Laboratory for Advanced Planning

and Simulation Project

Specifiche di *Hardware Abstraction Layer* del Sistema di Acquisizione Tridimensionale Ailun

Gianstefano Monni

Gems Area

Specifiche Hardware Abrstaction Layer

del Sistema di

Acquisizione Tridimensionale Ailun

SOMMARIO

1 Progetto dell'Hardware Abstraction Layer

Durante lo sviluppo del software è sorta l'esigenza di separare nettamente la gestione dell'hardware dal resto del codice.

Gli obiettivi che sono stati raggiunti sono molteplici :

1. un codice il più possibile portabile (da windows a Linux), manutenibile e modulare.
2. supporto dell'hardware più disparato, e rendere possibile la scelta a runtime del tipo di hardware tramite cui effettuare l'acquisizione.
3. disaccoppiamento tra la gestione dei diversi pezzi interni allo strumento di acquisizione (es. camera e obiettivo)
4. sistema unificato di gestione dei driver che segue un paradigma comune a tutti i driver (open, close, RTacquire e DSacquire) in modo tale da semplificare lo sviluppo da parte di personale esterno al team.

Per raggiungere questi obiettivi si è proceduto alla reingegnerizzazione di tutto il livello di astrazione dell'hardware (HAL) anche mediante l'uso di tecniche consolidate, note nell'ambito dell'ingegneria del software con il nome di design pattern.

1.1 I Design Pattern

1.1.1 Introduzione

Dice Christopher Alexander "*Ogni pattern descrive un problema che si ripete più volte nel nostro ambiente, descrive poi il nucleo della soluzione del problema, in modo tale che si possa usare la soluzione un milione di volte, senza mai applicarla nella stessa maniera*" [AIS 77, pag. x]. Anche se Alexander si riferiva a pattern architettonici per edifici e città, quanto detto è vero anche per i pattern per la programmazione a oggetti. Le nostre soluzioni sono espresse in termini di oggetti e interfacce invece che muri e porte, ma il concetto di base delle due tipologie di pattern è quello di trovare una soluzione a un problema in un determinato contesto.

Generalmente un pattern consta di quattro elementi essenziali:

1. **il nome del pattern**: è un nome simbolico che possiamo usare per descrivere in una parola, o due, un problema di progettazione, le sue soluzioni, e le conseguenze della soluzione scelta.
2. **il problema** descrive la situazione alla quale applicare il pattern. Può descrivere problemi di progettazione specifici o anche strutture di classi o di progetti non flessibili.
3. **la soluzione:** descrive gli elementi che costituiscono il progetto, le loro relazioni, responsabilità e collaborazioni.
4. **le conseguenze:** sono i risultati e i vincoli che si ottengono applicando il pattern.

il punto di vista modifica le interpretazioni della natura di un pattern: ciò che per una persona rappresenta un pattern, per un'altra può costituire un semplice punto di partenza. I design pattern non sono liste concatenate o tabelle di hash che possono essere definite con una singola classe e quindi riusate come sono. Non si tratta neppure di progetti complessi relativi a un'intera applicazione o a

un sottosistema. **I design pattern sono descrizioni di oggetti comunicanti e di classi, adattate in modo da poter risolvere un problema di progettazione in un particolare contesto.**

1.1.2 Classificazione dei pattern

I design pattern sono classificati seguendo due criteri: scopo e raggio d'azione.

Lo scopo indica ciò che il pattern fa. Un pattern può avere tre scopi diversi: creazionale, strutturale, comportamentale**.**

a. Un pattern creazionale riguarda il progesso di creazione degli oggetti.
b. Un pattern strutturale ha a che fare con la composizione di classi e oggetti.
c. Un pattern comportamentale si occupa del modo in cui classi o oggetti interagiscono reciprocamente e distribuiscono fra loro le responsabilità.

Il raggio d'azione specifica se il pattern si applica principalmente a classi o a oggetti.

a. I class pattern riguardano le relazioni tra classi e sottoclassi. Queste relazioni sono espresse attraverso l'ereditarietà, quindi sono statiche, fissate al momento della compilazione.
b. Gli object pattern riguardano relazioni tra oggetti le quali possono essere cambiate durante l'esecuzione e sono più dinamiche.

Quasi tutti i pattern usano in qualche misura l'ereditarietà, quindi i soli pattern definiti *class pattern* sono quelli in cui l'elemento principale è costituito dalle relazioni fra classi. Da notare che quasi tutti i pattern hanno come raggio d'azione gli oggetti.

1.2 Il design pattern Singleton

1.2.1 Introduzione

Il problema che ci si è presentato durante lo sviluppo del software:

c'è un insieme di parametri che descrivono la geometria, le diverse correzioni, e altri valori che sono fondamentali per tutte le altre parti dell'applicazione. Questo set di parametri deve essere condiviso da tutti gli oggetti che costituiscono l'applicazione. Deve inoltre essere possibile modificare velocemente tutto il set di parametri.

Il pattern: *avere una ed una sola istanza di una classe, e avere un punto di accesso globale a questa*.

Il Singleton definisce un'operazione Instance che consente ai client di accedere all'unica istanza esistente della classe. Instance deve essere un'operazione di classe, in altre parole deve essere una funzione membro statica in C++. Instance può essere responsabile della creazione della sua unica istanza.

1.2.2 Vantaggi

1. accesso controllato a un'unica istanza. Poiché la classe Singleton incapsula la sua unica istanza, può mantenere un controllo stretto sulle modalità e i tempi con cui i client accedono all'istanza.
2. riduzione dello spazio dei nomi. Il pattern *Singleton* rappresenta un miglioramento rispetto all'uso di variabili globali. Evita di inquinare lo

spazio dei nomi con variabili globali utilizzate per memorizzare i riferimenti alle singole istanze.

3. permette il raffinamento di operazioni di rappresentazione. È possibile definire delle sottoclassi della classe *Singleton* ed è semplice configurare un'applicazione in esecuzione in modo tale da utilizzare un'istanza di questa classe estesa.
4. permette di gestire un numero variabile d'istanze
5. maggiore flessibilità rispetto a operazioni di classe. Un altro modo di realizzare le funzionalità di un singleton consiste nell'utilizzare operazioni di classe, cioè funzioni membro statiche in C++. Purtroppo, entrambe queste tecniche fornite dai linguaggi rendono molto difficoltosa la modifica del progetto in modo da consentire l'utilizzo di più istanze di una classe. Inoltre, le funzioni membro statiche in C++ non sono mai virtuali, rendendo impossibile alle sottoclassi di sovrascrivere l'implementazione della superclasse e di sfruttare il polimorfismo.

1.2.3 Implementazione.

Il pattern Singleton è stato implementato in una classe, CParameters, che è stata poi compilata come DLL, e usata da tutti gli altri oggetti dell'applicazione.

```
class DLL_EXP CParameters
{
private:
        float Wv;
protected:
   static CParameters * parm;
```

```
  CParameters(CParameters&){};
  CParameters operator =(CParameters&){};
  CParameters();
  ~CParameters(){};

public

  static CParameters * getInstance();
  static void deleteInstance();

float getWv(){return Wv;}
}
```

Esempio d'uso:

```
CParameters * parm=CParameters::getInstance();
float inverse=1.0/parm->Wv;
```

I client accedono al singleton esclusivamente attraverso la funzione membro getInstance. L'oggetto parm è inizializzato a zero e la funzione membro statica getInstance restituisce il suo indirizzo, inizializzandolo con il riferimento all'istanza se questo è a 0: getInstance utilizza la "lazy initialization", in altre parole il valore restituito non viene creato e memorizzato fino a quando non si chiede di accedervi per la prima volta.
Si osservi che il costruttore della classe è dichiarato protected e un client che cerchi d'istanziare direttamente la classe CParameters ottiene un errore in fase di compilazione. Questo assicura l'unicità dell'istanza creata.

Sempre riguardo all'implementazione è bene notare che la deleteInstance **non deve essere chiamata** se non alla fine dell'esecuzione del programma, e la chiamata alla deleteInstance è un momento abbastanza critico visto che, di fatto, distrugge l'area di memoria di un oggetto condiviso tra tutti gli altri oggetti dell'applicazione.

1.3 Il pattern Proxy

1.3.1 Introduzione

Il problema che ci si è presentato durante lo sviluppo:

Dobbiamo supportare un sistema unificato di gestione dei driver: deve essere possibile, con la stessa interfaccia, scegliere un driver USB, PCI, Firewire o Frame grabber. I driver per i diversi bus devono avere la stessa interfaccia, e comportarsi in modo esattamente identico, la FSM che li descrive è unica per tutti i drivers.

I due pattern: *Definire un'interfaccia per la creazione di un oggetto lasciando alle sottoclassi la decisione sulla sottoclasse che deve essere istanziata.*
Fare in modo che l'accesso all'oggetto "reale" creato sia controllato.

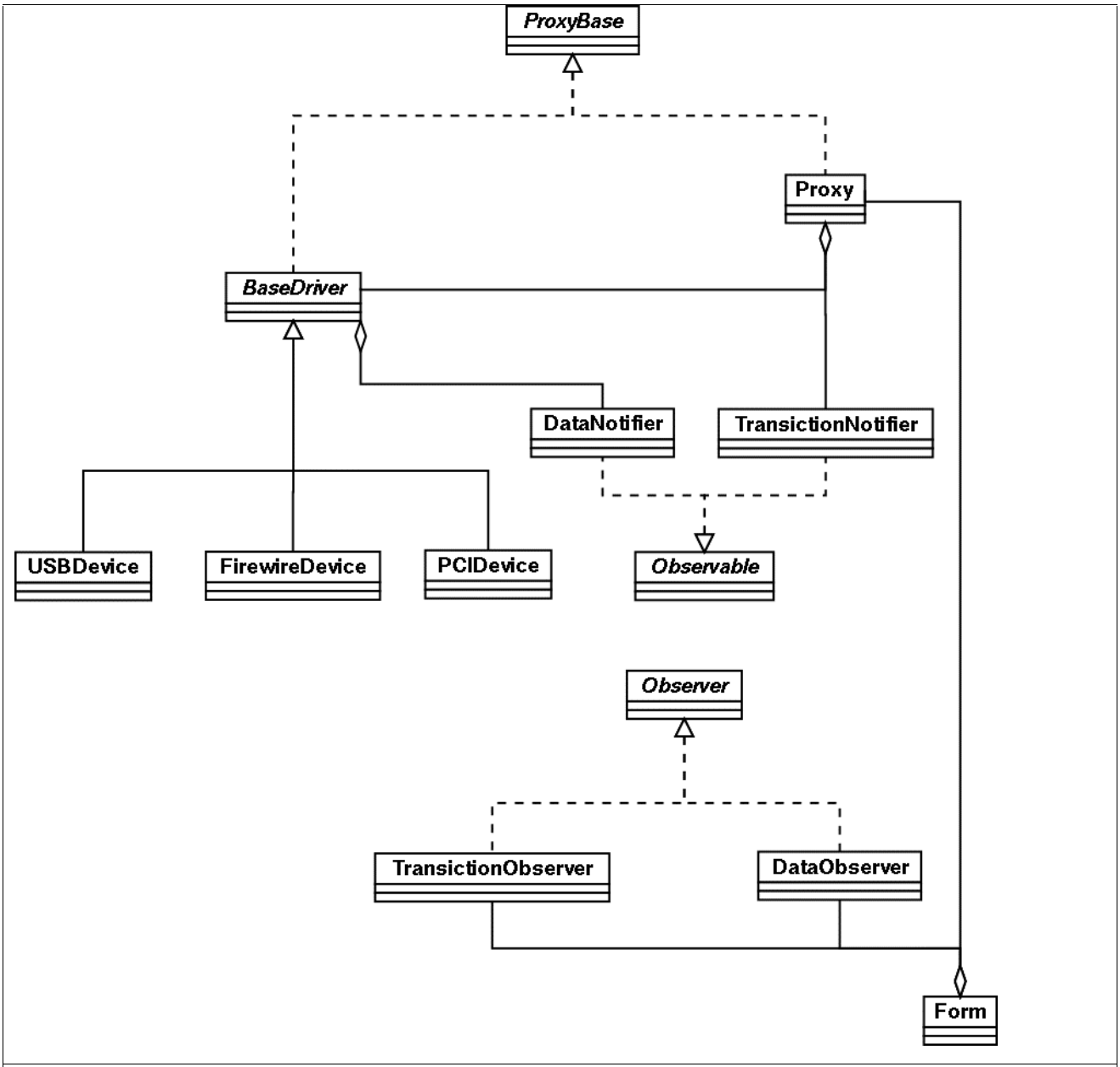


Fig. 1 UML Class Diagram

ProxyBase è l'interfaccia comune a BaseDriver e a Proxy.

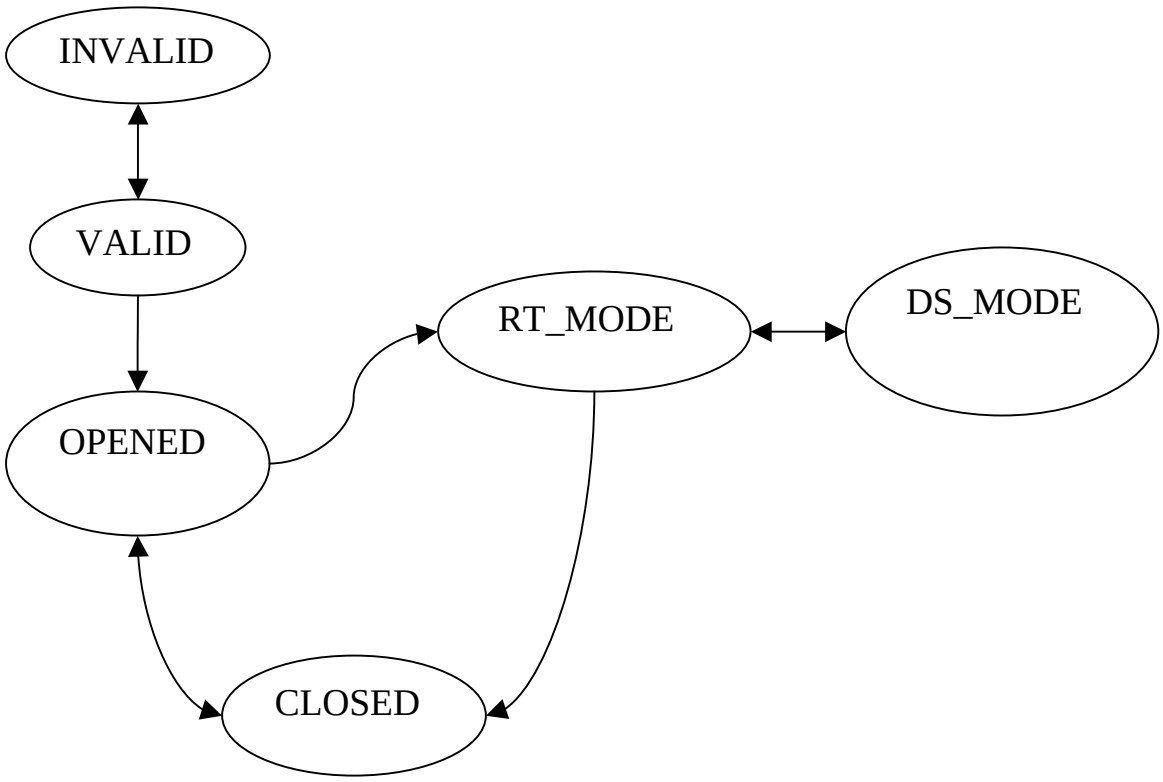
Proxy nasconde all'esterno la scelta del driver e l'esterno utilizza il metodo setBase (int idDriver) per fare in modo che l'oggetto reale usato da Proxy sia corretto. Proxy nasconde all'esterno le problematiche legate all'apertura e alla chiusura del driver, e soprattutto alla costruzione dell'oggetto Driver reale.

Proxy implementa tutte le funzionalità di BaseDriver, ma lo fa usando un oggetto interno che Proxy stesso crea/distrugge. Proxy,

BaseDriver è la superclasse (astratta) di tutti i Drivers reali e ne implementa alcuni metodi. Tutti i metodi di Basedriver sono virtuali, in particolare RTAcquire, DSAcquire, open e close sono virtuali e puramente astratti. L'idea di fondo è che, per la realizzazione di un driver minimale, bastino questi quattro metodi, e che questi quattro metodi non debbano essere implementati nella superclasse che, di fatto, non è agganciata ad alcun Device reale.

{Firewire/PCI/USB}Driver sono tre sottoclassi

1.3.2 La FSM di Proxy



2 Problemi relativi al driver di ogni camera

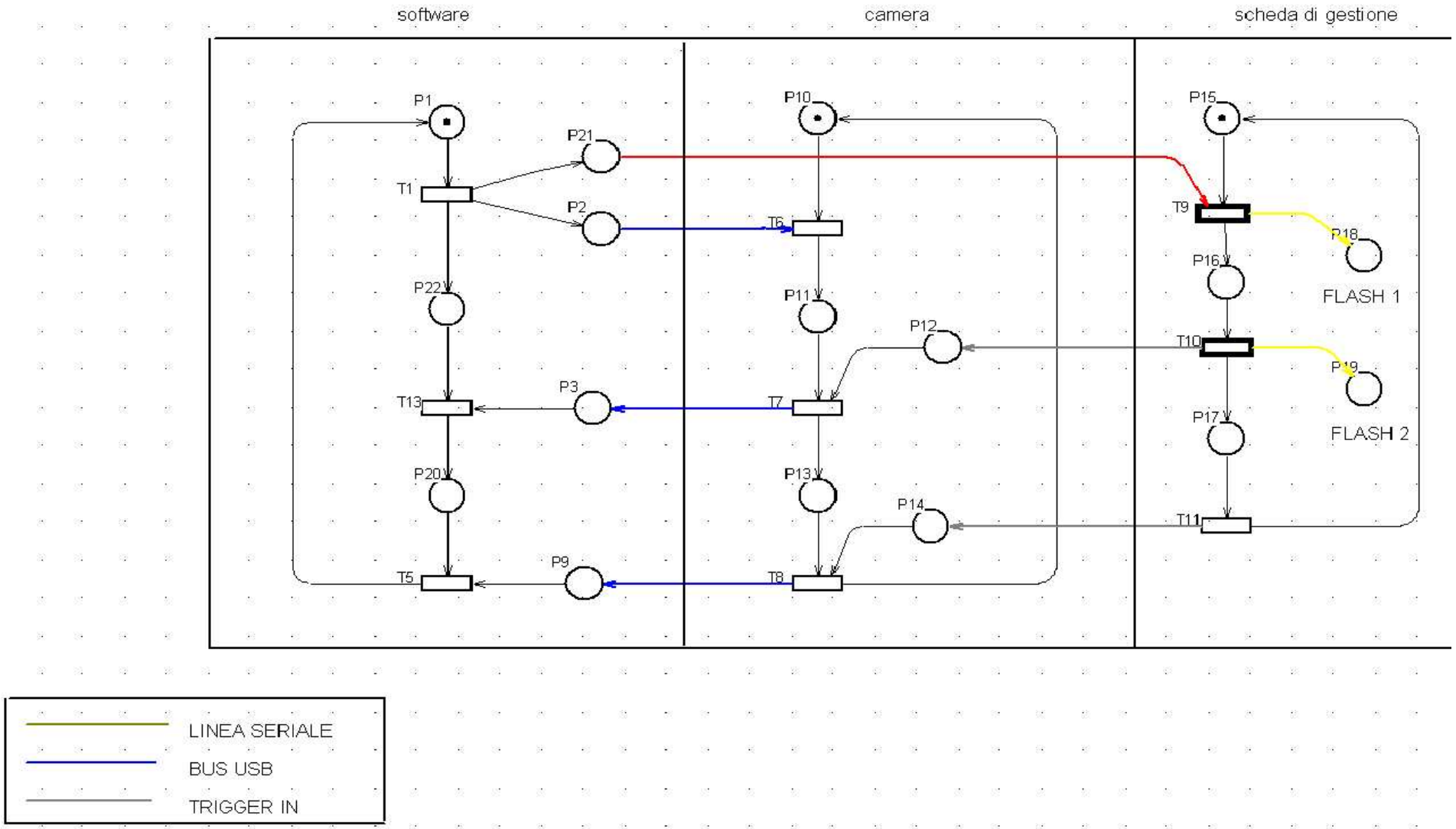
2.1 introduzione

Durante l'implementazione dei driver "reali" delle camere si sono affrontati alcuni problemi legati alle diverse implementazioni che i produttori danno dei VxD (Virtual Device Drivers). In particolare sono stati affrontati due aspetti:

1. progettazione di un protocollo asincrono per la gestione della camera: una delle camere fornite non era dotata, infatti, di un segnale di trigger OUTPUT.
2. implementazione della primitiva GrabTriggered atomica e multithread

2.2 Il protocollo asincrono progettato

In una delle camere di prova il segnale di TRIGGER OUTPUT non funzionava correttamente. Risultava quindi usare il protocollo sincrono che è stato usato per la coppia scheda frame-grabber ←→camera CCD esistente. Per poter implementare il driver di questa camera è stato necessario progettare un protocollo di gestione asincrono tra scheda di gestione, camera e modulo software.



Rete di Petri del protocollo asincrono progettato

2.3 Implementazione di GrabTriggered (unitCamera)

L'implementazione di GrabTriggered richiede la realizzazione di un thread che rimanga nello stato BLOCKED fino a quando non arriva il segnale che l'immagine è stata correttamente acquisita. L'implementazione di un Thread viene demandata alla classe CThread, superclasse di SICamera.
Il chiamante si mette in stato di wait su grabbedEvent (semaforo binario, stato iniziale non segnalato). Quando SICamera completa l'esecuzione (all'interno di una regione critica) segnala il semaforo e sblocca il chiamante.

```
void SICamera::Execute()
{
 CRITICAL_SECTION cs;
   InitializeCriticalSection(&cs);
   EnterCriticalSection(&cs)
...

   ResetEvent(grabbedEvent);

//actual grab
PrivateGrab(0);

 SI_DisarmTrigger(theCamID);
LeaveCriticalSection(&cs);
DeleteCriticalSection(&cs)
;
//signal semaphore: acquisition is completed
SetEvent(grabbedEvent);
}
```

2.3.1 Implementazione di CThread

Al fine di garantire la massima portabilità i Thread sono stati incapsulati in una superclasse , CThread, che fornisce il supporto minimo per il multithreading

```
/*
constructor of the class: use win32 API to create a thread, assigning propertly
callback functions to methods of the actual object that has been created.
*/
CThread::CThread()
{
  theStatus=tsIDLE;
  t_box= new ThreadBox(this);
   hThread=CreateThread(NULL,
     0,
   (unsigned long (__stdcall *)(void *))&(CThread::_execute),
   (void *)t_box, CREATE_SUSPENDED, NULL);

   if (hThread==NULL)
     throw new Exception("Can't create Thread");
}

CThread::~CThread()
{
  CloseHandle(hThread);
}


void CThread::_execute(void * PtrThis)
{
```

```
  ThreadBox * ptr = (ThreadBox * )PtrThis;
  CThread * this_t=ptr->this_thread;
  this_t->theStatus=tsRUNNING;
  this_t->Execute();
  this_t->theStatus=tsENDED;
}
void CThread::t_restart()
{
  ResumeThread(hThread);
}
void CThread::t_suspend()
{
  SuspendThread(hThread);
}
```

CThread è una classe astratta: ogni sottoclasse "reale" di CThread deve implementare il metodo Execute() che contiene il codice che viene eseguito all'interno del thread.
Il metodo _execute*(void * PtrThis)* è quello che viene chiamato, in callback, dalla win32API, e a tale metodo viene passato il puntatore all'oggetto corrente. Usando il this pointer viene poi chiamato il metodo (puramente virtuale in CThread, virtuale nelle altre classi) Execute.

2.3.2 Considerazioni sulla portabilità

Sia GrabTriggered che Cthread sono, per ovvie ragioni, non complemetamente portabili, tuttavia l'incapsulare in una classe a sé stante le funzionalità di GrabTriggering e multithreading semplifica di molto la portabilità in altri sistemi operativi che supportino l'IPC e in particolare i thread, i semafori e le regioni critiche.

3 La comunicazione mediante seriale (CcommLib)

All'interno del progetto è stato necessario, per ragioni di portabilità, manutenibilità e espansibilità del sistema, sviluppare una classe per la comunicazione mediante la porta seriale. Il driver, poi, è stato utilizzato sia per gestire l'accensione delle lampade flash sia per gestire il motore passo passo dell'obiettivo. Il driver incapsula anche una piccola FSM per la gestione degli eventi open, close, trigger.

3.1 Implementazione di COsiride

Osiride è il nome della scheda di gestione. COsiride è il driver, multithreaded, della scheda stessa. COsiride eredita da CThread, e in quanto tale può essere eseguito come un thread a parte. COsiride, inoltre, usa CcommLib per mandare i segnali hardware alla scheda attraverso la seriale. COsiride, inoltre, incapsula anche una FSM semplice che emula la FSM della scheda di gestione.
Un metodo di COsiride che può essere interessante analizzare è Execute (il metodo puramente virtuale di Cthread, che viene poi implementato in tutte le sue sottoclassi)

```
void COsiride::Execute()
{
  //open com and set some parameters.
   comm->open();
   comm->SetTimeout(500);
   comm->writeChar(actual_commands[g_cmd]) ;
   comm->writeChar(CR);

   BYTE val=comm->read();
```

```
//check what's the value osiride has sent back
   if (val==ACK_Y)
   {
     // OK: sw must sleep, wait for transmission to be completed and close port
      Sleep(total_time);
      comm->close();
      return      ;
   }
   else if (val==ACK_N)
   {
   // something went wrong: throw an excpetion
      Application->MessageBoxA("can't understand g command", "error",MB_OK);
   }
   else
      throw Application->MessageBoxA("trouble in comm", "error",MB_OK);
}
```

4 Il formato dei dati

L'importanza della memorizzazione dei dati bidimensionali, nell'applicazione in progetto, è legata a due fattori essenziali:

1. la "quantità di informazione" in un'immagine risulta molto più ricca rispetto alla forma tridimensionale estratta dall'immagine. A patto di avere il software necessario, è sempre possibile passare dall'immagine alla forma 3D, ma non è possibile (al momento) effettuare il passaggio inverso.
2. la dimensione dei files contenenti le immagini risulta molto più ridotta (almeno, con i formati da noi usati) rispetto alla dimensione dei files 3D.

fatte queste considerazioni, ci si è posto quindi il problema di trovare un formato per la memorizzazione delle immagini che supportasse una dinamica superiore ai 255 livelli di grigio, che fosse compresso, e che fosse LOSSLESS.

Un altro requisito che è emerso durante lo sviluppo è stata la necessità di "agganciare" al file d'immagine delle informazioni associate all'hardware, quali ad esempio la correzione della distorsione, la versione dello strumento, ecc ecc.

4.1 JPEG 12 bit

Per soddisfare le specifiche di cui al par. 4 in prima analisi si è preso in considerazione il JPEG a 12 bit, che supporta fino a 4096 livelli di grigio, dandogli come parametro di qualità 100. JPEG monocromatico a 12 bit e a qualità 100 non è lossless, ma l'errore introdotto sembra essere trascurabile per i nostri scopi.

Un altro vantaggio del JPEG è che permette di utilizzare dei marker: fino a 7 campi, di 64 kB ciascuno, che possono contenere uno stream di bytes che non viene processato dai viewer, ma è lasciato all'uso da parte dei programmatori.

Si è quindi incapsulata la IJG library (www.jpeg.org) in una classe, compilata per i dodici bit e fornito il supporto ai markers.

A seguito di questo lavoro ci si è poi resi conto che, in realtà, praticamente nessuno supporta il JPEG 12 bit, nonostante sia standard. L'uso dei marker, inoltre non era esattamente confacente alle nostre esigenze, e si è preferito usare un formato custom, compresso, realmente lossless e a precisione arbitraria.

4.2 Il Formato ZBI: Zipped Binary Image

A seguito della sperimentazione del JPEG a 12 bit, si è definito un nuovo formato lossless: ZBI. ZBI è, sostanzialmente, una bitmap a precisione arbitraria (8,10,12, 16 o più bit), con un complesso header contenente **tutti** i parametri hardware e software necessari ad una perfetta ricostruzione della forma tridimensionale estratta dall'immagine. ZBI, dopo essere costruito come stream di bytes viene compresso mediante un'implementazione win32 delle zlib. In questo modo ZBI risulta, in termini di occupazione, comparabile al JPEG corrispondente, senza tuttavia avere vincoli sull'errore nel passaggio da un formato all'altro, e fornendo inoltre, un migliore supporto per i dati non legati all'immagine.

4.3 La memorizzazione della forma 3D

Per memorizzare i dati tridimensionali estratti dall'immagine e renderli disponibili ad altri programmi si sono implementati i filtri per due formati standard:

1. VRML
2. STL

Per entrambi questi formati si è implementato un encoder-decoder. Si è, inoltre, implementato un sistema per la traduzione dall'uno all'altro. È inoltre disponibile il codice per estrarre, dal VRML o dal STL la nuvola dei punti da triangolare.