

Monday, 22 May 2006



IMAGE PROCESSING WITH VIPLIB HANDOUT

by © mmLab - DIT 2006, open source project

VIPLib Home: <http://mmlab.science.unitn.it/projects/VIPLib/>
VIPLib Package Studio Home: <http://mmlab.science.unitn.it/projects/vipPS/>

Written by Alessandro Polo, Revisioned by Nicola Conci

Sommario

Introduzione a VIPLib.....	3
1.1 – Esempi.....	4
Requisiti Software.....	6
Sviluppo di Pacchetti	7
4.1 – Convenzioni	9
4.2 – Specifiche della Documentazione.....	10
4.3 – Creazione del Pacchetto	10
4.4 – Compilare e Testare I Pacchetti	12
4.5 – Esempi e Note	14
4.5.1 – Classi nuove	14
4.5.2 – vipOutput	15
4.5.2 – vipFilter.....	15
4.5.3 – Note per vipFilter e vipOutput.....	16
4.5 – Licenza	17
4.6 – Maggiori Informazioni.....	17
4.7 – Ho trovato un Bug!	17
Appendix.....	18
5.1 – Buffering Interno per i Filtri	18
5.2 –Lavorare con i Frames.....	21
5.3 – Gestione dei Parametri per i Filtri.....	23
5.4 – Useful Components.....	26
vipFrameRGB24	26
vipWindowGTK.....	26
vipWindow32.....	27
vipDoctor	27
vipCodec_BMP	27
vipCodec_IMG.....	27
vipUtility	27
vipMatrix.....	27

Queste informazioni sono (o lo saranno a breve) disponibili online nelle rispettive sezioni dei siti:

VIPLib : <http://mmlab.science.unitn.it/projects/VIPLib>
VIPLib Package Studio : <http://mmlab.science.unitn.it/projects/vipPS>

Informazioni riguardo Subversion sono disponibili all'indirizzo:

<http://mmlab.science.unitn.it/services/subversion>

NOTE:

All'interno delle reti della facoltà di Scienze a Povo, il sito del Laboratorio Multimediale è accessibile con i classici protocolli e porte standard (HTTP: 80, HTTPS: 443, FTP: 21), mentre all'esterno della facoltà è necessario redirezionare il traffico rispettivamente sulle porte (HTTP: 8080, HTTPS: 4430, FTP: 2121), il server Subversion è ugualmente in ascolto sulla porta 3690.

Altre e più dettagliate informazioni sui servizi di mmLab sono disponibili all'indirizzo:

<http://mmlab.science.unitn.it/services>

INTRODUZIONE A VIPLIB

CHAPTER I

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 opensource quali libmpeg3, xvidcore, quicktime4linux per la (de)codifica video e imageMagick per la (de)codifica di immagini (praticamente tutti i formati).

Gli strumenti forniti da VIPLib 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 semplici (ogni componente ha un rispettivo progetto che ne evidenzia le caratteristiche e l'utilizzo).

Lo sviluppo di nuovi moduli (basati su interfacce standard) è stato semplificato anche grazie allo strumento Package Studio (sviluppato in ambiente Microsoft Visual Studio .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).

La documentazione di VIPLib è essenzialmente divisa in tre insiemi:

- documenti preliminari
- documentazione delle classi
- altri manuali (incluso il presente)

La fonte di riferimento principale rimane comunque la documentazione disponibile online insieme alla sezione *development* nel sito della libreria. La lingua privilegiata per la stesura di manuali e documentazione è l'inglese.

Le informazioni preliminari più interessanti in questo contesto consistono nei cosiddetti *readme* (in formato ASCII - plain text):

./README	:	Informazioni generali riguardo la libreria;
./USE	:	Come usare la libreria nelle proprie applicazioni;
./EXTEND	:	Come estendere la libreria;
./FAQS	:	Risposte alle domande più frequenti.
./ChangeLog	:	Lista degli aggiornamenti divisi per built;
./lib/README	:	Informazioni sui binari;
./docs/README	:	Informazioni sulla documentazione.

La documentazione, disponibile online (<http://mmlab.science.unitn.it/projects/VIPLib/docs/>), è generata semi-automaticamente nei formati più diffusi dallo strumento *DoxyGen* (*freeware*),

La sezione <http://mmlab.science.unitn.it/projects/VIPLib/dev/> raccoglie invece informazioni utili agli sviluppatori di pacchetti e a chi fosse interessato a contribuire all'estensione della libreria.

I componenti (alias moduli, packages) della libreria sono sempre 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 o nella fase di

test e sviluppo di un nuovo pacchetto (ad esempio è sempre utile un componente per la lettura e scrittura di immagini, uno per la visualizzazione in tempo reale, ecc).

1.1 – Esempi

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

```
#include "../source/vipFrameRGB24.h"
#include "../source/outputs/vipWindowGTK.h"
#include "../source/codecs/vipCodec_MPEG.h"

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

    vipWindowGTK myWin (mpegSource.getWidth(), mpegSource.getHeight());
    vipFrameRGB24 img24 (mpegSource.getWidth(), mpegSource.getHeight());

    myWin.show();

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

        mpegSource >> img24;
        myWin << img24;

        vipUtility::vipSleep(
            sleeptime - (long)(vipUtility::getTime_usec()-offset)/1000 );
    }
    return 0;
}
```

app_vipLinuxMPEGPlayerGTK.cpp

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

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

```

vipVideo4Linux cap("/dev/video0");
vipFrameRGB24 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);
vipWindowQT *myapp = new vipWindowQT(cap.getWidth(), cap.getHeight() );
myapp->show();
app.setMainWidget(myapp);

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

```

app_vipVideo4LinuxPlayer.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_vipVideo4Linux.cpp ../lib/VIPLib.a
-o test_vipVideo4Linux.out

```

```

vipDirectXInput 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_vipDirectXInput.cpp

REQUISITI SOFTWARE

CHAPTER II

Per creare e testare un pacchetto (estensione) di VIPLib sono ovviamente necessari una serie di elementi:

- Headers delle classi base e dei componenti che si intende usare.
- Binari della libreria per il sistema operativo adottato.

È possibile scaricare questi elementi separatamente o più semplicemente scaricare l'archivio di distribuzione VIPLib-SDK.zip (o l'installer per Windows VIPLib-SDK.msi) dal sito della libreria <http://mmlab.science.unitn.it/projects/VIPLib/>.

Questi archivi saranno aggiornati periodicamente, ma l'ultima versione disponibile (*nightly build*) è sempre accessibile tramite il repository *Subversion* del server mmLab.

È **consigliabile** utilizzare questo sistema per “scaricare” la libreria (anziché l'archivio ZIP), il vantaggio principale, infatti, consiste nella facilità di aggiornamento (completamente automatizzato da appositi clients).

Maggiori informazioni sul sistema Subversion e i links ai clients più diffusi sono disponibili all'indirizzo <http://mmlab.science.unitn.it/services/subversion>. Nella sezione riservata agli utenti Windows sono presenti anche i passi chiave per il primo approccio al repository.

L'indirizzo del progetto VIPLib (richiesto dai software) è

svn://mmlab.science.unitn.it/VIPLib/trunk

Inoltre, per semplificare ed automatizzare la fase di creazione di un nuovo pacchetto (creare i files di progetto e lo scheletro delle classi) è disponibile il software **VIPLib Package Studio** (attualmente solo per piattaforme Windows, con framework .NET).

L'ultima versione di **VIPLib Package Studio** è scaricabile dal sito web mmLab (<http://mmlab.science.unitn.it/projects/vipPS/>). Per informazioni sull'utilizzo e i requisiti del software consultare la documentazione online.

Anche il progetto Package Studio è disponibile tramite il sistema SVN, ma al fine dello sviluppo di pacchetti non è necessario utilizzare il codice sorgente. E' consigliabile piuttosto installare il software tramite il classico *installer* di Windows (*VIPLib-PKSTUDIO.msi*) .

Riassumendo:

0. Scaricare e installare un client Subversion. *
1. Connettersi a repository di VIPLib (comando: CHECKOUT) *
2. Scaricare e installare Package Studio
3. Eseguire Package Studio e configurare il percorso di installazione di VIPLib

* È possibile scaricare l'archivio SDK invece di utilizzare Subversion.

SVILUPPO DI PACCHETTI

CHAPTER IV

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

- vipInput (.h)	(data server)	(components in ./inputs)
- vipOutput (.h)	(data client)	(components in ./outputs)
- vipFilter (.h)	(data bridge)	(components in ./filters)
- vipCodec (.h)	(data bridge)	(components in ./codecs)
- vipVision (.h)	(data client)	(components in ./vision)
- vipBuffer (.h)	(data storage)	(components in ./buffers)
- vipFrame (.h)	(frame object)	(components in ./)
- vipObject (.h)	(general object)	(components in ./)

I componenti che forniscono algoritmi e operazioni matematiche non hanno restrizioni particolari, se non dettate dal buon senso e dallo staff, 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.

La convenzione (rispettata anche dal software *Package Studio*) prevede la seguente gerarchia:










- 📁 any directory (VIPLib root)
 - 📁 /lib/ {VIPLib.a, VIPLib.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).

I principali componenti di VIPLib (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.

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 invece completamente automatizzato dal software *VIPLib Package Studio*.

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

I primi due *file* sono il cuore del progetto e contengono rispettivamente la dichiarazione del componente e l'implementazione dei metodi, ovviamente non è presente una funzione *main* nel file sorgente, quindi per semplificare lo sviluppo è stato incluso il file `test_vipClassName.cpp` che permette di testare direttamente il componente e i suoi metodi nella funzione *main*. Vediamo ad esempio il componente *vipFilterGeometric*:

```
#include "../source/vipFrameRGB24.h"
#include "../source/codecs/vipCodec_BMP.h"
#include "../source/filters/vipFilterGeometric.h"
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("Testing vipFilterGeometric Development...\n");
    printf("\nCreating Instances...\n");
    vipFrameRGB24 img, img2;
    vipFilterGeometric geom;

    printf("Loading Frame...\n");
    vipCodec_BMP::load(img, "frame1.bmp", vipCodec_BMP::FORMAT_BMP_24);
    geom.getParameters().forceOutputSize(true);

    printf("Start Processing...ROTATING 90...\n");
    geom.getParameters().setRunMode(vipFilterGeometricParameters::ROTATE90);
    geom << img;
    geom >> img2;
    vipCodec_BMP::save(img2, "OUT_ROTATE90.bmp", vipCodec_BMP::FORMAT_BMP_24);

    printf("\nParameters Serialization (XML)...\n");
    geom.getParameters().forceOutputSize(true);
    geom.getParameters().setRunMode(vipFilterGeometricParameters::ROTATE);
    geom.getParameters().setRotationDegree(45.3);
    geom.getParameters().setResizeDimension(123, 456);
    geom.getParameters().saveToXML("geom.XML");
    vipFilterGeometric geom2;
    geom2.getParameters().loadFromXML("geom.XML");
    geom2.getParameters().saveToXML("geom_COPY.XML");

    printf("Test Completed. Type something to continue...\n");
    getchar();
    return 0;
}
```


4.1 – Convenzioni

Gli sviluppatori che vogliono estendere la libreria con nuovi moduli devono seguire le seguenti convenzioni ideate per mantenere VIPLib 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. *vipFilterColor*, ..)

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

I nomi riservati (non disponibili) sono elencati nel file *./templates/NAMESPACE* nella cartella di installazione di Package Studio (il software valida automaticamente il nome durante la creazione del pacchetto).

- 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>*, e, quando completi, possono in seguito 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 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 (*vip<..>Parameters*) e implementare I/O per serializzare la classe in formato XML. (Ex. *vipFilter*)
- 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 *VIPRESULT* (definito come intero) che indica il risultato del processo; i valori di ritorno possibili sono definiti in *vipDefs.h* (es. *VIPRET_OK*). Si ricorda che la convenzione è comunque: 0 → OK, errore altrimenti.
- Se si desidera implementare un plugin di *VIPLib 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 compatibili con queste specifiche non saranno presi in considerazione

4.2 – Specifiche della Documentazione

Il codice deve essere **documentato** dallo sviluppatore direttamente **nei file sorgenti** (cioè nell'header, e possibilmente copiare il testo anche nel sorgente) in **lingua inglese**, 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 VIPLib Frame RGB24 to be processed
 * @param[out] threshold optimal threshold for input image
 *
 * @return     VIPRET_PARAM_ERR if image is invalid, VIPRET_OK else.
 *
 * @see       realAlgorithm()
 */
VIPRESULT sampleFunction (vipFrameRGB24& 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.

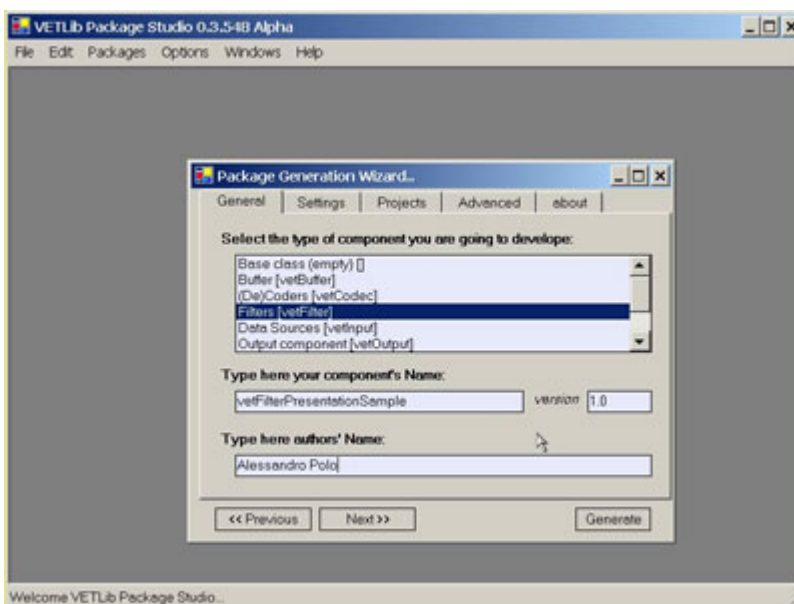
Questo sistema è largamente diffuso soprattutto per quanto riguarda librerie e SDK dedicati agli sviluppatori, eventuali modifiche e aggiornamenti del codice non implicano lunghe revisioni della documentazione finale, ma semplicemente la *ricompilazione* tramite il software (eventualmente la modifica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

Una sezione nella prima parte del file header deve riassumere la descrizione del componente ed eventuali note, oltre a informazioni relative all'autore, alla versione e alle date di aggiornamento.

Il software Package Studio è in grado di generare lo scheletro della documentazione e completare alcuni campi in base alle scelte progettuali.

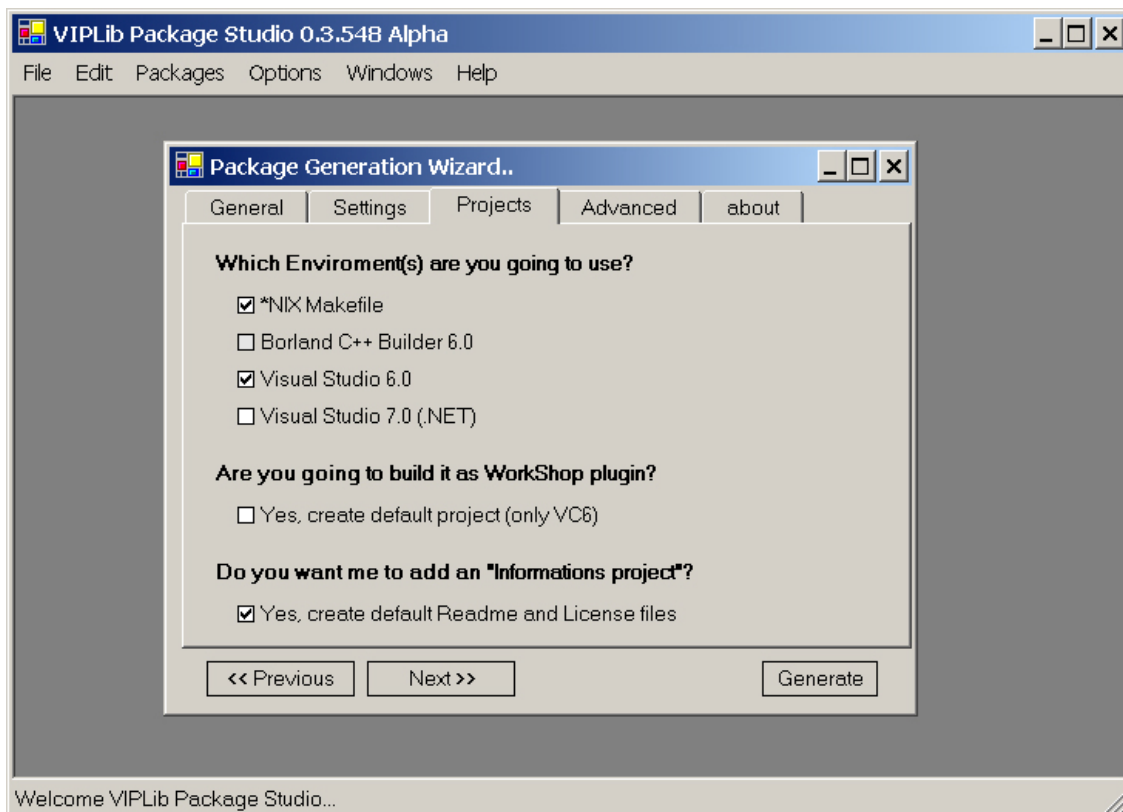
È fortemente consigliata la pratica (poco diffusa) di aggiungere commenti anche all'interno delle funzioni e di scomporre problemi complessi in più funzioni. In futuro, la convalida dei pacchetti terrà conto anche dello stile di programmazione e della documentazione attraverso metodi di analisi standard (come il *fog index*).

4.3 – Creazione del Pacchetto



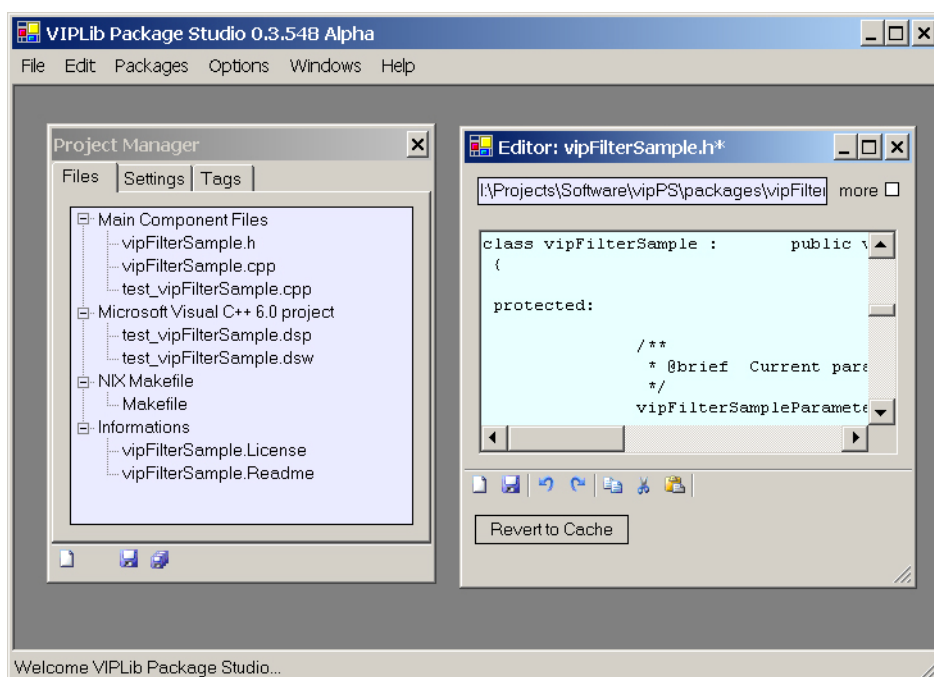
Nel menu *Packages*, scegliere la voce *Generation Wizard*, la nuova finestra presenta alcune schede di configurazione, la prima scheda permette di impostare i parametri fondamentali del componente:

0. Classe base
1. Nome
2. Autore/i



La terza scheda permette di scegliere i sistemi di sviluppo con cui si vuole scrivere e compilare il componente, la configurazione predefinita prevede di creare i progetti per Microsoft Visual Studio 6.0 e per Linux (utility Make).

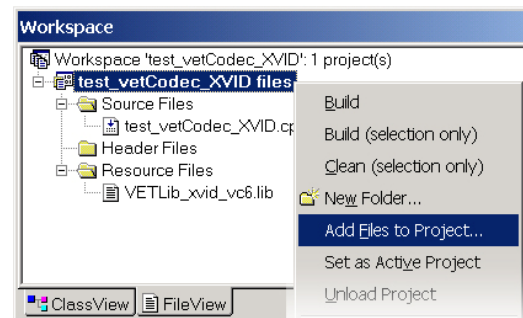
Non è necessario modificare altre impostazioni (tranne i parametri nella prima scheda), cliccare *Generate* per creare i files del pacchetto (viene creata una cartella con il nome del pacchetto all'interno della cartella /packages/ nella radice di VIPLib, in assenza della libreria verrà creata all'interno della cartella di installazione del software).



4.4 – Compilare e Testare I Pacchetti

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 VIPLib 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 *VIPLib*.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 VIPLib, il comando di compilazione (e linking) più semplice, dove non sono richieste altre librerie, è di questo tipo:

```
g++ myMain.cpp ../lib/VIPLib.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 *vipLinuxMPEGPlayerGTK*, incluso nella distribuzione e posizionato in *./tests/* necessita dei seguenti parametri:

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

Questo semplice programma è basato su due moduli: *vipCodec_MPEG* e *vipWindowGTK*. 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_vipLinuxMPEGPlayerGTK.cpp*) che fa riferimento a metodi contenuti in VIPLib (*../lib/VIPLib.a*). Ad un secondo livello 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 VIPLib (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 *VIPLib 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).

4.5 – Esempi e Note

VIPLib è distribuita con il sorgente di tutti i componenti, è altamente consigliata l'analisi di moduli simili per conformare lo stile e le convenzioni con la libreria stessa, i pacchetti creati con Package Studio includono funzioni dimostrative utili per un primo approccio.

La prima scelta progettuale consiste nel capire quale categoria di componente implementare, si sono citate le categorie possibili nella prima parte di questo capitolo, seguono alcuni esempi e note sulle modalità di sviluppo per i casi più comuni inerenti l'immagine processing.

4.5.1 – Classi nuove

Esempi:

- Trasformata Wavelet intera discreta
- Porting della trasformata DCT (implementazione veloce)
- Porting della trasformata Hadamard/Haar
- Implementazione dei descrittori di Fourier
- Porting implementazione di funzione per processing generico a blocchi nxm
- operazioni matriciali (trasposta, inversa, prodotto, prodotto scalare)

- Statistiche (medie, varianza, PSNR, MSE)
- Equalizzazione

Esempio:

```
vipStatistics {  
    template<class T>  
    VIPRESULT getAverage(T* data, long lenght, double* result) {  
        double sum = 0;  
        for (long i=0; i<lenght; i++)  
            sum +=data[i];  
        *result = sum/lenght;  
    }  
}
```

NOTA la media calcolata è passata come parametro, ciò è dovuto alla convenzione VIPRESULT: se "value" fosse il valore di ritorno, non saremmo in grado di distinguere un errore dal valore della media corretto.

NOTA2 dove possibile utilizzare template!, altrimenti ottimizzare il ciclo per i tipi più verosimili nel contesto.

4.5.2 – vipOutput

Le implementazioni dell'interfaccia vipOutput costituiscono l'ultimo blocco del flusso dati (il blocco importa frame), un esempio tipico è una finestra di visualizzazione, ma possono essere utili anche blocchi che eseguano statistiche, trasformate, e comuni operazioni matematiche, molto rapidamente si possono sviluppare blocchi vipOutput associati all'implementazione più astratta di un componente matematico, come:

- Statistiche (medie, varianza, PSNR, MSE)
- Estrazione piani di Bit
- Equalizzazione

Esempio:

```
vipOutStatistics {  
    vipStatistics* core;  
  
    VIPRESULT importFrom(vipFrameRGB24& src){  
        double value;  
        VIPRESULT ret = core->getAverage(src.data, width*height*3, &value);  
        ...  
    }  
}
```

in questo modo la media può essere calcolata su qualsiasi tipo standard, e il componente ausiliario vipOutStatistics semplifica l'utilizzo nei casi classici.

un simile schema può essere applicato agli altri operatori matematici/trasformate.

4.5.2 – vipFilter

Esempi basati su vipFilter:

- Porting di Image Compositing (materiale già disponibile)
- Filtri morfologici
- Dithering
- Halftoning
- Region Growing (materiale già disponibile)
- Split and Merge (materiale già disponibile)

L'implementazione si riduce allo sviluppo della funzione di processing (possibilmente scomposta in più funzioni) da inserire nel metodo " importFrom(IMAGE) ", inoltre è necessario gestire i parametri di lavoro tramite la classe vipFilterParameters (*vedi appendice*).

4.5.3 – Note per *vipFilter* e *vipOutput*

VIPLib è una libreria in grado di gestire flussi video, è sempre possibile applicare una sequenza di immagini ad un filtro o un uscita (*vipOutput*), è proprio questa peculiarità infatti che richiede la gestione della selezione del metodo corrente all'interno dei blocchi, ad esempio:

```
vipOutStatistics {
    vipStatistics* core;

    enum RUNMODE{ DO_NOTHING, AVERAGE, VARIANCE };

    RUNMODE runMode;

    double last_average;
    bool valid_parameters;

    VIPRESULT importFrom(vipFrameRGB24& src){
        switch ( runMode) {
            case DO_NOTHING:
                valid_parameters = false;
                return VETRET_OK;

            case DO_AVERAGE:
                VIPRESULT ret = VIPRET_INTERNAL_ERR;
                ret = core->getAverage(src.data, width*height*3, &last_average);
                if (ret == VIPRET_OK)
                    valid_parameters = true;
                return VIPRET_OK;

        }
        return VIPRET_NOT_IMPLEMENTED;
    }

    VIPRESULT getAverage(double* result){
        if (valid_parameters)
        {
            if ( result == NULL)
                return VIPRET_PARAM_ERR;
            *result = last_average;
            return VIPRET_OK;
        }
        return VIPRET_ILLEGAL_USE;
    }
}
```

Ciò vale anche per l'implementazione di filtri (*vedi vipFilterGeometric* o le funzioni dimostrative incluse nella generazione di Package Studio).

Sequenze di frame applicate ai filtri non comportano alcun cambiamento rispetto all'implementazione finalizzata al processing di una singola immagine, ma nel caso di classi di tipo *vipOutput* si nota l'evidente necessità della memorizzazione dei parametri estratti: altrimenti i dati relativi al frame corrente saranno sovrascritti dal successivo.

Questa gestione potrebbe essere implementata nel ciclo di redirezione dei frames, ad esempio:

```
int main() {

    vipFrameRGB24 globalBuffer;           // buffer for current processed frame
    vipOutStatistics myOutputClass;       // subject component
    vipCodec_MPEG mpegSource;
    mpegSource.load("football.mpg");
    myOutputClass.setRunMode(myOutputClass::AVERAGE);
    double* allAverages = new double[100];

    while (i++ < 100) {
        mpegSource >> globalBuffer
        myOutputClass << globalBuffer;
        myOutputClass.getAverage( &allAverages[i] );
    }
}
```

Questa soluzione è assolutamente deprecabile poiché complica i cicli di redirezione e mette in dubbio il vantaggio di utilizzare la classe `vipOutStatistics` (si potrebbe chiamare direttamente la funzione in `vipStatistics..`), il codice va infatti integrato nel modulo di uscita, ovviamente non si conosce a priori il numero di frame ed è necessario implementare o utilizzare un buffer dati dinamico (ad esempio si può usare la classe `./source/buffers/vipBufferArray`)

4.5 – Licenza

a 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 VIPLib devono accettare e **confermare questa licenza**. Qualora nel progetto siano incluse librerie esterne, anch'esse devono essere *open source*, la convenzione prevede di precisare eventuali note nel file `<ClassName>.License`.

4.6 – Maggiori Informazioni

Il server mmLab e i siti web dei progetti ospitati (così come l'interazione online degli utenti) sono ancora incomplete ed in fase di sviluppo; nonostante ciò, molte sezioni contengono informazioni utili e talvolta dettagliate riguardo l'installazione, l'utilizzo e lo sviluppo.

Fare riferimento a questo manuale come fonte più autorevole; considerare in seguito i siti web (sezioni: *documentation* e *development*) e in ultimo i file *readme* distribuiti con i progetti.

4.7 – Ho trovato un Bug!

Purtroppo è possibile, i progetti VIPLib e relativi sotto-progetti sono molto estesi e non sono ancora stati testati accuratamente; le operazioni più frequenti sono abbastanza affidabili, ma nell'eventualità in cui durante lo sviluppo scopriate un bug, siete pregati di contattare l'amministratore del progetto o l'ultimo sviluppatore attivo (vedere sul sito).

L'aggiornamento locale della libreria (sulla vostra stazione di lavoro) diventa estremamente banale se si è installata VIPLib tramite un client Subversion.

5.1 – Buffering Interno per i Filtri

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

I componenti di tipo *vipInput* e *vipOutput* 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:

```
VIPRESULT vipVideo4Linux::extractTo(vipFrameT<unsigned char>& img)
{
    if ( fd == -1 )
        return VIPRET_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 == vipFrame::VIPFRAME_BGR24  ) {

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

        return VIPRET_OK;
    }
    // [...]
}

VIPRESULT vipWindowGTK::importFrom(vipFrameRGB24& img)
{
    if (image == NULL)
        return VIPRET_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 VIPRET_OK;
}
```

I componenti derivati da *vipCodec* 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* ed inoltre il formato

dati dello stream può non essere direttamente compatibile con alcun oggetto frame base, ad esempio:

```
VIPRESULT vipCodec_MPEG::extractTo(vipFrameRGB24& img)
{
    if (file == NULL)
        return VIPRET_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);
    vipUtility::conv_rgb24_rgb96( buff, (unsigned char*)img.data[0],
                                width, height);
    return VIPRET_OK;
}
```

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

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

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

    myOutput << bufferGlobal;
}
```

Questo sistema è leggermente complicato dal fatto che ci sono tre possibili formati di frame ingresso (*vipFrameRGB24*, *vipFrameYUV420* e *vipFrameT<unsigned char>*), oltre ai metodi preposti all'accesso e alla gestione dei tre buffer si sono incluse direttamente in *vipFilter* anche le implementazioni delle funzioni di estrazione (imposte da *vipInput*):

```
VIPRESULT vipFilter::extractTo(vipFrameYUV420& img)
{
    INFO("VIPRESULT vipFilter::extractTo(vipFrameYUV420& img) [pushing data]")

    if ( !isBufferYUV() )
        return VIPRET_ILLEGAL_USE;

    img = *bufferYUV;

    return VIPRET_OK;
}
```

Il metodo relativo a *vipFrameRGB24* è simile, mentre per quanto riguarda *vipFrameT* la fase di controllo prende in considerazione anche il profilo (*vipFrameT::profile* distingue il formato). Come si può facilmente notare l'estrazione è valida solo se i formati corrispondono. La conversione automatica è stata deprecata; poichè 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 *vipFrameT*:

```
VIPRESULT vipDigitalFilter::importFrom(vipFrameYUV420& img)
{
    int ret = VIPRET_OK;
    if ( !isBufferYUV() ) {
        useBufferYUV(img.width, img.height);
        ret = VIPRET_OK_DEPRECATED;
    }
    if (myParams->currentKernel == NULL) {
        *bufferYUV = img;
        return ret;
    }
    else
        ret += doProcessing(    img, *bufferYUV,
                                *myParams->currentKernel,
                                myParams->clampNegative    );
    return ret;
}

VIPRESULT vipDigitalFilter::importFrom(vipFrameT<unsigned char>& img) {
    return VIPRET_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(vipFrameRGB24)* effettua il controllo 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 *vipFilterGeometric::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 vipFilter::useBufferYUV(unsigned int width, unsigned int height) {
    if ( bufferYUV == NULL )
        bufferYUV = new vipFrameYUV420(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 *releaseBuffers*.

5.2 –Lavorare con i Frames

Questa sezione presenta una serie di suggerimenti e consigli pratici relativi all'accesso ad oggetti frame, alcuni lettori potranno valutarli scontati e 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 vipFrameSample { int width; int height; unsigned char* data; };
void BADcutValue (vipFrameSample& 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 (vipFrameSample& 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 (vipFrameSample& 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 *vipFrameGrey.cpp* ottimizza il processo di copia grazie alle funzione *memset* e *memcpy*:

```
vipFrameGrey& vipFrameGrey::operator >> (vipFrameYUV420& 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:

```
vipFrameGrey& vipFrameGrey::operator >> (vipFrameYUV420& 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. Si tratta di un errore 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 nel momento in cui si tenta di delegare ad un'unica funzione (*reset*) l'inizializzazione. La versione proposta (corretta) è valida anche dal punto di vista dello stile.

5.3 – Gestione dei Parametri per i Filtri

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

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

La classe base *vipFilterParameters* 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 *vipFilterGeometric* crea il seguente XML:

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

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 spiare i filtri esistenti e modificarne il codice. La “serializzazione” deve comunque includere un selettore del buffer in uso. A tale scopo è sufficiente inserire il seguente estratto:

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

La variabile (enumerazione) *currentBuffer* è dichiarata nella classe madre *vipFilterParameters* 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 vipFilterGeometric :    public vipFilter {
protected:
    vipFilterGeometricParameters* myParams;
```

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

```
VIPRESULT setFilterParameters (vipFilterParameters* initParams) {
    return
        setParameters(static_cast<vipFilterGeometricParameters*>(initParams));
};

vipFilterParameters* getFilterParameters () {
```

```

return static_cast<vipFilterParameters*>(myParams);
};

```

Dove la classe *vipFilterGeometricParameters* è l'implementazione di *vipFilterParameters* per la classe *vipFilterGeometric*, la funzione di accesso ai parametri (senza casting) è banalmente:

```

vipFilterGeometricParameters& getParameters() { return *myParams; };

```

mentre la funzione che imposta realmente i parametri è:

```

VIPRESULT vipFilterGeometric::setParameters(vipFilterGeometricParameters* initParams)
{
    if (initParams != NULL && myParams == initParams)
        return VIPRET_PARAM_ERR;

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

        myParams = initParams;
    }
    allocateBuffer(myParams->currentBuffer);
    return VIPRET_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 *vipFilterGeometricParameters* inizializza le variabili tramite il costruttore:

```

vipFilterGeometricParameters(RUNMODE mode = DO_NOTHING) :
                                                                vipFilterParameters()
{
    runMode = mode;
}

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

```

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

```

void vipFilterGeometricParameters::reset()
{
    runMode = vipFilterGeometricParameters::DO_NOTHING;
    par_Rotation = 0;
    par_ResizeWidth = 0;
    par_ResizeHeight = 0;
    par_forzeSize = false;
    currentBuffer = vipFilterParameters::NONE;
}

```


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

```
VIPRESULT vipFilterGeometric::reset()
{
    releaseBuffers();

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

    return VIPRET_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 *vipFilter* inizializza i puntatori ai tre buffer).

Il costruttore dei filtri deve (non per ragioni di 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 cui tipica implementazione è banale:

```
vipFilterGeometric(vipFilterGeometricParameters* initParams = NULL) :
                                                                    vipFilter(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 *vipInput*, si è scelto di ignorare la gestione della *frame rate* con la chiamata *vipFilter(0)* che, come si può facilmente notare dal codice precedente, passa il parametro (0) al costruttore di *vipInput* (e quindi *sleeptime* = 0).

Infine l'istanza della classe *vipFilter<..>Parameters* viene liberata dal distruttore. I buffers sono invece liberati dal distruttore della classe madre *~vipFilter*, 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:

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

```
}
```

5.4 – Useful Components

vipFrameRGB24

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 (*VIPFRAME_PROFILE*) è *VIPFRAME_RGB24* e naturalmente il sistema di memorizzazione (*VIPFRAME_CHANNEL_TYPE*) è di tipo *VIPFRAME_CT_PACKED*.

vetFrameRGB24.h

```
VIPCLASS_TYPE_FRAME
VIPFRAME_RGB24
VIPFRAME_CT_PACKED
0x32424752
```

L'allocazione del buffer è estremamente banale:

```
void vipFrameRGB24::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
```

vipWindowGTK

Questo modulo, molto simile a *vipWindowQT*, 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(vipFrame_*)* per visualizzarli in una finestra. La creazione di una nuova applicazione nell'ambiente grafico è automatica (al contrario di *vipWindowQT*). È stata testata con tutte le sorgenti dati di *VIPLib* (su Linux), ed ha ottenuto risultati (anche se di poco) migliori rispetto alla gemella *vipWindowQT* (34 fps).

Implements: vipOutput

/outputs/vipWindowGTK.h

vipWindow32

La classe `vipWindow32` 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: `vipOutput`
for Windows32 only
`/outputs/vipWindow32.h`

vipDoctor

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.

Implements: `vipOutput`
Integrated in WorkShop
`/outputs/vipDoctor.h`

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

vipCodec_BMP

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

Implements: `vipCoder`
Serializable
`/codecs/vipCoder_BMP.h`

`vipCodec_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.

vipCodec_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: `vipCoder`
Serializable
`/codecs/vipCoder_IMG.h`

Attualmente il componente è in grado esclusivamente di caricare immagini in classi `vetFrameRGB24` (`vipCodec_IMG::load(vipFrameRGB24& img, char* filename)`);

vipUtility

`/vipUtility.h`

Questo modulo incorpora una serie di metodi comuni dichiarati *static* (accessibili anche senza un'istanza della classe: `vipUtility::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.

vipMatrix

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
`./math/vipMatrix.h`