void useBufferTuC( unsigned int width, unsigned int height,
                  vetFrame::VETFRAME_PROFILE profile      );
```

Il metodo imposto da *vetInput: bool EoF()* è direttamente implementato in *vetFilter* per semplificare lo sviluppo dei filtri più semplici, sarebbe ottimale aggiornare continuamente lo stato:

```
bool vetFilter::EoF() {
    if (bufferYUV != NULL || bufferRGB != NULL || bufferTuC != NULL)
        return false;

    return true;
};
```

Il sistema di buffering integrato influenza i metodi di estrazione dei frame, convenuto che il processing è implementato nelle funzioni di input (*importFrom*) e che il risultato è memorizzato in uno dei buffer, risulta evidente che le funzioni *extractTo* devono semplicemente copiare il buffer nell'immagine output:

```
VETRESULT vetFilter::extractTo(vetFrameYUV420& img)
{
    if ( !isBufferYUV() )
        return VETRET_ILLEGAL_USE;

    img = *bufferYUV;
    return VETRET_OK;
}
```

Si suppone che i frame in ingresso e in uscita siano nel medesimo formato ed in generale non si prevede di integrare la conversione (ad esempio tramite *vetUtility*) qualora i formati non corrispondano, la selezione del buffer dovrebbe avvenire anche in maniera automatica quando un frame viene importato.

Le interfacce *vetInput* e *vetOutput* impongono alcuni metodi per la lettura e la modifica della risoluzione, nella maggior parte dei casi si tratta solo di modificare il buffer interno e quindi *vetFilter* implementa direttamente queste funzioni chiamando il metodo *reAllocCanvas* del buffer attivo, sono stati ripostati due dei quattro metodi, il codice è simmetrico per la larghezza:

```
VETRESULT vetFilter::setHeight(unsigned int value)
{
    if (bufferYUV != NULL) {
        bufferYUV->reAllocCanvas(bufferYUV->width, value);
        return VETRET_OK;
    }
    else if (bufferRGB != NULL) {
        bufferRGB->reAllocCanvas(bufferRGB->width, value);
        return VETRET_OK;
    }
    else if (bufferTuC != NULL) {
        bufferTuC->reAllocCanvas(bufferTuC->width, value);
        return VETRET_OK;
    }
    return VETRET_NOT_IMPLEMENTED;
};

unsigned int vetFilter::getHeight() const
{
    if (bufferYUV != NULL)
        return bufferYUV->height;

    else if (bufferRGB != NULL)
        return bufferRGB-> height;

    else if (bufferTuC != NULL)
        return bufferTuC-> height;

    return 0;
};
```

Se l'aggiornamento della risoluzione implica altre procedure legati al filtro è ovviamente possibile sovrascrivere (*override*) queste implementazioni, ad esempio:

```
VETRESULT vetFilterImplementation::setHeight(unsigned int value)
{
    doMyStuffWithHeightValue(value);
    // [...]
    vetFilter::setHeight(value); // call original method
}
```

Questo sistema di supporto per i casi più comuni è integrato anche nell'inizializzazione e nella distruzione della classe *vetFilter*, in particolare il costruttore è:

```
vetFilter::vetFilter(float fps) : vetInput(fps), vetObject()
{
    bufferYUV = NULL;
    bufferRGB = NULL;
    bufferTuC = NULL;

    setName("Abstract Filter");
    setDescription("Abstract Class");
    setVersion(0.0);
};
```

I metodi *set\** sono implementati dalla classe *vetObject*, la descrizione approfondita è disponibile più avanti in questo capitolo, il risultato è che l'oggetto astratto *vetFilter* ha un nome, una descrizione e una versione di riferimento.

Il distruttore della classe *vetFilter* rilascia la memoria del buffer con una chiamata al metodo:

```
void vetFilter::releaseBuffers()
{
    if (bufferYUV != NULL)
        delete bufferYUV;

    if (bufferRGB != NULL)
        delete bufferRGB;

    if (bufferTuC != NULL)
        delete bufferTuC;

    bufferYUV = NULL; bufferRGB = NULL; bufferTuC = NULL;

    if ( getFilterParameters() != NULL )
        getFilterParameters()->currentBuffer = vetFilterParameters::NONE;
}
```

Con questo esempio si è introdotta la classe *vetFilterParameters*, i parametri di lavoro del filtro dovrebbero essere memorizzati (dinamicamente) nell'implementazione di in questa classe ausiliaria (ad esempio *vetFilterGeometricParameters*), i prototipi di accesso da *vetFilter* sono definiti:

```
virtual VETRESULT setFilterParameters (vetFilterParameters* params) = 0;
virtual vetFilterParameters* getFilterParameters () = 0;
```

La classe *vetFilterParameters* definisce anche un enumerazione per la selezione del buffer:

```
enum BUFFER_TYPE { NONE, YUV, RGB, TuC };
```

la classe *vetFilter* implementa un metodo scorciatoia (protetto) per l'allocazione del buffer:

```
void allocateBuffer(vetFilterParameters::BUFFER_TYPE bType);
```

utile nell'implementazione di *setFilterParameters* e del prototipo imposto per l'azzeramento della configurazione (e dei buffers):

```
virtual VETRESULT reset() = 0;
```

Inoltre *vetFilterParameters* impone l'implementazione di due funzioni dedicate alla serializzazione dei parametri:

```
virtual VETRESULT loadFromStreamXML(FILE *fp) = 0;
virtual VETRESULT saveToStreamXML(FILE *fp) = 0;
```

queste funzioni possono essere utilizzate direttamente o indirettamente tramite i rispettivi metodi che istanziano lo stream di lettura/scrittura (già implementati):

```
VETRESULT saveToXML(const char* filename);
VETRESULT loadFromXML(const char* filename);

int vetFilterParameters::loadFromXML(const char* filename)
{
    FILE *fp;
    if ( ( fp = fopen(filename, "r") ) == NULL )
        return VETRET_PARAM_ERR;
    //[..]
    ret = loadFromStreamXML(fp);
    //[..]
    fclose(fp);
    return ret;
}
```

Seguono alcuni esempi estratti dai filtri inclusi in questa release:

```
VETRESULT vetDigitalFilter::importFrom(vetFrameRGB24& img)
{
    int ret = VETRET_OK;

    if ( !isBufferRGB() ) {
        useBufferRGB(img.width, img.height);
        ret = VETRET_OK_DEPRECATED;
    }

    if (myParams->currentKernel == NULL) {
        *bufferRGB = img;
        return ret;
    }
    else
        ret += doProcessing(    img,
                               *bufferRGB,
                               *myParams->currentKernel,
                               myParams->clampNegative    );

    return ret;
}

VETRESULT vetFilterGeometric::importFrom(vetFrameT<unsigned char>& img)
{
    switch ( myParams->runMode )
    {
        {
            case vetFilterGeometricParameters::DO_NOTHING:
                if ( !isBufferTuC() ) {
                    useBufferTuC(img.width, img.height, img.profile);
                    *bufferTuC = img;
                    return VETRET_OK_DEPRECATED;
                }
                else {
                    *bufferTuC = img;
                    return VETRET_OK;
                }

            case vetFilterGeometricParameters::ROTATE90:
                return rotate90(img);

            default:
                return VETRET_PARAM_ERR;
        }
    }
}
```

Lo sviluppo di filtri è lo scopo principale della libreria, il capitolo terzo analizza nel dettaglio le convenzioni adottate, le modalità di creazione e gli strumenti a disposizione.

Le attuali implementazioni sono situate in *./source/filters/*

## 1.8 - vetCodec

Abstract Class  
vetCodec.h  
VETCLASS\_TYPE\_CODEC  
Header @ pg. 139

Questa interfaccia è la composizione delle classi *vetInput* e *vetOutput* (inoltre eredita *vetObject*), di fatto è estremamente simile alla precedente classe *vetFilter*, ma l'obiettivo è più specifico: il componente è un codificatore e/o un decodificatore. Il buffering è completamente delegato all'implementazione specifica e nella maggior parte dei casi è integrato in una libreria esterna di riferimento (che contiene i veri algoritmi di (de)codifica).

Oltre ai prototipi imposti dalle due interfacce base (*vedi sezioni precedenti*), i moduli di tipo *vetCodec* devono implementare anche una serie di metodi legati all'accesso e alla gestione degli streams:

```
virtual bool isEncodingAvailable() = 0;
virtual bool isDecodingAvailable() = 0;

virtual bool hasAudio(int stream = -1) = 0;
virtual bool hasVideo(int stream = -1) = 0;

virtual int getAudioStreamCount(int stream = -1) = 0;
virtual int getVideoStreamCount(int stream = -1) = 0;

virtual long getVideoStreamLength(int stream = -1) = 0;
virtual long getAudioStreamLength(int stream = -1) = 0;

virtual VETRESULT load(char *filename, int stream = -1) = 0;
virtual VETRESULT save(char *filename, int stream = -1) = 0;

virtual VETRESULT close() = 0;
```

I prototipi *isEncodingAvailable* e *isDecodingAvailable* identificano le capacità del modulo e saranno implementati *inline*, ad esempio il modulo *vetCodec\_MPEG* non è in grado di codificare e quindi:

```
bool vetCodec_MPEG::isEncodingAvailable() { return false; };
bool vetCodec_MPEG::isDecodingAvailable() { return true; };
```

Gli altri metodi identificano alcune caratteristiche dello stream, la convenzione prevede di ritornare il valore *-1* se il metodo o la caratteristica non è supportata, i video di ultima generazione (MPEG2, MPEG4, MOV) sono in grado di supportare più stream (esempio un video e due tracce audio), il parametro *stream* identifica la traccia se il valore è diverso da *-1* (valore predefinito).

I tre metodi *save*, *load* e *close* si occuperanno di istanziare, caricare e chiudere, rispettivamente, un file video.

## Esempio :

```
VETRESULT vetCodec_MPEG::extractTo(vetFrameRGB24& img)
{
    if (file == NULL)
        return VETRET_ILLEGAL_USE;
    if (width != img.width || height != img.height)
        img.reAllocCanvas(width, height);

    // read the frame to output image
    mpeg3_read_frame(file, (unsigned char*)img.data[0],
                    0, 0,
                    width, height,
                    width, height,
                    MPEG3_RGB888, 0);

    return VETRET_OK;
}

VETRESULT vetCodec_MOV::extractTo(vetFrameRGB24& img)
{
    if (file == NULL)
        return VETRET_ILLEGAL_USE;

    if (width != img.width || height != img.height)
        img.reAllocCanvas(width, height);

    // read the frame to output image
    quicktime_decode_video(file, (unsigned char**)img.data, 0);

    myParams->frameIndex++;

    if(myParams->frameIndex > lenghtFrames)
        streamEOF = true;

    return VETRET_OK;
}
```



## 1.9 - vetVision

```
Abstract Class
vetVision.h
VETCLASS_TYPE_VISION
Header @ pg. 140
```

Questa interfaccia è direttamente derivata da *vetOutput*, ma sono state aggiunte una serie di funzionalità per semplificare l'estrazione di informazioni ed un sistema di *callback*.

L'estrazione di informazioni da singole immagini o sequenze video è un ambito estremamente importante e sono ancora in corso molti studi (anche integrando l'AI) per risolvere problemi come la *Computer Vision* (classificazione e l'indipendenza di robots) e la video sorveglianza (riconoscimento e sicurezza). Nella libreria è incluso un banale modulo di *Motion Detection* basato sulla *DFD* (Frame Difference), il ciclo a livello di applicazione indirizza semplicemente frame tramite l'interfaccia di *vetOutput* (*importFrom*):

```
while (i++ < 100)
{
    myCameraSource >> globalBuffer;
    motionEstimationModule << globalBuffer;
}
```

L'evento di allerta (in questo caso viene alzato quando la differenza con il frame precedente supera una certa soglia) è gestito attraverso una i metodi offerti da *vetVision*:

```
void setAlertCall(void* (*functionCall)(void*)) ;
void setAlertCallArgument(void* arg);
void* getAlertCallArgument();
```

L'applicazione deve definire una funzione di allerta (potrebbe ad esempio accendere una sirena) che rispetti il prototipo *void\* (\*alertCall)(void\* argument)*, cioè, ad esempio:

```
void* myAlertFunction(void* myArgumentStruct)
{
    printf("\n -> Image has changed!\n");
    return NULL;
}
```

Il parametro puntatore a *myArgumentStruct* è memorizzato nella variabile *alertCallArgument* e viene passato alla funzione di alert dal metodo:

```
void vetVision::doAlert()
{
    if (alertCall != NULL)
        alertCall(alertCallArgument);
}
```

La configurazione quindi è:

```
motionEstimationModule.setAlertCall(myAlertFunction);
motionEstimationModule.setAlertCallArgument(NULL);
motionEstimationModule.getParameters().setDoAlert(true);
```

L'evento non è asincrono, il *threading* deve essere eventualmente gestito a livello di applicazione (nella funzione di alert), nel caso trattato infatti, il controllo sul frame successivo ad un evento viene analizzato solo quando il controllo è restituito dalla funzione *myAlertFunction*.

I moduli che devono disporre anche di un uscita frame sincrona (ad esempio sottolineano l'oggetto in movimento) dovrebbero implementare anche l'interfaccia *vetInput*, la sintassi è banale:

```
class myMotionDetection: public vetVision, public vetOutput {
```

Se si volesse sfruttare il sistema di buffering integrato in *vetFilter* è ovviamente possibile sostituire *vetOutput* con *vetFilter* nella dichiarazione precedente, la doppia definizione dell'interfaccia di acquisizione (*vetInput*) non crea alcun problema.

## 1.10 - vetBuffer

Abstract Class vetBuffer.h VETCLASS_TYPE_BUFFER Header @ pg.138
--

La classe definisce l'interfaccia di un generico buffer, i metodi imposti forniscono l'accesso ai singoli oggetti e il supporto di un indice del frame corrente.

La classe è un template (*vetBuffer*<class *T*>), le implementazioni possono mantenere questa astrazione o specificare un tipo definito.

Funzioni di configurazione del buffer:

```
void setDoDataCopy(bool value = true)
bool isDataCopyEnabled() const { return copyData; };
virtual int reset() = 0;
```

Funzioni di gestione dei frames:

```
virtual VETRESULT addFrame(T* newFrame) = 0;
virtual VETRESULT insertFrame(long index, T* newFrame) = 0;
virtual VETRESULT updateFrame(long index, T* newFrame) = 0;
virtual VETRESULT removeFrame(long index, bool freeData = false) = 0;
virtual VETRESULT removeFrame(T* frameToDel, bool freeData = false) = 0;
virtual VETRESULT deleteFrames() = 0;

long getFramesCount() const { return v_fcount; };
```

Funzioni di estrazione e posizionamento del frame corrente:

```
virtual T* getFrame(long index) = 0;
virtual long getCurrentFrameIndex() const = 0;

virtual VETRESULT goToNextFrame() = 0;
virtual VETRESULT goToPreviousFrame() = 0;
virtual VETRESULT goToFirstFrame() = 0;
virtual VETRESULT goToLastFrame() = 0;
virtual VETRESULT goToFrame(long index) = 0;
virtual VETRESULT goToStepFrame(long offset) = 0;

virtual T* getLastFrame() = 0;
virtual T* getFirstFrame() = 0;
virtual T* getNextFrame() = 0;
virtual T* getPreviousFrame() = 0;
virtual T* getCurrentFrame() = 0;
```

È opportuno sottolineare che la configurazione predefinita di un *vetBuffer* non prevede la copia né la distruzione degli oggetti memorizzati, la struttura gestisce i puntatori degli oggetti contenuti, il flag *copyData* abilita la creazione di nuovi frames nella fase di aggiunta/inserimento.

Inoltre solo la funzione *deleteFrames()* elimina completamente gli oggetti, nel caso della distruzione del buffer classica (chiamata al distruttore) gli oggetti non vengono cancellati e lo sviluppatore dell'applicazione deve liberare la memoria manualmente. Chiaramente ciò può essere fatto solo conoscendo l'indirizzo degli oggetti (puntatore) che viene perduto quando si distrugge il buffer.

```

template<class T>
class vetBufferArray : public vetBuffer<T>
{
protected:

    // brief data pointers storage.
    T** v_frames;    //frames* array

// [...]

VETRESULT addFrame(T* newFrame)
{
    if (v_fcount+1 > VETDEF_FCOUNT_MAX)
        return VETRET_ILLEGAL_USE;

    if (v_frames == NULL) // this one will be the first frame
    {
        v_storageLenght = VETDEF_STACK_FRAMECOUNT;
        // allocate a new frames array
        v_frames = new T*[v_storageLenght]; // first frames array allocation
        v_fcount = 1;
        // add the new frame
        if ( copyData )
            v_frames[0] = new T(*newFrame);
        else
            v_frames[0] = newFrame;
    }
    else if (v_fcount < (long)v_storageLenght) // there's enough space
    {
        if ( copyData )
            v_frames[v_fcount] = new T(*newFrame);
        else
            v_frames[v_fcount] = newFrame;

        // update frame count to VetInfo
        ++v_fcount;
    }
    else {
        // recalculates new storage (array) length using default step size (>1)
        v_storageLenght = v_fcount + VETDEF_STACK_FRAMEREALLOCSTEP;
        // allocate a new frames array
        T** new_frames = new T*[v_storageLenght];
        // copy frames* to the new array
        for (int i=0; i<v_fcount; i++) {
            new_frames[i] = v_frames[i];
        }
        // remove old pointer and update with new array
        delete v_frames;
        v_frames = new_frames;
        // add the new frame
        if ( copyData )
            v_frames[v_fcount++] = new T(*newFrame);
        else
            v_frames[v_fcount++] = newFrame;
    }

    return VETRET_OK;
};

```

Le attuali implementazioni sono situate in *./source/buffers/*

## 1.11 - vetObject

Abstract Class

vetObject.h

VETCLASS\_TYPE\_OBJECT

Header @ pg. 141

Questa classe è alla base di molti componenti VETLib ed implementa una serie di metodi comuni, va visto come un interfaccia ad un oggetto generico, attualmente si gestiscono solo alcune proprietà fondamentali (nome, descrizione, versione) attraverso i seguenti metodi:

```
void setName(const char* myName);
void setDescription(const char* myDesc);
void setVersion(const double myVersion);

char* getName() const { return f_name; };
char* getDescription() const { return f_description; };
double getVersion() const { return f_version; };

enum{ vetClassType = VETCLASS_TYPE_OBJECT };
```

In futuro si prevede di estendere l'oggetto con i seguenti prototipi:

```
virtual VETRESULT reset() = 0;
virtual vetObject* clone() = 0;
virtual DOUBLE getGUID() = 0;
virtual VETRESULT getChildObjects(void**, unsigned int objCount);
virtual VETRESULT getOwnerObjects(void**, unsigned int objCount);
```

## 1.12 - vetException

Class

vetException.h

VETCLASS\_TYPE\_EXCEPTION

Header @ pg. 126

Le eccezioni sono deprecate nella maggior parte delle implementazioni ottimizzate, la gestione di blocchi `try{}..catch(){}` è infatti onerosa per il compilatore e si ottiene un assembly più lento anche in assenza di errori. Nei casi in cui il tempo di esecuzione non è rilevante (o parzialmente) si può prendere in considerazione l'utilizzo delle eccezioni, è molto frequente nei contesti di accesso a files. Le informazioni si impostano attraverso il costruttore semplificato o completo:

```
vetException() ( std::string s ) : m_s ( s )
{
    caller = NULL;
    retCode = VETRET_INTERNAL_ERR;
};
vetException() ( std::string message, void* callerObject,
                 VETRESULT returnCode ) : m_s ( message )
{
    caller = callerObject;
    retCode = returnCode;
};
```

L'invocazione è banale:

```
myClass::myMethod () {
    // [...]
    throw vetException("file not found!", this, VETRET_ILLEGAL_USE);
}
try {
    myMethod();
}
catch (vetException& ex) {
    printf("%s\n", ex.getDescription() );
    //or cout << ex;
}
```

## 1.13 - Directory structure

I riferimenti interni (*relative paths*) sono molteplici e diffusi in tutti i progetti, è altamente sconsigliato rinominare o spostare qualunque file o directory di VETLib.

Il sorgente della libreria è interamente disponibile nella directory *./source/*, si è scelto di situare header e sorgente nel medesimo percorso e di suddividere i moduli in diverse categorie, la classificazione è arbitraria ma intuitiva:

<i>./source/</i>	:	Classi e interfacce base;
<i>./source/buffers</i>	:	Implementazioni di <i>vetBuffer.h</i> ;
<i>./source/codecs</i>	:	Implementazioni di <i>vetCodec.h</i> ;
<i>./source/filters</i>	:	Implementazioni di <i>vetFilter.h</i> ;
<i>./source/inputs</i>	:	Implementazioni di <i>vetInput.h</i> ;
<i>./source/outputs</i>	:	Implementazioni di <i>vetOutput.h</i> ;
<i>./source/vision</i>	:	Implementazioni di <i>vetVision.h</i> ;
<i>./source/libETI</i>	:	Classi per il supporto di <i>ETILib</i> ;
<i>./source/math</i>	:	Classi inerenti processi matematici;
<i>./source/network</i>	:	Classi relative al networking;

Il file header *./include/VETLib.h* include tutti gli header della libreria ed è utile nel caso di applicazioni finali, l'implementazione di moduli prevede di includere lo stretto necessario, il percorso relativo di inclusione fa riferimento alla directory *./source/*.

I progetti per la costruzione della libreria sono disponibili per Borland C++ Builder 6.0 (non sono supportate tutte le versioni, ad esempio il modulo *vetDirectXInput* non può essere compilato), e per Microsoft Visual Studio 6.0 nelle directory *./lib/bcb/* e *./lib/mvc/* rispettivamente.

Il file di configurazione per ambienti \*NIX è *./Makefile*.

I binari della libreria (*.lib*, *.a*) nelle tre release ufficiali (BCB, MVC, NIX) e delle *special builds* (versioni che includono o meno alcuni moduli) sono situati nella directory *./lib/*.

Ogni modulo di VETLib prevede un rispettivo progetto per test e dimostrazione, i files sono situati in *./tests/*, consultare l'appendice "Sample Applications" per maggiori informazioni.

Le librerie esterne su cui si appoggiano alcuni moduli (ad esempio *vetCodec\_XVID*) sono necessarie per la ricompilazione di VETLib, la distribuzione completa include i binari e quindi in generale non è necessario compilare anche quelle librerie, i progetti fanno riferimento a *./support/* e le relative sub-directories.

Gli strumenti legati alla libreria sono situati in *./Tools/*, attualmente sono disponibili sono per ambienti Windows (32 o superiori), la release corrente dispone di un software per il test e l'utilizzo dei componenti: *VETLib WorkShop* a cui è dedicato il capitolo quarto, un software per automatizzare la creazione di nuovi moduli: *VETLib Package Studio* ed uno strumento per semplificare la gestione della libreria: *VETLib Distribution Manager*, analizzati nel capitolo terzo.

La cartella *./Website* contiene la release 1.0.2 del sito web del progetto VETLib, è realizzato secondo lo standard XHTML 1.0 del consorzio W3C, attualmente il sito è statico e quindi consente la visualizzazione locale anche in assenza di un Web server.

Segue l'elenco esaustivo delle directory:

- ./distr : Archivi da distribuire;
- ./docs : Documentazione della libreria;
  - /html : Documentazione in formato HTML (web);
  - /latex : Documentazione in formato LaTeX;
  - /man : Documentazione in formato MAN (Linux Manual);
  - /pdf : Documentazione in formato PDF;
  - /xml : Documentazione in formato XML;
- ./extras : Add-Ons di VETLib;
- ./images : Una serie di immagini in vari formati per i test;
- ./videos : Una serie di video in vari formati per i test;
- ./include : Headers di VETLib;
- ./lib : Binari di VETLib;
  - /bcb : Progetto di Borland C++ Builder 6.0;
  - /mvc : Progetti di Microsoft Visual C++ 6.0;
  - /mvc7 : Progetti di Microsoft Visual C++ 7.0 (.NET);
- ./packages : Nuovi moduli e il Package Starter Kit;
- ./templates : Files necessari dal tool Package Studio per creare nuovi moduli;
- ./source : Sorgente di VETLib;
- /{category dirs} : ..diviso per categorie
- ./support : Librerie esterne;
  - /ccvt : Alcuni metodi di Color-Space conversion;
  - /directx : Files utili per la compilazione di VETLib con supporto DirectX
  - /ImageMagick : Libreria per leggere e scrivere immagini;
  - /libETI : Libreria ETI;
  - /libmpeg3 : Libreria per decodificare MPEG1-2;
  - /quicktime4linux : Libreria per (de)codificare video QuickTime (.MOV);
  - /xvidcore : Libreria per (de)codificare MPEG4 (.XVID);
- ./tests : Tests e Applicazioni dimostrative;
  - /bcb : Progetti dei test (C++ Builder);
  - /bin : Binari dei test;
  - /mvc : Progetti dei test (Visual C++);
  - /tmp : Files temporanei (dei test);
- ./tmp : Files temporanei (della libreria);
- ./Tools : Strumenti utili;
  - /7-Zip : Compressore ZIP e TAR usato da alcuni script;
  - /Distribution : Script per automatizzare la creazione degli archivi;
  - /DoxyGen : Strumento per creare la documentazione;
  - /MVC6-Updates : Compilatore NASM e aggiornamenti per Visual Studio 6.0;
  - /vetDM : Distribution Manager, un wizard per creare le distribuzioni;
  - /vetPS : Package Studio, uno strumento per creare nuovi moduli;
  - /vetWS : Workshop, un'applicazione per testare ed utilizzare VETLib;
- ./Website : Sito ufficiale di VETLib;

## 1.14 - Builts

Le interfacce, gli oggetti base e i componenti presentati nel prossimo capitolo sono inclusi in un unico file libreria statico (cioè sono compilati e gli oggetti sono “linkati” nella libreria), per semplificare la compilazione in locale sono stati sviluppati più progetti (per Visual Studio e nel Makefile) che contraddistinguono diverse *builts* della libreria, in particolare sono progressivamente esclusi alcuni moduli che necessitano delle librerie esterne.

La libreria compilata per piattaforme Windows offre le seguenti builts:

- ✚ VETLib\_base.lib [COFF]  
Libreria base, tutti i componenti che richiedono librerie esterne sono esclusi. (vetCodec\_XVID, vetCodec\_IMG, vetDirectXInput, vetDirectXInput2).
- ✚ VETLib\_im.lib [COFF]  
Libreria base e il modulo vetCodec\_IMG (libreria ImageMagick) (l’inclusione di questa libreria incrementa notevolmente le dimensioni, +10Mb).
- ✚ VETLib\_xvid.lib [COFF]  
Libreria base e il modulo vetCodec\_XVID (libreria xVidCore).
- ✚ VETLib\_dx.lib [COFF]  
Libreria base e i moduli legati a DirectX (vetDirectXInput, vetDirectXInput2).
- ✚ VETLib.lib, VETLib\_full\_vc6.lib [COFF]  
Libreria completa, sono inclusi tutti i moduli rilasciati.
- ✚ VETLib\_bcb.lib [OMF]  
Libreria base compilata con Borland C++ Builder 6.0, non è possibile includere i moduli citati per l’incompatibilità delle librerie statiche (problema OMF vs. COFF).

I binari per sistemi Linux (solo ./lib/VETLib.a) non include le librerie esterne ed è distribuita solo in una versione (*built*), il progetto finale deve impostare i corretti parametri del linker ed le librerie devono essere correttamente installate nel sistema, questo caso è analizzato nella prima parte del capitolo terzo, consultare il Makefile dei programmi dimostrativi (./tests/Makefile) per esempi pratici.

Non è attualmente disponibile alcuna versione dinamica della libreria (DLL), si prevede di rilasciare una serie di builts dinamiche dalla futura *Release 2.0*.

## 1.15 - Documentation

La documentazione di VETLib è essenzialmente divisa in tre insiemi: una serie di documenti preliminari, la documentazione delle classi ed alcuni manuali (incluso il presente), la lingua prediletta è l'inglese.

Le informazioni preliminari, cosiddetti *readme*, in formato ASCII (plain text) sono:

<code>./README</code>	:	Informazioni generali riguardo la libreria;
<code>./USE</code>	:	Come usare la libreria nelle proprie applicazioni;
<code>./COMPILE</code>	:	Come compilare la libreria;
<code>./EXTEND</code>	:	Come estendere la libreria;
<code>./FAQS</code>	:	Risposte alle domande più frequenti.
<code>./ChangeLog</code>	:	Lista degli aggiornamenti divisi per built;
<code>./lib/README</code>	:	Informazioni sui binari;
<code>./docs/README</code>	:	Informazioni sulla documentazione;
<code>./support/NOTES</code>	:	Note sulle librerie esterne (installazione, problemi riscontrati);
<code>./packages/README</code>	:	Informazioni sul Package Starter Kit;
<code>./packages/NAMESPACE</code>	:	Lista dei nomi riservati;
<code>./BUGS</code>	:	Lista dei problemi conosciuti;
<code>./TODO</code>	:	Lista delle estensioni e delle idee per il futuro;
<code>./AUTHORS</code>	:	Lista degli autori di VETLib e delle estensioni;
<code>./LICENSE</code>	:	Licenza di VETLib.

La documentazione, situata nella directory `./docs`, è generata semi-automaticamente nei formati più diffusi dallo strumento *DoxyGen* (freeware, incluso in `./Tools/DoxyGen`), il codice è documentato dallo sviluppatore direttamente nel file sorgente secondo uno standard semi-universale molto simile al Java, segue un esempio della descrizione di una funzione:

```
/**
 * @brief      Calculates optimal threshold value for input image.
 *
 * @param[in]  img VETLib Frame RGB24 to be processed
 * @param[out] threshold optimal threshold for input image
 *
 * @return     VETRET_PARAM_ERR if image is invalid, VETRET_OK else.
 *
 * @see       realAlgorithm()
 */
VETRESULT sampleFunction (vetFrameRGB24& img, uchar threshold) { [...] };
```

I *tags* riconosciuti sono auto-esplicativi e ampiamente descritti online, è anche possibile definire macro di compilazione della documentazione e personalizzare il progetto di generazione.

Questo sistema è largamente diffuso soprattutto per quanto riguarda librerie e SDK dedicati agli sviluppatori, eventuali modifiche e aggiornamenti del codice non implicano lunghe revisioni della documentazione finale, ma semplicemente la ricompilazione tramite il software (eventualmente la modifica del file di progetto), inoltre il sorgente vero e proprio è molto più chiaro e comprensibile grazie alle descrizioni integrate (inline).

La consultazione più diretta e integrata è sicuramente quella inclusa nei file header (e nei file sorgente) dei componenti stessi, per una navigazione rapida il formato consigliabile è senza dubbio l'HTML, gli altri formati disponibili sono XML, RTF, PDF, LaTeX, MAN (Linux Manual).

Gli indici generati da DoxyGen sono molto utili e di facile accesso, la pagina principale è `./docs/html/index.html`, essenzialmente raccoglie i collegamenti alle pagine indice e offre una navigazione attraverso *frames*.




















## 1.16 - VETLib Online


Questa prima release di VETLib è corredata anche da un sito web statico incluso nell'archivio completo e disponibile online sul mio server Linux (Apache 2) all'indirizzo:

<http://lnx.ewgate.net/vetlib>

Il sito è realizzato secondo gli standard XHTML 1.0 e CSS 1.0 del consorzio W3C, il browser consigliato è *Mozilla Firefox*, ma la visualizzazione è compatibile con tutti i software più diffusi, allo scopo di semplificare l'accesso locale e rendere le pagine indipendenti il sito non fa uso di *frames*. La struttura del sito è la seguente:

 html	Cascading Style Sheets file e immagini delle pagine web;
 distr	Copia online della libreria (non compressa, ultima release);
 documentation	Copia online della documentazione;
 downloads	Archivi disponibili (copiati da ./distr);
 screenshots	Screenshots di VETLib;
 tutorials	Articoli e manuali correlati;
 about.html	Informazioni riguardo il sito Web;
 documentation.html	Accesso alla documentazione online;
 downloads.html	Lista degli archivi disponibili per il download;
 extras.html	Lista dei files e degli archivi opzionali (video, tools);
 faqs.html	Risposte alle domande più frequenti;
 index.html	Indice, pagina principale;
 links.html	Collegamenti relativi a VETLib e al video processing;
 packages.html	Nuovi moduli non ancora inclusi nella libreria;
 sdk.html	Informazioni per gli sviluppatori;
 tutorials.html	Collegamenti (e downloads) di manuali;
 workshop.html	Pagina dedicata a software VETLib WorkShop.

In futuro si prevede un aggiornamento del sito in PHP5 e si auspica un server CVS per la distribuzione dei binari e del sorgente aggiornato.



*Per ogni problema complesso, c'è sempre una soluzione semplice.  
Ed è sbagliata.*

*George Luther Powell*

# VETLIB IMPLEMENTATION

## CHAPTER II

Nel capitolo precedente sono state presentate le classi principali che costituiscono il cuore della libreria, la corrente release include una serie di moduli che implementano le suddette classi e forniscono oggetti e metodi di comune utilizzo.

Le attuali categorie sono *buffers*, *inputs*, *math*, *outputs*, *filters*, *codecs* e *vision*, i moduli sono situati nelle sub-directories omonime in *./source/<category>/*, le classi astratte e i moduli più generici sono disponibili direttamente in *./source/*.

## 2.1 - Inputs

### ***vetNoiseGenerator***

È un implementazione estremamente banale di un generatore di rumore, offre una serie di metodi statici per la generazione di pixel casuali e la classica interfaccia di estrazione frames derivata da *vetInput*, ad ogni canale (RGB) è associato un valore casuale.

Questa versione non supporta le classiche informazioni (valor medio, potenza), i parametri di configurazione sono: normalizzazione (*pel\_value = rand()%NORM*), *spread* (numero di cicli di aggiunta rumore, *1.0 = image size*), la posizione del pixel rumoroso è casuale ed è possibile che un pixel sia aggiornato più volte.

Il modulo non è stato sviluppato per applicazioni reali, ma per alcuni test durante lo sviluppo di VETLib, i parametri configurabili e la risposta del generatore (fps) sono inefficienti, in futuro si prevede una versione più funzionale.

*Implements:* *vetInput*  
*Serializable*  
*/inputs/vetNoiseGenerator.h*

### ***vetPlainFrameGenerator***

Questo modulo genera frame monotematici, il codice è estremamente semplice e può essere analizzato dagli sviluppatori neofiti per comprendere l'implementazione di sorgenti dati in VETLib.

I pixel sono aggiornati direttamente nell'immagine output (passata come parametro alle funzioni *extractTo(vetFrame..)* ), non è stato implementato un buffer interno per velocizzare il processo, poiché è stato implementato principalmente per scopo didattico.

*Implements:* *vetInput*  
*Serializable*  
*/inputs/vetPlainFrameGenerator.h*

## vetVideo4Linux

Implements: vetInput  
for Linux only  
/inputs/vetVideo4Linux.h

Questa classe interfaccia VETLib al sistema di acquisizione dei sistemi NIX v4l (ver. 1), ovviamente è necessario che la libreria Video4Linux riconosca l'hardware collegato tramite i rispettivi driver, ciò è trasparente al modulo garantendo un ottimo livello di astrazione (simile a DirectX).

I test sono stati eseguiti con una Webcam Logitech Express USB, ottenendo una frame rate limitata solo dall'hardware (17 fps).

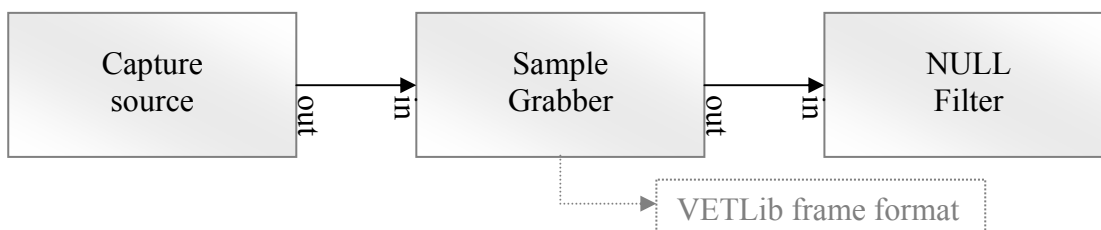
Le funzioni *connect(char\* inputFile = "/dev/video0")* e *disconnect()*, inizializzano e chiudono lo streaming (il device di acquisizione predefinito su Linux è appunto /dev/video0), le funzioni *get\*()* e *set\*()* permettono rispettivamente di leggere e modificare i principali parametri di lavoro (anche contrasto, saturazione, colore), l'estrazione dei dati è conforme allo standard VETLib tramite gli operatori di streaming e le classiche funzioni *extractTo(vetFrame..)*.

## vetDirectXInput

Implements: vetInput  
for Windows32 only,  
Integrated in WorkShop  
/codecs/vetDirectXInput.h

Tramite le API di DirectX questo modulo è in grado di enumerare i dispositivi di acquisizione disponibili e convertirne lo stream nel formato nativo di VETLib. A livello di compilazione è necessario disporre del DirectX SDK (versione 8 o 9, mentre la versione 10, attualmente in uscita sul mercato, non è compatibile con le precedenti), a livello di esecuzione il sistema deve disporre dei file di run-time (diffusi poiché utilizzati nelle applicazioni ludiche).

Le funzioni *enumerateDevices()* e *getDevice\*()* forniscono le informazioni su device di acquisizioni disponibili, tramite *connectTo(int)* e *disconnect()*, rispettivamente, si inizializza e si scioglie il *graph* di DirectShow (il sub-framework di DirectX destinato al video I/O) costruito come segue:



Sample Grabber è una classe interna al modulo, la risoluzione imposta è RGB 24 bit.

## vetDirectXInput2

Implements: vetInput  
for Windows32 only,  
Integrated in WorkShop  
/coders/vetDirectXInput2.h

Questo modulo è un'estensione complementare di *vetDirectXInput*, il grafo è molto più complesso e consente tre operazioni: visualizzare la preview (overlay mode), effettuare il rendering dello stream in formato non compresso (tutte le modalità supportate da DirectX), inoltre è possibile abilitare il grabbing (asincrono) dei frames che viene effettuato con un callback senza buffering (anche *vetDirectXInput* usa il callback ma questa versione ha prestazioni non paragonabili alla precedente).

Il componente è anche in grado di acquisire lo stream da device MPEG2 (ex. ADC con codificatore hardware) compatibili con DirectX (in questo caso il grafo è costruito manualmente).

## 2.2 - Outputs

### **vetWindowQT**

Questa classe interfaccia VETLib con la GUI di Linux (gnome, kde) tramite la libreria QT della Trolltech (gratuita, inclusa nella maggior parte delle distribuzioni di linux), dopo aver inizializzato l'istanza di un applicazione QT, è sufficiente indirizzare i frames tramite gli operatori di ingresso o i metodi *importFrom(vetFrame\_\*)* per visualizzarli in una finestra, la frame rate di lavoro è accettabile per fasi di testing anche nel caso di streaming video a bassa qualità/dimensione oltre che per immagini statiche. È stata testata con la classe *vetVideo4Linux* che si occupa dell'acquisizione e con i decodificatori video implementati, il risultato del re-indizzamento dei frames da una Webcam alla finestra è soddisfacente e limitato dall'hardware con una risposta di circa 15 frame al secondo; con una sorgente statica (da file) e decodifica (MPEG2 e QuickTime) in real time si è ottenuto una risposta di circa 33 frame al secondo (streaming video 320 x 240 x 24b).

*Implements:* *vetOutput*  
/outputs/vetWindowQT.h

### **vetWindowGTK**

Questo modulo, molto simile a *vetWindowQT*, permette di visualizzare immagini e video tramite la libreria GTK (gratuita, inclusa nella maggior parte delle distribuzioni di linux).

L'utilizzo della classe è banale, dopo aver creato un istanza, indirizzare i frames tramite gli operatori di streaming in ingresso oppure i metodi *importFrom(vetFrame\_\*)* per visualizzarli in una finestra, la creazione di una nuova applicazione nell'ambiente grafico è automatica (al contrario di *vetWindowQT*).

È stata testata con tutte le sorgenti dati di VETLib (su Linux), ha ottenuto risultati di poco migliori rispetto alla gemella *vetWindowQT* (34 fps).

*Implements:* *vetOutput*  
/outputs/vetWindowGTK.h

### **vetWindow32**

La classe *vetWindow32* consente la visualizzazione di frames anche con i sistemi operativi Windows a 32 bit, utilizza direttamente le API di Windows disegnando i pixel tramite le GDI. Questa soluzione non è assolutamente adatta a visualizzare video ma solo immagini statiche o alla fase di testing/debugging, per ottenere una performance paragonabile alle rispettive classi su Linux è necessario interfacciarsi a DirectX.

*Implements:* *vetOutput*  
for Widows32 only  
/outputs/vetWindow32.h

### **vetDoctor**

Durante l'implementazione di filtri o (de)codificatori è spesso necessario avere riscontri sul tempo di esecuzione, sulla frame rate ed eventualmente alcune statistiche comuni ricavate dai frames.

*vetDoctor* fornisce gli strumenti basilari per questi controlli e una serie di metodi statici multi-piattaforma per i casi più comuni.

*Implements:* *vetOutput*  
Integrated in WorkShop  
/outputs/vetDoctor.h

### **vetOutputVoid**

È una banale classe sviluppata per le fasi di test, non esegue alcuna operazione sui frame in ingresso ma ha un tempo di risposta che si avvicina a zero, può essere utile per misurare la frame rate ideale di un filtro o di una sorgente.

*Implements:* *vetOutput*  
/outputs/vetOutputVoid.h

## 2.3 - Codecs

### ***vetCodec\_BMP***

Questa classe interfaccia VETLib allo standard Bitmap, è in grado di leggere e scrivere file bmp, implementa le classi astratte vetInput (read) e vetOutput (write), il buffer interno è ereditato dalla classe vetFrameRGB e può essere disattivato per diminuire il tempo di esecuzione.

vetCodec\_BMP supporta sia in ingresso che in uscita l'accesso a file multipli, incrementando progressivamente il contatore che identifica il file (image13.bmp -> image14.bmp), ciò è particolarmente utile per automatizzare il salvataggio di sequenze.

*Implements:* vetCodec,  
vetFrameRGB  
Serializable  
/codecs/vetCodec\_BMP.h

### ***vetCodec\_IMG***

La libreria ImageMagick (gratuita, molto diffusa) consente a questo modulo di leggere e scrivere la maggior parte degli standard di codifica immagini esistenti (più di 90 formati diversi), il prezzo da pagare è la dimensione degli eseguibili che cresce drasticamente a causa della complessità di ImageMagick. Il modulo sarà aggiornato in futuro per diminuire questo inconveniente.

*Implements:* vetCoder  
Serializable  
/codecs/vetCoder\_BMP.h

### ***vetCodec\_MPEG***

Questo modulo utilizza la libreria libmpeg3 (open source) disponibile per ambienti Linux per decodificare lo standard MPEG 1-2.

Nonostante la codifica non sia abilitata si è scelto di implementare vetCodec invece di vetOutput.

Le prestazioni sono eccellenti e consentono largamente il playback tramite una delle interfacce di visualizzazione (vetWindow\*).

*Implements:* vetCodec  
for Linux only, Serializable  
/codecs/vetCodec\_MPEG.h

### ***vetCodec\_MOV***

Tramite la libreria quicktime4linux questa classe è in grado di decodificare e codificare stream in formato QuickTime (file .mov).

Sono stati riscontrati problemi con alcune versioni della libreria, si consiglia agli sviluppatori di installare la versione inclusa e leggere il file ./support/NOTES.

*Implements:* vetCodec  
for Linux only, Serializable  
/codecs/vetCodec\_MOV.h

### ***vetCodec\_XVID***

Lo standard open source MPEG4 più diffuso è comunemente noto come XVID, questo modulo sfrutta la libreria ufficiale xvidcore per codificare e decodificare lo stream.

Le prestazioni della decodifica sono paragonabili al modulo vetCodec\_MPEG, da sottolineare il fatto che la libreria è disponibile sia per ambienti Linux che Windows.

*Implements:* vetCodec  
Serializable  
Integrated in WorkShop  
/codecs/vetCodec\_XVID.h

## 2.4 - Filters

### ***vetFilterGeometric***

Questo filtro applica le operazioni geometriche più comuni sui frame in ingresso (rotazione, ridimensionamento, specchio), utilizza un buffer che può contenere un singolo frame, il buffer predefinito è `vetFrameCache` ma è possibile forzare l'utilizzo di un buffer a 24 bit per incrementare la frame rate. Sono disponibili anche una serie di metodi statici.

*Implements:* `vetFilter`  
`Serializable`  
`/filters/vetFilterGeometric.h`

### ***vetFilterColor***

Questo filtro applica le più comuni operazioni sul colore, utilizza un buffer che può contenere un singolo frame, il buffer predefinito è `vetFrameCache` ma è possibile forzare l'utilizzo di un buffer a 24 bit per incrementare la frame rate.

*Implements:* `vetFilter`  
`Serializable`  
`/filters/vetFilterColor.h`

### ***vetMultiplexer***

È l'implementazione software di un (de)multiplexer, la configurazione predefinita supporta 12 input e 12 output (il numero è definito come macro), è possibile selezionare la sorgente e la destinazione corrente ed eseguire cicli personalizzati di estrazione e caricamento di frames.

Attualmente non utilizza alcun threading.

*Implements:* `vetFilter`  
`Serializable`  
`/filters/vetMultiplexex.h`

### ***vetFilterNoiseChannel***

Questo modulo è l'implementazione di un canale non ideale, i frame in ingresso vengono perturbati con rumore (creato dal filtro o da una sorgente esterna) e memorizzati nel buffer (un frame) in attesa dell'estrazione.

*Implements:* `vetFilter`  
`Serializable`  
`/filters/vetFilterNoiseChannel.h`

## 2.5 - Buffers

### ***vetBufferArray***

Questa classe è un'implementazione della classica struttura dati basata su array, può contenere oggetti di qualunque tipo poiché è una classe template. La dimensione dell'array viene incrementata automaticamente quando necessario, è il buffer più indicato per medio-piccole quantità di oggetti.

*Implements:* `vetBuffer`  
Template Class [Data Structure]  
available as WorkShop PlugIn  
`/buffers/vetBufferArray.h`

### ***vetBufferLink***

Il modulo implementa i comuni metodi di `vetBuffer` con una struttura dati di tipo *Double Linked List*, la gestione dei frame è ottimizzata e solo l'accesso ai frame tramite indice implica una parziale scansione della lista (viene cominciata dalla testa, dalla coda oppure dal frame corrente in base alla distanza minima), il numero di oggetti massimo è indipendente dal modulo.

*Implements:* `vetBuffer`  
Template Class [Data Structure]  
available as WorkShop PlugIn  
`/buffers/vetBufferLink.h`



## 2.6 - Other Modules

### **vetHist**

Questo modulo facilita il calcolo dell'istogramma di immagini (o dati), vengono memorizzate le frequenze e successivamente le probabilità dei valori inseriti (0-255), è anche possibile serializzare i dati e realizzare un grafico, ovviamente per immagini multi-canale l'operazione va eseguita più volte.

for Windows32 only,  
Serializable  
Integrated in  
WorkShop  
/vetHist.h

### **vetThread**

Questo modulo fornisce un threading di alto livello indipendente dal sistema operativo, l'interfaccia base è comune alle due diverse implementazioni (Windows 32 e \*NIX) e quindi multi-piattaforma, ma sono disponibili anche metodi e caratteristiche legate al sistema operativo (ovviamente queste non sono condivise).

Different  
Implementations  
for Windows32, Linux  
/vetThread.h

### **vetUtility**

Questo modulo incorpora una serie di metodi comuni dichiarati *static* (accessibili anche senza un'istanza della classe: *vetUtility::method()* ), in particolare sono implementate alcune funzioni dedicate alla gestione del tempo e alcune conversioni di colore classiche, la maggior parte dei metodi accetta parametri *template* al fine di garantire un'alta elasticità ed ottimizzare gli algoritmi.

/vetUtility.h

### **vetMatrix**

Questa classe *template* memorizza una matrice bidimensionale di oggetti, è possibile serializzare i dati in formato XML (i valori sono salvati come floating point). L'utilizzo più probabile per questa classe è la memorizzazione di kernel per operazioni matematiche, per questo scopo la classe aggiorna automaticamente il valore somma (normalizzazione).

Serializable  
./vetMatrix.h

### **vetDFMatrix**

Questa classe è stata implementata per i filtri digitali, rappresenta una matrice  $N \times N + 1$  di valori *char* (-127÷127), l'ultimo valore è il divisore della normalizzazione, la dimensione è gestita dinamicamente, eventuali ottimizzazioni possono sfruttare il metodo *dup\_data()* che restituisce il buffer interno.

Sono stati integrati in maniera statica una serie di kernel predefiniti (3x3) accessibili tramite il metodo statico *vetDFMatrix\* createKernel\_3x3(int index)*, il parametro *index* indica il codice univoco che identifica il kernel richiesto, i codici sono definiti con un tag user-friendly all'inizio dell'header (*VETDF\_3x3\_<name>*), mentre i dati dei kernel (valori della matrice, *char*) sono definiti nel file *./source/filters/vetDigitalFilters.def*.

Serializable  
./vetDFMatrix.h

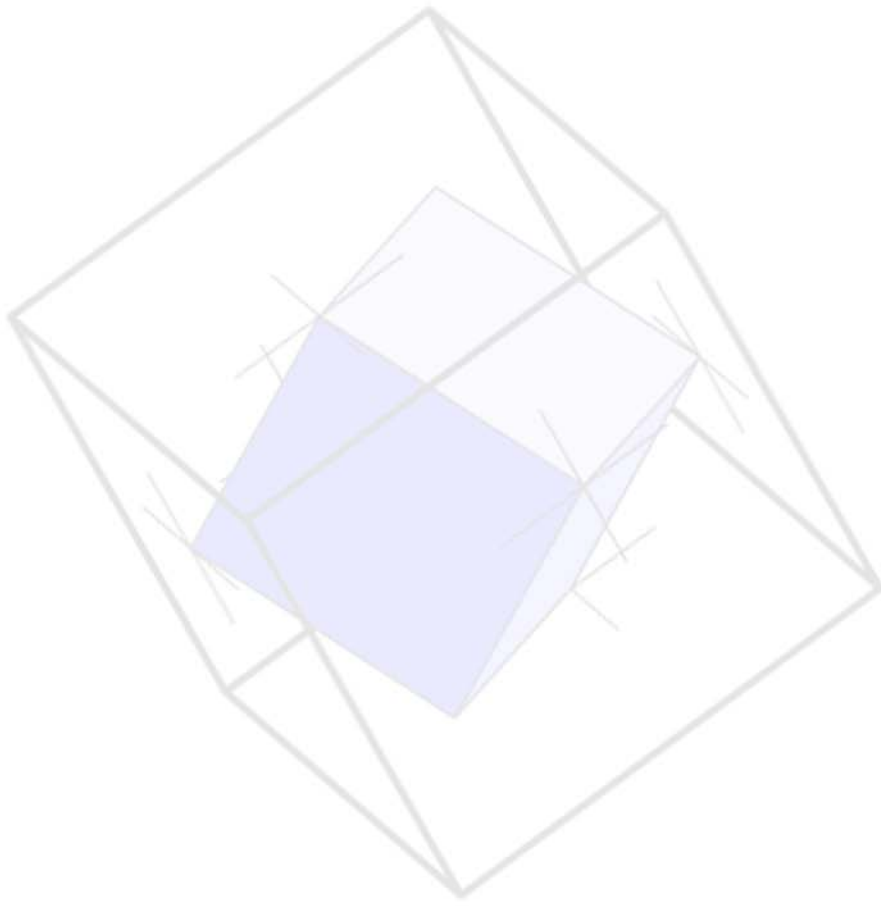
## 2.7 - Vision

### ***vetMotionLame***

Questo modulo è stato sviluppato per dimostrare l'utilizzo della classe base *vetMotion*, l'algoritmo è una banale *Frame Difference* con soglia, la risposta predefinita dei moduli *vision* consiste nella chiamata di una funzione data (callback), gli esempi dimostrano il funzionamento sia in ambiente Windows che \*NIX.

*Implements:* *vetMotion*  
*Serializable*

*/vision/vetMotionLame.h*



F  
I  
L  
L  
T  
R  
S

PACKAGES

*Everything should be made as simple as possible,  
but not simpler.*

*Albert Einstein*

# USING AND EXTENDING VETLIB

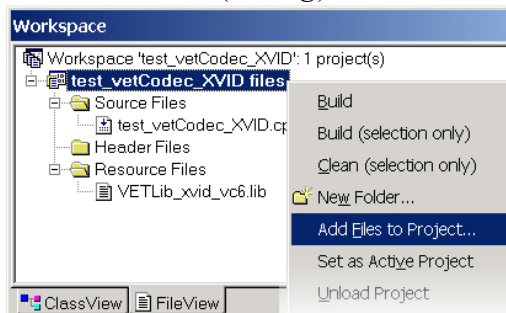
## CHAPTER III

Questo capitolo è focalizzato sull'utilizzo ed in particolare sull'estensione della libreria, le interfacce e gli oggetti principali sono stati presentati nel capitolo primo, la seguente analisi assume che il lettore conosca queste informazioni, mentre alcune, le più importanti, saranno ripetute ed approfondite con esempi pratici, il lettore "frettoloso" dovrebbe quantomeno analizzare gli headers degli oggetti principali prima di procedere.

### 3.1 - Overview

I corsi di programmazione spesso non approfondiscono uno degli aspetti più importanti della programmazione odierna: l'utilizzo di librerie, tutti i software complessi sono basati su una serie di librerie statiche e/o dinamiche più o meno standard e complesse.

La libreria VETLib non è differente dai casi classici, si tratta di includere (*linking*) i binari statici (*.lib* o *.a*) nella linea di comando del linker e ovviamente di includere gli headers necessari nei file sorgente, gli sviluppatori Windows sono avvantaggiati da due aspetti: le interfacce grafiche (GUI) dei software di sviluppo (*Visual Studio*, *Borland Builder*, ..) consentono di aggiungere la libreria in modo semplice e visuale (generalmente con il comando "Add to Project"), inoltre la libreria *VETLib\*.lib* ingloba anche le librerie di supporto necessarie, si tratta semplicemente di includere la versione preferita della libreria (vedi capitolo primo sezione *Builts*).



Il caso di piattaforme NIX è apparentemente differente, ove richiesto si devono aggiungere anche le librerie esterne richieste dal proprio progetto e dai moduli inclusi in VETLib, il comando di compilazione (e linking) più semplice, dove non sono richieste altre librerie, è di questo tipo:

```
g++ myMain.cpp ../lib/VETLib.a -o myOut.out
```

quando sono necessarie librerie esterne (MPEG, XVID, MOV, V4L, ..) il comando si complica ed è necessario consultare la documentazione delle singole librerie per i parametri di compilazione, ad esempio il programma dimostrativo *vetLinuxMPEGPlayerGTK*, incluso nella distribuzione e posizionato in *./tests/* necessita dei seguenti parametri:

```
g++ app_vetLinuxMPEGPlayerGTK.cpp ../lib/VETLib.a
-L/usr/lib/ -lpthread -lmpeg3
-o app_vetLinuxMPEGPlayerGTK.out
`pkg-config --cflags --libs gtk+-2.0`
```

Questo semplice programma è basato su due moduli: *vetCodec\_MPEG* e *vetWindowGTK*, la seconda e l'ultima linea di parametri include le librerie esterne necessarie: *libmpeg3* e *gtk2*, in pratica il linking viene fatto su due livelli: il primo dall'oggetto del sorgente principale (contenente

la funzione *main*, in questo caso *app\_vetLinuxMPEGPlayerGTK.cpp*) che fa riferimento a metodi contenuti in VETLib (*./lib/VETLib.a*), a sua volta i riferimenti non risolti dalla libreria (ma indirettamente richiesti dal progetto) sono risolti nelle librerie esterne passate come parametro al linker, il seguente frammento chiarisce il concetto:

```
#include "myReader.h"

int main(...)
{
    class myReader;
    myReader->read();
}
```

main.cpp

```
class myReader
{
    myReader();
    ~myReader();
    int read();
}
```

myReader.h

```
#include "myReader.h"
#include "../mySupportLibPath/RawReadLibrary.h"

//[..]
int myReader::read()
{
    file_open_thread(file, access);
    //..a method from external library
}
```

myReader.cpp (included as object in myReader.a)

```
g++ main.cpp myReader.a
-L/usr/lib/ -lRawReadLibrary
-o main.out
```

In questo caso si suppone che la libreria esterna *RawReadLibrary* sia disponibile nella directory */usr/lib/* e che sia conforme allo stile dei nomi classico (il file deve essere *libRawReadLibrary.a*).

Si è scelto di non includere le librerie esterne direttamente nella libreria per favorire l'indipendenza degli aggiornamenti e minimizzare la dimensione dei binari, può inizialmente sembrare una scelta controversa per i neofiti (sia della libreria che di Linux), ma tutte le distribuzioni software seguono questa prassi (basta considerare le dimensioni di librerie come *GTK* e *v4l*).

Consultare la sezione "External Libraries" nel file testo *./USE* ed eventualmente le note aggiuntive nel file *./support/NOTES* per maggiori informazioni, per esempi pratici visualizzare il Makefile della libreria (*./Makefile*) e quello dei tests (*./tests/Makefile*).

La documentazione di VETLib (i file *readme* sono inclusi in tutti gli archivi) è stata descritta nel capitolo primo, in caso di contraddizioni (a causa di aggiornamenti ad esempio) fare sempre riferimento ai file di testo, in particolare *./ChangeLog* elenca tutti gli aggiornamenti effettuati divisi per versione, gli aggiornamenti che rompono la continuità con i vecchi oggetti sono segnalati con un punto esclamativo, se si decide di aggiornare la versione inclusa nel proprio progetto con una più recente (banalmente si sostituisce il file statico e gli headers) occorre prestare attenzione a tutte le

possibili implicazioni, gli header di ogni classe aggiornata descrivono la natura dei cambiamenti e le ripercussioni pratiche.

Si consiglia di implementare i nuovi moduli partendo dalla libreria statica precompilata (eventualmente una *special built*), questo è anche il sistema predefinito dello strumento *VETLib Package Studio* (presentato più avanti in questo capitolo), tuttavia è ovviamente possibile compilare la libreria sul proprio sistema e aggiungere il nuovo modulo direttamente al progetto, in questo caso per testare il componente è necessario creare un nuovo progetto di test simile a quelli forniti in */tests/* (ovviamente il progetto di creazione della libreria non ha una funzione di sistema come *int main()* e non costruisce un eseguibile ma un oggetto statico).

La libreria include un'applicazione dimostrativa sviluppata in C++ .NET: *VETLib Workshop*, si tratta di un software completo, visuale, per ambienti Windows 32 (o superiori), realizzato per dimostrare le capacità della libreria e l'inclusione della stessa in un progetto complesso. *WorkShop* è in grado di gestire dinamicamente catene di filtri, consente di modificare i parametri in maniera visuale e veloce, gli oggetti possono essere caricati come *plugins* (DLL), è un ottimo laboratorio per il test preliminare e dimostrazioni didattiche nel campo del video/image processing. Il capitolo quarto è focalizzato su questo software e sulla creazione dei *plugins*.

La licenza di distribuzione del progetto (disponibile in appendice) è la classica *General Public License* che consente l'estensione e il riutilizzo del codice in contesto open source (mantenendo i crediti), ma limita l'integrazione in prodotti commerciali previa diretta autorizzazione.

Gli sviluppatori interessati all'estensione di *VETLib* devono accettare e confermare questa licenza, qualora nel progetto siano incluse librerie esterne, anch'esse devono essere *open source*, la convenzione prevede di precisare note o licenze diverse nel file *<ClassName>.License*.

## 3.2 - Samples

I componenti (alias moduli) della libreria sono corredati di un progetto di test e dimostrazione delle caratteristiche situato nella directory `./tests/`, il sorgente di tali applicazioni può essere estremamente utile per introdurre l'utilizzo di un oggetto nel proprio software, fare riferimento all'appendice per ulteriori informazioni sui progetti inclusi in questa release.

Seguono alcuni estratti utili per comprendere le potenzialità e la semplicità del framework VETLib:

```
#include "../source/vetFrameRGB24.h"
#include "../source/outputs/vetWindowGTK.h"
#include "../source/codecs/vetCodec_MPEG.h"

int main(int argc, char* argv[]) {
    int i = 0;
    long time = 0;
    float fps = 0 ;
    vetCodec_MPEG mpegSource;

    int ret = mpegSource.load("football.mpg");

    // Video StreamS Count:          mpegSource.getVideoStreamCount()
    // Video Stream [0] Frame Rate:   mpegSource.getVideoFrameRate()
    // Video Stream [0] Frame Count:   mpegSource.getVideoStreamLength()
    // Video Stream [0] Width:         mpegSource.getWidth()
    // Video Stream [0] Height:        mpegSource.getHeight()
    // Audio StreamS Count:            mpegSource.getAudioStreamCount()
    // Audio Stream [0] Channels:       mpegSource.getAudioChannels()
    // Audio Stream [0] Sample Rate:    mpegSource.getAudioSampleRate()
    // Audio Stream [0] Sample Count:   mpegSource.getAudioStreamLength()

    vetWindowGTK myWin (mpegSource.getWidth(), mpegSource.getHeight());
    vetFrameRGB24 img24 (mpegSource.getWidth(), mpegSource.getHeight());

    myWin.show();

    long sleeptime = (long) (1000 / mpegSource.getVideoFrameRate()) - 10;
    while (i++ < 100)// 100 frames
    {
        offset = vetUtility::getTime_usec();

        mpegSource >> img24;
        myWin << img24;

        vetUtility::vetSleep(
            sleeptime - (long)(vetUtility::getTime_usec()-offset)/1000 );
    }

    return 0;
}
```

app\_vetLinuxMPEGPlayerGTK.cpp

La libreria di supporto *libmpeg3* è disponibile solo per piattaforme \*NIX, i parametri per la compilazione sono:

```
g++ app_vetLinuxMPEGPlayerGTK.cpp ../lib/VETLib.a
-L/usr/lib/ -lpthread -lmpeg3
-o app_vetLinuxMPEGPlayerGTK.out
`pkg-config --cflags --libs gtk+-2.0`
```

```

vetVideo4Linux cap("/dev/video0");
vetFrameRGB24 img24;

// Stream Width:          cap.getWidth()
// Stream Height:         cap.getHeight()
// Stream Color Depth:    cap.getColorDepth()
// Stream Palette:        cap.getPalette()
// cap.setBrightness( value );
// cap.setContrast( value );
// cap.setHue( value );

QApplication app(argc, argv);
vetWindowQT *myapp = new vetWindowQT(cap.getWidth(), cap.getHeight() );
myapp->show();
app.setMainWidget(myapp);

int i = 0;
while (i++ < 100)
{
    cap >> img24;
    *myapp << img24;
}

```

app\_vetVideo4LinuxPlayer.cpp

Anche in questo caso il progetto è disponibile solo per piattaforme \*NIX e si basa sulle librerie di acquisizione *v4l* (video4linux) e di visualizzazione *QT*, la compilazione è molto semplice, basta includere la libreria:

```

g++ -g -Wall -O3 test_vetVideo4Linux.cpp ../lib/VETLib.a
-o test_vetVideo4Linux.out

```

```

vetDirectXInput cap;
cap.enumerateDevices();

printf("Devices Count: %d\n", cap.getDeviceCount() );

for (int i=0; i<cap.getDeviceCount(); i++)
    printf(" Device #%d: %s\n", i, cap.getDeviceDescription(i) );

int myDev = 0;
printf("\nConnecting to Device #%d...\n", myDev);
int ret = cap.connectTo(myDev);
if (ret) {
    printf("CANNOT connect to device : %d [ret=%d]\n", myDev, ret);
    return 1;
}
else
    printf("connected to device : %d\n", myDev);

int f, w, h;
char fdesc[255];
cap.getImageSize(&w, &h);
f = cap.getFormat();
cap.getFormat(f, &w, &h, fdesc);

printf("Device INFO:\n");
printf(" Device Width: %d\n", w );
printf(" Device Height: %d\n", h );
printf(" Device Format: %d\n", f );
printf(" Device Format Description: %s\n", fdesc );

```

test\_vetDirectXInput.cpp



Nel seguente esempio si sfrutta il metodo *dumb\_buffer\_RGB()* per eliminare le operazioni di copia, ammettendo che le classi *dataSource* e *visualizationWindow* non eseguano alcun buffering (verosimile), in questo ciclo non vi è alcuna operazione superflua, i dati sono elaborati e salvati in un'altra zona della memoria (già allocata) ad ogni passaggio:

```

unsigned int    w, h;
vetFrameRGB24* globalBuffer;
vetInput*      dataSource;
vetFilter*     imageFlipper;
vetFilter*     imageMirror;
vetOutput*     visualizationWindow;

w = dataSource->getWidth();
h = dataSource->getHeight();
globalBuffer->reAllocCanvas(w, h);
imageFlipper->useBufferRGB(w, h);
imageMirror->useBufferRGB(w, h);
visualizationWindow->setWidth(w);
visualizationWindow->setHeight(h);

while ( !dataSource->EoF() && (i++ < iMaxLoop) )
{
    dataSource->extractTo( *globalBuffer );
    imageFlipper->importFrom( *globalBuffer );
    imageMirror->importFrom( *imageFlipper->dumb_buffer_RGB() );
    visualizationWindow->importFrom( *imageMirror->dumb_buffer_RGB() );
}

```

Il sistema di estrazione classico dei filtri (*extractTo*) copia semplicemente il buffer interno al filtro nel buffer globale: *imageFlipper->extractTo(globalBuffer)*, ma i dati sono già disponibili nel buffer interno e quindi si tratta di un passaggio deprecabile che allunga il tempo di esecuzione, chiaramente il vantaggio consiste nell'astrazione e nella gestione di errori. Tramite l'accesso diretto al buffer è possibile eliminare questa operazione e guadagnare qualche decina di millisecondi per ogni frame, il codice risulta più "pericoloso" ed è consigliabile solo in applicazioni real-time.

Un'ultima nota sul controllo del flusso, in questo caso si suppone che *dataSource* implementi adeguatamente il prototipo *End Of File*, il secondo controllo deve essere sempre aggiunto per garantire che il ciclo non sia eterno (ad esempio se *dataSource* fosse *vetVideo4Linux* o *vetDirectXInput* l'unica causa che può fermare il loop è disconnettere il *device* via hardware!).

Lo sviluppatore deve accertarsi che i filtri siano compatibili con il formato scelto e preoccuparsi di eventuali conversioni dello spazio colore, inoltre i buffers dovrebbero essere inizializzati prima del ciclo per non intaccare il tempo di esecuzione sul primo frame (si suppone che il filtro adatti automaticamente il buffer al formato in ingresso).

Il tempo di processing, tipicamente variabile, può essere reso costante attraverso un buffer *FIFO* (First in First Out) posizionato al termine della catena.

### 3.3 - Tools of the Trade

VETLib è sviluppata in ANSI C++, quindi è necessario un qualunque compilatore che rispetti questo standard, lo sviluppo è stato testato con *Borland C++ Builder 6.0* e *Microsoft Visual Studio 6.0* su ambienti Windows e il classico *GNU Compiler* su Linux, i progetti per compilare la libreria e i tests (sample applications) per i compilatori citati sono disponibili.

Suggerisco agli sviluppatori Windows interessati a compilare la libreria in locale di utilizzare Visual Studio per problemi di compatibilità soprattutto con le librerie esterne (*DirectX*, *xvidcore*, *ImageMagick*), la versione 6.0 necessita di alcuni aggiornamenti disponibili in *./Tools*, (consultare il file di testo *./support/NOTES* per informazioni dettagliate).

Gli sviluppatori \*NIX devono installare le librerie esterne necessarie al proprio software, la soluzione più pratica è scaricare ed installare i pacchetti precompilati per il proprio sistema, ovviamente si preferisce la compilazione locale. (note in *./support/NOTES*)

Come descritto nella prima sezione, sia l'utilizzo che l'estensione della libreria non richiedono la compilazione in locale, in questo caso gli unici files necessari sono i binari compilati (*.a* o *.lib*) e gli headers, sono disponibili numerosi archivi complementari sia per piattaforma Windows che \*NIX, si possono ottenere dal sito di VETLib, è consigliabile scaricare la distribuzione completa più recente (VETLib-x.x.x.zip/tar.gz), altrimenti si può scaricare uno degli archivi di categoria SDK, destinati allo sviluppo di applicazioni basate su VETLib (applicazioni per l'utente finale, altre librerie, estensioni di VETLib):

- |                                  |  |
|----------------------------------|--|
| ✓ VETLib-SDK-x.x.x.tar.gz (.zip) | Completo                               |
| ✓ VETLib-SDK-LNX-x.x.x.tar.gz    | Solo i progetti e i moduli per Linux   |
| ✓ VETLib-SDK-WIN-x.x.x.zip       | Solo i progetti e i moduli per Windows |

Inoltre è disponibile l'archivio VETLib Package Starter Kit (VETLib-PSK-x.x.x.zip) destinato esclusivamente allo sviluppo di nuovi moduli, contiene:

- |                                       |   |
|---------------------------------------|---|
| ✓ Moduli Vuoti                        | <i>./packages/&lt;prefix&gt;Empty/</i>  |
| ✓ Moduli Esempio                      | <i>./packages/&lt;prefix&gt;Sample/</i> |
| ✓ Il Tool Package Studio              | <i>./packages/vetps.exe</i>             |
| ✓ Moduli Template necessari per vetPS | <i>./packages/templates/</i>            |

Il contenuto e l'utilizzo dell'archivio Package Starter Kit saranno analizzati nel dettaglio più avanti in questo capitolo. Consultare la pagina *downloads.html* per la lista dei file di ogni archivio.

## 3.4 - VETLib Component Conventions

Gli sviluppatori che vogliono estendere la libreria con nuovi moduli devono seguire le seguenti convenzioni ideate per mantenere VETLib uniforme, chiara e funzionale:

- Il nome della classe (cioè del modulo) deve essere formattato come segue:

*<prefix>ModuleName.h (.cpp)*

Il prefisso è il nome della classe base (ex. *vetFilterColor*, ..)

Il nome del file header e sorgente deve essere identico alla classe (come Java)

I nomi riservati (non disponibili) sono elencati nel file *./packages/NAMESPACE*

- Qualora siano necessarie più classi, dichiararle nell'header se utili all'utente finale, nel file sorgente se sono interne al modulo (quindi sono invisibili all'utente che include gli headers). Soprattutto nel caso di librerie esterne includere gli headers nel sorgente per massimizzare la trasparenza.
- I moduli aggiuntivi devono essere situati in *./packages/<ClassName>*, quando completi possono essere inviati ad uno sviluppatore designato che può decidere di includerli nella successiva distribuzione.
- Documentare il codice secondo lo standard utilizzato (doxygen) (sia header che possibilmente il sorgente), commentare i passaggi più complessi chiaramente. La documentazione (e i commenti) devono essere in lingua inglese.
- Se applicabile, seguire lo standard adottato per la gestione dei parametri (*vet<..>Parameters*) e implementare I/O per serializzare la classe in formato XML. (Ex. *vetFilter*)
- Scrivere un programma per il test/debugging e per la dimostrazione dei metodi e delle caratteristiche del modulo, nominarlo *test\_<ClassName>.cpp*, per programmi dimostrativi più complessi formattare il nome come *app\_<ClassName>.cpp*.
- Dichiarare le variabili interne *protected* (non pubbliche o private), ciò garantisce un corretto stile nell'interazione con gli altri moduli e la facilità di estensione in futuro.
- La maggior parte delle funzioni deve restituire il tipo *VETRESULT* (definito come intero) che indica il risultato del processo, i valore di ritorno possibili sono definiti in *vetDefs.h* (es. *VETRET\_OK*), ricordo che la convenzione è comunque:  $0 \rightarrow \text{OK}$ , errore altrimenti.
- Se si desidera implementare un plugin di *VETLib WorkShop* è comunque necessario sviluppare prima un modulo funzionante e solo successivamente convertirlo, il capitolo successivo è focalizzato sul funzionamento e sulla creazione dei plugins.

*I moduli non coerenti con queste specifiche non saranno presi in considerazione*

## 3.5 - Package Development







Lo sviluppo di componenti (alias *moduli*, *packages*), ovvero l'estensione della libreria, prevede di norma l'implementazione di una delle interfacce base:

- vetInput (.h) (data server) (components in ./inputs)
- vetOutput (.h) (data client) (components in ./outputs)
- vetFilter (.h) (data bridge) (components in ./filters)
- vetCodec (.h) (data bridge) (components in ./codecs)
- vetVision (.h) (data client) (components in ./vision)
- vetBuffer (.h) (data storage) (components in ./buffers)
- vetFrame (.h) (frame object) (components in ./)
- vetObject (.h) (general object) (components in ./)

I componenti che forniscono algoritmi e operazioni matematiche non hanno restrizioni e sono situati in *./source/math*.

La sintassi (codice) non si discosta dalla classica ereditarietà e composizione di classi del C++, la struttura modulare del framework semplifica al massimo questo processo, gli sviluppatori che vogliono proporre una nuova "categoria" di moduli possono contattare [vetlib@ewgate.net](mailto:vetlib@ewgate.net).

La fase preliminare prevede una serie di operazioni che possono essere automatizzate con lo strumento *vetPS.exe* incluso nel *Package Starter Kit*, il software sarà analizzato più avanti in questo capitolo. Un modulo standard comprende i seguenti files:

-  <ClassName>.h
-  <ClassName>.cpp
-  test\_<ClassName>.cpp
-  Makefile / BCB-MVC project files {...}
-  <ClassName>.Readme
-  <ClassName>.License (must be open source)

I primi due file sono il cuore del progetto, contengono rispettivamente la dichiarazione del componente e l'implementazione dei metodi, ovviamente non è presente una funzione *main* nel file sorgente, quindi per semplificare (velocizzare) lo sviluppo è stato incluso il file *test\_vetClassName.cpp* che permette di testare direttamente il componente e i suoi metodi.

La convenzione (del *Package Studio*) prevede la seguente gerarchia:

- any directory (VETLib root)
  - /lib/ {VETLib.a, VETLib.lib, ..}
  - /source/ {buffers, inputs, filters, math, ..}
  - /packages/
    - <ClassName>/

La directory *lib* contiene i binari statici della libreria (posizione legata ai riferimenti dei progetti predefiniti), la directory *source* contiene gli headers dei componenti rilasciati e delle interfacce principali (contiene anche il sorgente ma non è necessario), la directory *packages* contiene il vostro progetto (i files citati sopra).

Questa struttura delle directories offre due vantaggi: consente all'utente di sviluppare più componenti in cartelle separate e garantisce che l'inclusione degli headers (usare *relative paths*) sia valida anche spostando i due file principali (*vetClassName.h, cpp*) nelle sottodirectory di *source* (fa parte del processo di rilascio dei componenti).

### 3.5.1 - Working with Frames

Questa sezione presenta una serie di suggerimenti e consigli pratici relativi all'accesso a oggetti frame, alcuni potranno sembrare banali ma è utile sottolinearli per gli sviluppatori neofiti.

Innanzitutto è opportuno prestare molta attenzione all'accesso diretto ai buffer (dichiarati pubblici), soprattutto nei cicli, il puntatore deve essere sempre copiato per non modificare l'originale:

```
class vetFrameSample { int width; int height; unsigned char* data; };
void BADcutValue (vetFrameSample& img, unsigned char value)
{
    for (int i=0; i<img.height*img.witdh*3; i++)
    {
        if ( *img.data > value )
            *img.data = value;
            img.data++;
    }
}
void OK1cutValue (vetFrameSample& img, unsigned char value)
{
    unsigned char* dataPtr = img.data;
    for (int i=0; i<img.height*img.witdh*3; i++)
    {
        if ( *dataPtr > value )
            *dataPtr = value;
            dataPtr++;
    }
}
```

Il problema del primo metodo è che il puntatore *img.data* viene modificato e punterà al valore blu dell'ultimo pixel perdendo l'indirizzo fisico del primo pixel, la seconda funzione invece modifica la variabile temporanea *dataPtr* e non l'originale, anche l'accesso tramite l'operatore *[ ]* è corretto:

```
void OK2cutValue (vetFrameSample& img, unsigned char value)
{
    for (int i=0; i<img.height*img.witdh*3; i++)
    {
        if ( img.data[i] > value )
            img.data[i] = value;
    }
}
```

Di fatto il compilatore genera lo stesso *assembly* per le due funzioni corrette.

Alcune funzioni fornite dalla libreria standard (C++) possono essere molto utili per ottimizzare gli algoritmi, ad esempio la seguente implementazione dell'operatore di streaming estratto da *vetFrameGrey.cpp* ottimizza il processo di copia grazie alle funzione *memset* e *memcpy*:

```
vetFrameGrey& vetFrameGrey::operator >> (vetFrameYUV420& img)
{
    if ( width == 0 || height == 0 )
        throw "Cannot do that with empty image (no size)";

    if ( width != img.width || height != img.height)
        img.reAllocCanvas(width, height);

    memcpy(img.data, data, width * height);
    memset(img.U, '\0', width * height / 2); // u+v set to 0

    return *this;
}
```

Il precedente codice è di fatto valido poiché entrambi i buffer sono array di *unsigned char*, mentre il codice non ottimizzato sarebbe stato:

```
vetFrameGrey& vetFrameGrey::operator >> (vetFrameYUV420& img)
{
    if ( width == 0 || height == 0 )
        throw "Cannot do that with empty image (no size)";

    if ( width != img.width || height != img.height)
        img.reAllocCanvas(width, height);

    img.setBlack();
    const unsigned int size = width * height;
    for (unsigned int i=0; i < size; i++)
        img.data[i] = data[i];          // implicit cast

    return *this;
}
```

Questo metodo sarebbe valido anche se *img.data* (cioè il buffer YUV) fosse di interi o floating point, ma il ciclo for è estremamente più lento della copia diretta.

I controlli sono molto importanti per non incorrere in errori fatali (come l'accesso non valido alla memoria), questo frammento dimostra un errore comune:

```
char* buffer = new char[255];
// [...]
if (value)
    delete buffer;    // error, should be '{ delete buffer; buffer = NULL; }'
// [...]
if ( buffer == NULL ) // if ( buffer == 0 ) || if (!buffer)
{
    // never executed !
    buffer = new char[255];
}
```

La versione corretta imposta il buffer a *NULL* dopo aver liberato la memoria, questo errore è comune soprattutto se non si inizializzano i puntatori dopo la dichiarazione, il valore associato è di fatto casuale, diverso da *NULL*, quindi i controlli tentano di cancellare una zona di memoria non allocata:

```
class badAccess {
    char* buffer;
    badAccess() {
        // correct code: initialize pointer here (buffer = NULL)
        reset();
    }
    void reset()
    {
        // [...] width = 0; height = 0; .....
        if ( buffer != NULL )
            delete buffer;

        buffer = new char[256];
    }
}
```

Questo problema è molto comune quando si tenta di delegare ad un'unica funzione (*reset*) l'inizializzazione, la versione proposta (corretta) è valida anche dal punto di vista dello stile.

### 3.5.2 - Internal buffering

Il sistema di buffering predefinito, presentato nel capitolo primo (sezione *vetFilter*), semplifica incredibilmente lo sviluppo di filtri, questa sezione sottolinea alcuni aspetti importanti e presenta una serie di esempi pratici.

I componenti di tipo *vetInput* e *vetOutput* in generale non prevedono il buffering interno dei dati, ma si utilizza direttamente il frame in uscita o ingresso (rispettivamente) come sorgente o destinazione, ad esempio:

```
VETRESULT vetVideo4Linux::extractTo(vetFrameT<unsigned char>& img)
{
    if ( fd == -1 )
        return VETRET_ILLEGAL_USE;

    if (win.width != img.width || win.height != img.height)
        img.reAllocCanvas(win.width, win.height);

    if ( (vpic.palette == VIDEO_PALETTE_RGB24  &&
         img.profile == vetFrame::VETFRAME_BGR24  ) {

        read( fd, img.data[0], //it's a BGR source not RGB..
             win.width * win.height * 3 * sizeof(unsigned char) );

        return VETRET_OK;
    }
    // [...]
}

VETRESULT vetWindowGTK::importFrom(vetFrameRGB24& img)
{
    if (image == NULL)
        return VETRET_INTERNAL_ERR;

    gdk_draw_rgb_image(    image->window, image->style->fg_gc[image->state],
                          0, 0, img.width, img.height,
                          currDith,
                          (guchar*)img.data[0],
                          img.width*3          );

    return VETRET_OK;
}
```

I componenti derivati da *vetCodec* sono generalmente basati su librerie esterne che gestiscono il buffering internamente, a causa della stretta dipendenza con la libreria di (de)codifica questi componenti non hanno alcuna imposizione (o supporto) riguardo al buffering, inoltre il formato dati dello stream può non essere direttamente compatibile con alcun oggetto frame base, ad esempio:

```
VETRESULT vetCodec_MPEG::extractTo(vetFrameRGB24& img)
{
    if (file == NULL)
        return VETRET_ILLEGAL_USE;

    if (width != img.width || height != img.height)
        img.reAllocCanvas(width, height);

    mpeg3_read_frame( file, buff,
                    0, 0, width, height, width, height,
                    MPEG3_RGB888, 0);
    vetUtility::conv_rgb24_rgb96( buff, (unsigned char*)img.data[0],
                                width, height);

    return VETRET_OK;
}
```

Il caso dei filtri (derivati da *vetFilter*) è differente, un frame viene importato tramite la funzione *vetFilter::importFrom(vetFrame..)* che si occupa (verosimilmente attraverso altri metodi) di elaborare il frame e poi memorizza temporaneamente il risultato in un buffer, quando l'applicazione chiama il metodo *vetFilter::extractTo(vetFrame..)*, il frame modificato (temporaneo, nel buffer) viene semplicemente copiato nel frame output, il flusso dati descritto è classico ed è realizzato:

```
while ( i++ < iMax )
{
    mySource >> bufferGlobal;

    vetFilterGeometric << bufferGlobal;          //same as importFrom()
    vetFilterGeometric >> bufferGlobal;          //same as extractTo()

    myOutput << bufferGlobal;
}
```

Questo sistema è leggermente complicato dal fatto che ci sono tre possibili formati di frame ingresso (*vetFrameRGB24*, *vetFrameYUV420* e *vetFrameT<unsigned char>*), oltre ai metodi preposti all'accesso e alla gestione dei tre buffer si sono incluse direttamente in *vetFilter* anche le implementazioni delle funzioni di estrazione (imposte da *vetInput*):

```
VETRESULT vetFilter::extractTo(vetFrameYUV420& img)
{
    INFO("VETRESULT vetFilter::extractTo(vetFrameYUV420& img) [pushing data]")

    if ( !isBufferYUV() )
        return VETRET_ILLEGAL_USE;

    img = *bufferYUV;

    return VETRET_OK;
}
```

Il metodo relativo a *vetFrameRGB24* è simile, mentre per quanto riguarda *vetFrameT* la fase di controllo prende in considerazione anche il profilo (*vetFrameT::profile* distingue il formato). Come si può facilmente notare l'estrazione è valida solo se i formati corrispondono, la conversione automatica è stata deprecata, un generico filtro potrebbe essere in grado di gestire solo alcuni formati, le funzioni di acquisizione dei frame devono garantire la capacità di trattare quel formato ed in generale devono aggiornare lo stato del buffer interno a seconda del formato in ingresso, nel seguente esempio il filtro è in grado di processare frame di tipo YUV ma non oggetti *vetFrameT*:

```
VETRESULT vetDigitalFilter::importFrom(vetFrameYUV420& img)
{
    int ret = VETRET_OK;
    if ( !isBufferYUV() ) {
        useBufferYUV(img.width, img.height);
        ret = VETRET_OK_DEPRECATED;
    }
    if (myParams->currentKernel == NULL) {
        *bufferYUV = img;
        return ret;
    }
    else
        ret += doProcessing(    img, *bufferYUV,
                                *myParams->currentKernel,
                                myParams->clampNegative    );
    return ret;
}
```



```
VETRESULT vetDigitalFilter::importFrom(vetFrameT<unsigned char>& img) {
    return VETRET_NOT_IMPLEMENTED;
}
```

Il codice relativo al controllo *!isBufferYUV()* consente di impostare automaticamente il buffer corretto (se non è YUV, viene inizializzato con le dimensioni previste), ovviamente il metodo *importFrom(vetFrameRGB24)* effettua il controllo con sul buffer RGB.

In questo caso il filtro effettua una convoluzione con il kernel corrente (*currentKernel*) nella funzione *doProcessing*, lo stile di delegare il processo a funzioni statiche non è obbligatorio ma altamente consigliato, in particolare le funzioni che contengono l'algoritmo dovrebbero essere il più indipendenti possibile (come *vetFilterGeometric::doProcessing*).

Il sistema predefinito consente comunque allo sviluppatore di implementare una propria strategia di buffering, come in ogni altro processo di eredità in C++ si possono sovrascrivere (*override*) le funzioni già implementate e modificarne il comportamento, ad esempio se un filtro lavora su *n* frame, si potrebbe sostituire i 3 puntatori con tre array (in modo estremamente banale):

```
void vetFilter::useBufferYUV(unsigned int width, unsigned int height) {
    if ( bufferYUV == NULL )
        bufferYUV = new vetFrameYUV420(width, height)[n];
    else if ( bufferYUV->width != width || bufferYUV->height != height )
        for (unsigned int i=0; i < n; i++)
            bufferYUV[i].reAllocCanvas(width, height);

    if ( bufferRGB != NULL ) {
        delete [] bufferRGB;
        bufferRGB = NULL;
    }
    //[..]
};
```

Ovviamente anche gli altri metodi devono essere aggiornati, ma le modifiche sono banali e sporadiche, ad esempio le implementazioni di *isBuffer\**, *getHeight*, *getWidth*, *EoF* sono ancora compatibili, mentre è necessario aggiungere dei loop alle funzioni *setHeight*, *setWidth* e modificare come sopra (*delete []*) anche *relaseBuffers*.

### 3.5.3 - Parameters for Filters and Codecs

La serializzazione dei moduli è una caratteristica molto utile, quando è possibile i parametri di lavoro del modulo devono essere memorizzati direttamente nella classe *vet<..>Parameters* che quindi si può facilmente occupare anche di salvarli e caricarli in formato XML tramite i prototipi:

```
virtual VETRESULT saveToStreamXML(FILE *fp) = 0;
virtual VETRESULT loadFromStreamXML(FILE *fp) = 0;
```

La classe base *vetFilterParameters* fornisce l'implementazione dei metodi di accesso diretto a file tramite le funzioni di sistema *fopen* e *fclose*:

```
int saveToXML(const char* filename);
int loadFromXML(const char* filename);
```

L'implementazione inclusa in *vetFilterGeometric* crea il seguente XML:

```
<?xml version= "1.0" ?>
<vetFilterGeometricParameters>
  <runmode value="5" />
  <rotation value="45.65" />
  <resize width="320" height="240" />
  <forzeSize value="1" />
  <internalBufferType value="1"/>
</vetFilterGeometricParameters>
```

La formattazione del testo è semplificata dalle funzioni di sistema *fprintf* e *fscanf*, la cui documentazione è reperibile online o nei manuali di C++, il sistema più pratico è di spiare i filtri esistenti e modificare il codice, la serializzazione deve comunque includere un selettore del buffer in uso, è sufficiente inserire il seguente estratto:

```
if ( fprintf(fp, " <internalBufferType value=\"%u\" />\n",
              (int)currentBuffer) == EOF)
    return VETRET_INTERNAL_ERR;
```

La variabile (enumerazione) *currentBuffer* è dichiarata nella classe madre *vetFilterParameters* e viene trattata come intero (0 significa *NONE*, 3 significa *TuC* cioè il frame template), la lettura è:

```
int cB = (int)currentBuffer;
if ( fscanf(fp, " <internalBufferType value=\"%u\" />\n", &cB) == EOF )
    throw "error in XML file, unable to import data.";
currentBuffer = (BUFFER_TYPE)cB;
```

in questo caso si usa una variabile temporanea ed un casting per evitare il warning di alcuni compilatori.

La convenzione prevede la dichiarazione del puntatore *myParams* nel componente:

```
class vetFilterGeometric :    public vetFilter {
protected:
    vetFilterGeometricParameters* myParams;
```

L'accesso deve essere gestito dai due prototipi imposti (da *vetFilter*) che devono essere implementati in questo modo:

```
VETRESULT setFilterParameters (vetFilterParameters* initParams) {
    return
        setParameters(static_cast<vetFilterGeometricParameters*>(initParams));
};

vetFilterParameters* getFilterParameters () {
    return static_cast<vetFilterParameters*>(myParams);
};
```

Dove la classe *vetFilterGeometricParameters* è l'implementazione di *vetFilterParameters* per la classe *vetFilterGeometric*, la funzione di accesso ai parametri (senza casting) è banalmente:

```
vetFilterGeometricParameters& getParameters() { return *myParams; };
```

Mentre la funzione che imposta realmente i parametri è

```
VETRESULT vetFilterGeometric::setParameters(vetFilterGeometricParameters* initParams)
{
    if (initParams != NULL && myParams == initParams)
        return VETRET_PARAM_ERR;

    if ( initParams == NULL ) {
        if ( myParams != NULL )
            reset();
        else
            myParams = new vetFilterGeometricParameters();
    }
    else {
        if ( myParams != NULL )
            delete myParams;

        myParams = initParams;
    }

    allocateBuffer(myParams->currentBuffer);
    return VETRET_OK;
}
```

Le uniche differenze (da modificare) sono in grassetto, i nomi delle funzioni (*setParameters*, *getParameters*) non sono imposti dalla sintassi, ma è altamente consigliato mantenere questo stile. Si può notare che quando il parametro in ingresso (*initParams*) è *NULL*, viene creata una nuova istanza della classe oppure vengono resettati i parametri, ovviamente anche il costruttore predefinito della classe *vetFilterGeometricParameters* inizializza le variabili tramite il costruttore:

```
vetFilterGeometricParameters(RUNMODE mode = DO_NOTHING) :
                                                                    vetFilterParameters()
{
    runMode = mode;
}

// where super-class constructor is: vetFilterParameters() { reset(); };
```

Il metodo *void reset()* è imposto (*virtual*) dalla classe *vetFilterParameters*, e imposta le variabili ai valori predefiniti:

```
void vetFilterGeometricParameters::reset()
{
    runMode = vetFilterGeometricParameters::DO_NOTHING;
    par_Rotation = 0;
    par_ResizeWidth = 0;
    par_ResizeHeight = 0;
    par_forzeSize = false;
    currentBuffer = vetFilterParameters::NONE;
}
```

Il metodo di reset della classe *vetFilter* è praticamente standard, l'implementazione comune a tutti i filtri rilasciati è:

```
VETRESULT vetFilterGeometric::reset()
{
    releaseBuffers();

    if (myParams != NULL) {
        myParams->reset();
        allocateBuffer(myParams->currentBuffer);
    }
    else
        setParameters(NULL);

    return VETRET_OK;
}
```

Qualora presenti, le altre variabili dichiarate nel filtro devono essere inizializzate in questo metodo, occorre fare molta attenzione all'ordine di accesso soprattutto quando il reset viene chiamato indirettamente dal costruttore, in particolare i puntatori devono essere impostati a NULL prima di effettuare controlli (infatti il costruttore *vetFilter* inizializza i puntatori ai tre buffer).

Il costruttore dei filtri deve (non per sintassi ma per convenzione) deve accettare un puntatore alla propria classe di parametri, di solito con valore predefinito NULL, il compito che deve assolvere è di inizializzare le variabili ad un valore predefinito (impostate dalle due funzioni reset) oppure di impostare i parametri richiesti, la tipica implementazione è banale:

```
vetFilterGeometric(vetFilterGeometricParameters* initParams = NULL) :
                                                                    vetFilter(0)
{
    myParams = NULL;
    setParameters(initParams);

    setName("Geometric Editing Filter");
    setDescription("Resize, Crop, Rotation, Flip");
    setVersion(1.0);
}
```

Si può verificare che l'ordine di esecuzione delle chiamate garantisce la corretta inizializzazione dei parametri in entrambi i casi. Come suggerito nella sezione dedicata a *vetInput*, si è scelto di ignorare la gestione della frame rate con la chiamata *vetFilter(0)* che come si può facilmente notare dal codice precedente passa il parametro (0) al costruttore di *vetInput* (e quindi *sleeptime = 0*).

Infine l'istanza della classe *vetFilter<..>Parameters* viene liberata dal distruttore, invece i buffers sono liberati dal distruttore della classe madre *~vetFilter*, chiamato in maniera automatica dal compilatore secondo l'ordine dell'ereditarietà, se l'applicazione gestisce più istanze di parametri (ad esempio, risulta utile per i test) si deve occupare anche di liberare la memoria al termine, considerando che nella maggior parte dei casi la classe parametri è istanziata dal modulo (chiamata del metodo *setParameters(NULL)*) è necessario eliminare l'istanza nel distruttore:

```
vetFilterGeometric::~~vetFilterGeometric()
{
    if (myParams != NULL)
        delete myParams;
}
```

### 3.5.4 - Platform specific

Qualora siano necessarie implementazioni specifiche a seconda della piattaforma, mantenere comunque un solo header (ed un solo file sorgente) e sfruttare le macro di pre-compilazione:

```
#if defined(sun) || defined(__sun) || defined(linux) || defined(__linux) \
|| defined(__CYGWIN__) || defined(__FreeBSD__) || defined(__OPENBSD__) \
|| defined(__MACOSX__) || defined(__APPLE__) || defined(sgi) || defined(__sgi)

// linux specific code here

class vetMyFilter {

    //[...]
    int universalMethod(vetFrameCache24)
    {
        // here we may use linux-only services
    }

    void* linuxSpecificMethod(char* buffer) { }
};

#elif defined(_WIN32) || defined(__WIN32__)

// windows specific code here

class vetMyFilter {

    //[...]
    int universalMethod(vetFrameCache24)
    {
        // here we may use linux-only services
    }

    HWND windowsSpecificMethod(LPSTR buffer) { }
};

#endif
```

*esempio in ./source/vetThread.h*

Ovviamente la portabilità di un codice equivale alla più stretta portabilità degli oggetti inclusi (tipi/classi), il codice che include l'oggetto *vetMyFilter* e accede al metodo *universalMethod* è compatibile con sistemi NIX e Windows, mentre l'utilizzo dei metodi specifici legati alla piattaforma restringe la portabilità del software.

Ricordo che le principali differenze tra sistemi operativi che non sono completamente risolte dalle Standard C++ Libraries sono:

- ✓ Accesso binario a file
- ✓ Device I/O
- ✓ Threads
- ✓ Sockets

### 3.5.5 - Templates

La sintassi C++ di funzioni template è:

```
T resultGlobal;

template<class T>
T* myFunction(T& data)
{
    resultGlobal = data;

    return &resultGlobal;
};
```

mentre per intere classi è:

```
template<class T>
class myTemplateClass
{
    T* data;
    T meanValue;

    unsigned int size;
};
```

ed se si eredita da una classe template:

```
template<class T>
class myTemplateClassInherit : public myTemplateClass<T>
{ };
```

oppure si può restringere l'astrazione ad un tipo (o classe) definito:

```
class myTemplateClassInstance : public myTemplateClass<myObject>
{ };
```

Lo stile C++ prevede di definire direttamente nell'header sia funzioni che classi di questo tipo per ovviare a frequenti problemi di compilazione. Esempi completi si possono trovare nelle implementazioni di *vetBuffer* (./source/buffers).

La classe *vetFrameT<tipo>* è una struttura dati basata su un array di oggetti <tipo>, nel caso di immagini multi-canale è necessario gestire autonomamente la separazione dei canali ed impostare l'enumerazione *channelType* {VETFRAMET\_CT\_PIXELPACKED, VETFRAMET\_CT\_CHANNELPACKED}. Una descrizione più accurata della class *vetFrameT* è disponibile nel capitolo primo.

L'utilizzo di una oggetto come *vetFrameT<unsigned char>* è assolutamente identico a qualunque altro oggetto C++, si può pensare che il precompilatore sostituisca tutte le occorrenze della classe template (class *T*) con il tipo previsto (*unsigned char*), in pratica la prima funzione proposta diventa:

```
unsigned char* myFunction(unsigned char &data)
{
    unsigned char* resultGlobal = new unsigned char;
    resultGlobal = data;

    return &resultGlobal;
};
```

### 3.5.6 - Threading

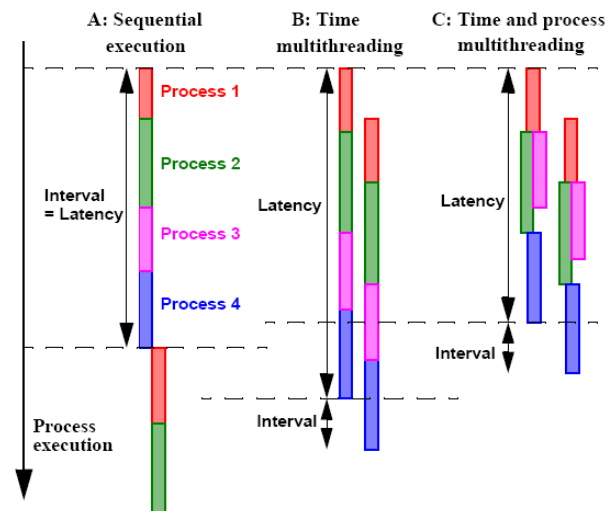
La maggior parte degli algoritmi di video processing operano ancora off-line, ovvero hanno un tempo di esecuzione superiore al tempo concesso dalla frame rate dello stream, ciò è prassi nella maggior parte dei codificatori software che per rispondere alla necessità del mercato di buon un compromesso tra qualità percepita e banda occupata (soprattutto nelle infrastrutture wireless) devono effettuare calcoli sempre più complessi.

Le operazioni coinvolte in questi processi sono legate alla teoria, cioè ad algoritmi matematici, più o meno approssimati, e all'hardware specifico della piattaforma; ottimizzare la performance di un processo simile prevede quindi sia uno studio teorico dell'algoritmo e delle approssimazioni accettabili che un'analisi relativamente pratica delle implementazioni specifiche per l'hardware e la codifica scelta.

La capacità di gestire la quantità di dati (bytes) su cui gli algoritmi operano è intrinsecamente legata all'hardware e all'I/O di basso livello. Oltre alla grande mole di dati il problema è la quantità di calcoli matematici effettuati dalla CPU ed eventuali sub-processor (matematico o grafici), il dato statistico di riferimento è il numero di operazioni in un determinato tempo.

I sistemi mono-processore funzionano intrinsecamente in modo seriale, i servizi di basso livello del sistema operativo emulano la capacità di gestire più processi in parallelo tramite il classico *TDM* (Time Division Multiplexing), mentre i sistemi *dual core* o distribuiti sono molto più complessi e non sono stati presi in considerazione in questo ambito.

Le prestazioni reali della piattaforma ovviamente sono costanti, il multi threading permette solo di ottimizzare l'utilizzo delle risorse, risulta evidente che eseguire troppi processi contemporaneamente sarebbe comunque controproducente.



L'aspetto ostico del threading è proprio la gestione dell'intera esecuzione e la suddivisione delle singole procedure che in generale non saranno completamente indipendenti, la correlazione dei threads può essere legata al tempo oppure ai dati elaborati, lo sviluppatore deve avere ben chiaro il percorso dell'*Instruction Pointer* (il puntatore alla corrente istruzione assembler e quindi alla successiva) soprattutto quando l'oggetto interagisce con altri oggetti e processi.

Alcuni contesti offrono terreno fertile al threading che risponde con ottime prestazioni, ad esempio operazioni su singoli pixel o blocchi di pixel indipendenti possono essere raggruppate (per righe o blocchi) ed eseguite come threads separati, quando l'ultimo thread ha concluso il controllo può essere restituito alla funzione chiamante. Di contro vi sono operazioni come la convoluzione che non possono essere scomposte (se non il calcolo matriciale) e sono solitamente approssimate diminuendo le dimensioni del kernel ( $3 \times 3 \rightarrow 10 * \text{pel} * \text{channels} * \text{size Op.}$ ).

La gestione di più processi (threads) è spesso complessa e prevede un sistema di sincronizzazione molto robusto, tuttavia ciò è ampiamente giustificato dalla performance risultante. Nel caso più comune, lo sviluppo di un filtro, è opportuno ricordare che se si esegue un multi-threading interno e si restituisce l'Instruction pointer (il controllo) alla funzione chiamante, essa è autorizzata a ritenere che il processo sia completo e ad estrarre i dati.

## Ad Esempio:

```
int main()
{
    vetFrameRGB24 temp(320,240);
    vetFilterBadSync myF;

    //[...] load data to temp

    myF << temp; // appena l'operatore di ingresso restituisce il controllo, copio i dati dal buffer
    myF >> temp; // se i processi interni al filtro non avessero tutti concluso i ciclo vitale una parte
                  dei dati potrebbe essere il residuo di un precedente processing.

    //[...] save data from temp
}

class vetFilterBadSync
{
    struct myThreadDataStr {
        void* data;
        unsigned int width;
        unsigned int row;
    }

    vetFrameRGB24* buffer; // supponiamo il buffer sia inizializzato dal costruttore
    myThreadDataStr* pData; // supponiamo che la struttura dati sia inizializzata dal costruttore

    //[...]

    int importFrom(vetFrameRGB24& img)
    {
        pData->width = img.width;
        pData->data = img.data[0][0];
        for (unsigned int i=0; i< img.height; i++)
        {
            pData->row = i;
            vetThread myThread(myFunction, pData);
        }

        // Qui è necessario controllare che tutti i processi siano completati,
        // altrimenti i dati estratti dal buffer non sono completamente aggiornati.

        // In questo particolare caso il codice funziona, poiché gli oggetti vetThread sono creati
        // all'interno di questo scope e vengono automaticamente cancellati tramite il relativo
        // distruttore, come si può notare nell'implementazione di vetThread i distruttori cercano di
        // attendere che il processo sia completato, in altre parole la sincronizzazione è automatica.
        // Utilizzare i threads in questo modo è rischioso, scrivere le routine di sincronizzazione
        // esplicitamente, sono disponibili funzioni che semplificano il meccanismo.
    }
}
```

Notare che in questo caso le funzioni lavoravano su ogni riga dell'immagine in modo indipendente, infatti processi che coinvolgono più pixel e sono correlati tra loro, come ad esempio la convoluzione, non possono essere ottimizzati tramite questo sistema, comunemente si ovvia al problema tramite algoritmi approssimati o kernel piccoli.

Consultare la documentazione della classe *vetThread* (`./source/vetThread.h`) per maggiori informazioni.



## 3.6 - VETLib Package Starter Kit

I principali componenti VETLib (filtri, codecs, inputs, outputs, ..) derivano direttamente dalle relative interfacce e devono rispondere ad un comportamento standard, la parte iniziale dello sviluppo prevede quindi l'analisi accurata delle funzioni da realizzare e la stesura dello scheletro che le implementa, secondo la filosofia *Extreme Programming* (write tests first) è anche necessario un file sorgente che verifichi le funzionalità della classe durante lo sviluppo e le dimostri all'utente quando il componente viene distribuito, sono inoltre necessari alcuni file di progetto (per Visual Studio, Borland, il Makefile).

Generalmente lo sviluppatore "frettoloso" e spesso anche quello esperto tende a cercare il progetto più coerente con il proprio e (..attraverso decine di backup..) modificarlo ad hoc, questo processo è stato completamente automatizzato dal software *VETLib Package Studio* (ex. vetPSK), tutti gli sviluppatori che intendono estendere la libreria con un nuovo componente dovrebbero scaricare l'archivio *VETLib Package Starter Kit* (*VETLib-PSK-x.x.x.zip*), il contenuto è il seguente:

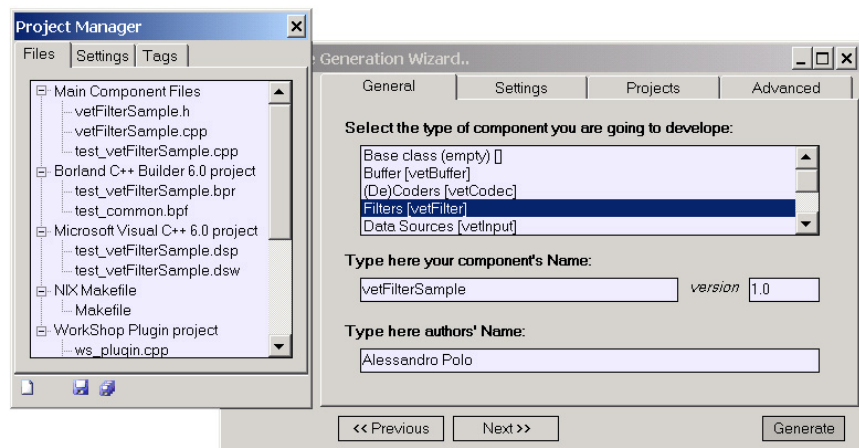
- ✓ Moduli vuoti ./packages/<categoryName>Empty/
- ✓ Moduli Esempio ./packages/<categoryName>Sample/
- ✓ Il software vetPS ./packages/vetps/
- ✓ Moduli template necessari al tool vetPS ./packages/templates/

L'archivio è disponibile anche come pacchetto di installazione (.MSI), il software è disponibile solo per ambienti Windows (è scritto in Managed C++ .NET), i requisiti necessari al corretto funzionamento sono tre:

- ✓ I files contenuti nella directory *templates*, sono inclusi negli archivi;
- ✓ Gli headers della libreria (*./source/\*.h*);
- ✓ I binari statici della libreria (*./lib/VETLib.lib*)










La ricerca delle directory presume che esista una directory *X* (root di VETLib) contenente il file *source/vetDefs.h*, la ricerca avviene dal percorso corrente fino alla root in modo recursivo (ad ogni ciclo si cerca nella directory superiore, e se non viene trovata è richiesta la locazione all'utente), una volta stabilito il percorso della root di VETLib il software suppone che contenga */lib*, */source*, */packages* e soprattutto */package/templates*, quest'ultima cartella contiene infatti i moduli base e i file di configurazione XML che vengono caricati all'avvio, se non fossero presenti il software chiede di selezionarli manualmente: *vetGroups.xml*, *vetTypes.xml*. Nella directory */packages* risiede inoltre il file *NAMESPACE*, questo file di testo elenca tutti i nomi riservati della libreria, il software valida il nome attraverso il prefisso predefinito dell'interfaccia e controllando la lista dei nomi riservati.

*Package Studio* è ancora in fase di sviluppo, ma la sua principale funzionalità è attiva, lo sviluppatore può utilizzare il *Generator Wizard* per creare i files principali e i file dei progetti selezionati, sono disponibili una serie di opzioni che vanno modificate raramente, nella maggior parte dei casi è sufficiente compilare la prima scheda selezionando la











tipologia del componente e inserendo il nome della classe, le interfacce supportate sono: *vetInput*, *vetOutput*, *vetFilter*, *vetCodec*, *vetVision*, *vetBuffer*, *vetFrame*, *vetObject* ed una classe vuota.

Segue la lista dei file di un progetto generico, completo, il tag `<className>` è il nome della classe convalidato con una serie di restrizioni (vedi *Package Conventions*):

 <code>&lt;className&gt;.h</code>	Dichiarazione della classe;
 <code>&lt;className&gt;.cpp</code>	Implementazione della classe;
 <code>test_&lt;className&gt;.cpp</code>	File sorgente contenente il metodo <i>main</i> del progetto;
 <code>Makefile</code>	File di configurazione dell'utility Make;
 <code>test_&lt;className&gt;.bpr</code>	File del progetto per Borland C++ Builder 6.0;
 <code>test_&lt;className&gt;.dsp</code>	File di progetto per Microsoft Visual Studio 6.0;
 <code>test_&lt;className&gt;.dsw</code>	File del Workspace per il progetto (Visual Studio);
 <code>&lt;className&gt;.License</code>	Licenza del progetto;
 <code>&lt;className&gt;.Readme</code>	Informazioni sul progetto;

Opzionali (WorkShop PlugIn project):

 <code>ws_&lt;className&gt;.h</code>	Header dell'interfaccia tra componente e plugin;
 <code>ws_&lt;className&gt;.cpp</code>	Sorgente dell'interfaccia tra componente e plugin;
 <code>ws_plugin_def.h</code>	File di sistema (WS PlugIn);
 <code>ws_plugin_func.h</code>	File di sistema (WS PlugIn);
 <code>ws_plugin_src.cpp</code>	File di sistema (WS PlugIn);
 <code>ws_plugin.def</code>	File di sistema (WS PlugIn);
 <code>ws_plugin.dsp</code>	File del Progetto (Visual Studio);
 <code>ws_plugin.dsw</code>	File di WorkSpace (Visual Studio);

Lo sviluppo (conversione) dei plugins per il software WorkShop è analizzato nel dettaglio nel prossimo capitolo.

La parte restante di questa sezione presenta il funzionamento del software e la formattazione dei file di configurazione, gli sviluppatori di pacchetti non sono tenuti a conoscere il meccanismo e possono saltare fin d'ora la lettura del capitolo.

Il progetto generico che include tutti i files elencati (sopra) è diviso sia nell'interfaccia (nel *Project Manager*), sia nella fase di creazione in sotto-progetti, ogni sotto-progetto è gestito da un istanza della classe *vetPkgProject* che racchiude una serie di informazioni (*ID*, *friendlyName*, *OS*, *comment*) e un array di oggetti di tipo *vetPKGFile* (massimo 32).

La classe *vetPKGFile* rappresenta completamente un generico file (sono trattati tutti allo stesso modo) e implementa alcuni metodi per l'accesso e il parsing (sostituzione di *tags*), ad ogni file output è quindi associato un file sorgente template, una serie di proprietà (*description*, *myForm*) ed in particolare un oggetto *vetTagHash* contenente la lista dei tags (stringhe univoche) e dei relativi valori da aggiornare (ad esempio il tag *%CLASSNAME%* è sostituito con il nome della classe), in generale si condivide un'unica lista per tutti i progetti e files.

In termini semplici Package Studio seleziona i progetti e i files sorgente (moduli template) e sostituisce una lista di tags con i valori corretti in base alle scelte dell'utente, questo processo è realizzato con un sistema elastico e molto modulato, i files e i progetti base sono aggiornabili in modo (parzialmente) indipendente con il software.

I sotto-progetti sono definiti nel file *vetGroups.xml* e caricati dinamicamente all'avvio, la modifica delle proprietà di un progetto non implica alcun aggiornamento del software, ma non è attualmente possibile aggiungere nuovi progetti poiché la selezione visuale è statica (nel *Generation Wizard*).

```
<?xml version="1.0" ?>
<vetPkgProjectsGroup>
  <vetPkgProject>
    <id value="8" />
    <friendlyName value="Borland C++ Builder 6.0 project" />
    <files>
      <file value="test_Template.bpr" friendlyName="BCB project"/>
      <file value="test_common.bpf" friendlyName=""/>
    </files>
    <os value="WIN" />
    <comment>BCB project for developing/testing Your module.</comment>
  </vetPkgProject>
  {..}
</vetPkgProjectsGroup>
```

vetGroups.xml

È possibile visualizzare i progetti predefiniti (template) caricati attraverso il comando “*view current*”, “*vetPkgProjects*” nel menu “*Packages*”, oltre a questi progetti che sono aggiunti al progetto principale a seconda della selezione nella seconda scheda del “*Generation Wizard*”, viene creato un nuovo sotto-progetto contenente i tre files sorgente *<classname>.h*, *.cpp* e il file di test (infatti sono sempre presenti e sono i file da modificare..).

La configurazione minima prevede di selezionare la tipologia dei componente ed il nome dello stesso (prima scheda del “*Generation Wizard*”), il tipo di componente scelto ha delle ripercussioni sul nome del componente (*Package Conventions*) e soprattutto sui file (sorgente) template che saranno modificati tramite la *vetTagHash* del progetto. I tipi supportati sono descritti nel file *vetTypes.xml* e sono caricati dinamicamente all'avvio, in questo caso è possibile sia modificare le proprietà dei nodi *vetClassType* che aggiungerne di nuovi, segue un estratto del file XML:

```
<?xml version="1.0" ?>
<vetTypes>
  {..}
  <vetClassType>
    <classID value="7000" />
    <friendlyName value="Filters" />
    <interfaceFileName value="vetFilter" />
    <classPrefix value="vetFilter" />
    <implementationsDirName value="filters" />
    <templateFileName value="vetFilterTemplate" />
    <comment></comment>
  </vetClassType>
  {..}
</vetTypes>
```

vetTypes.xml

La classe *vetClassType* rappresenta un singolo tipo e si occupa di caricare le informazioni dai nodi XML, il comando “*view current*”, “*vetClassType*” nel menu “*Packages*” consente di visualizzare i tipi disponibili.

## Contenuto della directory `./packages/templates/`

	<code>vetGroups.xml</code>	Definizione dei sotto-progetti disponibili;
	<code>vetTypes.xml</code>	Definizione dei tipi di componenti ( <i>fully editable</i> );
	<code>Template.License</code>	Licenza predefinita (GPL) del componente;
	<code>Template.Readme</code>	Leggimi preformattato del componente;
	<code>Makefile</code>	File di configurazione di Make preformattato;
	<code>test_common.bpf</code>	File di Progetto Borland C++ Builder 6.0;
	<code>test_Template.bpr</code>	File di progetto Borland C++ Builder 6.0;
	<code>test_Template.dsp</code>	File di progetto Visual C++ 6.0;
	<code>test_Template.dsw</code>	File di workspace Visual C++ 6.0;
	<code>vetTemplate.cpp</code>	Sorgente di una generica classe;
	<code>vetTemplate.h</code>	Header di una generica classe;
	<code>vetTemplate_test.cpp</code>	Sorgente di test di una generica classe;
	<code>vetBufferTemplate.cpp</code>	Sorgente di un componente Buffer;
	<code>vetBufferTemplate.h</code>	Header di un componente Buffer;
	<code>vetBufferTemplate_test.cpp</code>	Sorgente di test di un componente Buffer;
	<code>vetCodec_Template.cpp</code>	Sorgente di un componente Codec;
	<code>vetCodec_Template.h</code>	Header di un componente Codec;
	<code>vetCodec_Template_test.cpp</code>	Sorgente di test di un componente Codec ;
	<code>vetFilterTemplate.cpp</code>	Sorgente di un componente Filter;
	<code>vetFilterTemplate.h</code>	Header di un componente Filter;
	<code>vetFilterTemplate_test.cpp</code>	Sorgente di test di un componente Filter;
	<code>vetFrameTemplate.cpp</code>	Sorgente di un componente Frame;
	<code>vetFrameTemplate.h</code>	Header di un componente Frame;
	<code>vetFrameTemplate_test.cpp</code>	Sorgente di test di un componente Frame;
	<code>vetInputTemplate.cpp</code>	Sorgente di un componente Input;
	<code>vetInputTemplate.h</code>	Header di un componente Input;
	<code>vetInputTemplate_test.cpp</code>	Sorgente di test di un componente Input;
	<code>vetObjectTemplate.cpp</code>	Sorgente di un componente Object;
	<code>vetObjectTemplate.h</code>	Header di un componente Object;
	<code>vetObjectTemplate_test.cpp</code>	Sorgente di test di un componente Object;
	<code>vetOutputTemplate.cpp</code>	Sorgente di un componente Output;
	<code>vetOutputTemplate.h</code>	Header di un componente Output;
	<code>vetOutputTemplate_test.cpp</code>	Sorgente di test di un componente Output;
	<code>vetVisionTemplate.cpp</code>	Sorgente di un componente Vision;
	<code>vetVisionTemplate.h</code>	Header di un componente Vision;
	<code>vetVisionTemplate_test.cpp</code>	Sorgente di test di un componente Vision;
	<code>WS_DLL_13.cpp</code>	Header dell'interfaccia tra il componente e il plugin;
	<code>WS_DLL_13.h</code>	Sorgente dell'interfaccia tra il componente e il plugin;
	<code>WS_DLL_13.dsp</code>	File del Progetto (Visual Studio);
	<code>WS_DLL_13.dsw</code>	File di WorkSpace (Visual Studio);
	<code>ws_plugin_def.h</code>	File di sistema (WS PlugIn);
	<code>ws_plugin_exp.def</code>	File di sistema (WS PlugIn);
	<code>ws_plugin_func.h</code>	File di sistema (WS PlugIn);
	<code>ws_plugin_src.cpp</code>	File di sistema (WS PlugIn);
	<code>WARNING</code>	File di testo che suggerisce di non modificare i file.

Il seguente estratto imposta le principali proprietà del progetto e crea la lista dei tags:

```
vetDirectories* dirs = dynamic_cast<vetPKStudio*>(this->MdiParent)->Directories;
vetClassType* vct = dynamic_cast<vetClassType*>(lB_type->SelectedItem);
Array* vpp = dynamic_cast<vetPKStudio*>(this->MdiParent)->vetTemplateProjects;

vetPkgProject* newProject = new vetPkgProject();
newProject->FriendlyName = S"Main Component Files";

String* pathProject;
if ( tB_dir->Text->Length > 1 )
    pathProject = tB_dir->Text;
else
    pathProject = String::Concat(dirs->packages, tB_name->Text, S"\\");

if ( !Directory::Exists(pathProject) )
    Directory::CreateDirectory(pathProject);

if ( !Directory::Exists(pathProject) )
{
    this->Cursor = Cursors::Default;
    MessageBox::Show(this, S"Error", S"Directory name is NOT valid!");
    return;
}

vetTagHash* prjHash = new vetTagHash();
prjHash->loadDefaultHash();
prjHash->disableAll();

prjHash->editSimpleTag( S"%CLASSNAME%", tB_name->Text, true);
prjHash->editSimpleTag( S"%CLASSDEFINE%", tB_name->Text->ToUpper(), true);
prjHash->editSimpleTag( S"%VETTYPE%", vct->FriendlyName, true);
prjHash->editSimpleTag( S"%VERSION%", tB_version->Text, true);
prjHash->editSimpleTag( S"%AUTHOR%", tB_authors->Text, true);
prjHash->editSimpleTag( S"%FILENAME%", tB_name->Text, true);
prjHash->editSimpleTag( S"%TODAY%", DateTime::Now.ToString(), true);
prjHash->editSimpleTag( S"%SOURCEDIR%", dirs->vetSource, true);
prjHash->editSimpleTag( S"%LIBDIR%", dirs->vetBinaries, true);

prjHash->editDoubleTag( S"%VFI_START%", cB_funcv->Checked);
prjHash->editDoubleTag( S"%EFI_START%", cB_funcce->Checked);
prjHash->editDoubleTag( S"%DOCVAR%", cB_docs_v->Checked);
prjHash->editDoubleTag( S"%DOCFUN%", cB_docs_f->Checked);

newProject->TagHash = prjHash;
```

wizardForm.cpp :: Generate function

Il metodo *vetTagHash::loadDefaultHash()* carica la lista predefinita (i tag sono disabilitati), le chiamate multiple di *vetTagHash::editSimpleTag(String\*, String\*, bool)* aggiornano i valori da sostituire ai tag e li attivano, nonostante sia possibile impostare una tabella di tags diversa per ogni file del progetto, si condivide sempre una singola tabella.

Il seguente estratto (direttamente consecutivo al precedente) si occupa di creare i due files principali (header e sorgente della classe) ed il file sorgente di test (contenente il *main*), le tre istanze vengono quindi aggiunte al progetto e tramite le funzioni *vetPkgProject::ApplyHashes()* e *vetPkgProject::SaveOutputs()* si effettua la sostituzione dei tag ed il salvataggio dei files (la stringa "Template" nel nome dei files predefiniti, se presente, è sostituita con il nome della classe). La seconda parte del codice analizza la selezione dell'utente in base ai progetti caricati dal sistema (*vetGroups.xml*), la configurazione dei ogni (sotto)progetto consiste nell'impostazione della tabella di tag corrente, della directory del progetto e del nome della classe (per la rinominazione).

```

vetPKGFile* fileH = new vetPKGFile(
    String::Concat(dirs->packagesTemplate, vct->TemplateFileName, S".h"),
    String::Concat(pathProject, tB_name->Text, S".h") );

vetPKGFile* fileS = new vetPKGFile(
    String::Concat(dirs->packagesTemplate, vct->TemplateFileName, S".cpp"),
    String::Concat(pathProject, tB_name->Text, S".cpp") );

vetPKGFile* fileT = new vetPKGFile(
    String::Concat(dirs->packagesTemplate, vct->TemplateFileName, S"_test.cpp"),
    String::Concat(pathProject, S"test_", tB_name->Text, S".cpp") );

newProject->addFile(fileH);
newProject->addFile(fileS);
newProject->addFile(fileT);

newProject->ApplyHashes();
newProject->SaveOutputs();

prjManForm* newPrj = new prjManForm();
newPrj->MdiParent = this->MdiParent;

newPrj->AddProject(newProject);

IEnumerator* en = vpp->GetEnumerator();
while ( en->MoveNext() )
{
    vetPkgProject* obj = __try_cast<vetPkgProject*>( en->Current );

    if (obj == NULL)
        continue;

    if ( ( obj->ID == 1  && cB_prj_1->Checked ) ||
        ( obj->ID == 2  && cB_prj_2->Checked ) ||
        ( obj->ID == 4  && cB_prj_4->Checked ) ||
        ( obj->ID == 8  && cB_prj_8->Checked ) ||
        ( obj->ID == 16 && cB_prj_16->Checked ) ||
        ( obj->ID == 32 && cB_prj_32->Checked ) )
    {
        vetPkgProject* nPrj = new vetPkgProject( obj );
        nPrj->TagHash = prjHash;
        nPrj->Directory = String::Copy( pathProject );
        nPrj->SetOutputNameFromClassname( tB_name->Text );
        nPrj->loadFiles();
        nPrj->ApplyHashes();
        nPrj->SaveOutputs();
        newPrj->AddProject(nPrj);
    }
}
}

```

wizardForm.cpp :: Generate function

## 3.7 - Releasing VETLib

Nonostante l'integrazione di un nuovo pacchetto nelle distribuzioni ufficiali sia un'operazione relativamente semplice, il processo può complicarsi drasticamente sia a causa di conflitti con librerie esterne o interne che per la negligenza dell'autore del pacchetto, è quindi complesso automatizzare l'intero processo di controllo e aggiunta di nuovi moduli.

Nei casi più semplici è sufficiente aggiungere il file sorgente al progetto (con Borland Builder o Visual Studio: Project→"Add file to Project..", su Linux aggiungere una nuova linea alla lista dei sorgenti), includere l'header in `./include/VETLib.h`, aggiornare i test (Makefile e progetti).

Segue un elenco diviso in 2 fasi delle operazioni da svolgere per aggiungere un nuovo componente:

### Fase A: (Compilazione)

- Spostare i file (.h, .cpp) nella directory `./source/<category>` (modificare i *relative paths*);
- Se il pacchetto dipende da librerie esterne aggiungerle in `./support/<libName>`;
- Spostare il file di test in `./tests` (modificare i *relative paths*);
- Aggiungere il sorgente ai progetti BCB / MVC / Makefile;
- Aggiornare i progetti del programma di test `./tests/Makefile`, `test_ModuleName.dsp` (.bpr);
- Verificare le funzionalità tramite i programmi di test;
- Compilare la libreria ed eventualmente le special builds.

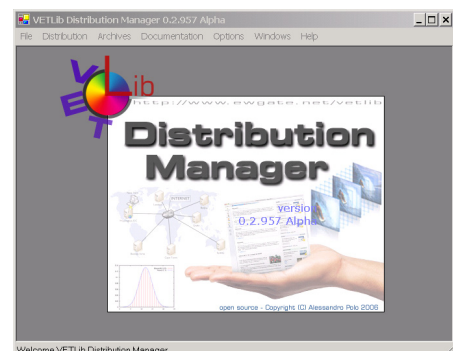
### Fase B: (Documentazione)

- Compilare la documentazione con il software *Doxygen* (setup file: `./docs/Doxygen`);
- Aggiornare i file READMEs:
  - `./README`
  - `./ChangeLog`
  - `./packages/NAMESPACE`
  - Eventualmente `./AUTHORS`, `./USE`, `./EXTEND`, `./support/NOTES`
- Creare gli archivi aggiornati;
- Aggiornare il sito con i nuovi files rilasciati:
  - `./downloads/`
  - `./documentation/`
  - `./distr/`
  - Il numero di versione in tutte le pagine;
  - Aggiornamento di `packages.html`, `downloads.html`

E' attualmente in corso lo sviluppo del software *VETLib Distribution Manager* che automatizza la gestione della libreria ed il processo di creazione degli archivi da distribuire.

La configurazione è basata sui files XML: `vetArchives.xml` (caratteristiche di ogni archivio), `vetBUILTS.xml` (progetti per costruire VETLib) e `vetDocumentation.xml` (proprietà e formati della documentazione).

L'utilizzo del software richiede ovviamente la versione più completa della libreria (tutti i Tools e le librerie esterne).



Segue la lista degli archivi da distribuire:

**📁 COMPLETE Lib** *VETLib-x.x.x.tar.gz (.zip)*

```
./*. *  
./docs/*. *  
./include/*. *  
./lib/*. * [exclusions: *~*, *.ncb, *.opt, *.plg, *.pch]  
./packages/*. *  
./source/*. *  
./support/*. *  
./tests/*. * [exclusions: *.obj, *~*, *.ncb, *.opt, *.plg, *.pch]
```

**📁 BINARIES only** *VETLib-x.x.x.tar.gz (.zip)*

```
./*. *  
./include/*. *  
./lib/*. * [exclusions: sub-folders]
```

**📁 SOURCE only** *VETLib-SRC-x.x.x.tar.gz (.zip)*

```
./*. *  
./include/*. *  
./lib/bcb/*. * [exclusions: *~*]  
./lib/mvc/*. * [exclusions: *.ncb, *.opt, *.plg, *.pch]  
./source/*. *
```

**📁 SOURCE + Support Libs** *VETLib-SRCx-x.x.x.tar.gz (.zip)*

```
./*. *  
./include/*. *  
./lib/bcb/*. * [exclusions: *~*]  
./lib/mvc/*. * [exclusions: *.ncb, *.opt, *.plg, *.pch]  
./source/*. *  
./support/*. *
```

**📁 SDK** *VETLib-SDK-x.x.x.tar.gz (.zip)*

```
./*. *  
./docs/html/*. *  
./include/*. *  
./lib/*. * [exclusions: *~*, *.ncb, *.opt, *.plg, *.pch]  
./source/*. *  
./tests/*. * [exclusions: *.obj, *.exe, *.out, *~*, *.ncb, *.opt, *.plg, *.pch]
```



**■ SDK Windows ONLY***VETLib-SDK-WIN-x.x.x.zip*

```
./*. * [exclusions: Makefile.*]
./docs/html/*. *
./include/*. *
./lib/*. * [exclusions: *.a, *~*, *.ncb, *.opt, *.plg, *.pch]
./source/*. *
./tests/*. * [exclusions: Makefile.*, *.obj, *.exe, *.out, *~*, *.ncb, *.opt, *.plg, *.pch]
```

**■ SDK Linux ONLY***VETLib-SDK-LNX-x.x.x.tar.gz*

```
./*. *
./docs/html/*. *
./include/*. *
./lib/*. * [exclusions: *.lib, sub-folders]
./source/*. *
./tests/*. * [exclusions: sub-folders]
```

**■ DOCUMENTATION (all)***VETLib-DOCS-x.x.x.tar.gz (.zip)*

```
./*. * [exclusions: Makefile.*]
./docs/*. *
./Website/*. * [exclusions: sub-folders]
./Website/html/*. *
./Website/tutorials/*. *
./Website/screenshots/*. *
```

**■ HTML DOCUMENTATION***VETLib-DOCS-HTML-x.x.x.tar.gz (.zip)*

```
./*. * [exclusions: Makefile.*]
./docs/html/*. *
```

**■ PACKAGE STARTER KIT***VETLib-PSK-x.x.x.tar.gz (.zip)*

```
./packages/*. * [exclusion: released packages]
```

**■ SAMPLES***VETLib-SAMPLES-x.x.x.tar.gz (.zip)*

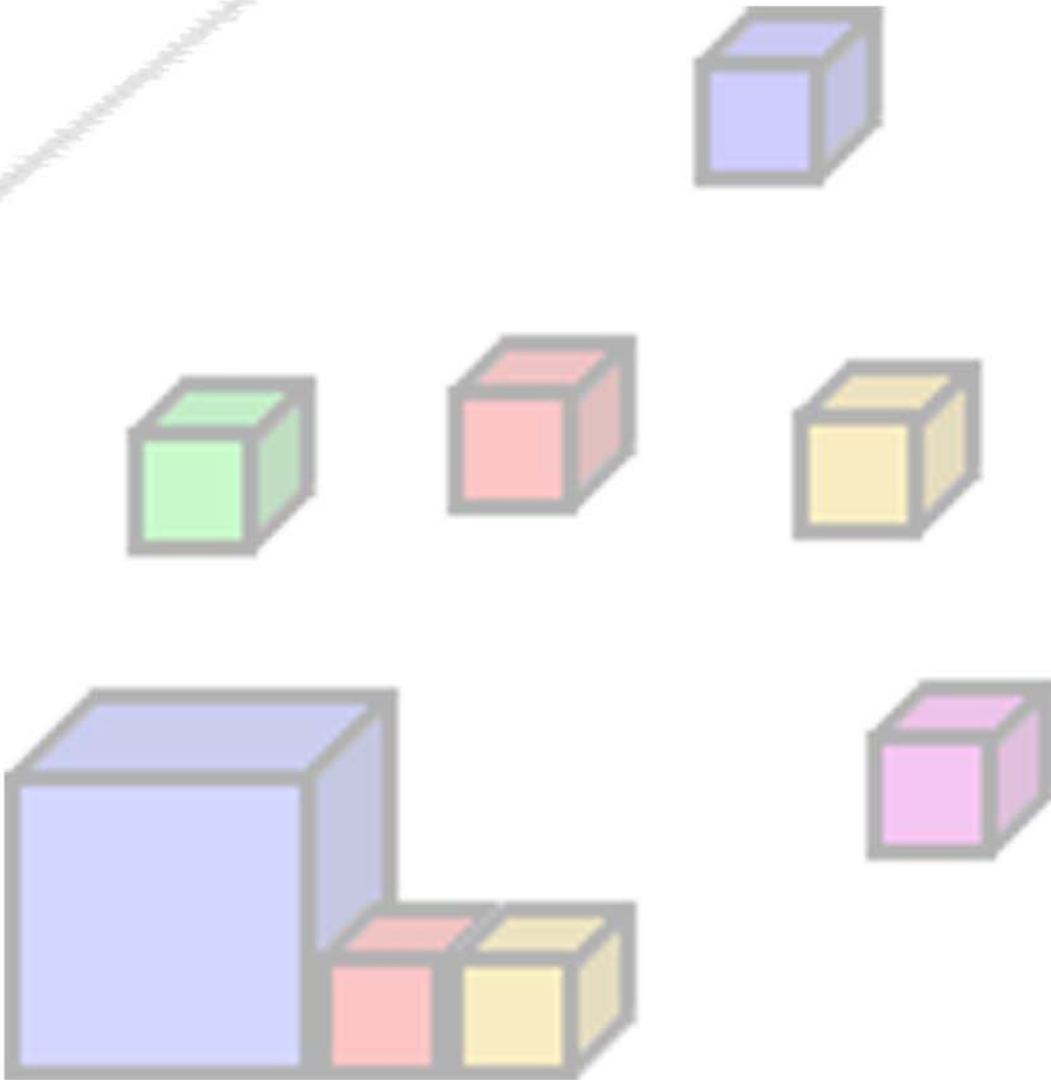
```
./tests/*. *
```

**■ PACKAGES***VETLib-PACKAGES-x.x.x.tar.gz (.zip)*

```
./packages/ [include only released packages, it is probably
empty if packages are included in official built.]
```

*Real Users never know  
what they want,  
but they always know  
when your software  
doesn't deliver it.*

*Anonymous*



# VETLIB WORKSHOP & PLUGINS

## CHAPTER IV

### 4.1 - Overview

VETLib WorkShop è un applicazione sviluppata in C++ .NET (Managed C++) per sistemi Windows, è destinata all'utilizzo pratico e al test di componenti VETLib (e future estensioni), in particolare ha la capacità di gestire in modo dinamico catene di filtri e sorgenti multiple, infatti verificare via codice (ad esempio in un *main()*) la funzionalità di una serie di filtri può risultare problematico soprattutto nella fase di ottimizzazione (o deduzione) dei parametri, il principale vantaggio di WorkShop è l'accesso rapido ai parametri di ogni componente e la possibilità di visualizzare in tempo reale sia la sequenza finale che i passi intermedi.

L'interfaccia grafica è di tipo *MDI* (Multiple Document Interface), i *child forms* (finestre figlie) sono le interfacce visuali dei componenti VETLib, queste finestre sono quasi completamente indipendenti, il componente vero può essere compilato in modo statico (linkato con il software) oppure può essere implementato in modo indipendente e caricato durante l'esecuzione (sono comunemente definiti *plugins*), i principali componenti già integrati (statici) sono:

- ✓ Sorgente statica (immagini, formati più comuni)  
Converte nello standard VETLib *vetFrameRGB24* un immagine caricata tramite il componente *System.Drawing.Bitmap* di .NET framework.
- ✓ Sorgente Playback video in formato MPEG4 (XVID)  
Interfaccia grafica del componente *vetCodec\_XVID*.
- ✓ Capture/Grabbing/Preview Real-Time DirectX (VideoIn, IEEE1394, CAMs)  
Interfaccia grafica del componente *vetDirectXInput2*.
- ✓ Sorgente Playback (tutte codifiche supportate dal sistema)  
Interfaccia grafica del componente *vetDXMovieLoader*.
- ✓ Sorgente (grabbing) tramite DirectX (CAMs)  
Interfaccia grafica del componente *vetDirectXInput*.
- ✓ Finestra di Visualizzazione  
Converte i frame dallo standard *vetFrameRGB24* al formato supportato da .NET (GDI+).

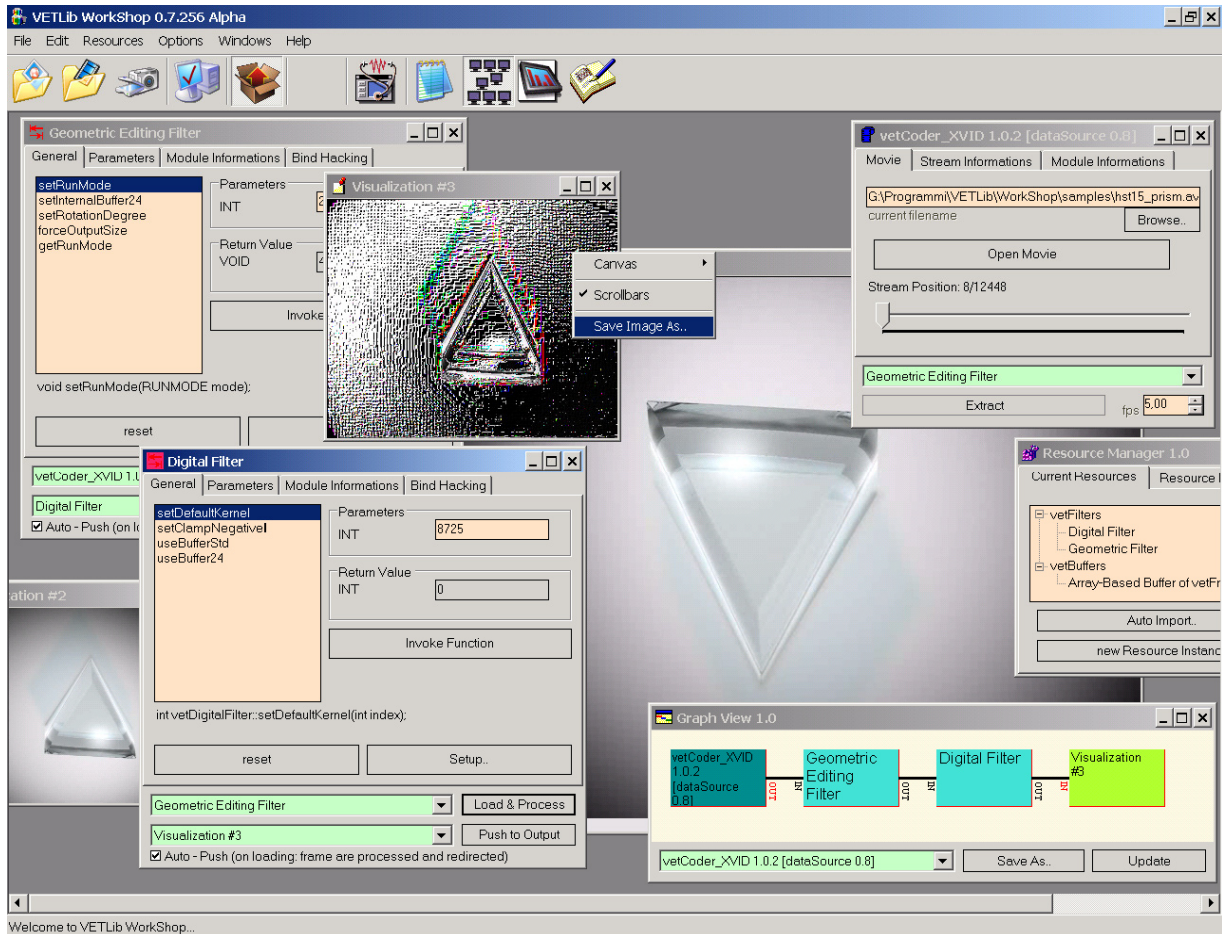
WorkShop è in grado di caricare dinamicamente nuovi componenti incapsulati in file DLL (plugins), analizzeremo più avanti nel dettaglio le modalità di creazione dei plugins e l'implementazione del sistema, la versione corrente include alcuni plugins derivati dai componenti *vetFilterGeometric*, *vetDigitalFilter*, *vetBufferArray<vetFrameRGB24\*>*, *vetBufferLink<vetFrameRGB24\*>*.

Attualmente WorkShop supporta solo lo standard I/O *vetFrameRGB24*, si prevede di superare questa limitazione nel prossimo aggiornamento attraverso l'utilizzo del componente *vetProcess* (non ancora rilasciato).

La piattaforma .NET introduce un innovativo sistema di gestione delle eccezioni, qualora non siano gestite esplicitamente dal programma è spesso possibile continuare l'esecuzione ed almeno salvare il lavoro svolto. I componenti direttamente integrati nel WorkShop sono incapsulati staticamente dalla libreria completa compilata con Microsoft Visual Studio 6.0 (*VETLib\_full\_vc6.lib*), quindi eventuali aggiornamenti di questi moduli implicano anche la ricompilazione dell'applicazione (diversamente dai plugins).

I plugins sono incapsulati in DLL (Dynamic Link Library) standard (senza estensioni), simili alle DLL del kernel di Windows, è possibile caricare al massimo *n* componenti, un istanza ciascuno, con

$n$  definito alla compilazione (attualmente 64). In futuro si prevede di aggiornare il sistema di gestione dei plugins in modo da supportare più istanze di un singolo plugin e anche DLL .NET, ciò necessita un ambiente di sviluppo Microsoft di ultima generazione anche per lo sviluppatore della DLL, ma consente interessanti sviluppi come un'interfaccia grafica di controllo del componente estremamente semplice da sviluppare e perfettamente integrata in WorkShop.



WorkShop è distribuito in due diversi formati: un archivio compresso ZIP e un pacchetto di installazione per Windows (.msi), il pacchetto è creato con il sistema di *deployment* integrato in Microsoft Visual Studio 7.0 (.NET), il file progetto è *vetWS3\_Setup.vdproj* ed è incluso nel progetto principale di WorkShop *vetWS3.vcproj*, le dipendenze sono gestite automaticamente.

La directory di installazione predefinita è `<Programmi>\VETLib\WorkShop`, i files distribuiti sono:

- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>📁 data</li> <li>📁 layout</li> <li>📁 plugins</li> <li>📁 reference</li> <li>📁 samples</li> <li>📁 shots</li> <li>📄 ChangeLog.txt</li> <li>📄 README</li> <li>📄 ToDo</li> <li>📄 vetWS3.log</li> <li>📄 vetWS3.exe</li> </ul> | <ul style="list-style-type: none"> <li>File essenziali ed opzionali per l'esecuzione del software;</li> <li>File di configurazione del layout di WorkShop (background, ..);</li> <li>I plugins inclusi nella distribuzione corrente (DLL);</li> <li>La documentazione di WorkShop in formato RTF;</li> <li>Contiene una serie di immagini e video per i tests;</li> <li>Cartella designata a memorizzare gli screenshots;</li> <li>Lista degli aggiornamenti divisi per built;</li> <li>Informazioni preliminari;</li> <li>Lista degli aggiornamenti futuri;</li> <li>File log delle sessioni (nuove sessioni aggiunte al termine);</li> <li>Eseguibile principale.</li> </ul> |
|---|--|

WorkShop è *open source*, la licenza è la classica *General Public License* (in appendice).

Informazioni, screenshots e links sono raccolti nella pagina `./Website/workshop.html` disponibile online <http://lnx.ewgate.net/vetlib/workshop.html>.

## 4.2 - How It Works

Il linguaggio *Managed C++* consente solo parzialmente l'interazione tra oggetti classici e oggetti Managed (cioè gestiti dal *garbage collector*), quindi per non incorrere in scomode limitazioni (specialmente in futuro), si sono definite due nuove interfacce similari a *vetInput* e *vetOutput* (nonostante le relazioni tra i componenti siano praticamente identiche al framework VETLib).

Segue la definizione delle interfacce:

```
public __gc __interface sourceInterface
{
    VETRESULT extractTo(vetFrameRGB24* img);

    vetProcess* getMyProcess(); //currently not implemented

    outputInterface* getOutputInterface();
    System::Object __gc* getObjectInstance();
    void viewsUpdate(void);
};

public __gc __interface outputInterface
{
    VETRESULT importFrom(vetFrameRGB24* img);

    vetProcess* getMyProcess(); //currently not implemented

    sourceInterface* getSourceInterface();
    System::Object __gc* getObjectInstance();
    void sourcesUpdate(void);
    int setSource(sourceInterface* sF);
};
```

Come si può notare attualmente lo standard I/O interno è solo *vetFrameRGB24*, i due metodi che consentono ai componenti di ricevere e trasmettere frames sono gli omonimi del framework VETLib, mentre gli altri prototipi sono di supporto, queste interfacce sono implementate da *forms* (praticamente finestre) definiti come *child* dell'applicazione principale (MDI), segue un estratto della *GUI* (Graphic User Interface) del componente *vetDirectXInput*:

```
public __gc class dxinputForm : public System::Windows::Forms::Form,
                                public sourceInterface
{
public:
    dxinputForm(void);

    void Init() {
        myCap = new vetDirectXInput();
    }

    VETRESULT extractTo(vetFrameRGB24* img) {
        if (img != NULL && myCap->getCurrentDevice() != -1)
            return myCap->extractTo( *img );
        return VETRET_ILLEGAL_USE;
    }

    void dxinputForm::Dispose(Boolean disposing) {
        delete myCap;
    }

private: vetDirectXInput* myCap;
private: outputInterface* vF;
public: outputInterface* getOutputInterface() { return vF; }
public: System::Object __gc* getObjectInstance() { return this; };
// [...]
```

Il componente standard per la visualizzazione dei frame è *viewForm*, l'implementazione dell'interfaccia *outputInterface* è molto semplice poiché non c'è nessun controllo sul flusso dati e l'immagine non viene bufferizzata ma copiata direttamente nel buffer grafico di Windows:

```
public __gc class viewForm : public System::Windows::Forms::Form,
                             public outputInterface
{
public:
    viewForm(void) {
        InitializeComponent();
        bm = new Bitmap(400, 400); // will be resized on first call
        pictureBoxBuffer->Image = dynamic_cast<Image*>(bm);
    }

    VETRESULT importFrom(vetFrameRGB24* img) {
        if (img == NULL)
            return VETRET_PARAM_ERR;
        if (img->width == 0 || img->height == 0 || img->data[0] == NULL)
            return VETRET_PARAM_ERR;
        if (img->width != bm->Width || img->height != bm->Height) {
            bm->Dispose();
            bm = new Bitmap(img->width, img->height);
            pictureBoxBuffer->Image = dynamic_cast<Image*>(bm);
        }

        BitmapData* bitData;
        Rectangle rec(0,0, img->width, img->height);
        bitData = bm->LockBits(    rec, ImageLockMode::WriteOnly,
                                PixelFormat::Format24bppRgb    );

        unsigned char* dest;
        dest = static_cast<unsigned char*>( bitData->Scan0.ToPointer() );

        // this would be great but microsoft LIES about it..
        // memcpy( dest, img.data[0],
        //         img.height * img.width * 3 * sizeof(unsigned char) );
        //
        // in fact it's BGR not RGB..

        vetUtility::conv_bgr_rgb( dest, (unsigned char*)img->data[0],
                                img->width, img->height);

        bm->UnlockBits(bitData);
        pictureBoxBuffer->Refresh();

        return VETRET_OK;
    }

    void sourcesUpdate(void) { }
    int setSource(sourceInterface* sF) { return 0; }
}
```

L'accesso al buffer GDI+ è ottimizzato tramite le pericolose funzioni di accesso al puntatore (*LockBits* e *UnlockBits*), le prestazioni sono più che accettabili ed attualmente superiori alle necessità dei processi standard, la massima frame rate supportata da un comune PC domestico è circa 20-25 fps.

La gestione del flusso dati (*vetFrameRGB24*) non è ottima, sia per il controllo sulla frame rate (tempo di elaborazione non considerato) sia per l'assenza di un vero multi-threading, il vantaggio è che è possibile controllare il flusso dati in modo estremamente elastico grazie all'opzione *AutoPush* disponibile nei filtri (*filterForm.cpp*):

```
VETRESULT filterForm::importFrom(vetFrameRGB24* img)
{
    VETRESULT ret = VETRET_OK;
    if ( img != NULL && buffer != NULL && myFilter != NULL )
    {
        ret += myFilter->importFrom(*img);
        ret += myFilter->extractTo(*buffer);

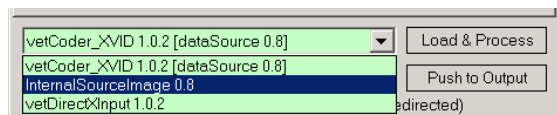
        if (cBautoEx->Checked)
            button1_Click(NULL, NULL);
    }
    return ret;
}

System::Void filterForm::button1_Click(...)
{
    if (vF && buffer != NULL)
        vF->importFrom(buffer);
}
```

Il sistema è stato implementato in questo modo come soluzione temporanea, nell'attesa dello sviluppo completo del componente *vetProcess*. Questo oggetto si occupa (occuperà) di gestire una catena di filtri in modo ottimizzato e semi-automatico attraverso threads e l'accesso diretto ai buffer per minimizzare le inutili operazioni di copia, come si può notare dall'estratto di codice, WorkShop non sfrutta il sistema di accesso diretto al buffer interno (*dump\_buffer\**).

Dopo l'elaborazione, il valore del checkbox *cBautoEx* abilita la redirectione del un frame al successivo componente (qualora esista), *vF* è il puntatore a un implementazione dell'interfaccia di output e quindi si tratta di un altro filtro oppure di un componente di tipo *vetOutput* (finestra di visualizzazione).

Le finestre di tipo sorgente (per esempio *dxinputForm.h*) e di tipo filtro (*filterForm.h*) condividono alcuni controlli del flusso dati (in fondo alla finestra), la lista scorrevole *cbViews* (combo box) elenca i nomi delle possibili destinazioni (in pratica aggiorna il puntatore *vF*), il prototipo *viewsUpdate()* imposto dall'interfaccia *sourceInterface* è chiamato dal framework (WorkShop) quando viene istanziata una nuova finestra, l'implementazione che segue è estratta dalla classe *filterForm* (*filterForm.h*) ed aggiorna la lista:



```
void viewsUpdate(void)
{
    int oldSel = cBviews->SelectedIndex;

    cBviews->Items->Clear();
    cBviews->Items->Add( new System::String(S"NULL Output" ) );

    for (int i=0; i<this->MdiParent->MdiChildren->Count; i++) {
        if ( this->MdiParent->MdiChildren->get_Item(i)->GetType()->GetInterface("outputInterface") &&
            !this->MdiParent->MdiChildren->get_Item(i)->Equals(this) )
            cBviews->Items->Add( (static_cast<Form*>(this->MdiParent->MdiChildren->get_Item(i)))->Text );
    }
    // [...]
}
```

Il controllo dentro il ciclo assicura che la finestra presa in considerazione nell'array `MdiChildren` sia una valida destinazione dati, la seguente funzione è chiamata dal controllo `cBviews` quando l'utente seleziona un elemento:

```
System::Void cBviews_SelectedIndexChanged(System::Object *sender, System::EventArgs *e)
{
    for (int i=0; i<this->MdiParent->MdiChildren->Count; i++) {
        if (static_cast<Form*>(this->MdiParent->MdiChildren->get_Item(i))->Text->Equals(cBviews->SelectedItem->ToString()))
        {
            vF = static_cast<outputInterface*>(this->MdiParent->MdiChildren->get_Item(i));
            vF->setSource(this);
            break;
        }
    }
}
```

La selezione di un elemento prevede di aggiornare il puntatore `outputInterface* vF` (utilizzato nel codice `button1_click()` ) e nel caso della sorgente il puntatore `sourceInterface* sF` tramite il prototipo `setSource()`:

```
public: int setSource(sourceInterface* NsF) {
    if (NsF == NULL) {
        cBsources->SelectedIndex = -1;
        return 0;
    }
    for (int i=0; i<cBsources->Items->Count; i++) {
        if (dynamic_cast<Form*>(NsF)->Text->Equals(cBsources->Items->Item[i]->ToString()))
        {
            cBsources->SelectedIndex = i;
            sF = NsF;
            return 0;
        }
    }
    cBsources->SelectedIndex = -1;
    return 1;
}
```

Sia durante lo sviluppo di `WorkShop` che nell'utilizzo pratico, si rivela utile un sistema di logging, la maggior parte delle informazioni sono inviate dai componenti (filtri, sorgenti) al sistema che le scrive in fondo al file (`append`) preposto `vetWS3.log`:

```
static_cast<Form1*>(this->MdiParent)->writeLog(S"Logged String", this);
```

il metodo chiamato dai `forms` è implementato in questo modo:

```
void Form1::writeLog(String* message, Form* sender)
{
    if (logWriter == NULL)
        return;
    if ( sender != NULL && sender != logLastSender )
        logWriter->Write("\r\n___from_ {0}:", sender->ToString());
    logWriter->WriteLine(message);
    if (sender)
        logLastSender = sender;
    else
        logLastSender = this;
}
```



Il codice è reso più complesso in modo da evitare la ripetizione del componente chiamante in operazioni consecutive, l'oggetto *logWriter* è uno *StreamWriter* ed è allocato con il metodo statico:

```
logWriter = File::AppendText( logFilePath );
```

Segue l'estratto di una breve sessione:

```
#####  
#####  
# VETLib WorkShop Log File NEW SESSION  
#  
# WorkShop Built: 0.7.256 Alpha  
#  
# Execution Time: martedì 6 dicembre 2005, 22.24.28  
#  
  
Starting Up..  
  Initializing Child Windows..  
  Creating GUI..  
  Initializing Reasources Array.. [SUPPORT 64 RES MAX]  
  Shutting SplashForm down..  
Loading last Job.. DISABLED  
System is Ready.  
  
Initializing Resource Manager..  
___from_ vetWS3.resManForm, Text: Resource Manager 1.0:  
Initializing: Loading Resources..  
Plug-In System supports version: 1 . 3  
Importing a new Resource..  
Allocation Plug-In Interface Object.. [loaderDLL]  
Loading file: ..\vetDigitalFilterPlugIn.dll  
DLL Loaded. Validating Plug-in..  
Plug-in Validated..  
  Resolved: Digital Filter [vetFilter]  
Adding Resource to enviroment..  
  
Adding new resource to menu..  
New InternalSourceImage Instance..Updating Child Windows..  
___from_ vetWS3.intSourceImageForm, Text: InternalSourceImage 0.8:  
Updating buffer..  
Filling buffer..  
New Visualization Window Instance..Updating Child Windows..  
PUSHING frame..  
  
Opening a new PlugIn Interface..  
  Getting loaderDLL object..  
  Detecting PlugIn ClassType.. [DLL File: vetDigitalFilterPlugIn.dll]  
  Creating a new sourceForm Interface.. [vetFilter ClassType PlugIn]  
  Creating a new PlugIn Core Instance..  
  Initializing sourceForm Interface.. [Core: vetFilter, PLS: ver1.rev3, DLL:0.2.2]  
___from_ vetWS3.filterForm, Text: vetFilter Interface 1.0:  
  Initializing: Loading Filter Informations..  
  Initializing: Loading Filter Functions Informations..  
Updating Child Windows..Initializing GUI..  
  Digital Filter is ready.  
New Visualization Window Instance..  
Updating Child Windows..  
Invoking function: INTsetDefaultKernel(INT)  
  
___from_ vetWS3.intSourceImageForm, Text: InternalSourceImage 0.8:  
  Extracting frame.. (copying frame data)  
  
Shutting WorkShop Down...  
  Checking Child Windows... 4 found  
  Closing Child Resource Manager 1.0...  
  Closing Child Visualization #1...Updating Child Windows..  
  Closing Child InternalSourceImage 0.8...  
Updating Child Windows..  
Freeing Buffer..  
Destroying Interface..  
  Closing Child vetFilter Interface 1.0...Updating Child Windows..  
___from_ vetWS3.filterForm, Text: vetFilter Interface 1.0:  
  Destroying Filter Functions Informations..  
  Destroying Interface..  
  Closing Threads...  
  Removing Resources..  
  Closing Threads...  
  Removing Resources..  
___from_ vetWS3.resManForm, Text: Resource Manager 1.0:  
  Destroying Interface..
```

## 4.3 - Dynamic PlugIn System

La classe *loaderDLL* è l'interfaccia tra WorkShop ed il plugin, si occupa di caricare la risorsa (.dll), convalidare la versione, inizializzare l'istanza del componente e liberare la memoria in seguito.

Il set di funzioni per l'accesso a DLL standard sono dichiarate in *winbase.h* (incluso in *windows.h*) e implementate nel kernel di Windows (*kernel32.lib*):

```
HMODULE LoadLibrary(LPCTSTR lpFileName);
BOOL FreeLibrary(HMODULE hModule);
FARPROC GetProcAddress(HMODULE hModule, LPCTSTR lpProcName);
```

Il plugin esporta una serie di metodi che consentono l'accesso diretto al componente, l'interfaccia è riconosciuta grazie al numero di versione e numero di revisione, attualmente sono supportati i plugins fino alla versione 1 revisione 3, la gestione del modulo sarà differente per valori superiori.

Ovviamente WorkShop può integrare e utilizzare solo componenti che rispondono ad una serie di caratteristiche note a priori, nel capitolo primo si è parlato ampiamente del concetto di interfaccia, ad esempio *vetFilterGeometric* implementa *vetFilter*, anche se WorkShop non conosce il funzionamento della classe derivata, sicuramente le funzioni imposte dalla super-classe sono disponibili, è come un semplice (*down*)casting:

```
vetFilterGeometric* myF = new vetFilterGeometric();
vetFilter* myFup = static_cast<vetFilter*>(myF);
[... ]
myFup->reset();
myFup->importFrom(*buffer);
myFup->extractTo(*buffer);
```

I metodi imposti dalle super-classi riconosciute da Workshop sono quindi completamente integrati nel sistema, ma ovviamente un componente implementa funzioni proprietarie e normalmente necessita di una serie di parametri, in effetti la principale comodità di un'interfaccia grafica è la velocità e la semplicità del setup delle variabili in run-time.

Questi metodi e parametri sono sconosciuti al sistema poiché non sono stati dichiarati durante la compilazione (non conosco la dichiarazione della classe del componente, ma solo delle super-classi), ciò complica notevolmente la configurazione dei componenti di cui comunque si fa carico il WorkShop ed alcuni metodi predefiniti inclusi nei plugin.

Il sistema riconosce il tipo di componente in base al valore *static const vetClassType* che viene ereditato dalle super-classi (0 = UNKNOWN), i *ClassType* compatibili con la versione corrente (*VETLib WorkShop 0.7.256*) sono *vetInput*, *vetOutput*, *vetFilter*, *vetBuffer*.

Lo sviluppo e l'integrazione di DLL può essere molto ostico, senza entrare troppo nel dettaglio analizziamo la struttura di un plugin 1.3, la risorsa esporta le seguenti funzioni:

```
int __callback_function (__vetPlugInFuncParam_ID,
                        __vetPlugInParams*,
                        __vetPlugInParams** );
int __callback_function_list_count ( void );
int __callback_function_list_info ( __vetPlugInFuncInfo** );
HRESULT openSetupDialog ( HWND );
void* constructInstance ( void );
void destructInstance ( void );
void getVETPlugInInternalName ( char* );
void getVETPlugInFullName ( char* );
DWORD getVETPlugInClassType ( void );
DWORD getVETPlugInSystemVersion ( void );
DWORD getVETPlugInSystemRevision ( void );
HRESULT DllGetVersion ( dllversioninfo* );
```

Le due funzioni *constructInstance* e *destructInstance* si occupano rispettivamente di creare una nuova istanza del componente VETLib e di rimuoverla, la variabile che memorizza il puntatore dell'oggetto durante la sessione è statica, il metodo *constructInstance* ritorna un puntatore generico (*void\**) che viene convertito da WorkShop con un casting nella superclasse relativa, ciò è dovuto al fatto che WorkShop non conosce il vero componente e non può stanziarlo direttamente. Il codice di questi metodi è universale per tutti i plugin ed è contenuto nel file *ws\_plugin\_src.cpp*:

```
extern VETPLUGIN_CLASSNAME* vetClassHistance;

WS_DLL_13_API void* constructInstance(void)
{
    vetClassHistance = NULL;
    vetClassHistance = new VETPLUGIN_CLASSNAME();
    return static_cast<void*>(vetClassHistance);
}










WS_DLL_13_API void destructInstance(void)
{
    if (vetClassHistance)
        delete vetClassHistance;
}
```

Dove *VETPLUGIN\_CLASSNAME* è definita (precompilazione) nel file *WS\_DLL\_13.h*:

```
#define VETPLUGIN_CLASSNAME    vetDigitalFilter
```

Nel file *ws\_plugin\_src.cpp* sono contenuti anche gli altri metodi citati, non devono assolutamente essere modificati dallo sviluppatore poiché sono necessari al funzionamento dell'interfaccia tra WorkShop e il componente "reale" (nel caso precedente *vetDigitalFilter*), ad esempio le funzioni *getVETPlugIn\** contraddistinguono la versione del plugin, modificando la configurazione predefinita il componente potrebbe non essere in grado di interagire con il sistema (ad esempio impostando un valore diverso da *ver.1 rev.3* WorkShop 0.7.256 si rifiuta di caricare la DLL).

Quindi i files contenuti in un progetto di un plugin standard sono i seguenti:

 WS_DLL_13.cpp	Sorgente dell'interfaccia plugin ( <i>da modificare</i> )
 WS_DLL_13.h	Header dell'interfaccia plugin ( <i>da modificare</i> )
{your prj files..}	Header e Sorgente del componente (o libreria se incluso)
 VETLib.lib	La libreria necessaria al componente
 WS_DLL_13.dsp	Visual Studio 6.0 Project file
 WS_DLL_13.dsw	Visual Studio 6.0 Workspace file
 ws_plugin_def.h	Definizioni di sistema ( <i>non modificare</i> )
 ws_plugin_func.h	Dichiarazioni di sistema ( <i>non modificare</i> )
 ws_plugin_src.cpp	Metodi di sistema ( <i>non modificare</i> )
 ws_plugin_exp.def	Exports (lista di metodi esportati) ( <i>non modificare</i> )

Il problema principale è la configurazione del componente, le soluzioni sono essenzialmente due, entrambe sono ammesse dal sistema:

- Delegare la configurazione dei parametri del componente alla DLL stessa, tramite un'interfaccia visuale di tipo *Dialog*, tale soluzione è ampiamente diffusa in tutti i sistemi e sotto-sistemi Windows. Il codice di start-up dell'interfaccia deve essere inserito nella funzione *openSetupDialog*, questo esempio, estremamente banale, visualizza un messaggio di errore.

```
WS_DLL_13_API HRESULT openSetupDialog(HWND hParentWnd = NULL)
{
    //add your setup dialog here

    MessageBox( hParentWnd,
                "Sorry, Setup dialog has not been implemented jet.",
                "vetDigitalFilter PlugIn",
                MB_OK | MB_ICONWARNING
                );

    return 1;
}
```

Lo sviluppo di applicazioni GUI su sistemi Windows può creare grossi problemi agli sviluppatori neofiti, quanto prima saranno disponibili interfacce template per semplificare al massimo lo sviluppo, la prossima versione 2 dei plugin prevede la compatibilità con DLL .NET che permettono lo sviluppo rapido e visuale di interfacce (RAD).

- Permettere a WorkShop di interagire direttamente con il componente senza conoscere la sua interfaccia a priori, la funzione di sistema *GetProcAdd()* ricerca nell'*export table* l'indirizzo fisico di una funzione basandosi sul nome, attraverso una serie di prototipi definiti è quindi possibile chiamare un qualunque metodo di cui si conoscano le caratteristiche principali, i metodi *\_\_callback\_\** costituiscono il cuore del sistema.

Il tipo *\_\_vetPlugInFuncParam\_ID* (definito come *int*, progressivo in *[0, methodCount[* ) contraddistingue il codice univoco che identifica un metodo, la struttura *\_\_vetPlugInFuncInfo* raccoglie una serie di informazioni (id, nome, parametri, valore di ritorno), *\_\_vetPlugInParams* è la struttura dedicata al passaggio di parametri tra il sistema e il componente.

La funzione *\_\_callback\_function\_list\_count* restituisce il numero di funzioni accessibili (è necessario al WorkShop per istanziare la memoria che conterrà le informazioni sui metodi) e *\_\_callback\_function\_list\_info* fornisce le informazioni sui metodi esportati, quindi WorkShop è in grado di selezionare il prototipo corretto per chiamare i metodi esportati dal componente, l'accesso al componente è effettuato attraverso un proxy: la funzione *\_\_callback\_function* interpreta il codice del metodo richiesto e passa i relativi parametri.

La conversione di un componente a plugin prevede solo di modificare l'header e queste funzioni nei file *WS\_DLL\_13.h* e *WS\_DLL\_13.cpp* rispettivamente, la prossima sezione esplicita i passi coinvolti nella creazione di un nuovo plugin, gli sviluppatori interessati al funzionamento del sistema di caricamento DLL possono analizzare il codice sorgente dei files: *loaderDLL.h*, *loaderDLL.cpp*, *resManForm.cpp*, *filterForm.cpp*.

Seguono alcuni frammenti significativi, si suppone che le variabili siano già inizializzate:

```
loaderDLL* myPlugInObj;           // assume instance of..
vetFilter* myFilterInstance;     // assume instance of..

int func_infoCount = myPlugInObj ->get__callback_function_list_count();
func_infoAr = new __vetPlugInFuncInfo*[func_infoCount];
```

### ■ Caricamento informazioni sui metodi esportati:

```
for (int i=0; i<func_infoCount; i++)
    { func_infoAr[i] = new __vetPlugInFuncInfo; }
myPlugInObj->get__callback_function_list_info(func_infoAr);
for (int i=0; i<func_infoCount; i++)
    {
        lBcb->Items->Add(new String( func_infoAr[i]->name ));
    }
```

### ■ Chiamata di un metodo:

```
__vetPlugInParams* f_param = new __vetPlugInParams;
String* sParStr = new String(func_infoAr[lBcb->SelectedIndex]->param1);
String* data = tBparam1->Text;

if ( sParStr->Equals(S"BOOL") )
    f_param->bool_value = Convert::ToBoolean( data );
else if ( sParStr->Equals(S"INT") )
    f_param->int_value = Convert::ToInt32( data );
else if ( sParStr->Equals(S"DOUBLE") )
    f_param->double_value = Convert::ToDouble( data );
else if ( sParStr->Equals(S"CHARP") )
    f_param->charP_value =
        (char*)(Marshal::StringToHGlobalAnsi(data)).ToPointer();
else if ( !sParStr->Equals(S"VOIDP") )
    // error.. Not implemented jet.
else if ( !sParStr->Equals(S"VOID") )
    // error.. Unknown Parameter Type

f_param->result = NULL;
int res = 0;
try {
    res = myPlugInObj->do__callback_function(
        func_infoAr[lBcb->SelectedIndex]->id, f_param );
}
catch (System::Exception* ex) { return; } // [...]

if (f_param->result != NULL) {
    String* retType = func_infoAr[lBcb->SelectedIndex]->result;
    if (retType->Equals(S"INT") ) {
        int val = *( static_cast<int*>(f_param->result) );
        tBreturn->Text = val.ToString();
    }
    else if (retType->Equals(S"DOUBLE") ) {
        double val = *( static_cast<double*>(f_param->result) );
        tBreturn->Text = val.ToString();
    }
    else if (retType->Equals(S"BOOL") ) {
        bool val = *( static_cast<bool*>(f_param->result) );
        tBreturn->Text = val.ToString();
    }
    else
        // unknown / not implemented..

    delete val;
}
```

## 4.4 - WorkShop Plugin Development

Lo sviluppo di un plugin basato su un componente esistente è reso molto più semplice dal sistema integrato, segue l'analisi dettagliata dei passi da compiere:

0. Procurarsi un progetto template (standard), l'utility *vetPS.exe* genera anche questo progetto (abilitare l'opzione "WorkShop plugin" nella scheda "Projects").
1. Aprire il progetto ed eventualmente aggiungere i propri file sorgente e le librerie esterne richieste dal componente.
2. Modificare il file header del plugin (*WS\_DLL\_13.h*).

```
#include "[...]\source\filters\vetFilterGeometric.h"
// include here your module class header

#define VETPLUGIN_CLASSNAME      vetFilterGeometric
// type your class name here (same as class declaration)

#define VETPLUGIN_CLASSNAMEq    "vetFilterGeometric"
// type your class name with QUOTES here (so it's a string)

#define VETPLUGIN_FULLNAME      "Geometric Editing Filter"
// type full extended name of your module

#define VETPLUGIN_MAJORVERSION  0x00000001
#define VETPLUGIN_MINORVERSION  0x00000000
#define VETPLUGIN_BUILDNUMBER   0x00000002
// your version informations, as HEX
```

3. Modificare il file sorgente del plugin (*WS\_DLL\_13.cpp*)

```
// update value (5) with the number of methods your will export
DLL_1_API int __callback_function_list_count ( void )
{
    return 5;    // means infoPArray[5]
}

// this function exports informations about internal methods
// any error will be fatal!

// values are CASE SenSiTiVe, id must be PROGRESSIVE [0, methodCount[

DLL_1_API int __callback_function_list_info( __vetPlugInFuncInfo** infoPArray )
{
    infoPArray[0]->init( 0,           // progressive id
                        "setRunMode", // name
                        "INT",        // param1
                        "VOID",       // result
                        "void setRunMode(RUNMODE mode);" // c_decl - optional
                      );

    // [... informations for method 1, 2, 3 ...]

    infoPArray[4]->init( 4,           // progressive id
                        "getRunMode", // name
                        "VOID",       // param1
                        "INT",        // result
                        "RUNMODE getRunMode();" // c_decl - optional
                      );

    return 0;
}
```

Resta da modificare la funzione proxy tra Workshop ed il componente:

```
DLL_1_API int __callback_function(__vetPlugInFuncParam_ID f_ID,
                                __vetPlugInParams* f_param,
                                __vetPlugInParams** f_param_xtra = NULL )
{
    switch ( f_ID )
    {
    case 0:
        vetClassHistance->getParameters().setRunMode(
            (vetFilterGeometricParameters::RUNMODE) f_param->int_value );
        break;

    // [... Cases of method 1, 2, 3 ...]

    case 4:
        {
            vetFilterGeometricParameters::RUNMODE ret =
                vetClassHistance->getParameters().getRunMode();
            int* retReal = new int;
            *retReal = (int) ret;
            f_param->result = static_cast<void*>(retReal);
            break;
        }

    default:      return 1;    // unknow method id
    }
    return 0;
}
```

Notare che il valore di ritorno è un puntatore generico, quindi è necessario istanziare un oggetto ed effettuare un down-casting (statico), WorkShop in base alle informazioni sul metodo effettua un up-casting e valuta il risultato, la memoria è liberata dall'applicazione host (Workshop). Il sistema corrente (ver1.rev3) supporta solo i seguenti parametri e valori di ritorno:

```
int      →   int_value;
bool     →   bool_value;
double   →   double_value;
char*    →   charP_value;
```

la selezione è basata sulla dichiarazione del metodo (pagina precedente), quindi occorre prestare molta attenzione visto che anche una lettera minuscola è fatale.


La funzione di start-up di un interfaccia di configurazione visuale è

```
_DLL_1_API HRESULT openSetupDialog(HWND hParentWnd = NULL)
{
    //add your setup dialog here, here we show an error.

    MessageBox( hParentWnd,
                "Sorry, Setup dialog has not been implemented jet.",
                "vetFilterGeometric PlugIn",
                MB_OK | MB_ICONWARNING
    );

    return 1;
}
```

4. Compilare e costruire la DLL.
5. Testare la DLL.



*Certi libri sembrano scritti  
non perché leggendoli si impari,  
ma perché si sappia che l'autore  
conosceva qualche cosa.*

*Joann Wolfgang Von Goethe*



### ./README

```

                                V E T L i b
-----
                                Video Elaboration & Transmission LIBrary
                                Version
                                1.0.2.25-i686-32bit
                                20/01/2006

```

```
Coding Language:      ANSI C++ (ISO/IEC 14882:2003)
Category:             Image & Video Processing FrameWork
Platform:             *NIX, WIN32
License:              Open source, GPL
Developer (founder): Alessandro Polo
WebSite:              http://www.ewgate.net/vetlib/

```

```
Star(t) date:        13/06/2005
Last Update:         20/01/2006

```

\*\*\*\*\*

#### What is it?

=====

Video Processing C++ Library, designed for testing and developing filters, (de)coders.

VETLib basic built is fully compatible with Windows and NIX operative systems, some special builds (not available on all platforms) require external libraries such as libmpeg3, quicktime4linux, xvidcore, v4l, qt, gtk, DirectX.

VETLib implements many tools for Video processing developing, with few line of code you may create a MPEG to QuickTime converter, Cam Recorder, Video Player and much more.

#### More informations

=====

Project's documentation is available in HTML and other formats in ./docs directory. Frequently Asked Questions are listed in ./FAQS document, VETLib developers are named in ./AUTHORS.

An updated version of this document (and all distribution) is available online at VETLib WebCenter: <http://lnx.ewgate.net/vetlib/distr/README>.

#### Under Linux

=====

Library and new packages are currently compiled with GNU C++ Compiler 3.3.4 or later. Special builds supported on Linux:

- o V4L [require video4linux library installed]
- o QT (GUI) [require Trolltech QT library installed]
- o GTK (GUI) [require GTK library installed]

- o ImageMagick [require ./support/ImageMagick/]
- o MPEG1-2 [require ./support/libmpeg3/]
- o XVID (MPEG4) [require ./support/xvidcore/]
- o QuickTime [require ./support/quicktime4linux/]

#### Under Windows

-----

Library and new packages are currently compiled with Visual C++ 6.0, Borland C++ 6.0.  
Special builds supported on Windows:

- o DirectX [require\* DirectX SDK]
- o XVID (MPEG4) [require\* ./support/xvidcore/]
- o ImageMagick [require\* ./support/ImageMagick/]

\* Borland project for these special builds are NOT available  
(OMF error, check ./support/NOTES)

#### Using

-----

Binaries (all) are stored in ./lib folder, read ./lib/README document  
for informations about each built (and its content).

Read ./USE for informations about using VETLib in your applications.  
Check Tests and Samples applications in ./tests folder.

VETLib WorkShop is a great tool for testing and using VETLib components,  
it comes with some integrated plugins (image/mpeg4/directx source,  
visualization, ..) but you may add new plugins at run-time.  
(coded in C++ .NET, for Windows only)

#### Compiling

-----

Read COMPILER for indepth informations;

#### Extending

-----

VETLib is very modular, it's quite easy for developers to add new packages  
or extend existing packages.

Developers interested in extending library should read ./EXTEND document and  
use Package Starter Kit (./packages), the tool Package Studio creates base  
source and project files (available for Windows systems only).

Read ./TODO if you wish join VETLib project;  
Read ./BUGS for well know bugs and problems list;

#### License

-----

GNU General Public License, read LICENSE for more informations.

#### Contact

-----

Questions, Comments and Tips to [vetlib@ewgate.net](mailto:vetlib@ewgate.net), [alessandro.polo@ewgate.net](mailto:alessandro.polo@ewgate.net)

## ./USE

Sample applications and tests are located in ./tests/ directory.  
Here are listed some simple task which may be easily build with VETLib:

### MULTI-PLATFORM

=====

- o Use frame objects (data structure)  
classes: ./source/vetFrame\*.h
- o Read/Write any image  
class: ./source/codecs/vetCodec\_BMP.h  
class: ./source/codecs/vetCodec\_IMG.h
- o Decode streams  
class: ./source/coders/vetCodec\_XVID.h
- o Process images  
classes: ./source/filters/\*.h
- o Digital Filtering (lowpass, sobel, ..)  
class: ./source/filters/vetDigitalFilter.h  
class: ./source/math/vetDFMatrix.h
- o Motion Estimation related applications  
classes: ./source/motion/\*.h
- o Thread Support  
class: ./source/vetThread.h

### LINUX only

=====

- o Interface with most capture devices through Video4Linux  
class: ./source/inputs/vetVideo4Linux.h
- o Show images and videos (QT library)  
class: ./source/outputs/vetWindowQT.h  
class: ./source/outputs/vetWindowGTK.h
- o Decode MPEG1-2 stream  
class: ./source/coders/vetCodec\_MPEG.h
- o Decode MPEG4 stream (XVID)  
class: ./source/coders/vetCodec\_XVID.h
- o Encode/Decode QuickTime stream (MOV files)  
class: ./source/coders/vetCodec\_MOV.h

### WINDOWS only

=====

- o Interface with most capture devices through DirectX (DirectShow)  
class: ./source/inputs/vetDirectXInput.h,  
class: ./source/inputs/vetDirectXInput2.h
- o Show images and videos (Windows GDI)  
class: ./source/outputs/vetWindow32.h

## SOURCE CODE Hierarchy

=====

Framework's standard class hierarchy:

```
./source/           Base classes & interfaces
./source/buffers/   Buffers utilities (inherit vetFilter)
./source/codecs/    (de)coders (images and streams)
./source/filters/   Filters (inherit vetFilter)
./source/inputs/    Image/Video data sources
./source/libETI/    ETILib support (image, picture)
./source/math/      Math functions and utilities
./source/network/   Networking related components
./source/outputs/   Image/Video data output
./source/vision/    Motion Detection processes
```

## STATIC Library

=====

VETLib binaries are stored in ./lib folder, there is a full built and some special builds (useful for local building), read ./lib/README document for informations about each built and its content.

Not-Expert developers should (download and) use FULL library for Windows (VETLib.lib) and on Linux platform it would be easier to download packages of external libraries (then compile them by yourself).

If your interested to compile VETLib on your system, read ./COMPILE document.

You should have a look inside ./tests/Makefile and check projects in ./tests/mvc.

## LINUX Tips

=====

A very simple compilation command is:

```
g++ myMainApp.cpp ../lib/VETLib.a -o outputBin.out
```

here, we have all required objects included in static library, next sample is the current command for compiling vetLinuxMPEGPlayerGTK test:

```
g++ app_vetLinuxMPEGPlayerGTK.cpp ../lib/VETLib.a
-L/usr/lib/ -lpthread -lmpeg3
-o app_vetLinuxMPEGPlayerGTK.out
'pkg-config --cflag --lib gtk+-2.0'
```

second and last lines configure two external libraries: libmpeg3 and gtk2, they are required by vetCodec\_MPEG and vetWindowGTK components.

You may read ./tests/Makefile for more compilation parameters samples

## ./COMPILE

All projects share many source files, stored in ./source directory and sub-directories, headers are located in the same folder of source file, ./include/VETLib.h includes most class' headers.

Project files for Windows are located in ./lib/mvc/ and ./lib/bcb/, Makefile for Linux is ./Makefile, default output folder is ./lib/ (as relative path), Windows' binaries are named as project's filename with classic .lib extension, linux's Makefile writes to VETLib.a (.i don't like libVET.a), see ./lib/README for details about each built.

External libraries (required by special builds) are located in ./support folder, check ./support/NOTES for more informations.

### LINUX

=====

Compatible with most GNC C++ Compilers, make configuration is stored in

./Makefile

Execute:

```
make VETLib : build standard VETLib framework;
              [fully MultiPlatform, only C++ Standard Lib. is required]

make v4l : build VETLib framework with Video4Linux support;
           [require v4l library]

make xvid : build VETLib framework with XVID (MPEG4) support;
           [require xvidcore library]

make quicktime : build VETLib framework with quicktime (MOV) support;
                [require quicktime4linux library]

make mpeg : build VETLib framework with MPEG 1-2 support;
            [require libmpeg3 library]

make gui : build VETLib framework with GUI support;
           [require QT, GTK library]

make magick : build VETLib framework with ImageMagick support;
              [require ImageMagick library]

make all : build VETLib with all available options;
           [all external libraries are required]
```

### WINDOWS

=====

Projects' files are available in ./lib/ folder for following environments:

```
o Borland C++ Builder 6.0 (or later) * {Menu->Project->Build VETLib}

    /lib/bcb/VETLib_bcb.bpr          [build minimal VETLib framework]
                                   BCB MakeFile (VETLib_bcb.mak)
                                   {execute MAKE .-fVETLib_bcb.mak}
```

\* BCB NOTE:

Base library only is supported by BCB compilers, external libraries (such as DirectX, ImageMagick, xVidcore) are NOT compatible with Borland Linker (mainly because of COFF<->OMF problem), so you will need Microsoft Visual Studio to build complete library.

- o Microsoft Visual C++ 6.0 {Menu->Build->Build VETLib}
  - /lib/mvc/VETLib\_base.dsw [build minimal VETLib framework]  
output: VETLib\_base.lib
  - /lib/mvc/VETLib\_dx.dsw [build VETLib framework with DirectX support,  
require DirectX SDK 9c, see ./support/NOTES  
and last part of this document for details]  
output: VETLib\_dx.lib
  - /lib/mvc/VETLib\_xvid.dsw [build VETLib framework with MPEG4 support,  
require xvidcore library, to build xvidcore.lib  
use project ./support/xvidcore/build/win32/xvidcore.dsw]  
output: VETLib\_xvid.lib
  - /lib/mvc/VETLib\_im.dsw [build VETLib framework with ImageMagick support,  
require imagemagick library, to build follow  
READMEs and add include folder to MVS]  
output: VETLib\_im.lib
  - /lib/mvc/VETLib\_full.dsw [build FULL VETLib framework,  
require all listed..]  
output: VETLib.lib, VETLib\_full.lib
  
- o Microsoft Visual C++ 7.0 (.NET) [ALPHA PROJECT]
  - /lib/mvc7/VETLib\_full.vcproj [build complete VETLib framework]  
output: VETLib\_vc7.lib

---

## EXTERNAL LIBRARIES

---

## EXTERNAL LIBRARIES

In ./support/ folder are located all required libraries for VETLib, standard framework uses only C++ standard library and it's fully multi-platform (ANSI C++).

When building with MPEG, MOV, XVID or DirectX support, VETLib needs some external libraries, they may be unavailable for some systems and they probably need to be installed or built on your system.

./support/NOTES report some nice tips and known bugs for installing libraries, read it.

You may download libraries package from VETLib website or (better) download each Library from its updated homepage and install on your system, read file ./support/NOTES before proceeding.

### Current External Libraries:

- o ccvt Multi-Platform  
required in ./source/vetUtility.cpp
- o qt Multi-Platform  
required in ./source/outputs/vetWindowQT.cpp
- o gtk Multi-Platform  
required in ./source/outputs/vetWindowGTK.cpp
- o imagemagick Multi-Platform  
required in ./source/coders/vetCodec\_IMG.cpp
- o libmpeg3 [1.5.4-i686] Linux only  
required in ./source/coders/vetCodec\_MPEG.cpp
- o quicktime4linux [1.4] Linux only  
required in ./source/coders/vetCodec\_MOV.cpp
- o xvidcore [1.1.0beta2-i686] Linux and Windows32  
required in ./source/coders/vetCodec\_XVID.cpp

- o DirectX 9c [before 10.0] Window32 + DirectX runtime  
     required in ./source/inputs/vetDirectXInput.cpp  
               ./source/inputs/vetDirectXInput2.cpp

LINUX  
 =====

- o v4l : Install: - Usually Included by system  
       Home: http://www.exploits.org/v4l
- o xvidcore : Install: ./support/xvidcore/doc/install (or package)  
       Home: http://www.xvid.org  
       Parameters: -L/usr/lib/ -lxvidcore
- o quicktime : Build: ./support/quicktime4linux/docs/index.html  
       Home: http://www.heroinewarrior.com/quicktime4linux  
       Parameters: -lpthread -lpng -ldl -lz -lglib  
                  -L../support/quicktime4linux/i686 -lquicktime
- o libmpeg3 : Install: Default library installation technique  
                  (build or package)  
       Home: http://www.heroinewarrior.com/libmpeg3  
       Parameters: -lpthread -lmpeg3
- o qt : Install: Usually installed  
       Home: http://www.trolltech.com  
       Parameters: -L/usr/lib/qt/lib -lqt-mt
- o gtk : Install: Usually installed  
       Home: http://www.gtk.org  
       Parameters: `pkg-config --cflags --libs gtk+-2.0`
- o imagemagick: Install: Default library installation technique  
                  (build or package)  
       Home: http://www.imagemagick.org/  
       Parameters: -L/usr/local/lib -L/usr/X11R6/lib -L/lib/graphviz  
                  -lfreetype -lz -L/usr/lib -lMagick -llcms -ltiff  
                  -lfreetype -ljpeg -lpng -ldpstk -ldps -lXext -lXt  
                  -lSM -lICE -lX11 -lbz2 -lxml2 -lz -lpthread -lm  
                  -lpthread

WINDOWS  
 =====

- o DirectX : Install: SDK installation, update include and library  
                  in your compiler / RAD development system.  
       Home: http://www.microsoft.com/directx  
       Link: ./support/directx/amstrmid.lib, strmbasd.lib,  
               dbghelp.lib  
       Note: DirectX is NOT compatible with Borland Compilers  
              (COFF<->OMF).
- o xvid : Build: With Visual Studio, read ./support/NOTES,  
                  build static library (.lib)  
       Home: http://www.xvid.org  
       Link: ./support/xvidcore/build/win32/bin/libxvidcore.lib
- o imagemagick: Build: With Visual Studio, read ./support/NOTES,  
                  build static library (.lib)  
       Home: http://www.imagemagick.org/  
       Link: ./support/ImageMagick/VisualMagick/lib/\*.\*

---

## COMPILING TESTS

## COMPILING TESTS

All projects share same source files, stored in /tests directory, named /test\_<ClassName>.cpp, where <ClassName> is VETLib class to test.

### LINUX

-----

Compatible with most GNC C++ Compilers, make configuration is stored in

./tests/Makefile

Execute:

```
./tests/make all           :    build all tests;
./tests/make <ClassName>  :    build selected test;

./tests/make help         :    show available tests;
./tests/make clean        :    clean output files and objects;
```

Reference to VETLib static library (VETLib.a) is at "../lib/VETLib.a", some tests may need special VETLib built (MPEG, MOV, XVID, V4L support):

```
./tests/test_vetQWindow.cpp           Show how to use QWindow class
./tests/test_vetCodec_XVID.cpp        Show how to use vetCodec_XVID class
./tests/test_vetCodec_MOV.cpp         Show how to use vetCodec_MOV class
./tests/test_vetCodec_MPEG.cpp        Show how to use vetCodec_MPEG class

./tests/app_vetVideo4LinuxPlayer.cpp  Play /dev/video0 stream to a window
./tests/app_vetLinuxMOVPlayer.cpp     Play a QuickTime movie in a window
./tests/app_vetLinuxMPEGPlayer.cpp    Play a MPEG1-2 movie in a window (QT)
./tests/app_vetLinuxMPEGPlayerGTK.cpp Play a MPEG1-2 movie in a window (GTK)
./tests/app_vetLinuxXVIDPlayer.cpp    Play a MPEG4 movie in a window
```

### WINDOWS

-----

Built project for

- o Borland C++ Builder 6.0 (or later)

./tests/bcb/test\_<ClassName>.bpr

{Menu->Project->Run}

- o Microsoft Visual C++ 6.0 (or later)

./tests/mvc/test\_<ClassName>.dsw

{Menu->Build->Execute test\_<ClassName>.vc6.exe}

Following tests need DirectX SDK Library (9c, not version 10)

```
./tests/app_vetDirectXLamePlayer.cpp  Play first capture device in a window
```

Reference to VETLib static library is at ./lib/VETLib.lib for BCB and ./lib/VETLib\_vc6.lib for MVC.

Object and temp files are stored in ./tests/tmp folder, output binaries are moved to ./tests/bin/.

To build VETLib WorkShop, Package Studio or Distribution Manager you need Visual Studio 7.0 (.NET) and the full library binary for Windows (./lib/VETLib.lib), see README document in application's folder for details.



## ./FAQS

An HTML version of this document is available online at

<http://lnx.ewgate.net/vetlib/faqs.html>

---

|o  
| I just want to use DirectX / Video4Linux component to capture data  
| for my application, ..without reading manuals :P  
|

---

Easy, with a text pad open one of these components (or all):

```
./source/inputs/vetDirectXInput.h  
./source/inputs/vetDirectXInput2.h  
./source/inputs/vetVideo4Linux.h
```

and their own test files:

```
./tests/test_vetDirectXInput.cpp  
./tests/test_vetDirectXInput2.cpp  
./tests/test_vetVideo4Linux.cpp
```

but you are probably more interested in these simple "applications":

```
./tests/app_vetDirectXLamePlayer.cpp  
./tests/app_vetVideo4LinuxPlayer.cpp
```

check them out, code is often commented and quite simple, you may take a look to tests' project files for building your own (or just edit them):

```
./tests/mvc/  
./tests/bcb/  
./tests/Makefile
```

for more information check ./USE.

|  
\

---

---

|o  
| I just want to use movie files related components to built a player, is  
| it possible?  
|

---

Yes, but there are some many open source players.....  
Anyway, related components are located in ./source/codecs/ folder, read headers and their tests (similar filename), your should take a look to:

```
./tests/app_vetLinuxMOVPlayer.cpp  
./tests/app_vetLinuxMOVPlayerGTK.cpp  
./tests/app_vetLinuxMPEGPlayer.cpp  
./tests/app_vetLinuxMPEGPlayerGTK.cpp  
./tests/app_vetLinuxXVIDPlayer.cpp
```

|  
\

---

Contact

-----

Questions, Comments and Tips to [vetlib@ewgate.net](mailto:vetlib@ewgate.net), [alessandro.polo@ewgate.net](mailto:alessandro.polo@ewgate.net)

## ./EXTEND

Extending library means to develop a standard VETLib component, it is NOT required to build library on your platform, you may use static binaries but you should already know how to use VETLib, first read ./USE.

### Requirements

=====

You will need following items to code a VETLib component:

./lib/{..}	Static library binary for your platform (ex. VETLib.lib or VETLib.a)
./packages/*.*	Templates, sample and empty components and VETLib Package Studio software.
./source/*.h	Headers are required by source code.

### Package Starter Kit

=====

Package Starter Kit is an archive designed for extending VETLib, it comes with a great tool: Package Studio, the simpler (and best) way is to download last updated version of VETLib SDK (FULL) and PSK archives.

### Conventions

=====

- o Name header and source files as classname (Java like).  
Class Name must be formatted as follow: <prefix><ClassName>, where prefix is the name of base interface, examples:  

vetFilter	->	vetFilterGeometric
vetCodec	->	vetCodec_BMP
- o Check ./packages/NAMESPACE and choose a valid classname
- o Develop your module in directory ./packages/<className>
- o Most functions that return a state value, such as extractTo(..) or importFrom(..), must use the type VETRESULT, defined as an integer in vetDefs.h, return codes convention is 0<->OK, error else.
- o Filters should implement a parameters class (vet<className>Parameters) that serialize filter in XML format.

### Package Development

=====

A standard VETLib package contains following files:

vet<ClassName>.h	Class Header
vet<ClassName>.cpp	Class Source
test_vet<ClassName>.cpp	Test source [int main()]
Makefile	Make configuration file
test_vet<ClassName>.bpr	Borland C++ Builder 6.0 project file
test_common.bpf	Borland C++ Builder 6.0 project file
test_vet<ClassName>.dsp	Microsoft Visual Studio 6.0 project file
test_vet<ClassName>.dsw	Microsoft Visual Studio 6.0 Workspace file
vet<ClassName>.Readme	Readme of the component
vet<ClassName>.License	License of the component

Please keep all source code in ONE file (vet<ClassName>.cpp) and all classes useful for users in header vet<ClassName>.h (for example a filter includes also the vet<ClassName>Parameters class).

#### Notes

-----

If you plan to extend library you should be able to built it, read COMPILER for more informations, anyway default configuration uses static library binaries.

Then respect following conventions and tips:

- o If you are going to (re)implement streaming operators, remember that you need to redefine all (streaming) operators, also the old ones.  
Example:

To add >> vetFrameRGB96 support you have to define

```
void operator << (vetFrameRGB& img) { importFrom(img); };
```

and related

```
VETRESULT importFrom(vetFrameRGB96& img);
```

but you must also REdefine:

```
void operator << (vetFrameRGB24& img) { importFrom(img); };  
void operator << (vetFrameYUV420& img) { importFrom(img); };  
void operator << (vetFrameT<unsigned char>& img)  
                { importFrom(img); };
```

- o Implement testing code as a simple application, located in your package's folder and named 'test\_<classname>.cpp', create Makefile, Borland and Microsoft project files.

I Wish include my Package in distribution

-----

Great, please send your package to [vetlib@ewgate.net](mailto:vetlib@ewgate.net):

Note: when a package is released, files are moved and integrated into VETLib Builds, Authors' credits will be added to ./AUTHORS.

# ./ChangeLog

Here is the list of updates for each built:

NOTE: "!" means that has been cut backward compatibility on that item,  
if your contest includes it, please see details in header  
file before upgrading your implementation.

version 1.0.2.25 First Official Release [OnLine]

-----

- !o Updated Base Interfaces (vetFilter buffering reviewed)
- !o Updated Package Starter Kit
- !o Upgraded VETLib WorkShop to version 0.7.256
- !o Upgraded PlugIn System (of WorkShop) to version 1 revision 3
  - o Added new Module: vetDirectXInput2
  - o Released some plugins (included in WS distribution package)
  - o Released VETLib Presentation [ITALIAN]
  - o Released VETLib Manual [ITALIAN]
  - o Updated WebSite (./Website/)

version 1.0.1.5

-----

- !o Updated Base Interfaces
- !o Updated Base Frames { vetFrame, vetFrameRGB24, vetFrameRGB32, vetFrameYUV420, vetFrameT }
- o Added VETLib WorkShop 0.5.101 (./Tools/vetWS/vetws3.exe)
- o Updated WebSite (./Website/)
- o Updated Doxygen configuration file
- o Updated READMEs files

version 1.0.1.4

-----

- o Added VETLib Distribution Manager (./Tools/vetDM/vetdistr.exe)
- o Added VETLib Package Starter Kit (./packages)  
[include software for Win32 and templates for developers]

version 1.0.1.3

-----

- o Added vetVision Modules (./source/vision)
- o Added vetException class

version 1.0.1.2

-----

- o Added ImageMagick support (vetCodec\_IMG)

version 1.0.1.0

-----

- o Added MPEG4-XVID support (vetCodec\_XVID)
- o Added MPEG1-2 support (vetCodec\_MPEG)

version 1.0.0.0

-----

- o Beta Release

# ./TODO

## Concept ToDo List

=====

- \* Update object and framework as Java/.NET style (namespaces, static methods)
- \* Inherit all classes form vetObject? (useful for workshop, threading, qos?)
- \* Implement static methods to filters for vetFrameT<> objects
- \* Develop some Driver for DirectX e Video For Windows (I/O vetFrame\*, vetFilter)

## VETLib ToDo List

=====

- \* Debug and Update Color-Space Conversions
- \* MultiThreading Testing and Debugging
- \* vetProcess (require vetThread)
- \* MPEG interface debugging
- \* MPEG encoding (FFMPEG)
- \* MOV interface debugging
- \* XVID interface debugging, writing
- \* vetBufferSequential (Array Queue)
- \* intel Open source computer vision library interface
- \* Add support for MPEG4IP Library
- \* Add support for IEEE1394 on Linux (libraw1394)  
and DV Interface (libdv + libavc1394)
- \* upgrade Package Starter Kit in C++ .NET
- \* upgrade Distribution Manager in C++ .NET
- \* see ./Tools/vetWS/ToDo

## More ToDo List

=====

- \* Image Processing:
  - Contour Processing (Finding, manipulation, simplification of image contours)
  - Geometry (Line and ellipse fitting)
  - Features (1st & 2nd Image Derivatives)
  - Image Statistics (Count, Mean, Norm, Moments, in regions)
  - Morphology (Erode, dilate, open, close, Gradient)
- \* Video Processing:
  - Camera Calibration (checkerboard calibration pattern)
  - Active Contours (Snakes)
- \* More Utilities:
  - CSP, Template matching)
  - Matrix Math (SVD, inverse, cross-product)
- Can COFF <-> OMF problem be resolved in BCB?  
Should try to compile DirectShow base classes and xvid library  
with borland compiler and use that static library. Do we really need that?

## ./BUGS

This document refers to version 1.0.2.25-i686-32bit  
Here is the list of know bugs and errors, please report new bugs.

Known BUGS

-----

- \* vetDirectXInput: First captured frame sometimes is black, grabbing is designed for low-quality devices.
- \* vetCodec\_IMG writing image not implemented.
- \* vetCodec\_XVID encoding not implemented.
- \* vetFilterGeometric implementation is incomplete.
- \* vetThread implementation is uncompleted.
- \* vetMotionLame implementation is incomplete.
- \* vetMotionIlluminationInvariant implementation is incomplete.
- \* DirectX SDK is NOT compatible with Borland Compilers.

I Found another Bug, so what to do?

-----

Report bug and patch if you resolved the problem to VETLib Staff.

vetlib@ewgate.net or alessandro.polo@ewgate.net

Thanks for help!

I Wrote the patch for a Bug, so what to do?

-----

Great, please send the patch to vetlib@ewgate.net or alessandro.polo@ewgate.net,  
we will include in next release and report your credits.

Thanks for help!

## ./AUTHORS

List of all VETLib developers:

Alessandro Polo

-----

Originally conceived of, authored, and still maintains the VETLib project.  
He developed everything has been released until version 1.0.2.25 (current),  
including VETLib WorkShop 0.7.256, Package Studio 0.3.548 and  
Distribution Manager 0.2.957.

Email: alessandro.polo@ewgate.net  
Web: http://www.ewgate.net/alex/

Here is the list of know bugs and errors, please report new bugs.

## ./lib/README

Current built 1.0.2.25-i686-32bit (20/01/2006)

Static binaries have been built with GNU C++ Compiler 3.3.4 on Linux platform and Visual Studio 6.0 / Borland C++ Builder 6.0 on Windows platform. Note that ECB and MVC binaries are different (COFF <-> OMF).

### Base Library Content

=====

Compiled resources are:

- o Base Classes (vetObject, vetInput, vetFilter, vetVision)
- o vetFrames (vetFrameRGB24, vetFrameYUV420, vetFrameRGBA32, vetFrameHSV, vetFrameRGB96, vetFrameGrey)
- o vetUtility
- o vetHist
- o vetCodecs (vetCodec\_BMP)
- o vetFilters (vetDigitalFilter, vetFilterGeometric, vetFilterColor, vetMultiplexer, vetFilterNoiseChannel)
- o vetInputs (vetNoiseGenerator, vetPlainFrameGenerator)
- o vetOutputs (vetDoctor, vetOutputVoid)
- o vetVisions (vetMotionLame, vetMotionIlluminationInvariant)
- o vetMaths (vetStatistics)

### Linux Binaries

=====

Binaries for NIX platforms are distributed only as full version, but you will need support (external) libraries to be installed in your system if your software requires them directly or indirectly (in fact library doesn't include them).

- o VETLib.a

All objects but external libraries are missing (not linked).

Check ./tests/Makefile for linker's parameters of external libraries.

### Windows Binaries

=====

Binaries for Windows are available as (classic) static library (.lib), just include it in your linker's parameters (usually also "Add to project.." works fine), all external libraries are included (linked) into VETLib.a (or VETLib\_full.lib), you may choose a special built to save bandwidth (downloading) but some components are missing, using the full built you won't have any problems.

- o VETLib\_base.lib [COFF]

Base library, compiled with Microsoft Visual Studio 6.0, following components are not included:

- vetCodec\_XVID
- vetCodec\_IMG
- vetDirectXInput
- vetDirectXInput2

- o VETLib\_dx.lib [COFF]
 

Base library plus DirectX components (vetDirectXInput, vetDirectXInput2), it's compiled with Microsoft Visual Studio 6.0.
- o VETLib\_im.lib [COFF]
 

Base library plus vetCodec\_IMG component (ImageMagick support) (this component produce a BIG file size, about +20Mb) XVID and DirectX components are NOT included, library is compiled with Microsoft Visual Studio 6.0.
- o VETLib\_xvid.lib [COFF]
 

Base library plus vetCodec\_XVID component (xVidCore support), vetCodec\_IMG and DirectX components are NOT included, compiled with Microsoft Visual Studio 6.0.
- o VETLib.lib [COFF] ( VETLib\_full.lib )
 

Complete library, compiled with Microsoft Visual Studio 6.0, include everything has been released.  
Currently it means: Base library + vetCodec\_XVID + vetCodec\_IMG +  
+ vetDirectXInput + vetDirectXInput2
- o VETLib\_vc7.lib \* [COFF]
 

Complete library, compiled with Microsoft Visual Studio 7.0, include everything has been released.  
Currently it means: Base library + vetCodec\_XVID + vetCodec\_IMG +  
+ vetDirectXInput + vetDirectXInput2

\* Actually this is a BETA built, we had some problems with ImageMagick library.
- o VETLib\_bcb.lib [OMF]
 

Base Library compiled with Borland C++ Builder 6.0, following components are not included:

  - vetCodec\_XVID
  - vetCodec\_IMG
  - vetDirectXInput
  - vetDirectXInput2



Quando ci si trova davanti ad un ostacolo,  
la linea più breve tra due punti può essere una curva.

Bretolt Brecht



Gli headers sono contenuti nella directory `./source` e sotto-directory, associati al relativo sorgente, nelle distribuzioni dove il sorgente non è incluso sono invece disponibili in `./include` e sotto-directory, il file `./include/VETLib.h` include le singole classi e le dichiarazioni globali, nelle applicazioni basate su VETLib è consigliabile includere questo file mentre nel caso di estensioni dei singoli moduli includere gli header necessari (considerarli in `./source/`).

### Abstract Classes :

<code>vetFrame</code>	<code>[vetFrame.h]</code>
<code>vetInput</code>	<code>[vetInput.h]</code>
<code>vetOutput</code>	<code>[vetOutput.h]</code>
<code>vetFilter</code>	<code>[vetFilter.h]</code>
<code>vetCodec</code>	<code>[vetCodec.h]</code>
<code>vetVision</code>	<code>[vetVision.h]</code>
<code>vetBuffer</code>	<code>[vetBuffer.h]</code>
<code>vetObject</code>	<code>[vetObject.h]</code>

### Included Implementations:

<code>vetFrameRGB24</code>	<code>[vetFrameRGB24.h]</code>
<code>vetFrameYUV420</code>	<code>[vetFrameYUV420.h]</code>
<code>vetFrameT</code>	<code>[vetFrameT.h]</code>
<code>vetFrameRGBA32</code>	<code>[vetFrameRGBA32.h]</code>
<code>vetFrameHSV</code>	<code>[vetFrameHSV.h]</code>
<code>vetFrameGrey</code>	<code>[vetFrameGrey.h]</code>
<code>vetFrameRGB96</code>	<code>[vetFrameRGB96.h]</code>
<code>vetMatrix</code>	<code>[math/vetMatrix.h]</code>
<code>vetDFMatrix</code>	<code>[math/vetDFMatrix.h]</code>
<code>vetHist</code>	<code>[vetHist.h]</code>
<code>vetException</code>	<code>[vetException.h]</code>
<code>vetThread</code>	<code>[vetThread.h]</code>
<code>vetUtility</code>	<code>[vetUtility.h]</code>
<code>vetBufferArray</code>	<code>[buffers/vetBufferArray.h]</code>
<code>vetBufferLink</code>	<code>[buffers/vetBufferLink.h]</code>

## More Classes:

vetCodec_BMP	[codecs/vetCodec_BMP.h]
vetCodec_IMG	[codecs/vetCodec_IMG.h]
vetCodec_MPEG	[codecs/vetCodec_MPEG.h]
vetCodec_XVID	[codecs/vetCodec_XVID.h]
vetCodec_MOV	[codecs/vetCodec_MOV.h]
vetNoiseGenerator	[inputs/vetNoiseGenerator.h]
vetPlainFrameGenerator	[inputs/vetPlainFrameGenerator.h]
vetVideo4Linux	[inputs/vetVideo4Linux.h]
vetDirectXInput	[inputs/vetDirectXInput.h]
vetDirectXInput2	[inputs/vetDirectXInput2.h]
vetDXMovieLoader	[inputs/vetDXMovieLoader.h]
vetFilterNoiseChannel	[filters/vetFilterNoiseChannel.h]
vetFilterGeometric	[filters/vetFilterGeometric.h]
vetFilterColor	[filters/vetFilterColor.h]
vetDigitalFilter	[filters/vetDigitalFilter.h]
vetMultiplexer	[filters/vetMultiplexer.h]
vetOuputVoid	[outputs/vetOuputVoid.h]
vetDoctor	[outputs/vetDoctor.h]
vetWindowQT	[outputs/vetWindowQT.h]
vetWindowGTK	[outputs/vetWindowGTK.h]
vetWindow32	[outputs/vetWindow32.h]

```

typedef unsigned char uchar;

typedef int VETRESULT;

#define VETRET_OK 0 /* no errors found */
#define VETRET_PARAM_ERR 1 /* illegal parameter(s) */
#define VETRET_INTERNAL_ERR 2 /* internal routine error */
#define VETRET_ILLEGAL_USE 4 /* illegal use of function (forbidden, empty video/frame, ..) */
#define VETRET_DEPRECATED_ERR 8 /* */

#define VETRET_OK_DEPRECATED 16 /* */

#define VETRET_NOT_IMPLEMENTED 666 /* */

#define VETCLASS_TYPE_UNKNOWN 0
#define VETCLASS_TYPE_OBJECT 10
#define VETCLASS_TYPE_EXCEPTION 90
#define VETCLASS_TYPE_FRAME 1000
#define VETCLASS_TYPE_BUFFER 2000
#define VETCLASS_TYPE_INPUT 3000
#define VETCLASS_TYPE_OUTPUT 4000
#define VETCLASS_TYPE_FILTER 7000
#define VETCLASS_TYPE_CODER 5500
#define VETCLASS_TYPE_VISION 4500

// #define __VETLIB_DEBUGMODE__

// define DEBUG utility, use: DEBUG('variablename')
#ifdef __VETLIB_DEBUGMODE__

#include <stdio.h>

#define INFO(x) printf(" _NFO: %s\n");
#define DEBUG(x) printf(" _DBG: %s = %p \n", #x, x);
#define DEBUGMSG(msg, x) printf(" _DBG: %s %s = %p \n", msg, #x, x);

#else

#define INFO(x) ;
#define DEBUG(x) ;
#define DEBUGMSG(msg, x) ;

#endif // __VETLIB_DEBUGMODE__

```

```

class vetException
{
protected:

    std::string m_s;
    void* caller;
    VETRESULT retCode;

public:

    vetException() ( std::string s ) : m_s ( s );

    vetException() ( std::string message, void* callerObject, VETRESULT returnCode ) : m_s ( message );

    virtual ~vetException() { }

    std::string getDescription() { return m_s; };
    void* getCallerObject() { return caller; };

    VETRESULT getReturnCode() { return retCode; };

    friend ostream& operator << (ostream& os, vetException& p);

    enum{ vetClassType = VETCLASS_TYPE_EXCEPTION };
};

/***** USAGE

try {
    throw vetException("simple Error");
    // or throw vetException("complex Error", NULL, VETRET_ILLEGAL_USE);
}
catch ( vetException& myEx )
{
    printf("%s\n", myEx.getDescription() );
}
*/

```

```

class vetFrame
{
    protected:

        unsigned int      width;
        unsigned int      height;

        long timeStamp;

        vetFrame() : width(0), height(0) { }

        vetFrame(unsigned int w, unsigned int h) : width(w), height(h) { }

    public:

        enum VETFRAME_PROFILE
        {
            VETFRAME_NONE,           //empty
            VETFRAME_MONO,           //grayscale

            // RGB formats
            VETFRAME_RGB24,          //standard
            VETFRAME_BGR24,          //also standard
            VETFRAME_RGB32,          //
            VETFRAME_BGR32,          //
            VETFRAME_RGB96,          //
            VETFRAME_BGR96,          //

            VETFRAME_RGB565,         //pixel 16bit!
            VETFRAME_BGR565,         //pixel 16bit!
            VETFRAME_RGB555,         //pixel 16bit!
            VETFRAME_BGR555,         //pixel 16bit!

            VETFRAME_ARGB32,         //alpha + .
            VETFRAME_ABGR32,         //alpha + .
            VETFRAME_RGBA32,         //. + alpha
            VETFRAME_BGRA32,         //. + alpha

            // YUV formats
            VETFRAME_I420,           //4:2:0 planar (=IYUV)
            VETFRAME_YV12,           //4:2:0 planar (MPEG software: NxM Y + N/2*M/2 V U )
            VETFRAME_YUY2,           //4:2:2 packed (common in AVI and hardware devices)
            VETFRAME_UYVY,           //4:2:2 packed (cinepack, 2nd choice of mpeg codecs)
            VETFRAME_YVYU,           //4:2:2 packed

            //
            VETFRAME_AYUV,           //alpha + 4:4:4 planar

            VETFRAME_HSV,
            VETFRAME_CUSTOM          //not listed here..
        };

        enum VETFRAME_CHANNEL_TYPE
        {
            VETFRAME_CT_NONE,
            VETFRAME_CT_PACKED,
            VETFRAME_CT_PLANAR,
            VETFRAME_CT_CUSTOM
        };

        virtual ~vetFrame() { }

        virtual VETRESULT reAllocCanvas(unsigned int w, unsigned int h) = 0;

        virtual VETRESULT extractBrightness(unsigned char* buffer, unsigned int* size = 0) = 0;

        virtual void* dump_buffer() = 0;

        virtual unsigned int getBufferSize() = 0;

        virtual VETRESULT setBlack() = 0;
        virtual VETRESULT setWhite() = 0;

        unsigned int getWidth() const { return width; };
        unsigned int getWidth() { return width; };

        unsigned int getHeight() const { return height; };
        unsigned int getHeight() { return height; };

        virtual unsigned int getBpp() = 0;

        virtual VETFRAME_PROFILE getProfile() = 0;

        virtual VETFRAME_CHANNEL_TYPE getChannelType() = 0;

        virtual int getFOURCC() = 0;

        enum{ vetClassType = VETCLASS_TYPE_FRAME };
};

```

**Current Implementations :**

vetFrameRGB24.h, vetFrameYUV420.h, vetFrameT.h, vetFrameRGBA32.h, vetFrameHSV.h,  
vetFrameRGB96.h

```

/**
 * @class   vetFrameRGB24
 *
 * @brief   This Class implements standard VETLib I/O Frame format.
 *          An image consisting of red, green and blue pixels.
 *          PixelRGB24 Array [width*height] (raster scan)
 *          Currently PixelRGB24 is defined as 3 char (3 * 8 = 24bits)
 *
 *
 * @see     PixelRGB24
 * @see     vetFrame
 *
 * @version 0.6
 * @date    12/07/2005 - //2005
 * @author  Alessandro Polo
 *
 *
 * *****
 * VETLib Framework 1.0.2
 * Copyright (C) Alessandro Polo 2005
 * http://www.ewgate.net/vetlib
 *
 * *****/
#endif
#define __VETLIBVETFRAMERGB24_H__

#define VETFRAMERGB24_SLOWMODE

#include "vetDefs.h"

#include "PixelRGB24.h"
#include "vetFrame.h"

class vetFrameRGB24;
#include "vetFrameYUV420.h"

class vetFrameRGB24 : public virtual vetFrame
{
public:
    unsigned int width;
    unsigned int height;

    bool autoFreeData;

    PixelRGB24 *data;

public:
    enum ChannelRGB { RED, GREEN, BLUE };

    vetFrameRGB24();
    vetFrameRGB24(unsigned int width, unsigned int height);
    vetFrameRGB24(vetFrameRGB24& img);

    ~vetFrameRGB24();

    void* dump_buffer() { return static_cast<void*>(data); };

    unsigned int getBufferSize() { return (unsigned int)( width * height * 3); };

    VETRESULT reAllocCanvas(unsigned int w, unsigned int h);

    unsigned int getBpp() { return sizeof(PixelRGB24) * 8; };
    VETRESULT setBlack();
    VETRESULT setWhite();

    VETRESULT extractBrightness(unsigned char* buffer, unsigned int* size = NULL );

    vetFrameRGB24& clearWith(PixelRGB24& bg);

    VETRESULT setPixel(unsigned int x, unsigned int y, PixelRGB24 p);

    VETRESULT getPixel(unsigned int x, unsigned int y, PixelRGB24& p);

    vetFrameRGB24& operator = (vetFrameRGB24& img);
    vetFrameRGB24& operator += (vetFrameRGB24& img);
    vetFrameRGB24& operator *= (vetFrameRGB24& img);

    void operator << (const vetFrameYUV420& img);

    vetFrameRGB24& operator >> (vetFrameYUV420& img);

    VETFRAME_PROFILE getProfile() { return vetFrame::VETFRAME_RGB24; };
    VETFRAME_CHANNEL_TYPE getChannelType() { return vetFrame::VETFRAME_CT_PACKED; };

    int getFOURCC() { return 0x32424752; }; // same as RGB32

};

#endif // __VETLIBVETFRAMERGB24_H__

```

```

/** @file    vetFrameYUV420.h
 * @class   vetFrameYUV420
 *
 * @brief   This Class implements standard VETLib I/O Frame format.
 *          An image consisting of red, green and blue pixels.
 *          PixelRGB24 Array [width*height] (raster scan)
 *          Currently PixelRGB24 is defined as 3 char (3 * 8 = 24bits)
 *
 *
 * @see     PixelRGB24
 * @see     vetFrame
 *
 * @version 0.6
 * @date    12/07/2005 - //2005
 * @author  Alessandro Polo
 *
 *
 * *****
 * VETLib Framework 1.0.2
 * Copyright (C) Alessandro Polo 2005
 * http://www.ewgate.net/vetlib
 *
 * *****/

#ifndef __VETLIB_VETFRAMEYUV420_H__
#define __VETLIB_VETFRAMEYUV420_H__

#define _VETFRAMECACHE24_SLOWMODE

#include "vetDefs.h"

#include "vetFrame.h"
class vetFrameYUV420;
#include "vetFrameRGB24.h"

class vetFrameYUV420 : public virtual vetFrame
{
public:
    unsigned int width;
    unsigned int height;

    bool autoFreeData;

    unsigned char *data;

    unsigned char *Y; // = data[0]
    unsigned char *U; // = data[ width*height* ]
    unsigned char *V; // = data[ width*height*1.25 ]

public:
    enum ChannelYUV { Lum, Cb, Cr }; // YUV

    vetFrameYUV420();
    vetFrameYUV420(unsigned int width, unsigned int height);
    vetFrameYUV420(vetFrameYUV420& img);

    ~vetFrameYUV420();

    void* dump_buffer() { return static_cast<void*>(data); };

    VETRESULT reallocCanvas(unsigned int w, unsigned int h);

    unsigned int getBpp() { return 12; };
    VETRESULT setBlack();
    VETRESULT setWhite();

    VETRESULT extractBrightness(unsigned char* buffer, unsigned int* size = NULL );

    unsigned int getBufferSize() { return (unsigned int)( width * height * 1.5); };

    vetFrameYUV420& clearWith(unsigned char* bg, ChannelYUV channel);

    VETRESULT setPixel(unsigned int x, unsigned int y, unsigned char& value, ChannelYUV channel);
    VETRESULT getPixel(unsigned int x, unsigned int y, unsigned char& value, ChannelYUV channel);

    vetFrameYUV420& operator = (vetFrameYUV420& img);
    vetFrameYUV420& operator += (vetFrameYUV420& img);
    vetFrameYUV420& operator *= (vetFrameYUV420& img);

    void operator << (const vetFrameRGB24& img);
    vetFrameYUV420& operator >> (vetFrameRGB24& img);

    VETFRAME_PROFILE getProfile() { return vetFrame::VETFRAME_I420; };
    VETFRAME_CHANNEL_TYPE getChannelType() { return vetFrame::VETFRAME_CT_PLANAR; };

    int getFOURCC() { return 0x30323449; };

};

#endif // __VETLIB_VETFRAMEYUV420_H__

```

```

#ifndef __VETLIB_VETFRAMET_H__
#define __VETLIB_VETFRAMET_H__

#define _VETFRAMET_SLOWMODE

#include "vetDefs.h"
#include "vetFrame.h"

template<class T>
class vetFrameT : public virtual vetFrame
{
    public:

        unsigned int width;
        unsigned int height;

        bool autoFreeData;

        VETFRAME_PROFILE profile;

        VETFRAME_CHANNEL_TYPE      dataType;

        T *data;

    public:

        vetFrameT()

        vetFrameT(unsigned int w, unsigned int h);

        vetFrameT(unsigned int w, unsigned int h,
                  vetFrame::VETFRAME_PROFILE prof,
                  vetFrame::VETFRAME_CHANNEL_TYPE dataTy = vetFrame::VETFRAME_CT_PLANAR)

        vetFrameT(vetFrameT& img)

        ~vetFrameT()

        bool isBuiltInSupportedProfile(VETFRAME_PROFILE pr)

        unsigned int getWidth() const { return width; };
        unsigned int getWidth() { return width; };

        unsigned int getHeight() const { return height; };
        unsigned int getHeight() { return height; };

        void* dump_buffer() { return static_cast<void*>(data); };

        VETRESULT reAllocCanvas(unsigned int w, unsigned int h)

        VETRESULT extractBrightness(unsigned char* buffer, unsigned int* size = NULL )

        unsigned int getBpp()

        unsigned int getBufferSize()

        VETRESULT setBlack()
        VETRESULT setWhite()

        vetFrameT& clearWith(T& bg)

        VETRESULT setPixel(unsigned int x, unsigned int y, T p)

        VETRESULT getPixel(unsigned int x, unsigned int y, T& p)

        vetFrameT&      operator = (vetFrameT& img)

        vetFrameT& operator += (vetFrameT& img)

        vetFrameT& operator *= (vetFrameT& img)

        VETFRAME_PROFILE getProfile() { return profile; };
        VETFRAME_CHANNEL_TYPE getChannelType() { return dataType; };

        int getFOURCC()

};

#endif // __VETLIB_VETFRAMET_H__

```



```

#ifndef __VETLIBVETFRAMERGBA32_H__
#define __VETLIBVETFRAMERGBA32_H__

#define VETFRAMERGB24_SLOWMODE

#include "vetDefs.h"

// #include "PixelRGB24.h"
#include "vetFrame.h"

class vetFrameRGBA32;
#include "vetFrameYUV420.h"
#include "vetFrameRGB24.h"
#include "vetFrameRGB96.h"

typedef char* PixelRGBA32;

class vetFrameRGBA32 : public virtual vetFrame
{
public:
    bool autoFreeData;
    unsigned char *data;

public:
    enum ChannelRGBA { RED, GREEN, BLUE, ALPHA };

    vetFrameRGBA32();
    vetFrameRGBA32(unsigned int width, unsigned int height);
    vetFrameRGBA32(vetFrameRGBA32& img);
    vetFrameRGBA32(vetFrameRGB24& img);
    vetFrameRGBA32(vetFrameRGB96& img);

    ~vetFrameRGBA32();

    void* dump_buffer() { return static_cast<void*>(data); };

    unsigned int getBufferSize() { return (unsigned int)( width * height * 4); };

    VETRESULT reAllocCanvas(unsigned int w, unsigned int h);

    unsigned int getBpp() { return 32; };
    VETRESULT setBlack();
    VETRESULT setWhite();

    VETRESULT extractBrightness(unsigned char* buffer, unsigned int* size = NULL );

    vetFrameRGBA32& clearWith(unsigned char* bg);

    VETRESULT setPixel(unsigned int x, unsigned int y, unsigned char* p);
    VETRESULT getPixel(unsigned int x, unsigned int y, unsigned char* p);

    vetFrameRGBA32& operator = (vetFrameRGBA32& img);
    vetFrameRGBA32& operator += (vetFrameRGBA32& img);
    vetFrameRGBA32& operator -= (vetFrameRGBA32& img);

    void operator << (const vetFrameRGB24& img);
    void operator << (const vetFrameRGB96& img);

    vetFrameRGBA32& operator >> (vetFrameRGB24& img);
    vetFrameRGBA32& operator >> (vetFrameRGB96& img);

    VETFRAME_PROFILE getProfile() { return vetFrame::VETFRAME_RGBA32; };
    VETFRAME_CHANNEL_TYPE getChannelType() { return vetFrame::VETFRAME_CT_PLANAR; };

    int getFOURCC() { return 0x41424752; };

};

#endif // __VETLIBVETFRAMERGBA32_H__

```

```

#ifndef __VETLIB_VETFRAMERGB96_H__
#define __VETLIB_VETFRAMERGB96_H__

#include "vetFrame.h"
#include "PixelRGB96.h"

class vetFrameRGB96; // cos of "double-face" type definitions in operators
#include "vetFrameYUV420.h"
#include "vetFrameRGB24.h"
#include "vetFrameRGBA32.h"
#include "vetFrameGrey.h"

class vetFrameRGB96 : public virtual vetFrame
{
public:

    bool autoFreeData;

    PixelRGB96 *data;

    enum ChannelRGB { RED, GREEN, BLUE };

    vetFrameRGB96();

    vetFrameRGB96(unsigned int width, unsigned int height);

    vetFrameRGB96(vetFrameRGB96& img);
    vetFrameRGB96(vetFrameRGB24& img);
    vetFrameRGB96(vetFrameRGBA32& img);
    vetFrameRGB96(vetFrameGrey& img);

    ~vetFrameRGB96();

    VETRESULT setWidth(unsigned int newWidth);
    VETRESULT setHeight(unsigned int newHeight);

    void* dump_buffer() { return static_cast<void*>(data); };

    VETRESULT reAllocCanvas(unsigned int w, unsigned int h);

    unsigned int getBufferSize() { return (unsigned int)( width * height * 4); };

    unsigned int getBpp() { return sizeof(PixelRGB96) * 8; };
    VETRESULT setBlack();
    VETRESULT setWhite();

    VETRESULT extractBrightness(unsigned char* buffer, unsigned int* size = NULL );

    VETRESULT setPixel(unsigned int x, unsigned int y, PixelRGB96 p);
    VETRESULT setRGB(unsigned int x, unsigned int y, int red, int green, int blue);

    VETRESULT setChannel(unsigned int x, unsigned int y, ChannelRGB ch, int value);

    VETRESULT getPixel(unsigned int x, unsigned int y, PixelRGB96& p) const;

    const PixelRGB96& getPixel(unsigned int x, unsigned int y) const;

    int getChannel(unsigned int x, unsigned int y, ChannelRGB ch) const;

    vetFrameRGB96& clear(int = 0);
    vetFrameRGB96& clearWith(PixelRGB96& bg);
    vetFrameRGB96& clearChannel(ChannelRGB ch, int bg = 0);

    vetFrameRGB96& copy(vetFrameRGB96& img);

    vetFrameRGB96& operator = (vetFrameRGB96& img) { return this->copy(img); };
    vetFrameRGB96& operator += (vetFrameRGB96& img) ;
    vetFrameRGB96& operator -= (vetFrameRGB96& img) ;

    vetFrameRGB96& operator >> (vetFrameYUV420& img);
    vetFrameRGB96& operator >> (vetFrameRGB24& img);
    vetFrameRGB96& operator >> (vetFrameRGBA32& img);
    vetFrameRGB96& operator >> (vetFrameGrey& img);

    void operator << (const vetFrameYUV420& img);
    void operator << (const vetFrameRGB24& img);
    void operator << (const vetFrameRGBA32& img);
    void operator << (const vetFrameGrey& img);

    VETFRAME_PROFILE getProfile() { return vetFrame::VETFRAME_RGB96; };
    VETFRAME_CHANNEL_TYPE getChannelType() { return vetFrame::VETFRAME_CT_PACKED; };

    int getFOURCC() { return 0x32424752; }; // same as RGB24

};

#endif // __VETLIB_VETFRAMERGB96_H__

```

```

#ifndef __VETLIB_VETFRAMEHSV_H__
#define __VETLIB_VETFRAMEHSV_H__

#include "vetFrame.h"
#include "PixelHSV.h"

/*
struct PixelHSV {
    unsigned short int hue;    //16bit |
    unsigned char sat;        //8bit  |> 32bit
    unsigned char vat;        //8bit  |
}
*/

#include "vetFrameRGB96.h"
#include "vetFrameGrey.h"

class vetFrameHSV : public virtual vetFrame
{
public:

    PixelHSV *data;

    enum ChannelHSV { HUE, SAT, VAL };

    vetFrameHSV();

    vetFrameHSV(unsigned int width, unsigned int height);

    vetFrameHSV(vetFrameHSV& img);
    vetFrameHSV(vetFrameRGB96& img);
    vetFrameHSV(vetFrameGrey& img);

    ~vetFrameHSV();

    VETRESULT setWidth(unsigned int newWidth);
    VETRESULT setHeight(unsigned int newHeight);

    void* dump_buffer() { return static_cast<void*>(data); };

    VETRESULT reAllocCanvas(unsigned int w, unsigned int h);

    unsigned int getBpp() { return sizeof(PixelHSV) * 8; };
    VETRESULT setBlack();
    VETRESULT setWhite();

    VETRESULT extractBrightness(unsigned char* buffer, unsigned int* size = NULL);

    VETRESULT setPixel(unsigned int x, unsigned int y, PixelHSV p);

    VETRESULT setHSV( unsigned int x, unsigned int y, unsigned short int hue, unsigned char sat,
        unsigned char val);

    VETRESULT setChannel(unsigned int x, unsigned int y, ChannelHSV ch, unsigned int value);

    VETRESULT getPixel(unsigned int x, unsigned int y, PixelHSV& p);
    unsigned int getChannel(unsigned int x, unsigned int y, ChannelHSV ch);

    vetFrameHSV& clearWith(PixelHSV& bg);
    vetFrameHSV& clearChannel(ChannelHSV ch, unsigned int value = 0);

    vetFrameHSV& copy(vetFrameHSV& img);

    vetFrameHSV& operator = (vetFrameHSV& img) { return this->copy(img); };
    vetFrameHSV& operator += (vetFrameHSV& img);
    vetFrameHSV& operator -= (vetFrameHSV& img);
    vetFrameHSV& operator /= (vetFrameHSV& img);
    vetFrameHSV& operator *= (vetFrameHSV& img);

    vetFrameHSV& operator >> (vetFrameRGB24& img);
    vetFrameHSV& operator >> (vetFrameHSV& img);
    vetFrameHSV& operator >> (vetFrameRGB96& img);
    vetFrameHSV& operator >> (vetFrameGrey& img);

    void operator << (const vetFrameRGB24& img);
    void operator << (const vetFrameHSV& img);
    void operator << (const vetFrameRGB96& img);
    void operator << (const vetFrameGrey& img);

    static void convPixel_RGB32toHSV (const PixelRGB96&, PixelHSV&);
    static void convPixel_HSVtoRGB32 (const PixelHSV&, PixelRGB96&);

    static void convPixel_RGB24toHSV (const PixelRGB24&, PixelHSV&);
    static void convPixel_HSVtoRGB24 (const PixelHSV&, PixelRGB24&);

    static void convPixel_GREYtoHSV (const PixelGrey&, PixelHSV&);
    static void convPixel_HSVtoGREY (const PixelHSV&, PixelGrey&);

    VETFRAME_PROFILE getProfile() { return vetFrame::VETFRAME_CUSTOM; };
    VETFRAME_CHANNEL_TYPE getChannelType() { return vetFrame::VETFRAME_CT_PACKED; };

    int getFOURCC() { return 0;};

};

#endif // __VETLIB_VETFRAMEHSV_H__

```

```

#ifndef __VETLIB_VETFRAMEGREY_H__
#define __VETLIB_VETFRAMEGREY_H__

#include "vetDefs.h"

#include "vetFrame.h"
class vetFrameGrey; // cos of "double-face" type definitions in operators
#include "vetFrameYUV420.h"
#include "vetFrameRGB24.h"
#include "vetFrameRGB96.h"
#include "vetFrameRGBA32.h"

/// Single pixel consisting of an u char.
typedef unsigned char PixelGrey;

class vetFrameGrey : public virtual vetFrame
{
public:
    PixelGrey *data;

public:
    vetFrameGrey();
    vetFrameGrey(unsigned int width, unsigned int height);

    vetFrameGrey(vetFrameGrey& img);
    vetFrameGrey(vetFrameYUV420& img);
    vetFrameGrey(vetFrameRGB24& img);
    vetFrameGrey(vetFrameRGB96& img);
    vetFrameGrey(vetFrameRGBA32& img);

    ~vetFrameGrey();

    int setWidth(unsigned int newWidth);
    int setHeight(unsigned int newHeight);

    void* dump_buffer() { return static_cast<void*>(data); };

    VETRESULT reAllocCanvas(unsigned int w, unsigned int h);

    unsigned int getBpp() { return sizeof(PixelGrey) * 8; };
    VETRESULT setBlack();
    VETRESULT setWhite();

    unsigned int getBufferSize() { return (unsigned int)(width * height); };

    VETRESULT extractBrightness(unsigned char* buffer, unsigned int* size = NULL );

    VETRESULT setPixel(unsigned int x, unsigned int y, PixelRGB24 p);
    VETRESULT setPixel(unsigned int x, unsigned int y, PixelGrey level);

    VETRESULT getPixel(unsigned int x, unsigned int y, PixelGrey& p) const;

    const PixelGrey& getPixel(unsigned int x, unsigned int y) const;
    vetFrameGrey& clearWith(PixelGrey bg = 0);

    vetFrameGrey& copy(vetFrameGrey& img);

    VETRESULT invert();

    VETRESULT threshold(PixelGrey thresh) ;

    vetFrameGrey& operator = (vetFrameGrey& img) { return this->copy(img); };
    vetFrameGrey& operator += (vetFrameGrey& img) ;
    vetFrameGrey& operator -= (vetFrameGrey& img) ;
    vetFrameGrey& operator += (PixelGrey offset) ;
    vetFrameGrey& operator -= (PixelGrey offset) ;
    vetFrameGrey& operator /= (float factor) ;
    vetFrameGrey& operator *= (float factor) ;

    vetFrameGrey& operator >> (vetFrameYUV420& img);
    vetFrameGrey& operator >> (vetFrameRGB24& img);
    vetFrameGrey& operator >> (vetFrameRGB96& img);
    vetFrameGrey& operator >> (vetFrameRGBA32& img);

    void operator << (const vetFrameYUV420& img);
    void operator << (const vetFrameRGB24& img);
    void operator << (const vetFrameRGBA32& img);
    void operator << (const vetFrameRGB96& img);

    VETFRAME_PROFILE getProfile() { return vetFrame::VETFRAME_MONO; };

    VETFRAME_CHANNEL_TYPE getChannelType() { return vetFrame::VETFRAME_CT_PACKED; };

    int getFOURCC() { return 0; };
};

#endif // __VETLIB_VETFRAMEGREY_H__

```

```

/**
 * @class   vetOutput
 *
 * @brief   Abstract class for all data outputs, implementations should
 *          encode VETLib data stream to a device (visualization or storage
 *          for example), inherited classes must implement data encoding
 *          throw importFrom() methods, base input operators (<<) make a
 *          direct call to these functions, inherited classes may obviously
 *          override current operators behaviour.
 *
 * @version 1.0.2
 * @date    24/12/2005
 * @author   Alessandro Polo
 *
 * *****
 * VETLib Framework 1.0.2
 * Copyright (C) Alessandro Polo 2005
 * http://www.ewgate.net/vetlib
 *
 * *****/

#ifndef __VETLIB_VETOUTPUT_H__
#define __VETLIB_VETOUTPUT_H__

#include "vetDefs.h"
#include "vetFrameYUV420.h"
#include "vetFrameRGB24.h"
#include "vetFrameT.h"

class vetOutput
{
public:
    /**
     * @brief Default constructor is protected because this is an
     *        abstract class and instance cannot be created directly.
     */
    vetOutput() { }

    virtual ~vetOutput() { }

    /**
     * @brief Set current canvas' height.
     *
     * @return height in pixel.
     */
    virtual VETRESULT setHeight(unsigned int value) = 0;

    /**
     * @brief Set current canvas' width.
     *
     * @return width in pixel.
     */
    virtual VETRESULT setWidth(unsigned int value) = 0;

    /**
     * @brief Inherited class must implement this function,
     *        processing frames to specific output (device or stream)
     *
     * @param[in] img VETLib Cache Frame to be processed (encoded for example)
     *
     * @return VETRET_OK if everything is fine, VETRET_PARAM_ERR if frame
     *         is not valid, VETRET_INTERNAL_ERR or VETRET_ILLEGAL_USE else.
     *
     * @note Input operator (<<) call directly this function.
     * @see operator << (vetFrameYUV420&)
     */
    virtual VETRESULT importFrom(vetFrameYUV420& img) = 0;

    virtual VETRESULT importFrom(vetFrameRGB24& img) = 0;

    virtual VETRESULT importFrom(vetFrameT<unsigned char>& img) = 0;

    /**
     * @brief Input operator, import standard VETLib frame formats,
     *        current implementation calls directly importFrom() method.
     *
     * @param[in] img VETLib Cache Frame to be processed (encoded for example)
     *
     * @see importFrom(vetFrameYUV420&)
     */
    void operator << (vetFrameYUV420& img) { importFrom(img); };

    void operator << (vetFrameRGB24& img) { importFrom(img); };

    void operator << (vetFrameT<unsigned char>& img) { importFrom(img); };

    /**
     * @brief Ignore this, it's a class-type definition, mostly used
     *        by VETLib WorkShop, syntax is a bit more complex than
     *        usual because of a VC6 BUG, it's the same as:
     *        const int vetClassType = VETCLASS_TYPE_OUTPUT;
     */
    enum{ vetClassType = VETCLASS_TYPE_OUTPUT };
};

#endif // __VETLIB_VETOUTPUT_H__

```

```

/** @file    vetInput.h
 * @class    vetInput
 *
 * @brief    Abstract class for all data input, implementations should
 *           decode stream from a device (capture or storage for example)
 *           and convert to VETLib standard formats, inherited classes must
 *           implement data extraction throw extractTo() methods, base
 *           extraction operators (>>) call these functions for output,
 *           if frameRate is different from 0, after extractTo() call, process
 *           will stop for (1/fps)-(extractTo() execution time) seconds.
 *           This base implementation of operators and timer management has
 *           been coded for low-complex data sources, inherited classes may
 *           obviously override current operators behaviour.
 *
 * @version 1.0.2
 * @date    24/12/2005
 * @author  Alessandro Polo
 *
 * *****
 * VETLib Framework 1.0.2
 * Copyright (C) Alessandro Polo 2005
 * http://www.ewgate.net/vetlib
 * *****/

#ifndef __VETLIB_VETINPUT_H__
#define __VETLIB_VETINPUT_H__

#include "vetDefs.h"
#include "vetFrameYUV420.h"
#include "vetFrameRGB24.h"
#include "vetFrameT.h"

#include <time.h>

class vetInput
{
protected:
    float v_framerate;
    long v_sleeptime;
    clock_t v_elab_start;
    inline void setElaborationStart();
    inline long getElaborationTime();

public:
    vetInput(float fps = 0);

    virtual ~vetInput() { }

    virtual unsigned int getHeight() const = 0;
    virtual unsigned int getWidth() const = 0;

    virtual VETRESULT reset() = 0;

    virtual bool EoF() = 0;

    float getFrameRate() const { return v_framerate; };
    int setFrameRate(float fps);

    /**
     * @brief Inherited class must implement this function, processing
     *        hardware/format specific stream to standard VETLib frame formats.
     *
     * @param[out] img VETLib Cache Frame to store data.
     *
     * @return VETRET_OK if everything is fine, VETRET_PARAM_ERR if frame
     *         is not valid, VETRET_INTERNAL_ERR or VETRET_ILLEGAL_USE else.
     *
     * @note Ouput operator (>>) call directly this function.
     * @see operator >> (vetFrameYUV420&)
     */
    virtual VETRESULT extractTo(vetFrameYUV420& img) = 0;
    virtual VETRESULT extractTo(vetFrameRGB24& img) = 0;
    virtual VETRESULT extractTo(vetFrameT<unsigned char>& img) = 0;

    /**
     * @brief Ouput operator, export to standard VETLib frame formats,
     *        current implementation calls directly extractTo() method
     *        and if framerate isn't zero waits untill clock is synchronized,
     *        if elaboration time is greater than sleeptime, no delay is applied.
     *
     * @param[out] img VETLib Cache Frame to store data.
     *
     * @return Address of current instance.
     *
     * @see importFrom(vetFrameYUV420&)
     * @see vetsleep()
     * @see setElaborationStart()
     * @see getElaborationTime()
     */
    vetInput& operator >> (vetFrameYUV420& img);
    vetInput& operator >> (vetFrameRGB24& img);
    vetInput& operator >> (vetFrameT<unsigned char>& img);

    enum{ vetClassType = VETCLASS_TYPE_INPUT };
};

#endif // __VETLIB_VETINPUT_H__

```



## Segue vetFilter.h

```
#include <stdio.h>
#include <fstream>

class vetFilterParameters
{
protected:
    vetFilterParameters() {}

public:
    ~vetFilterParameters() {}

    //setOperation(int?);
    //int? getOperation();

    int saveToXML(const char* filename);
    int loadFromXML(const char* filename);

    virtual int saveToStreamXML(FILE *fp) = 0;
    virtual int loadFromStreamXML(FILE *fp) = 0;

};

#endif // __VETLIB_VETFILTER_H__
```

## vetBuffer.h

```
#ifndef __VETLIB_VETBUFFER_H__
#define __VETLIB_VETBUFFER_H__

#include "vetDefs.h"

template<class T>
class vetBuffer
{
protected:
    long v_fcount;

    bool copyData;

    vetBuffer(float fps = 0) { }

public:
    long getFramesCount() const { return v_fcount; };

    void setDoDataCopy(bool value = true) { copyData = value; };

    bool isDataCopyEnabled() const { return copyData; };

    virtual int reset() = 0;

    virtual int deleteFrames() = 0;

    virtual int addFrame(T& newFrame) = 0;
    virtual int insertFrame(long index, T &newFrame) = 0;
    virtual int updateFrame(long index, T &newFrame) = 0;
    virtual int removeFrame(long index) = 0;
    virtual int removeFrame(T &frameToDelete) = 0;

    virtual int goToNextFrame() = 0;
    virtual int goToPreviousFrame() = 0;
    virtual int goToFirstFrame() = 0;
    virtual int goToLastFrame() = 0;
    virtual int goToFrame(long index) = 0;
    virtual int goToStepFrame(long offset) = 0;

    virtual long getCurrentFrameIndex() const = 0;
    virtual T& getFrame(long index) = 0;
    virtual T& getLastFrame() = 0;
    virtual T& getFirstFrame() = 0;
    virtual T& getNextFrame() = 0;
    virtual T& getPreviousFrame() = 0;
    virtual T& getCurrentFrame() = 0;

};

#endif // __VETLIB_VETBUFFER_H__
```



```

#ifndef __VETLIB_VETCODEC_H__
#define __VETLIB_VETCODEC_H__

#include "vetDefs.h"
#include "vetObject.h"
#include "vetInput.h"
#include "vetOutput.h"
#include "vetFilter.h"
#include "vetFrameYUV420.h"
#include "vetFrameRGB24.h"
#include "vetFrameT.h"

class vetCodecParameters;

class vetCodec : public vetInput,
                public vetOutput,
                public vetObject
{
public:
    vetCodec(float fps = 0) {} // : vetFilter(fps) { }

    virtual ~vetCodec() {}

    virtual VETRESULT reset() = 0;

    virtual int getAudioStreamCount(int stream = -1) = 0;
    virtual int getVideoStreamCount(int stream = -1) = 0;

    virtual bool EoF() = 0;

    virtual bool isEncodingAvailable() = 0;
    virtual bool isDecodingAvailable() = 0;

    virtual long getVideoStreamLength(int stream = -1) = 0;
    virtual long getAudioStreamLength(int stream = -1) = 0;

    virtual bool hasAudio(int stream = -1) = 0;
    virtual bool hasVideo(int stream = -1) = 0;

    virtual VETRESULT close() = 0;

    virtual VETRESULT load(char *filename, int stream = -1) = 0;
    virtual VETRESULT save(char *filename, int stream = -1) = 0;

    virtual VETRESULT extractTo(vetFrameYUV420& img) = 0;
    virtual VETRESULT extractTo(vetFrameRGB24& img) = 0;
    virtual VETRESULT extractTo(vetFrameT<unsigned char>& img) = 0;

    virtual VETRESULT importFrom(vetFrameYUV420& img) = 0;
    virtual VETRESULT importFrom(vetFrameRGB24& img) = 0;
    virtual VETRESULT importFrom(vetFrameT<unsigned char>& img) = 0;

    enum{ vetClassType = VETCLASS_TYPE_CODER };
};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <fstream>

class vetCodecParameters : public vetFilterParameters
{
protected:
    char fileName[128];
    long frameIndex;
    int stream;

public:
    vetCodecParameters() {}

    virtual ~vetCodecParameters() {}

    virtual VETRESULT saveToStreamXML(FILE *fp) = 0;
    virtual VETRESULT loadFromStreamXML(FILE *fp) = 0;
};

#endif // __VETLIB_VETCODEC_H__

```

```

#ifndef __VETLIB_VETVISION_H__
#define __VETLIB_VETVISION_H__

#include "vetDefs.h"
#include "vetOutput.h"
#include "vetObject.h"

#include "vetFrameYUV420.h"
#include "vetFrameRGB24.h"
#include "vetFrameT.h"

class vetVision      :      public vetOutput,
                          public vetObject
{
protected:

    void* (*alertCall)(void* argument);
    void* alertCallArgument;

    void doAlert();

public:
    vetVision();

    virtual ~vetVision();

    void setAlertCall(void* (*functionCall)(void*)) { alertCall = functionCall; };
    void setAlertCallArgument(void* arg) { alertCallArgument = arg; };
    void* getAlertCallArgument() { return alertCallArgument; };

    /**
     * @brief Inherited class must implement this function,
     *        should reset all filters' parameters, something like
     *        constructor initialization.
     * @return VETRET_OK if everything is fine, VETRET_INTERNAL_ERR or VETRET_ILLEGAL_USE else.
     */
    virtual VETRESULT reset() = 0;

    /**
     * @brief Inherited class must implement this function,
     *        processing frames to specific output (device or stream)
     * @param[in] img VETLib Cache Frame to be processed (encoded for example)
     * @return VETRET_OK if everything is fine, VETRET_PARAM_ERR if frame
     *         is not valid, VETRET_INTERNAL_ERR or VETRET_ILLEGAL_USE else.
     * @note   Input operator (<<) call directly this function.
     * @see    operator << (vetFrameYUV420&)
     */
    virtual VETRESULT importFrom(vetFrameYUV420& img) = 0;
    virtual VETRESULT importFrom(vetFrameRGB24& img) = 0;
    virtual VETRESULT importFrom(vetFrameT<unsigned int>& img) = 0;

    enum{ vetClassType = VETCLASS_TYPE_VISION };      //because of VC6 BUG
};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

class vetVisionParameters
{
protected:

    bool doAlert;
    bool doEval;

public:
    vetVisionParameters() { }
    virtual ~vetVisionParameters() {}

    void setDoEval(bool value = true);
    void setDoAlert(bool value = true);

    VETRESULT saveToXML(const char* filename);

    VETRESULT loadFromXML(const char* filename);

    virtual VETRESULT saveToStreamXML(FILE *fp) = 0;

    virtual VETRESULT loadFromStreamXML(FILE *fp) = 0;
};

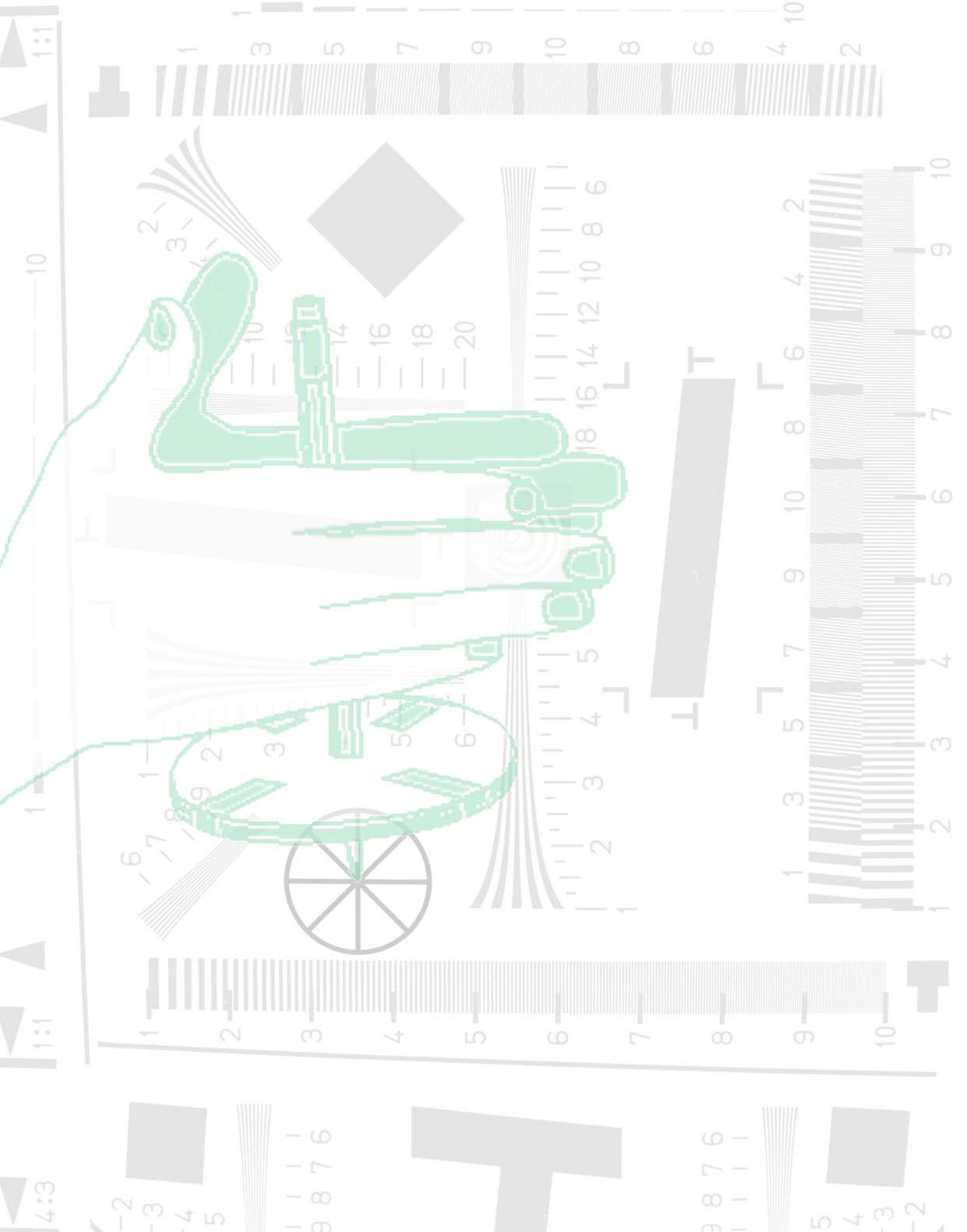
#endif // __VETLIB_VETVISION_H__

```



Il lavoro mi piace, mi affascina.  
Potrei starmene seduto per ore a guardarlo.

*Jerome Klapka Jerome*



# SAMPLE APPLICATIONS

## APPENDIX III

Ogni componente VETLib ha un corrispettivo file di test che verifica le funzionalità e dimostra l'utilizzo dei metodi, il sorgente è situato nella directory */tests/*, mentre i relativi progetti per la compilazione sono situati nelle directory *tests/bcb/* per Borland C++ Builder 6.0 e *tests/mvc/* per Microsoft Visual Studio 6.0, per i sistemi NIX il file di configurazione di Make è */tests/Makefile*.

I progetti sono configurati per posizionare il file eseguibile in */tests/bin/* e i file temporanei in */tests/tmp/*, il link predefinito alla libreria VETLib è */lib/VETLib.a* per sistemi NIX, */lib/VETLib\_bcb.lib* per Borland C++ Builder e */lib/VETLib.lib* per Visual Studio.

I programmi con prefisso *app\_* (invece di *test\_*) sono dimostrativi più complessi che nella maggior parte dei casi coinvolgono vari moduli di VETLib e sono esempi di applicazioni reali per l'utente finale.

Tutti i progetti sono stati ampiamente testati sui due sistemi:

- Windows 2000 SP4 - Athlon 1GHz 768Mb
- Linux 2.4.29 - Slackware 10.1 – Intel 1.6GHz 512Mb.

Segue la descrizione di alcuni dei progetti più significativi:

### ■ test\_vetVideo4Linux

```
* @brief   Testing code for class vetVideo4Linux.
*
*         Connect to first capture device (/dev/video0), print
*         size, colour depth and palette, capture a frame, print
*         processing time (milliseconds) and store image to file.
*
* @warning need VETLib with V4L support
*         need a valid capture device, Video4Linux library.
*
* @see     vetVideo4Linux, test_vetVideo4LinuxPlayer.cpp
*
* @version 1.0
* @date    5/08/2005
```

### ■ test\_vetDirectXInput.cpp

```
* @brief   Testing code for class vetDirectXInput.
*
*         Enumerate available devices, print list with description,
*         try to connect to first device and save 3 frames.
*
* @warning requires VETLib with DirectX support
*
* @todo    frame loop and fps estimation
*
* @see     vetDirectXInput
*
* @version 0.50
* @date    12/09/2005
```

## ■ app\_vetVideo4LinuxPlayer.cpp

```
* @brief   Testing code for class vetVideo4Linux and vetWindowQT.
*
*         Load video stream from first device (/dev/video0) and
*         show preview in a window with the same size, stream
*         is passed through vetFrameRGB24 objects.
*         Frame rate should be the higher possible, after 100 frames,
*         loop will exit printing average fps; this number depends
*         on source rate and window's redrawing both (sum).
*         Then last captured frame is saved.
*
*         Tested on a Linux 2.4.29 kernel with an USB Logitech
*         Express Webcam, frame rate is around 15 fps, that is
*         hardware limit, in example test_LinuxMPEGPlayer it was
*         34 fps (decoding + displaying).
*
* @warning require VETLib with V4L support and GUI support
*         require a valid capture device, Video4Linux library,
*         QT library.
*
* @see     vetVideo4Linux, vetWindowQT, vetDoctor
* @see     app_LinuxMPEGPlayer.cpp, app_LinuxMOVPlayer.cpp
*
* @version 1.0.2
* @date    12/09/2005
```

## ■ test\_vetLinuxMPEGPlayer.cpp

```
* @brief   Testing code for class vetCodec_MPEG and vetWindowQT.
*
*         Load first video stream from an MPEG1-2 format file and
*         show preview in a window , stream is passed through
*         vetFrameRGB24 objects.
*         Frame rate should be the higher possible, after 100 frames,
*         loop will exit printing average fps; this number depends
*         on source rate and window's redrawing both (sum).
*         Then last captured frame is saved.
*
* @warning need VETLib with MPEG support and GUI support (make all)
*
* @see     vetCodec_MPEG, vetWindowQT
* @see     app_Video4LinuxPlayer.cpp, app_LinuxMOVPlayer.cpp
*
* @version 1.0.2
* @date    11/09/2005
```

## ■ test\_vetFilterGeometric.cpp

```
* @brief   Testing code for class vetFilterGeometric.
*
*         Performs all available operations on a source image and
*         save output to BMP file. Then tests settings serialization.
*
* @see     vetFilterGeometric
*
* @version 1.0.2
* @date    02/09/2005
```

## ■ test\_vetLinuxMPEGPlayer.cpp

```
* @brief   Testing code for class vetCodec_MPEG and vetWindowQT.
*
*         Load first video stream from an MPEG1-2 format file and
*         show preview in a window , stream is passed through
*         vetFrameRGB24 objects.
*         Frame rate should be the higher possible, after 100 frames,
*         loop will exit printing average fps; this number depends
*         on source rate and window's redrawing both (sum).
*         Then last captured frame is saved.
*
* @warning requires VETLib with MPEG support and GUI support
*         (make all)
*
* @see     vetCodec_MPEG, vetWindowQT
* @see     app_Video4LinuxPlayer.cpp, app_LinuxMOVPlayer.cpp
*
* @version 1.0.2
* @date    11/09/2005
```

## ■ test\_vetWindowQT.cpp

```
* @brief   Testing code for class vetWindowQT.
*
*         Load two frames using vetFrameRGB32 and vetFrameRGB24 both,
*         show QWindow sliding some times between the two frames.
*         Frame rate should be the higher possible, after 100 frames,
*         loop will exit printing average fps.
*         During tests average fps was around 20.
*
* @warning require VETLib with GUI support (make gui)
*         require Linux OS, QT Library and a desktop environment
*         (KDE, GNOME)
*
* @see     vetWindowQT, vetCodec_BMP
* @see     test_Video4LinuxPlayer.cpp, test_LinuxMPEGPlayer.cpp,
*
* @version 1.0.2
* @date    08/09/2005
```

## ■ test\_vetWindowQT.cpp

```
* @brief   Testing code for class vetMatrix.
*
*         vetMatrix template class is designed for filters'
*         and plugins' developing, this simple code tests an
*         integer matrix, populates the matrix incrementing
*         value (a[0,0] = 0, a[10,10] = 99), serializes matrix
*         to a file and tests also data loading to a new int matrix,
*         then prints the cloned matrix to stdout.
*
* @see     vetMatrix
*
* @version 1.0
* @date    15/08/2005
```



Your best idea is already copyrighted.

*Murphy's Law*





## APPENDIX IV

```
/*
 * VETLib Framework 1.0.2
 * Copyright (C) 2005 Alessandro Polo
 * http://www.ewgate.net/vetlib
 *
 * *****
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * *****/
http://www.gnu.org/copyleft/gpl.html
```

## GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.  
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### **PREAMBLE**

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it. For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software. Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all. The precise terms and conditions for copying, distribution and modification follow.

## **TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION**

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
  - a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
  - b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
  - c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
  - a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
  - b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
  - c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for non-commercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

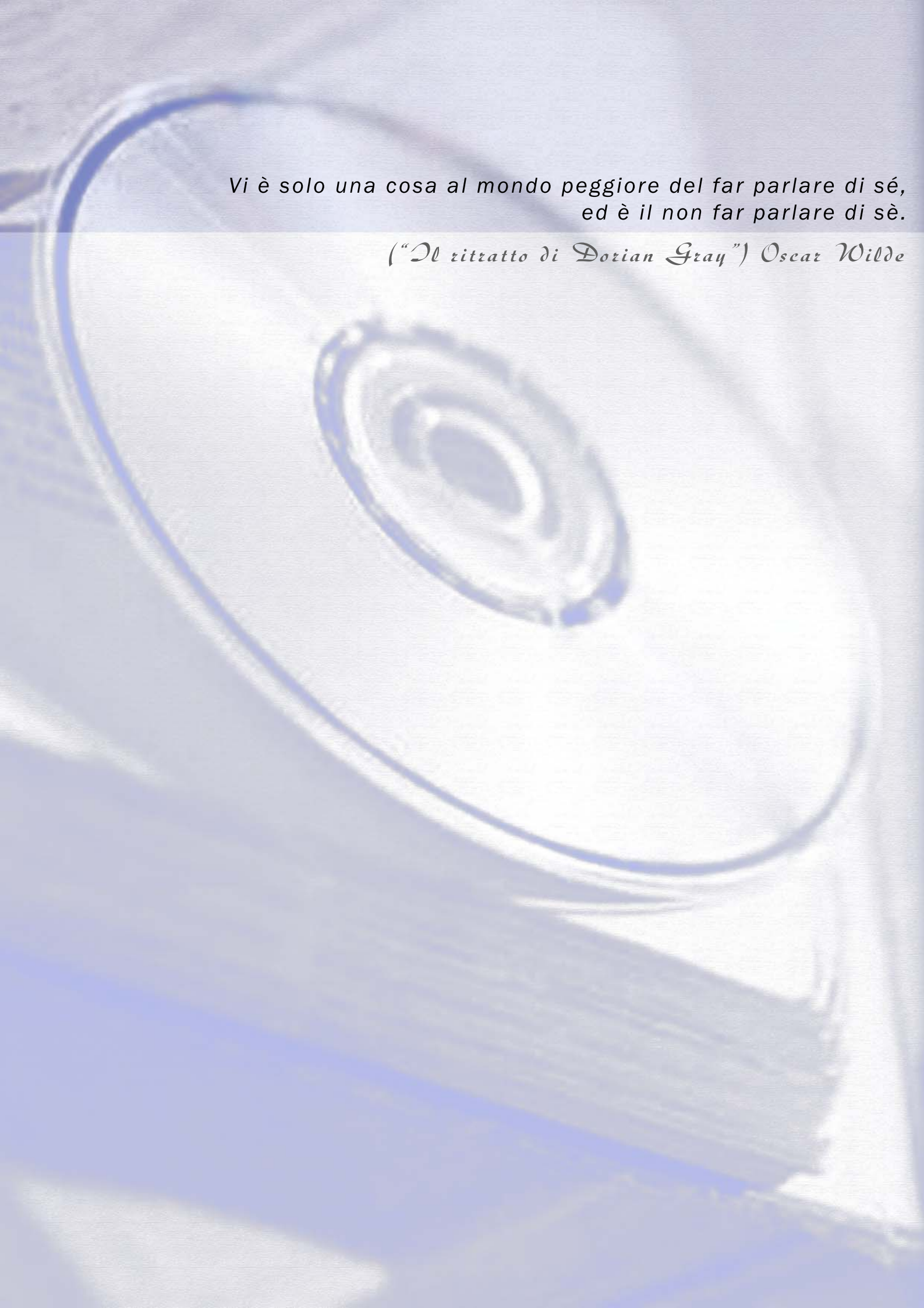
Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

## **NO WARRANTY**

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

**END OF TERMS AND CONDITIONS**



*Vi è solo una cosa al mondo peggiore del far parlare di sé,  
ed è il non far parlare di sé.*

*("Il ritratto di Dorian Gray") Oscar Wilde*

# Bibliography

---

- ✓ “A Modular Software Architecture for Real-Time Video Processing”, Alexandre R.J. François and Gérard G. Medioni
- ✓ “Thinking in C++”, Bruce Eckel, ISBN 0-13-979809-9
- ✓ “Digital Image Processing”, William K. Pratt, ISBN 0-471-37407-5
  
- ✓ MSDN Documentation on <http://msdn.microsoft.com>
- ✓ DirectX SDK Samples (Source code)
  
- ✓ XviD Documentation on <http://www.xvid.org>
- ✓ QuickTime for Linux documentation on <http://www.heroinwarrior.com/quicktime>
- ✓ LibMPEG3 documentation on <http://www.heroinwarrior.com/libmpeg3>
  
- ✓ Video 4 Linux documentation on <http://www.exploits.org/v4l>
- ✓ Trolltech QT Library documentation on <http://doc.trolltech.com>
- ✓ GTK Library documentation <http://www.gtk.org>