Università degli Studi di Salerno

Facoltà di Scienze Matematiche Fisiche e Naturali

Tesi di Laurea di I livello in

Informatica

Una libreria per l'interazione hand-based in ambienti virtuali

Relatore
Prof. Vittorio Scarano

Candidato
Dario Scarpa

Anno Accademico 2006-2007

Dediche e ringraziamenti

Ringrazio la mia famiglia, i miei amici e tutto lo staff dell'ISISlab per il costante supporto ricevuto, in diverse forme, durante la stesura di questa tesi. Un ringraziamento speciale a tutti gli sconosciuti che, passando davanti al laboratorio, non sono rimasti straniti vedendomi gesticolare concentrato di fronte al monitor. Se ce ne sono stati.

...dedicata a tutti quelli che hanno bisogno di una mano

Questa tesi è stata sviluppata in 

Indice

1 Introduzione **1**

2 Strumenti utilizzati **9**
2.1 Hardware . . . 9
2.1.1 5DT Data Glove 16 . . . 9
2.1.2 InterSense PCTracker . . . 11
2.2 Software . . . 13
2.2.1 wxWidgets . . . 13
2.2.2 Quest3D . . . 17

3 Realizzazione dell'infrastruttura **20**
3.1 ISISgloveAPI . . . 23
3.1.1 5DT Data Glove: l'SDK fornita . . . 23
3.1.2 Obiettivi di ISISgloveAPI . . . 30
3.1.3 Descrizione generale e cenni sull'implementazione . . . 31
3.1.4 I/O e confronto dei tipi user-defined . . . 41
3.1.5 Un confronto tra librerie . . . 45
3.1.6 Sviluppi futuri . . . 47
3.2 ISISgloveManager . . . 48
3.2.1 Utilizzo dell'applicazione . . . 48
3.2.2 Architettura . . . 52
3.2.3 Implementazione . . . 54
3.2.4 Sviluppi futuri . . . 64
3.3 ISISgloveManagerInput . . . 66
3.3.1 Il supporto nativo al 5DT Data Glove, e perché sostituirlo . . . 66
3.3.2 Sockets in Quest3D . . . 67
3.3.3 Lua scripting: canali con logica personalizzata . . . 68
3.3.4 L'interfacciamento con ISISgloveManager . . . 69

3.3.5 Sviluppi futuri 75
3.4 ISISpcTracker 77
3.4.1 Il supporto nativo ai trackers *InterSense*, e perché sostituirlo 77
3.4.2 Cenni sull'SDK dell'Intersense PCTracker 79
3.4.3 Estendere *Quest3D*: la creazione di un channel 82
3.4.4 Implementare il supporto all'*Intersense PC Tracker* 84

4 SkeleTronDemo **92**
4.1 La mano virtuale 92
4.1.1 Animare un modello di mano scheletrica 92
4.1.2 Motion tracking del polso 96
4.2 Una walkthrough camera tracker-based 98
4.3 Un esempio di interazione gesture-based 99

5 Conclusioni **102**

Bibliografia **106**

A Appendice **109**
A.1 Listati 109
A.1.1 ISISgloveAPI 109
A.1.2 ISISgloveManager 118
A.1.3 ISISgloveInput 154
A.1.4 ISISpcTracker 156

Capitolo 1

Introduzione

> *"Back in the early 1980s, my colleagues and I spent a lot of time exploring how the human body could best interact with the virtual world. The iconic VR device that emerged from that work was the DataGlove, a sensor-filled glove that would direct a real-time virtual model of your hand into a computer-generated world."* [22]
>
> Jaron Lanier, pioniere della realtà virtuale

Il primo *"DataGlove"* fu sviluppato a metà degli anni '80 dalla *VPL Research*, la prima azienda a occuparsi della vendita di dispositivi per la *realtà virtuale*, termine coniato proprio dal suo giovane fondatore, Jaron Lanier.

Il guanto fu realizzato per fare da periferica di input adatta a suonare una "air guitar", tradendo la passione di Lanier, compositore e visual artist oltre che informatico, per gli strumenti musicali *alternativi*.

Per la prima volta, grazie al *DataGlove*, una mano era usata per l'interazione in un mondo 3D simulato. Da allora, insieme agli *Head Mounted Displays*, i *"Wired Gloves"* (termine "neutro" usato per indicare questa categoria di dispositivi, dato che *"DataGlove"* e *"CyberGlove"*, come spesso li si sente chiamare, sono due marchi registrati) sono alla base della realizzazione di sistemi di realtà virtuale più o meno complessi. D'altro canto ciò é abbastanza naturale, considerando che é proprio con le mani che compiamo gran parte delle interazioni col mondo che ci circonda.

Negli anni '80-'90 ci fu molta attenzione attorno alla realtà virtuale, anche grazie a film come l'innovativo "Tron" (1982) e il mediocre "The Lawnmower Man" ("Il Tagliaerbe", 1992), oltre che alla nascita della letteratura *cyberpunk* ad opera di *William Gibson*, che coniò il termine *"cyberspace"*

in "Burning Chrome" (1982) e lo rese popolare con l'opprimente distopia del suo "Neuromancer" (1984).

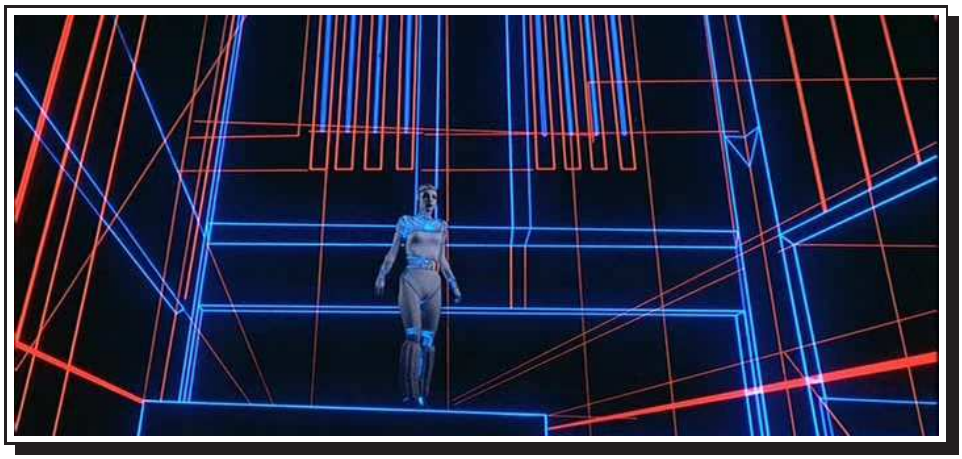


Figura 1.1: Per quanto tecnicamente e narrativamente ingenuo, "Tron" ha enormi meriti visivi: per la prima volta ha portato sul grande schermo l'idea di uno spazio creato al computer, e ha fissato nell'immaginario collettivo il suo stile grafico a base di wireframe fluorescente.

> Cyberspace. A consensual hallucination experienced daily by billions of legitimate operators, in every nation, by children being taught mathematical concepts... A graphic representation of data abstracted from banks of every computer in the human system. Unthinkable complexity. Lines of light ranged in the nonspace of the mind, clusters and constellations of data. Like city lights, receding

Figura 1.2: la definizione canonica di cyberspazio, di *William Gibson* [38]

L'hype attorno alle periferiche per la realtà virtuale fu tale che, nel 1989, la Nintendo tentò di portare "qualcosa del genere" in tutte le case commercializzando il *Power Glove*, un controller alternativo per la sua storica console, il *Nintendo Entertainment System*. Fu un adattamento del brevetto originale del *DataGlove* della VPL research, pesantemente limitato per abbassarne il costo e renderlo utilizzabile con la poco performante console a

Figura 1.3: Una mano proiettata nella realtà virtuale con un Data Glove ne "Il tagliaerbe", in cui, misto al materiale di scena, é presente del vero hardware per la realtà virtuale, come dimostra il marchio VPL visibile sul guanto.

8 bit. Ne uscì fuori un dispositivo limitatissimo e - come se non bastasse - supportato molto male dai giochi, tanto che il *Power Glove* ha guadagnato negli anni la fama di essere stato una delle peggiori periferiche per il gaming mai progettate. Nonostante ciò, ne furono venduti circa 100000 nei soli USA. Di recente la Nintendo é tornata a giocare sulle periferiche alternative di input, col suo *Nintendo WII*, indubbiamente meglio riuscito...

A ogni modo, non c'é mai stato un vero e proprio "boom" della realtà virtuale, il cui utilizzo é rimasto confinato in settori e ambienti ben precisi. Uno dei motivi di ciò risiede probabilmente nel costo dei sofisticati dispositivi coinvolti, oltre al fatto che appare ormai evidente che la strada dell'immersione totale in un ambiente virtuale non é pratica per la maggioranza dei compiti che si svolgono quotidianamente con i computer: tastiera e mouse, tra l'altro decisamente meno invasivi a livello fisico/psicologico per l'utente, restano la soluzione più efficace, a dispetto degli scenari cyberpunk.

Anche potendosi permettere di ignorare l'aspetto *costi*, la tecnologia esistente all'epoca dei primi esperimenti nel campo della realtà virtuale limitava - non poco - la qualità dell'esperienza immersiva: grafica 3D primitiva, sensori lenti e spesso imprecisi, dispositivi hardware pesanti e voluminosi. L'evoluzione delle tecnologie coinvolte é stata quindi cruciale nello svilupparsi di tali sistemi, e continuerà ad esserlo in futuro.

Ad esempio, una tecnologia strettamente correlata all'uso dei wired gloves é, da sempre, quella del *motion tracking*: i sensori presenti nei guanti

Figura 1.4: *Johnny Mnemonic*: trama e personaggi dimenticabili, ma splendida messa in scena del cyberspazio, in cui il protagonista naviga proprio tramite due *data gloves*, in maniera credibile e spettacolare.

provvedono a descrivere la posizione delle dita, ma é tipicamente necessario hardware a parte che permetta di localizzare la mano nello spazio. Alcuni guanti incorporano essi stessi dei meccanismi per il motion tracking, ma più frequentemente é necessario abbinarne l'utilizzo a un sistema separato.

Ci sono molte tecniche per effettuare il motion tracking, alcune basate sulla visione artificiale (e che quindi prevedono l'utilizzo di telecamere e marker da individuare) e altre che coinvolgono sensori di vario genere (inerziali, ultrasonici, meccanici, magnetici), ognuna con peculiarità e costi che le rendono più o meno adatte a un determinato tipo di utilizzo (frequentissimo l'uso nel *motion capture*, per animare personaggi di film e videogiochi). Non essendo il caso di estendere troppo il discorso, ci limitiamo a considerare il motion tracking una tecnologia di supporto usata per rilevare lo spostamento della mano "guantata".

Lasciando da parte la realtà virtuale, é opportuno parlare dell'utilizzo dei wired gloves come periferiche di input in altri contesti. Un'applicazione piuttosto ovvia é quella del riconoscimento automatico del linguaggio dei segni, per cui esistono molteplici implementazioni. Le più interessanti, di recente sviluppo, sono probabilmente quelle che accoppiano il sistema di riconoscimento dei gesti a un sintetizzatore vocale, permettendo ai sordomuti

Figura 1.5: Oltre a 4 sensori di flessione delle dita (niente da fare per il pollice) a due bit di risoluzione, il *Nintendo Power Glove* aveva due emettitori di ultrasuoni sul polso, e l'asta ad "L" in figura, che andava poggiata sul televisore, conteneva tre microfoni che, tramite triangolazione, fornivano alcune informazioni sul posizionamento della mano. Per quanto impreciso e limitato, il sistema valeva indubbiamente i soldi che costava e, all'inizio degli anni '90, la rivista "Byte" pubblicò le informazioni necessarie a modificare il dispositivo e collegarlo alla porta parallela di un PC. A breve, prevedibilmente, seguì il reverse engineering del protocollo.

di comunicare, con i limiti del caso, anche con chi non conosce il linguaggio dei segni. Altra applicazione tipica é quella della riproduzione a distanza dei movimenti, tipicamente per mezzo di una mano robotica potenzialmente di dimensioni e forza diverse dalla mano dell'utilizzatore, o magari collocata in ambienti dove non si può interagire di persona, ad esempio a causa di radioattività. Anche la *"remote surgery"*, ovvero la chirurgia a distanza, può comprendere l'utilizzo di un wired glove: in questo caso é indiscutibilmente necessario utilizzare un dispositivo "aptico", ovvero in grado di fornire feedback tattile al chirurgo. E' da dire però che la strada della chirurgia remota e robotizzata sembra, ragionevolmente, andare oltre la riproduzione robotica della mano del chirurgo: ad esempio la "Intuitive Surgical" ha sviluppato un sistema per la chirurgia basato su dei bracci robotici attrezzati e miniaturizzati (comandati da joystick ad hoc), pensati per agire meno invasivamente e più efficacemente di una mano umana.

E' poi possibile pensare di utilizzare dei data gloves per interagire con interfacce che, non tentando la riproduzione virtuale delle mani, usano gesti e movimenti per effettuare determinate azioni: é il caso di una famosa scena

del film *Minority Report*, in cui un sistema glove-based viene usato, con spettacolarità, per una sorta di manipolazione/riproduzione video.

Discutendo di questa tipologia di sistemi interattivi, risulta interessante questa recentissima (maggio 2007) citazione, sempre di Jaron Lanier:

"Many years after the glove era, I had a fun gig helping to brainstorm the movie *Minority Report* (...). The glove interface ended up with a starring role, and the filmmakers expended tremendous effort making that style of virtual interaction look viable when it actually isn't. I thought it was a fitting memorial: a real design that didn't quite work used to symbolize a fictitious future world that wasn't quite working. Fortunately, sensors and microprocessors have gotten much better and cheaper since then..." [22]

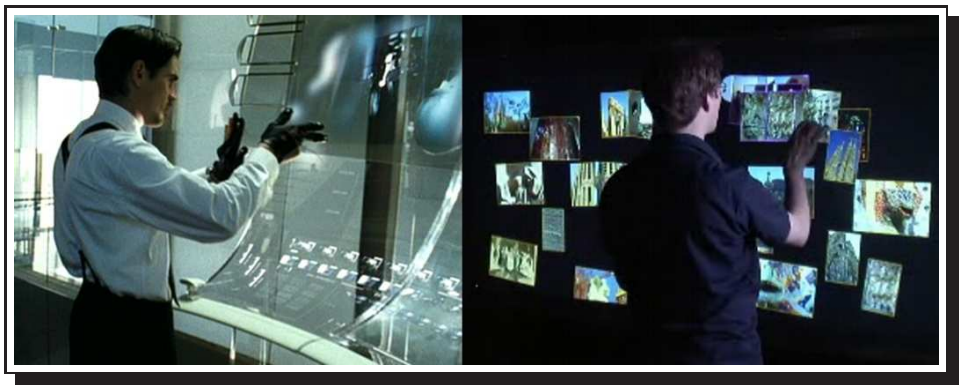


Figura 1.6: *Minority Report* e *PerceptivePixel*: sci-fi e realtà a confronto. Dato l'uso "bidimensionale o quasi" fatto delle mani nella famosa interfaccia glove-based di *Minority Report*, questa ricorda decisamente le interfacce multitouch-screen a parete o da tavola, tecnologie sviluppate da anni alla *PerceptivePixel* e che sembrano ormai prossime alla diffusione commerciale.

Lanier, che dalla fondazione della *VPL Research* (poi acquisita dalla *Sun Microsystems*) non ha mai smesso di essere attivo nel campo, considera quindi rilevante anche l'evoluzione dell'hardware degli ultimissimi anni (la sua consulenza alla preproduzione di *"Minority Report"*, uscito nelle sale nel 2002, risale al 1999) e, specialmente non avendo i mezzi di una produzione hollywoodiana, dei militari o dei laboratori all'avanguardia nella ricerca sulla realtà virtuale, probabilmente solo di recente si sono resi disponibili strumenti, sia hardware che software, che consentano di ottenere risultati concreti in questo campo con sforzo moderato.

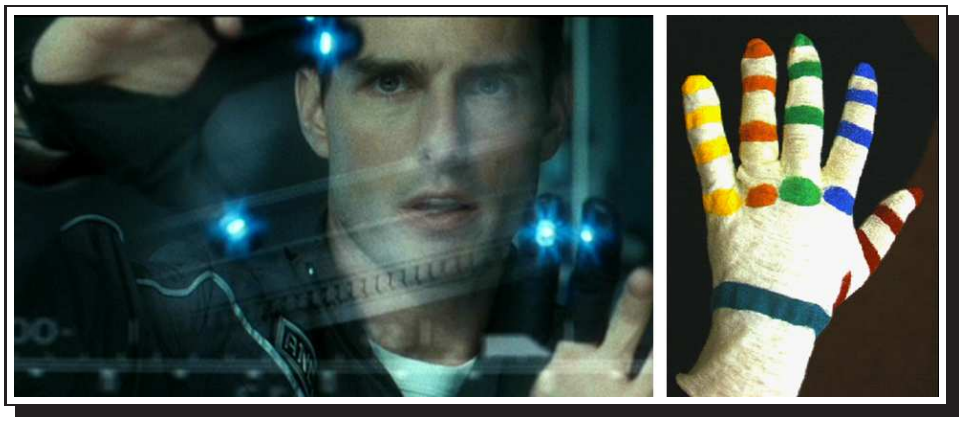


Figura 1.7: A sinistra, un'altra scena di *Minority Report* in cui un'interfaccia glove-based ha un ruolo centrale. I led azzurri sui curiosi guanti a tre dita del film, probabilmente aggiunti più che altro per esigenze scenografiche, fanno pensare all'approccio alternativo al motion tracking con sensori, ovvero quello basato su telecamere e "computer vision". E' infatti tipico di queste tecniche, per semplificare il software di riconoscimento, mettere in evidenza in qualche modo i punti da individuare. A destra, una foto estratta da un paper scientifico [8] al riguardo: sicuramente permette un hand tracking più accurato di tre led azzurri, ma é difficile non sorridere immaginandosi il protagonista del film, serio e concentrato, che gesticola indossando dei guanti del genere...

Ed é su strumenti del genere che si é basato il nostro lavoro: l'interfacciamento di un *wired glove* e di un dispositivo di *motion tracking* (dotato di due stazioni localizzabili) all'engine *Quest3D*, allo scopo di ottenere un sistema di interazione hand-based in ambienti virtuali.

La mano sinistra sarà usata per gestire lo spostamento della telecamera tramite una delle due stazioni di tracking (dotata anche di un joystick e di alcuni bottoni), mentre la mano destra, grazie al data glove e alla seconda stazione di tracking, sarà proiettata nello spazio virtuale in prossimità del punto di vista dell'utente, e permetterà di interagire con l'ambiente 3D in base al movimento e al riconoscimento di gesti o sequenze di gesti.

Particolare attenzione é stata rivolta al supporto software del guanto, creando un'utility di gestione cross-platform e riutilizzabile anche in altri contesti, ad esempio per fare da supporto a un'interfaccia gesture-driven che non coinvolga un engine 3D.

Nel prossimo capitolo sono presentati gli strumenti hardware e software uti-

lizzati, mentre nel terzo vengono descritti prima prima l'architettura generale della libreria e poi lo sviluppo di ogni singola componente. Per concludere viene mostrata, nel capitolo 4, una semplice applicazione dimostrativa che utilizza l'infrastruttura creata.

Capitolo 2

Strumenti utilizzati

Thus spake the master programmer:
"Without the wind, the grass does not move.
Without software, hardware is useless."
The Tao of Programming [33], Book 8

Presentiamo brevemente l'hardware utilizzato (guanto e dispositivo di tracking), la libreria wxWidgets (utilizzata per sviluppare ISISgloveManager) e l'engine Quest3D, approfittandone per introdurre la metodologia di svilupppo *channel-based* che lo caratterizza.

2.1 Hardware

2.1.1 5DT Data Glove 16

La 5DT *(Fifth Dimension Technologies)* è un'azienda specializzata in sistemi hardware/software per la realtà virtuale.

Il **5DT Data Glove 16** è un wired glove progettato per rilevare con accuratezza i movimenti della mano, come dimostra il suo frequente utilizzo nel motion capture.

E' realizzato in lycra, leggermente elasticizzato, e basa il suo funzionamento su dei filamenti in fibra ottica cuciti al suo interno e collegati a un'unità optoelettronica (collocata sul dorso della mano).

A un'estremità di ogni fibra c'è una piccola sorgente luminosa, mentre all'altra è collegato un fotorecettore: la quantità di impulsi luminosi dispersi permette di calcolare la flessione a cui è sottoposto il filamento di fibra ottica.

L'unità optoelettronica è a sua volta collegata a un'*interface box* posta sul polso che gestisce il protocollo di comunicazione al computer.

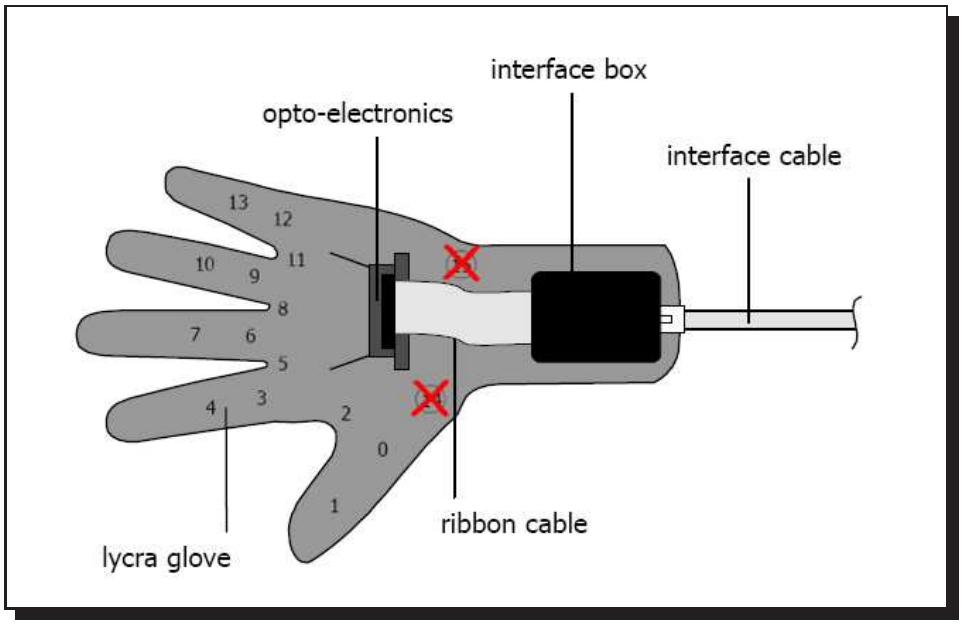


Figura 2.1: Il 5DT Data Glove 16: componenti e posizione dei sensori

Il guanto si collega al computer su porta seriale, e va precisato che (ragionevolmente) quella che avviene è una comunicazione one-way (dal guanto verso il computer). Si tratta quindi di un guanto input-only: un'altra categoria di dispositivi (dai costi molto più alti), quello delle interfacce *aptiche*, preve anche l'invio di informazioni in senso inverso, consentendo di descrivere come applicare forze, vibrazioni e movimenti alla mano dell'utilizzatore, allo scopo di ottenere con tali stimolazioni meccaniche la simulazione del contatto tattile con l'ambiente virtuale.

Il 5DT Data Glove 16 ha 14 sensori (sarebbero dovuti essere 16, come indica il nome del modello, ma due non sono mai stati implementati). Dieci sensori rilevano la flessione delle dita (due sensori per dito), mentre gli altri quattro indicano la divaricazione tra ogni coppia di dita. I due sensori mai implementati avrebbero dovuto rilevare la traslazione del pollice rispetto al palmo della mano e la flessione del polso.

Il corredo software del dispositivo consiste in un utility di gestione che consente di testare il funzionamento e la calibrazione del guanto. Inoltre, più interessantemente, il driver (per Windows e Linux) fornisce un'API per l'accesso alle funzionalità del device, sotto forma di una libreria C.

La documentazione fornita descrive sia il protocollo seriale usato, in caso

sia necessario sviluppare per piattaforme non supportate dal driver, sia le funzioni della libreria C.

2.1.2 InterSense PCTracker

La InterSense è un'azienda specializzata nell'hardware per il motion tracking di precisione.

Il **PCTracker** è un tracker *6-DOF*, ovvero a sei gradi di libertà: traccia l'orientamento (yaw, pitch, roll) e la posizione nello spazio (X, Y, Z).

Le stazioni di tracking sono basate su tecnologia ibrida, una combinazione di tracking inerziale e ultrasonico.

I dati forniti da accelerometri e giroscopi presenti in ogni stazione di tracking *"MiniTrax"* vengono combinati con le misurazioni ottenute dal sistema a ultrasuoni e sottoposti a filtraggio e correzione degli errori, allo scopo di ottenere un tracciamento pulito e coerente nel tempo.

Altri prodotti della InterSense usano un processore hardware indipendente (IS-900) per effettuare il merge e la correzione dei dati (secondo la tecnologia che la InterSense ha battezzato *"SensorFusion"*), mentre il PCTracker sfrutta l'hardware dei comuni personal computer per eseguire in software tali calcoli, attraverso la libreria dinamica standard della InterSense (*isense.dll* su piattaforma Win32) e il software *IServer*.

Il sistema a ultrasuoni della InterSense si basa sull'interazione tra PSE (Position Sensing Element) mobili e fissi. I "SoniDiscs" (emettitori di ultrasuoni) sono PSE fissi, e formano una "costellazione" che viene usata come riferimento per il tracking dei PSE mobili, ovvero gli URM (dei microfoni, Ultrasonic Receiver Module) presenti nelle stazioni *MiniTrax* da tracciare.

L'InterSense PCTracker, nella configurazione a nostra disposizione, consiste di queste componenti:

- un frame rigido (SoniFrame) al quale sono fissate tre "SoniStrips", ovvero delle aste che incapsulano i "SoniDiscs". La costellazione montata in laboratorio è la "4-2-4", una delle configurazioni stardard previste dalla InterSense (due SoniStrips laterali da 4 piedi, e una centrale da 2 piedi);

- due stazioni di tracking *MiniTrax*, di tipo "Head Tracker" e "Wand". La stazione "Head Tracker" è pensata per essere fissata sugli HMD (Head Mounted Display), mentre quella "Wand", fatta per essere tenuta in mano, è caratterizzata dalla presenza di un joystick e di 5 bottoni;

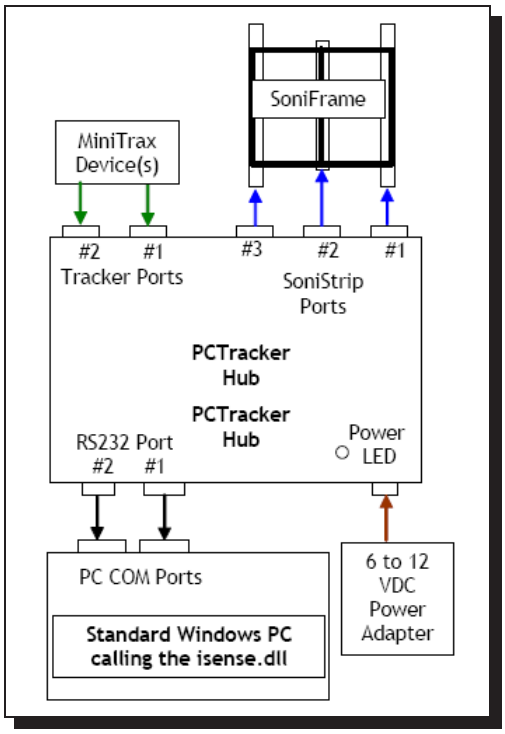


Figura 2.2: Schema del PCTracker

- il PCTracker HUB, al quale si collegano le "SoniStrips" e le stazioni *MiniTrax*. L'HUB si collega al computer tramite due interfacce seriali (requisito essenziale al corretto funzionamento del tracker è che il sistema operativo e l'hardware consentano un rate minimo di refresh di 100Hz sulle porte seriali, caratteristica - si è verificato sperimentalmente - non comunissima come si potrebbe pensare).

L'HUB permette al massimo il collegamento di due stazioni *MiniTrax* e di tre SoniStrips (per un massimo di 9 SoniDiscs, considerando i diversi tipi di SoniStrips). Esistono altri tipi di stazioni *MiniTrax* oltre la "Wand" e l'"Head Tracker", come la "Hand Tracker" (pensata per l'uso che noi abbiamo fatto dell'"Head Tracker" e che paradossalmente, essendo più ingombrante, sarebbe stata difficilmente utilizzabile insieme al 5DT Data Glove, come mostra la figura 2.3).

E' inoltre possibile utilizzare delle "SoniStrips" di diverse dimensioni, in caso ad esempio sia necessario coprire un'area più vasta, oppure montarle in configurazioni personalizzate, slegate dal "SoniFrame". In questi casi, si

Figura 2.3: a sinistra, un'immagine promozionale dell'ingombrante stazione Hand Tracker - a destra, il nostro guanto con allacciata, grazie a delle fascette in velcro, la stazione Head Tracker

ottengono delle "costellazioni" di SoniDiscs che vanno accuratamente descritte nel software di configurazione del tracker, in modo che posizione ed orientamento degli emettitori di ultrasuoni siano note e possano quindi fare da riferimento nel tracciamento delle stazioni *MiniTrax*.

L'hardware è corredato da un SDK e da programmi di esempio in Visual C++ e Visual Basic. Supportato lo sviluppo su Windows, Linux, Mac OS X e altri sistemi Unix-like.

2.2 Software

2.2.1 wxWidgets

wxWidgets è un framework per lo sviluppo di applicazioni cross-platform.

Principalmente orientato alla creazione di GUI, fornisce astrazione anche per numerose funzionalità di utilizzo comune ma tipicamente legate all'API della piattaforma su cui si sta sviluppando (sockets, multithreading, accesso al file system).

wxWidgets è implementato in C++, ma sono stati sviluppati nel corso degli anni numerosi bindings, più o meno completi, per l'utilizzo della libreria anche con altri linguaggi di programmazione (Python, Perl, Ruby, C#, Lua, Smalltalk...). Curiosa l'esistenza di un binding della libreria per Java, che affronta e risolve diversamente i problemi di portabilità (una famosa citazione del creatore del C++, Bjarne Stroustrup, recita [10]: *"Java isn't platform independent; it is a platform"*).

Grazie ai ports stabili attualmente disponibili (*wxMSW, wxGTK, wxX11,*

wxMac...) e al supporto per i compilatori C++ più comuni, si potrebbe dire che l'approccio dello sviluppo con *wxWidgets*, almeno idealmente, è quello *"write once, *compile* and run everywhere"*.

Un buon modo per capire come *wxWidgets* affronti il problema non banale dello sviluppo nativo cross-platform è descriverne brevemente l'architettura e l'organizzazione del codice.

L'architettura del framework prevede quattro layer concettuali:

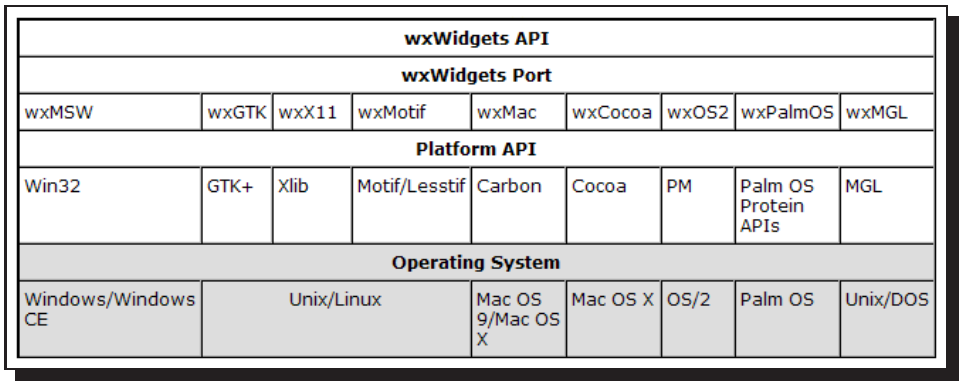


Figura 2.4: l'architettura del framework.

- La *wxWidgets API*, unica, si basa su un certo numero di ports.
- Un determinato port si basa sull'API fornita da una specifica piattaforma.
- L'API della piattaforma in oggetto si basa sulle primitive fornite dal sistema operativo.

Il codebase di *wxWidgets*, partendo dal basso, è organizzato in sei livelli:

1. **Common code**: usato in tutti i ports, include classi di strutture dati utilizzate internamente e le classi base (come `wxWindowBase`) che permettono di separare il codice comune a tutte le implementazioni di una classe.
2. **Generic code**: implementa componenti dell'interfaccia utente avanzati indipendentemente dalla piattaforma, permettendo di emulare i controlli non disponibili nativamente in alcuni casi (ad esempio, `wxCalendarCtrl` per la selezione della data).

3. **wxUniversal**: è un'implementazione di componenti base per quelle piattaforme che non hanno il proprio insieme di widgets nativi, come *X11*.

4. **Platform-specific code**: implementazioni delle classi dipendenti da un specifica piattaforma, utilizzando le funzionalità native a disposizione.

5. **Contributed code**: classi non essenziali ma utili, tenute in una gerarchia separata *"contrib"*.

6. **Third-party code**: librerie sviluppate indipendentemente da *wxWidgets*, ma usate per implementare caratteristiche importanti (gestione di immagini JPEG e PNG, compressione Zlib).

Quando si include un header di *wxWidgets*, ad esempio "`wx/textctrl.h`", si include in effetti un file specifico per la piattaforma su cui si sta compilando (ad esempio "`wx/msw/textctrl.h`" sotto win32), poichè "`wx/textctrl.h`" contiene, oltre a una parte comune a tutti i ports, una serie di direttive al preprocessore per l'inclusione condizionale di un file platform-dependant:

```
#if defined(__WXX11__)
    #include "wx/x11/textctrl.h"
#elif defined(__WXUNIVERSAL__)
    #include "wx/univ/textctrl.h"
#elif defined(__SMARTPHONE__) && defined(__WXWINCE__)
    #include "wx/msw/wince/textctrlce.h"
#elif defined(__WXMSW__)
    #include "wx/msw/textctrl.h"
#elif defined(__WXMOTIF__)
    #include "wx/motif/textctrl.h"
#elif defined(__WXGTK20__)
    #include "wx/gtk/textctrl.h"
#elif defined(__WXGTK__)
    #include "wx/gtk1/textctrl.h"
#elif defined(__WXMAC__)
    #include "wx/mac/textctrl.h"
#elif defined(__WXCOCOA__)
    #include "wx/cocoa/textctrl.h"
#elif defined(__WXPM__)
    #include "wx/os2/textctrl.h"
#endif
```

In pratica, *wxWidgets* fa da *wrapper* attorno a ogni API nativa e, quando possibile, si limita ad introdurre un sottile layer d'astrazione attorno a un *widget* della piattaforma in uso. Quando ciò non è possibile, la libreria si preoccupa di implementare/emulare i controlli e le funzionalità non presenti su una data piattaforma.

L'obiettivo è quindi quello di conservare il "native look-and-feel" e l'efficienza dell'uso dei controlli nativi, senza però dover rinunciare ai componenti e alle funzionalità presenti solo su alcune piattaforme.

Il caso limite di questo approccio è quello del supporto alle piattaforme che non prevedono affatto dei componenti nativi per l'interfaccia utente: il port *wxX11* utilizza i widgets implementati nel layer 3 dell'architettura (*wxUniversal*) e si basa direttamente su `Xlib` per "disegnare" le componenti.

wxWidgets ha il valore aggiunto di essere un progetto open source e di adottare una politica di licensing decisamente vantaggiosa per gli sviluppatori: la libreria si può utilizzare tranquillamente, senza dover sostenere costi, anche in applicazioni commerciali e anche utilizzando il linking statico ai propri eseguibili. L'unico obbligo verso la comunità *wxWidgets* è, più che giustamente, quello di rendere disponibili eventuali cambiamenti/miglioramenti che si apportano al codice sorgente della libreria stessa, in modo che tutti possano beneficiarne.

Il framework si può tranquillamente definire attivamente in sviluppo (la versione 2.8.4 è stata rilasciata il 18/05/2007) e maturo, in quanto la pubblicazione della versione 1.0 di *wxWidgets* risale al settembre 1992. Una nota a sfavore dell'anzianità del progetto è l'utilizzo di tipi di dati custom che, a detta degli autori, al tempo della progettazione iniziale del framework, non avevano un'implementazione cross-platform affidabile basata sulla STL del C++. Ci si trova quindi a lavorare con tipi di dati e contenitori non standard (`wxString`, `wxArray`, `wxList`...) e a dover conseguentemente ricorrere, in alcuni casi, a *cast* e conversioni di tipo poco eleganti. E' però ragionevole auspicare, anche sbirciando la road-map degli sviluppatori di *wxWidgets*, che ci sarà un graduale passaggio all'utilizzo della STL ovunque possibile. Altro segno di vecchiaia (più che di maturità) della libreria, è il mancato utilizzo delle eccezioni: nulla impedisce di utilizzarle nel proprio codice, ma la libreria stessa non ne solleva in alcun caso, e regna il controllo degli errori *C-style* basato sul valore restituito dalle funzioni.

2.2.2 Quest3D

Quest3D è un software per la creazione rapida di scene 3D interattive. E' stato progettato pensando principalmente allo sviluppo di presentazioni 3D, visualizzazioni per l'architettura, applicazioni videoludiche e di realtà virtuale.

Oltre a incapsulare le funzionalità offerte dalle *DirectX*, su cui l'engine si basa (vincolandosi purtroppo alla piattaforma Win32), Quest3D offre una serie di facilities di comune utilizzo nei contesti applicativi a cui si rivolge (simulazione della fisica con ODE, accesso ai database e alla rete, supporto per alcuni dispositivi per la realtà virtuale).

Il software cerca di rendere immediato ed intuitivo il processo di creazione di una scena consentendo l'editing 3D real-time: il progetto attivo viene continuamente renderizzato all'interno dell'editor, che permette di cambiare *on-the-fly* le proprietà e il comportamento delle componenti in scena.

Pur essendo pensato per essere utilizzato con profitto anche dai non programmatori, grazie alla metodologia di sviluppo visuale dei *channels*, Quest3D può essere esteso secondo le proprie necessità implementando in C++ i propri componenti oppure, più limitatamente, usando lo scripting LUA per definire dei channels con logica personalizzata all'interno del software stesso. Entrambe le tecniche saranno utilizzate nello sviluppo della nostra infrastruttura.

Spieghiamo brevemente in cosa consiste lo sviluppare con i *channels*.

Lo sviluppo channel based

In Quest3D l'assemblaggio della scena e l'implementazione della logica applicativa vengono effettuate tramite un particolare sistema di progettazione visuale component-based, quello dei channels.

Un channel è una componente modulare, rappresentata nell'editor come un blocco rettangolare, capace di effettuare un'azione e/o ospitare dei dati.

I channels vengono messi in comunicazione tra loro grazie ai **link squares**, dei quadratini neri che permettono di stabilire dei collegamenti tra diversi channels. Ogni channel, nell'editor, ha un link square sul bordo superiore (che permette di collegare il channel stesso ad altri) e da zero a un numero arbitrario di link squares sul bordo inferiore (per collegare dei "figli" al channel).

Un collegamento tra channels, che stabilisce sempre una relazione padre-figlio, si effettua quindi unendo un "top" link square del figlio a un "bottom" link square del padre.

Da puntualizzare il fatto che è consentito che il "top link square" di un channel si possa collegare a un numero arbitrario di altri channels: è normale (e accade di frequente) che un channel abbia "più padri".

Relativamente al collegamento di channels, va descritto il loro meccanismo di tipizzazione.

Ogni channel è caratterizzato da un *nome*, un *Type* e un *BaseType*.

Il *BaseType* determina come un channel può essere utilizzato: ad esempio un channel `ExpressionValue`, pur essendo di *Type* `ExpressionValue`, ha *BaseType* `Value`, e può essere quindi utilizzato ovunque può essere usato un channel di tipo `Value` (ragionevolmente, il valore è appunto il risultato dell'espressione). Se quindi il *Type* è univoco per ogni channel, più channels hanno lo stesso *BaseType*.

Il meccanismo di tipizzazione gerarchica dei channels permette di determinare dei vincoli sul modo in cui i channels si possono collegare: i link squares per i figli possono infatti essere vincolati a un certo *BaseType*, evitando gli errori di collegamento più banali a livello di editor (sarà impossibile collegare un channel di tipo `Text` a un channel che si aspetta come figlio un `Vector`).

Descritto il modo in cui si possono collegare i channels, è essenziale capire quale sia il flusso dell'applicazione che si ottiene.

Un unico channel del progetto si può impostare come "Start Channel", ed è da lì che inizia l'esecuzione dell'applicazione. Tipicamente, ogni channel "chiama" dei channels suoi figli, ricorsivamente, e questa visita del grafo di channels avviene una volta per fotogramma.

Chiamare un channel significa far sì che sia eseguita la sua funzione `CallChannel`, che normalmente definisce la logica del channel. Ovviamente l'interazione tra padri e figli non si limita a `CallChannel`, essendo necessario un modo per scambiare informazioni: un channel padre può prelevare/assegnare valori dai/ai figli.

Non tutti i figli devono per forza essere chiamati, questo dipende dalla logica definita nel channel padre: nell'esempio in figura 2.5 è presente un `If/Else`.

Il channel `If/Else`, chiamato da `Project Start`, ha tre link square inferiori: al primo si deve collegare un valore vero/falso, agli altri due dei channels qualsiasi. Se il valore del primo figlio è vero, viene chiamato il secondo figlio, se è falso viene chiamato il terzo. In figura, secondo e terzo figlio sono due `CallChannel`, ovvero un tipo predefinito di channel che, banalmente, chiama tutti i figli che vi si collegano.

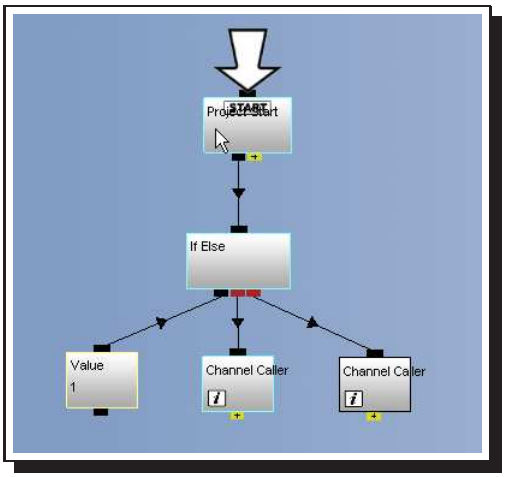


Figura 2.5: un semplice if/else implementato con i channels

Entrando più nel dettaglio, è interessante notare l'implicazione pratica del fatto che, avendo channels con più nodi padre, non si ha a che fare con la visita di un albero ma con la visita di un grafo, anche se l'engine cerca di nascondere le complicazioni a ciò correlate. Infatti, i channels che hanno piu' "padri" collegati vengono "chiamati" più volte, ma normalmente si vuole che un channel sia eseguito (e quindi che calcoli il suo valore) solo una volta per frame. E' per questo che l'engine mantiene un "tree count": se "visitando" un channel il tree count è lo stesso rispetto a quello osservato nella visita precedente di quel channel, vuol dire che non si tratta della prima visita di quel channel relativamente al frame che si sta renderizzando, e quindi l'engine salta l'esecuzione del channel e si limita a restituirne il valore attuale. In casi particolari si potrebbe invece volere che il channel sia calcolato più volte per frame, e quindi questo "tree count check" si puo' disabilitare. Ed ecco che col "tree count check" disabilitato, se c'è dipendenza ciclica tra i channels, si può avviare un loop infinito che porta inevitabilmente al crash dell'applicazione, come avverte il manuale di Quest3D.

Capitolo 3

Realizzazione dell'infrastruttura

Make everything as simple as possible,
but not simpler.
Albert Einstein

L'infrastruttura realizzata è nata partendo dal supporto nativo di *Quest3D* per l'hardware posseduto, e si è sviluppata man mano con la realizzazione di componenti software aggiuntive, per sopperire alle mancanze di quanto disponibile e per ottenere funzionalità avanzate.

Le componenti create sono state pensate per essere potenzialmente riutilizzate in altri contesti, anche indipendentemente l'una dall'altra. Lo stesso "SkeleTronDemo", descritto in seguito e che ha l'obiettivo di presentare la libreria nel suo insieme, è stato sviluppato con un occhio di riguardo alla modularità, in modo che, con i limiti dell'editor di *Quest3D*, possa fare da esempio a livello globale oppure se ne possano scorporare/modificare soltanto alcune parti.

La libreria si basa principalmente su queste componenti:

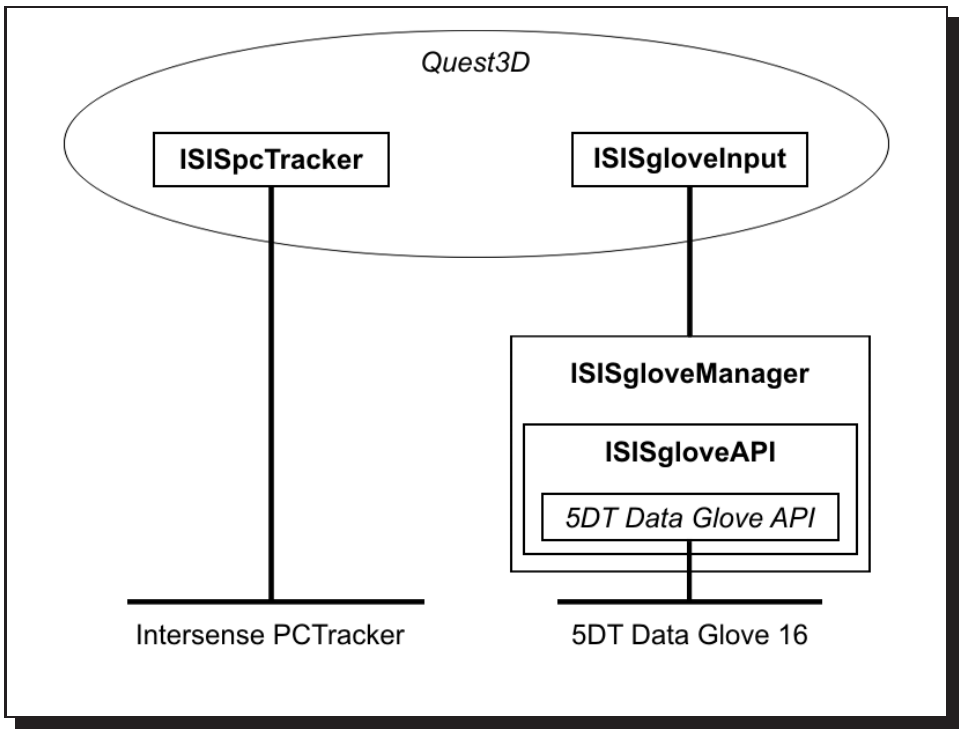
- ISISgloveAPI
- ISISgloveManager
- ISISgloveManagerInput
- ISISpcTracker

ISISgloveAPI è una libreria C++ basata sull'API del driver del guanto, che aggiunge funzionalità orientate al riconoscimento dei gesti. La libreria è pensata per essere usata in sostituzione dell'API del driver della 5DT.

ISISgloveManager è un'applicazione C++ cross-platform (testata su Windows e Linux) che, basata su ISISgloveAPI e sul framework *wxWidgets*, fa da utility di gestione del guanto. Permette la calibrazione e il salvataggio/riconoscimento dei gesti, e implementa un server TCP che provvede a inviare in broadcast ai clients i valori letti dal dispositivo e le informazioni sui gesti osservati. Obiettivo di **ISISgloveManager** è quello di implementare una volta per tutte la gestione di calibrazione e *gesture-recognition* e fornire quindi un accesso ad alto livello, tramite socket, alle funzionalità del guanto, semplificando lo sviluppo di qualsiasi tipo di applicazione *glove-based*.

ISISgloveManagerInput è un template di *Quest3D* che va a sostituire il supporto nativo al 5DT Data Glove. Aggiunto al progetto *Quest3D* in sviluppo, grazie allo scripting *Lua* e al supporto dei sockets dell'engine, il template fa da componente client per ISISgloveManager e, oltre a fornire i valori letti dai sensori, preleva la stringa indicante il riconoscimento di un gesto. Questa viene usata come input in una struttura, semplicemente espandibile, che consente di utilizzare i gesti come trigger che azionano logica *Quest3D* ad-hoc.

ISISpcTracker è un template contenente un "custom channel" per *Quest3D*, ovvero una componente sviluppata in C++ per integrarsi nel modello di sviluppo dell'engine. Mentre il supporto nativo di *Quest3D* ai dispositivi di tracking *InterSense* prevede il collegamento di un'unica stazione *MiniTrax*, e permette di prelevare unicamente i suoi vettori posizione e orientamento, il nostro channel `IntersensePCtracker` è pensato per l'utilizzo di entrambe le stazioni *MiniTrax* a disposizione e, oltre ai vettori posizione/orientamento di entrambe, permette l'utilizzo del joystick e dei bottoni presenti sulla stazione di tipo *Wand*.


Quest3D
ISISpcTracker
ISISgloveInput
ISISgloveManager
ISISgloveAPI
5DT Data Glove API
Intersense PCTracker
5DT Data Glove 16

3.1 ISISgloveAPI

Iniziamo col descrivere la libreria C fornita con l'hardware, per poi spiegare i criteri e le tecniche con cui questa è stata incorporata ed ampliata sviluppando ISISgloveAPI.

3.1.1 5DT Data Glove: l'SDK fornita

A corredo del 5DT Data Glove è fornita una libreria C, per *Windows* e *Linux*, che consente l'interfacciamento col dispositivo.

Piuttosto tipicamente, il kit consiste in un header file (`fglove.h`) che dichiara le funzioni implementate in una libreria compilata per una specifica piattaforma, e nel nostro caso:

- su Win32: una dll (`fglove.dll`) e una libreria Microsoft Visual C++ (`fglove.lib`)
- su Linux: una shared library (`fglove.so`)

Per utilizzare l'API è quindi necessario, in sintesi

- installare sul sistema la dll o la shared library
- nella propria applicazione, includere l'header file e aggiungere al processo di linking la libreria

Informazioni dettagliate in merito sono presenti sul "Reference Manual" del driver 5DT che, va precisato, è unico per tutti i modelli di Data Glove dell'azienda. Pur non disponendo di altri dispositivi oltre al 5DT Data Glove 16, si è cercato durante lo sviluppo, prima di **ISISgloveAPI** e poi di **ISISgloveManager**, di non perdere tale generalità.

Le funzioni messe a disposizioni dall'API si possono cosi' raggruppare:

- accesso al dispositivo e alla descrizione delle sue caratteristiche
- accesso al valore dei sensori (in forma `raw` o `scaled`)
- gestione della calibrazione e dello scaling dei valori
- rilevamento *gestures*

Nel file `fglove.h` sono inoltre definite alcune enumerazioni di costanti che è possibile usare con le funzioni, di cui va citata almeno `EfdSensors`, utile a individuare non numericamente i singoli sensori:

```
enum EfdSensors {
    FD_THUMBNEAR=0, FD_THUMBFAR,    FD_THUMBINDEX,
    FD_INDEXNEAR,   FD_INDEXFAR,    FD_INDEXMIDDLE,
    FD_MIDDLENEAR,  FD_MIDDLEFAR,   FD_MIDDLERING,
    FD_RINGNEAR,    FD_RINGFAR,     FD_RINGLITTLE,
    FD_LITTLENEAR,  FD_LITTLEFAR,   FD_THUMBPALM,
    FD_WRISTBEND,
    FD_PITCH,
    FD_ROLL
};
```

Analizziamo brevemente la libreria e l'utilizzo che ne va fatto.

Accesso al dispositivo

Questo gruppo di funzioni, non particolarmente interessanti, provvede all'ottenimento/rilasciamento del dispositivo e al prelievo di informazioni sulle sue caratteristiche.

- `fdGlove *fdOpen(char *pPort);`
  Indicata la porta seriale a cui è connesso il guanto, restituisce un puntatore alla struttura `fdGlove` che lo rappresenta.

- `int fdClose(fdGlove *pFG);`
  Rilascia la risorsa guanto.

- `int fdGetGloveHand(fdGlove *pFG);`
  Restituisce `FD_HAND_LEFT` o `FD_HAND_RIGHT` a seconda di se si tratti di un guanto per la mano sinistra o per la mano destra.

- `int fdGetGloveType(fdGlove *pFG);`
  Restituisce `FD_GLOVENONE` se non è connesso alcun dispositivo, o un'altra costante che ne identifica il modello (`FD_GLOVE7`, `FD_GLOVE7W`, `FD_GLOVE16`, `FD_GLOVE16W`).

- `int fdGetNumSensors(fdGlove *pFG);`
  Il numero di valori dei sensori che il driver gestisce/rende disponibili.

- `void fdGetGloveInfo(fdGlove *pFG, unsigned char *pData);`
  Ottiene l'"information data block" del guanto connesso (32 bytes)

- `void fdGetDriverInfo(fdGlove *pFG, unsigned char *pData);`
  Ottiene l'"information data block" del driver (32 bytes, NULL terminated string)

Unica nota da fare è riguardo la funzione `fdGetNumSensors`, che non restituisce il numero di sensori effettivamente presenti in hardware, come ci si potrebbe aspettare: risultato della chiamata è il numero di valori gestiti dal driver, che nelle implementazioni attuali, tra l'altro, è 18 indipendentemente dal tipo di dispositivo. In ogni caso, come vedremo in seguito analizzando **ISISgloveAPI**, non si è fatto affidamento su questa particolarità e si è allocata dinamicamente la memoria in funzione di quanto restituito da `fdGetNumSensors`, prevedendo di dover gestire un numero diverso (e non definito a priori) di valori.

Accesso al valore raw dei sensori

Il valore "raw" di un sensore è un intero senza segno di 12 bit (ovvero un numero compreso tra 0 e 4095), e l'API memorizza tale tipo di dato in un `unsigned short`.

Si può prelevare un singolo valore con la funzione

- `unsigned short fdGetSensorRaw(fdGlove *pFG, int nSensor);`

Per ottenere, ad esempio, il valore di flessione dell'indice al primo punto di flessione (ovvero alla nocca), dopo aver ottenuto un handle al dispositivo (glove), useremo

```
unsigned short indexKnuckleFlex = fdGetSensorRaw(glove, FD_INDEXNEAR);
```

Per prelevare in blocco tutti i valori, va invece usata la funzione

- `void fdGetSensorRawAll(fdGlove *pFG, unsigned short *pData);`

dove `pData` deve essere un array di `unsigned short` di dimensione adeguata.

Ecco quindi il codice C da usare per ottenere correttamente l'array di valori raw dal driver:

```
unsigned short *sensorValues;
sensorValues =
  (unsigned short *) malloc(fdGetNumSensors(glove)*sizeof(unsigned short));
fdGetSensorRawAll(glove, sensorValues);
```

Molte funzioni dell'API hanno, come in questo caso, una duplice versione per l'utilizzo riferito a un singolo sensore o a tutti in blocco.

Per completezza in merito alla manipolazione dei valori raw, vanno citate le funzioni che permettono di "forzare" dei valori nel buffer dei valori grezzi gestito dal driver

- `void fdSetSensorRawAll(fdGlove *pFG, unsigned short *pData);`
- `void fdSetSensorRaw(fdGlove *pFG, int nSensor, unsigned short nRaw);`

Lo scopo di queste funzioni è alterare, a livello di driver e non di applicazione utente, il valore dei sensori non presenti in hardware, che di default sarebbe zero. Forzare il valore di un sensore che è invece presente non ha senso, in quanto verrebbe immediatamente sovrascritto dal blocco successivo di dati forniti dal dispositivo.

Gestione della calibrazione e dello scaling dei valori

Definiamo come *Dynamic Range* di un sensore la differenza del valore *raw* del sensore con mano completamente chiusa e completamente aperta (nel caso dei sensori di flessione) e di mano con le dita unite e con le dita divaricate (nel caso dei sensori tra le coppie di dita):

$$DynamicRange = RawMax - RawMin$$

Con mani diverse si hanno `DynamicRange` diversi, ed è qui che entra in gioco la calibrazione software del dispositivo, che prevvede a normalizzare i valori `raw` in funzione del `DynamicRange` rilevato.

Una calibrazione del guanto consiste infatti nella definizione dell'intervallo di valori *raw* rilevato (con una certa mano) per ognuno dei sensori, ovvero in due array contenenti i valori *RawMin* e *RawMax*.

Se ad esempio si desidera scalare i dati *raw* entro l'intervallo 0–Max, un valore *scaled* sarà così calcolato:

$$ScaledVal = \left(\frac{RawVal - RawMin}{RawMax - RawMin} \right) \cdot Max$$

Come vedremo a breve in dettaglio, i valori scalati sono quindi dei `float`, e di default `Max` vale 1.

Il driver implementa una routine di autocalibrazione *dinamica*, nel senso che, per ogni sensore, ogni valore letto viene confrontato con gli estremi

RawMin/*RawMax* del *DynamicRange* rilevato fino a quel momento, ed eventualmente sostituisce uno dei due valori.

Ecco perchè in pratica, per calibrare il guanto, è sufficiente:

- aprire/chiudere la mano a pugno, per impostare *RawMin*/*RawMax* dei sensori di flessione
- unire/divaricare le dita, per impostare il *RawMin*/*RawMax* dei sensori di divaricazione

Per ottenere una buona calibrazione è bene non forzare i movimenti, in modo da non estendere troppo il *DynamicRange*, cosa che comporterebbe avere una minore risoluzione quando si effettuano movimenti "normali".

Le funzioni dell'API per la gestione della calibrazione permettono di assegnare e prelevare tramite codice i valori *RawMin*/*RawMax*. Anch'esse sono presenti in coppia, per agire su un singolo sensore o su tutti.

- ```
  void fdGetCalibrationAll(fdGlove *pFG,
                           unsigned short *pUpper,
                           unsigned short *pLower);
  ```

  I due array passati, precedentemente allocati, come nell'esempio relativo a `fdGetSensorRawAll`, vengono riempiti con tutti i valori *RawMax* (`pUpper`) e *RawMin* (`pLower`) considerati dal driver al momento della chiamata.

- ```
  void fdGetCalibration(fdGlove *pFG,
                                int nSensor,
                        unsigned short *pUpper,
                        unsigned short *pLower);
  ```

  Gli `unsigned short` agli indirizzi passati in `pUpper` e `pLower` conterranno rispettivamente, dopo la chiamata, i valori *RawMax* e *RawMin* relativi al sensore `nSensor`.

- ```
  void fdSetCalibrationAll(fdGlove *pFG,
                           unsigned short *pUpper,
                           unsigned short *pLower);
  ```

  Imposta la calibrazione del guanto, assegnando come valori *RawMax*/*RawMin* utilizzati dalla routine di scaling del driver quelli passati attraverso gli array `pUpper` e `pLower`, precedentemente allocati e riempiti dei valori che si desidera assegnare.
  ```

- `void fdSetCalibration(fdGlove *pFG,`
  `int nSensor,`
  `unsigned short nUpper,`
  `unsigned short nLower);`

  Imposta gli estremi per il singolo sensore `nSensor`, assegnando `nUpper` come *RawMax* e `nLower` come *RawMin*.

E' inoltre presente un'ulteriore funzione per annullare la calibrazione corrente del guanto e far sì quindi che si inizino a calcolare nuovi *RawMax*/*RawMin*:

- `void fdResetCalibration(fdGlove *pFG);`

  Equivale a chiamare `fdSetCalibrationAll` con tutti i valori dell'array `pUpper` uguali a `0` e tutti i valori dell'array `pLower` uguali a `4095`.

Volendo, si può ignorare del tutto l'autocalibrazione dinamica del guanto, e basare una propria routine di scaling sui valori *raw*. Ad esempio si potrebbe voler evitare che, caricata una calibrazione, questa possa essere variata a runtime se vengono rilevati nuovi *RawMax*/*RawMin*.

Tralasciando questi casi particolari, è ragionevole sfruttare lo scaling e l'autocalibrazione del guanto, prevedendo però il salvataggio e il caricamento delle calibrazioni. **ISISgloveManager** permetterà all'utente di effettuare una calibrazione accurata e salvarla, per poi caricarla agli utilizzi successivi evitando il comportamento anomalo del guanto non calibrato, all'inizio delle sessioni.

Accesso al valore scaled dei sensori

I valori scaled non sono che i valori *raw* normalizzati in funzione dell'autocalibrazione effettuata dal guanto, come appena spiegato.

Le funzioni per ottenere i valori scalati sono simili a quelle da utilizzare per prelevare i valori *raw*: l'unica differenza è nel tipo, `float` invece che `unsigned short`.

- `void fdGetSensorScaledAll(fdGlove *pFG, float *pData);`
- `float fdGetSensorScaled(fdGlove *pFG, int nSensor);`

Di default il range dei valori scalati è [0..1], ma volendo si può cambiare l'estremo superiore con la coppia di funzioni

- `void fdSetSensorMaxAll(fdGlove *pFG, float *pMax);`

- `void fdSetSensorMax(fdGlove *pFG, int nSensor, float fMax);`

e si può verificare l'impostazione corrente con le simmetriche

- `void  fdGetSensorMaxAll(fdGlove *pFG, float *pMax);`
- `float fdGetSensorMax(fdGlove *pFG, int nSensor);`

Rilevamento gestures

L'API offre inoltre un rudimentale sistema di *gesture recognition*: le dita della mano, escluso il pollice, sono viste come una sequenza di quattro cifre binarie: dito piegato 0, dito disteso 1. L'indice è la cifra meno significativa.

- `int fdGetNumGestures(fdGlove *pFG);`

  Restituisce il numero di gesti che il driver può riconoscere: nell'implementazione attuale, è 16.

- `int fdGetGesture(fdGlove *pFG);`

  Restituisce l'id (0-15) del gesto riconosciuto, o -1 se non risconosciuto alcun gesto.

Le dita vengono considerate piegate o meno in base a un certo valore di tolleranza, che si può impostare a piacimento, sensore per sensore, con queste funzioni:

- ```
  void fdGetThresholdAll(fdGlove *pFG,
                         float *pUpper,
                         float *pLower);
  ```

- ```
  void fdGetThreshold(fdGlove *pFG,
                      int nSensor,
                    float *pUpper,
                    float *pLower);
  ```

- ```
  void fdSetThresholdAll(fdGlove *pFG,
                         float *pUpper,
                         float *pLower);
  ```

- ```
  void fdSetThreshold(fdGlove *pFG,
                      int nSensor,
                    float fUpper,
                    float fLower);
  ```

Questo basilare sistema di gesture recognition era stato probabilmente implementato pensando al **Data Glove 7**, modello dotato di un unico sensore di flessione per dito, e anche in quel caso non si capisce perchè gli sviluppatori della 5DT abbiano deciso di non considerare il pollice come ulteriore bit, restringendo la funzionalità a 16 (2^4) combinazioni e non 32 (2^5) come sarebbe stato più naturale.

Basarsi su queste funzioni per effettuare il riconoscimento dei gesti significherebbe sotto-utilizzare in maniera folle l'hardware a nostra disposizione, che con 14 sensori (a 12 bit di risoluzione massima) permette di fare molto di meglio, quindi ignoreremo quest'ultimo gruppo di funzioni nello sviluppo di **ISISgloveAPI**.

3.1.2 Obiettivi di ISISgloveAPI

ISISgloveAPI si propone di incapsulare l'API C fornita in una libreria C++, strutturata secondo i principi dell'OOP, che

- sia più immediata da usare:
- supporti il riconoscimento dei gesti sfruttando appieno le potenzialità dell'hardware (differentemente dal grezzo sistema di gesture recognition "rimosso" dall'API)
- fornisca un minimale supporto alla persistenza dei dati

Facilità d'uso

Si è scelto di "nascondere" le funzionalità ritenute di dubbia utilità allo scopo di ottenere un'API piu' snella, oltre che meglio organizzata grazie all'OOP. Queste le funzionalità "bocciate":

- la possibilità di forzare i valori raw dei sensori non presenti in hardware.
- l'impostazione del valore di massimo dei valori scaled, che sarà quindi sempre 1.0.
- il *gesture recognition* "primitivo" e la relativa impostazione della tolleranza.

Riconoscimento dei gesti

La classe `ScaledHandPosition` è stata implementata fornendo operatori di confronto che permettono l'ordinamento dei gesti (e quindi la ricerca efficiente) e il controllo di uguaglianza in base a una tolleranza all'errore definibile a piacimento.

Supporto alla persistenza

Le classi `GloveCalibration`, `ScaledHandPosition` e `RawHandPosition` sono dotate di operatori di I/O che, sfruttando l'astrazione degli *streams*, rendono immediato il save/load dei dati (definito nella classe astratta *SaveLoad*, da cui derivano) e la trasmissione su socket in un semplice formato text based.

3.1.3 Descrizione generale e cenni sull'implementazione

ISISgloveAPI si compone di queste classi, organizzate come in figura 3.1:

- Glove
- SaveLoad
- GloveCalibration
- ScaledHandPosition
- RawHandPosition

Analizziamole una per una spiegando man mano le scelte di sviluppo e fornendo degli esempi che mostrino l'utilizzo della libreria.

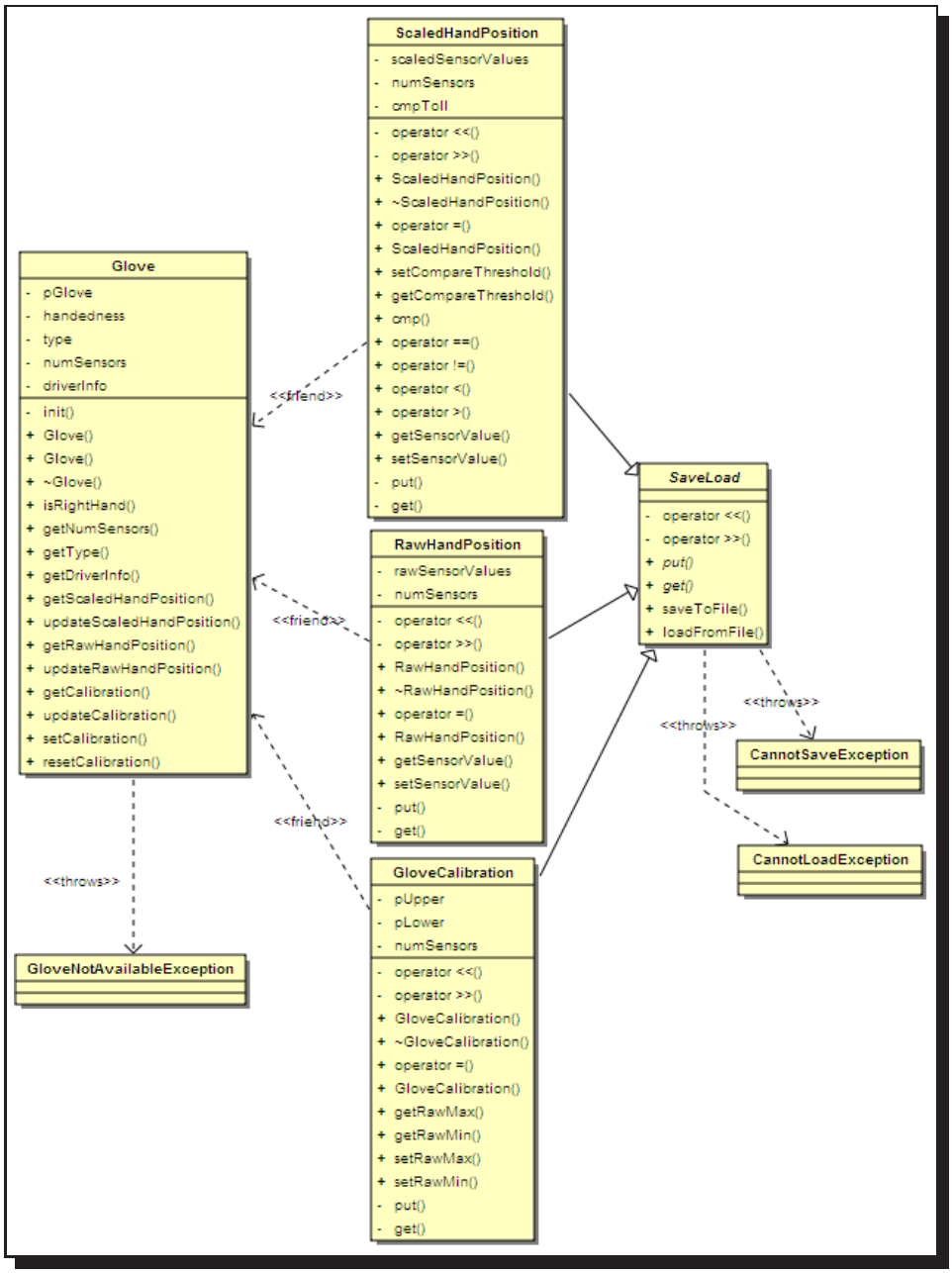


Figura 3.1: ISISgloveAPI: class diagram

La classe Glove

Glove è la classe principale, che permette di interagire con la risorsa "guanto": il costruttore stesso della classe tenta di ottenere un *handle* al dispositivo e, in caso di problemi, solleva una `GloveNotAvailableException`.

Si noti che sarebbe stato sbagliato utilizzare il design pattern del *singleton* per la classe `Glove`: sebbene apparisse sensato nel nostro caso specifico, avrebbe precluso la possibilità di utilizzare più di un guanto, mentre con l'implementazione scelta sarebbe sufficiente istanziare diversi oggetti `Glove`:

```
Glove *destro = new Glove("COM1");
Glove *sinistro = new Glove("COM2");
```

Non specificando la porta, il costruttore tenta di accedere al dispositivo con uno scan da COM1 a COM8.

Un oggetto `Glove` offre tutti i metodi necessari a interagire col guanto, mentre le altre classi della libreria sono in effetti dei tipi di dati personalizzati, che incapsulano

- una lettura di tutti i valori raw (`RawHandPosition`)
- una lettura di tutti i valori scaled (`ScaledHandPosition`)
- una calibrazione del guanto (`GloveCalibration`)

Un discorso a parte vale per `SaveLoad`, classe astratta da cui derivano le tre appena citate, che implementa le funzionalità di salvataggio e caricamento.

Sebbene siano abbastanza ovvi, sopratutto dopo aver discusso l'API 5DT che viene incapsulata, esaminiamo rapidamente i metodi della classe `Glove`, raggruppandoli similmente a quanto fatto in precedenza.

Recupero di informazioni sul dispositivo

- `bool isRightHand();`

  Permette di rilevare se si tratta di un guanto destro o sinistro.

- `int getNumSensors();`

  Fornisce il numero di sensori gestiti dal driver, cruciale nel dimensionare gli array contenuti nelle altre strutture dati. Corrispettiva di `fdGetNumSensors(fdGlove *pFG)`.

- `char *getType();`

  Mentre `fdGetGloveType(fdGlove *pFG)` restituisce una costante per individuare il modello, questo metodo restituisce una stringa, più immediatamente utilizzabile in output. Le stringhe associate alle costanti sono `None/Glove7/Glove7W/Glove16/Glove16W`.

- `unsigned char *getDriverInfo();`

  Banale wrapper di `fdGetDriverInfo(fdGlove *pFG, unsigned char *pData)`.

Accesso ai valori raw e scaled

- `RawHandPosition *getRawHandPosition();`
- `ScaledHandPosition *getScaledHandPosition();`

  Allocano, rispettivamente, un oggetto `RawHandPosition` o `ScaledHandPosition` contenente una lettura dei valori *raw* o *scaled* dei sensori, e restituiscono l'indirizzo dell'oggetto creato.

- `void updateRawHandPosition(RawHandPosition *hp);`
- `void updateScaledHandPosition(ScaledHandPosition *hp);`

  Aggiornano, rispettivamente, l'oggetto `RawHandPosition/ScaledHandPosition` puntato da `hp`, precedentemente allocato, con una lettura dei sensori *raw/scaled*.

Gestione della calibrazione

- `GloveCalibration *getCalibration();`

  Preleva la calibrazione dal guanto: alloca un oggetto `GloveCalibration` e lo inizializza con gli attuali valori $RawMin/RawMax$ rilevati dal dispositivo.

- `void updateCalibration(GloveCalibration *gc);`

  Aggiorna un oggetto `GloveCalibration` precedentemente allocato con gli attuali valori $RawMin/RawMax$ rilevati dal dispositivo.

- `void setCalibration(GloveCalibration *gc);`

  Assegna al guanto la calibrazione passata come parametro.

- `void resetCalibration();`

  Annulla la calibrazione attuale del guanto, permettendo la ricalibrazione.

La classe astratta SaveLoad

`SaveLoad` è una classe astratta che sfrutta l'overloading degli operatori di I/O, effettuato nelle classi "dato" sue eredi e di cui si parlerà estensivamente in seguito, per implementare, attraverso i metodi `saveToFile` e `loadFromFile`, un generico meccanismo di save/load. La classe è sufficientemente breve ed interessante da meritare una visione d'insieme del codice che la implementa:

```
class CannotLoadException : public std::exception {};
class CannotSaveException : public std::exception {};

class SaveLoad {
    friend ostream& operator<<(ostream& s, const SaveLoad& sl) {
        return sl.put(s); //uses the right put()
    };

    friend istream& operator>>(istream& s, SaveLoad& sl) {
        return sl.get(s); //uses the right get()
    }

    public:
        virtual ostream& put(ostream& s) const = 0; //write *this to s
        virtual istream& get(istream& s) = 0; //read *this from s

        void saveToFile(std::string filename);
        void loadFromFile(std::string filename);
};

void SaveLoad::saveToFile(std::string filename) {
    std::ofstream out(filename.c_str());
    if (!out.is_open()) throw CannotSaveException();
    out << *this;
    if (!out.good()) throw CannotSaveException();
    out.close();
}
```

```
void SaveLoad::loadFromFile(std::string filename) {
    std::ifstream in(filename.c_str());
    if (!in.is_open()) throw CannotLoadException();
    in >> *this;
    if (!in.good()) throw CannotLoadException();
    in.close();
}
```

Il C++ è stato progettato tenendo sempre d'occhio l'efficienza e quindi, essendo le operazioni di input/output tipiche candidate a creare colli di bottiglia nelle applicazioni, gli operatori di I/O non sono definiti come `virtual`, in modo da avere il *binding statico* (e potenzialmente l'*inlining*) delle *operator functions* che li implementano. Questa scelta implica un piccolo artificio in una situazione come la nostra: gli operatori di I/O di `SaveLoad` non possono essi stessi essere `virtual`, ma possono incorporare delle funzioni private della classe, `get` e `put`, che in `SaveLoad` sono delle *pure virtual functions*, ovvero dei metodi astratti che andranno per forza di cose implementati nelle classi discendenti da `SaveLoad`. In tutte e tre le classi, `get` e `put` saranno delle semplicissime funzioni private che provvederanno a richiamare l'operatore di I/O corretto.

Per rendere chiaro il meccanismo è doveroso un esempio: consideriamo di avere un'istanza della classe `ScaledHandPosition` di cui chiamiamo il metodo `saveToFile`, ereditato da `SaveLoad`:

```
void SaveLoad::saveToFile(std::string filename) {
    std::ofstream out(filename.c_str());
    if (!out.is_open()) throw CannotSaveException();
    out << *this;
    if (!out.good()) throw CannotSaveException();
    out.close();
}
```

Tralasciando l'accesso al file e il controllo delle condizioni di errore, la riga di intetesse è naturalmente

```
out << *this;
```

Se `operator<<` fosse virtual, non sarebbe necessario fare nient'altro per ottenere il comportamento polimorfico che desideriamo: verrebbe usato l'operatore << definito in `ScaledHandPosition`. Ma come abbiamo premesso,

questo non è possibile, quindi quello che viene usato è l'operatore << definito nella classe SaveLoad:

```
friend ostream& operator<<(ostream& s, const SaveLoad& sl) {
    return sl.put(s); //uses the right put()
};
```

Ed è qui che viene ottenuto il comportamento polimorfico, poiché la funzione sceglie a runtime il metodo put da utilizzare, e nel nostro caso di esempio quello definito in ScaledHandPosition (in SaveLoad put è astratto!).

I metodi put delle sottoclassi, tutti uguali, fanno quindi da "ponte" richiamando staticamente l'operatore di output della propria classe:

```
std::ostream& put(std::ostream& s) const { return s << *this; };
```

Lo stesso meccanismo, simmetricamente, è usato per l'input (loadFromFile, operator>> e get).

La classe GloveCalibration

La classe GloveCalibration modella una calibrazione del guanto, contenendo due array di valori *raw* minimi e massimi, come spiegato in precedenza. Un oggetto GloveCalibration si ottiene tipicamente utilizzando un oggetto Glove a mo' di *factory*:

```
Glove *g = new Glove();
GloveCalibration *gc = g->getCalibration();
```

In alternativa, noto il numero di sensori, si può costruire un oggetto GloveCalibration "vuoto" e lo si può riempire col metodo updateCalibration di Glove:

```
GloveCalibration *gc = new GloveCalibration(18);
Glove *g = new Glove();
g->updateCalibration(gc);
```

Questa sequenza non ha però molto senso, mentre tipicamente quello che si farà sarà ottenere un oggetto GloveCalibration dal guanto e poi, in caso si necessiti di valori aggiornati dal dispositivo, usare l'*update*:

```
Glove *g = new Glove();
GloveCalibration *gc = g->getCalibration();
//altro codice
g->updateCalibration(gc);
```

Questo vale anche per le classi `ScaledHandPosition` e `RawHandPosition`: inizialmente i metodi di *update* non erano stati implementati, pensando che si potesse semplicemente chiamare nuovamente il rispettivo metodo "get" di `Glove` e ottenere un nuovo oggetto con i dati aggiornati. Questo avrebbe però avrebbe reso la libreria inefficiente: tipicamente le operazioni di aggiornamento sono effettuate in loop ogni tot millisecondi, e sarebbe stato folle allocare ogni volta un nuovo oggetto (e costringere l'utente al `delete` dell'oggetto puntato prima di ottenere il nuovo, pena *memory leak*). Si è quindi scelto di usare la *friendship* del C++ per lasciare alla classe `Glove` libero accesso ai dati privati di `ScaledHandPosition`, `RawHandPosition` e `GloveCalibration`, consentendo l'aggiornamento "in place" tramite le chiamate dell'API 5DT sottostante, come mostra il frammento di codice seguente:

```
//*** estratto da GloveCalibration.h ********
    private:
        unsigned short *pUpper; //array dei RawMax
        unsigned short *pLower; //array dei RawMin
        int numSensors;         //numero sensori e dim array

    friend class Glove; //lascia accesso libero a Glove
//****************************************
//*** estratto da Glove.cpp *****************
GloveCalibration *Glove::getCalibration() {
    GloveCalibration *gc = new GloveCalibration(getNumSensors());
    fdGetCalibrationAll(pGlove, gc->pUpper, gc->pLower);
    return gc;
}
void Glove::updateCalibration(GloveCalibration *gc) {
    fdGetCalibrationAll(pGlove, gc->pUpper, gc->pLower);
}
void Glove::setCalibration(GloveCalibration* gc) {
    fdSetCalibrationAll(pGlove, gc->pUpper, gc->pLower);
}
void Glove::resetCalibration() {
```

```
    fdResetCalibration(pGlove);
}
//****************************************
```

Un'altra idea di progettazione della libreria comportava tenere un puntatore a un oggetto `Glove` negli oggetti `ScaledHandPosition`, `RawHandPosition` e `GloveCalibration`, il che avrebbe portato a codice del genere:

```
Glove *g = new Glove();
ScaledHandPosition *shp = new ScaledHandPosition(g);
shp->update();
```

...ma questo avrebbe obbligato a gestire varie situazioni (ad esempio chiamare `update()` su una `ScaledHandPosition` caricata da file non avrebbe avuto senso, almeno senza prima assegnarle un oggetto `Glove` con un metodo `ScaledHandPosition::setGlove(Glove *g)`). Si è scelto allora di tenere `GloveCalibration`/`ScaledHandPosition`/`RawHandPosition` come classi "stupide" che contengano i dati e permettano di salvarli e caricarli, lasciando il "comando" alla classe `Glove`.

Il save/load su file delle istanze di queste classi si effettua, grazie al meccanismo implementato da `SaveLoad` e all'overloading degli *operatori di I/O*, che sarà illustrato a breve, così:

```
// esempio di salvataggio
Glove *g = new Glove();
GloveCalibration *daSalvare = g->getCalibration();
daSalvare->saveToFile("miacalibrazione.cal");

// esempio di caricamento
Glove *g = new Glove();
GloveCalibration *daCaricare = new GloveCalibration(g->getNumSensors());
daCaricare->loadFromFile("miacalibrazione.cal");
g->setCalibration(daCaricare);
```

Normalmente un oggetto `GloveCalibration` si usa nel suo insieme, come appena mostrato. Se però fosse necessario accedere da codice ai singoli valori $RawMax/RawMin$ di una calibrazione, sono forniti i seguenti metodi getter/setter, autoesplicativi:

- `unsigned short getRawMax(EfdSensors sensorId)`
- `unsigned short getRawMin(EfdSensors sensorId)`

- `void setRawMax(EfdSensors sensorId, unsigned short max)`
- `void setRawMin(EfdSensors sensorId, unsigned short min)`

La classe ScaledHandPosition

`ScaledHandPosition` incapsula un array di `float` corrispondente a una lettura di valori *scaled* dal guanto.

In caso si vogliano gestire singolarmente i valori, sono forniti i due metodi:

- `unsigned short getSensorValue(EfdSensors sensorId)`
- `void setSensorValue(EfdSensors sensorId, unsigned short val)`

L'idea è che un insieme di valori scalati, sotto forma di un oggetto `ScaledHandPosition`, rappresenti una certa posizione della mano in maniera non dipendente dalla calibrazione in uso, almeno con una certa tolleranza, ed è quindi ovvio basare su questa classe il sottisistema di riconoscimento dei gesti. E' da dire che in effetti andrebbe verificato quanto la calibrazione riesce a normalizzare i valori, effettuando dei test di riconoscimento dei gesti su un campione di utenti e magari tentando di migliorare il criterio di confronto in funzione dei risultati. In ogni caso, la classe fornisce, attraverso il metodo `cmp` e l'overloading degli operatori di confronto, la possibilità di trattare `ScaledHandPosition` come un tipo base del linguaggio:

```
//salva un gesto e rimane bloccato finchè non si muove la mano
Glove *g = new Glove();
ScaledHandPosition *current = g->getScaledHandPosition();
ScaledHandPosition saved = *current;
while (saved == *current) { // confronto tra i due gesti
    g->updateScaledHandPosition(current);
    Sleep(20);
}
```

Per tutte e tre le classi `ScaledHandPosition`, `RawHandPosition` e `GloveCalibration` sono stati implementati sia l'assegnamento che il costruttore di copia, permettendo il funzionamento corretto di linee di codice come

```
ScaledHandPosition saved = *current;
```

Per quanto riguarda gli operatori relazionali, sebbene possa sembrare un pò forzato parlare di posizioni della mano "maggiori" o "minori" di altre, rendere ordinabile il tipo di dato permette la ricerca efficiente, e farlo utilizzando i meccanismi standard del linguaggio, implementando gli operatori relazionali, permette di usare i container standard del C++, come vedremo in seguito più in dettaglio.

La classe RawHandPosition

`RawHandPosition` si limita ad incapsulare un array di `unsigned short` corrispondente a una lettura dei valori *raw* dal guanto. Dato che il riconoscimento dei gesti si basa normalmente sui valori *scaled*, non ci si è preoccupati di implementare gli operatori di confronto come in `ScaledHandPosition`.

Anche in questo caso, se non si sta usando l'oggetto nel suo insieme ma si necessita di intervenire sui singoli valori contenuti, sono presenti i metodi:

- `unsigned short getSensorValue(EfdSensors sensorId)`
- `void setSensorValue(EfdSensors sensorId, unsigned short val)`

3.1.4 I/O e confronto dei tipi user-defined

L'overloading degli operatori è una caratteristica molto interessante del C++ che, usata correttamente, permette di creare dei tipi di dato perfettamente inseriti nella sintassi del linguaggio.

Nello sviluppo di ISISgloveAPI tale feature è stata utilizzata con due obiettivi:

- integrare i tipi `ScaledHandPosition`/`RawHandPosition`/`GloveCalibration` nel meccanismo di I/O su stream del C++, implementando in tali classi gli *inserters* (>>) e gli *extractors* (<<);
- poter considerare un'istanza della classe `ScaledHandPosition` un "gesto" e quindi far sì che tali oggetti siano confrontabili, implementando nella classe gli operatori di confronto.

Descriviamo in maniera approfondita queste due particolarità dell'implementazione, già citate più volte nella descrizione generale della libreria.

L'I/O su streams

Il sistema degli I/O streams è il meccanismo generico adottato dal C++ per l'input/output. Tralasciando le funzionalità che riguardano il buffering, la formattazione locale-based e altro, concentriamoci su input/output nel senso di trasformazione da dati in memoria a una sequenza di caratteri (in output) e nella conversione inversa (in input).

Un I/O text-based è particolarmente adatto al nostro caso poiché:

- torna utile nel debugging
- rende immediata l'implementazione di un semplice meccanismo di persistenza su files di testo *human-readable* (nonché semplicemente modificabili in fase di testing/debugging)
- si presta alla trasmissione su socket (che sarà effettuata da ISISgloveManager)
- consente un semplice parsing all'interno di Quest3D, in ricezione (come vedremo descrivendo il template ISISgloveInput)

L'output su stream è associato all'operatore <<, detto *extractor* o "put to", nel senso che inserisce l'oggetto nello stream. Simmetricamente, l'input è associato all'operatore >>, detto `inserter` o "get from", che preleva dallo stream il flusso di caratteri con cui riempie un oggetto.

La classe standard `ostream` (stream di output) definisce essa stessa l'operatore << per i tipi *built-in* del linguaggio: un operatore << è definito da una *operator function* `operator<<()`, che restituisce un riferimento all'`ostream` per la quale era stata chiamata, in modo che vi si possa applicare in cascata un'altra `operator<<()`, permettendo codice del genere

```
int x = 5;
cout << " x = " << x;
```

che sarà interpretato come

```
operator<<(cout, " x = ").operator<<(x)
```

e visualizzerà sullo standard output, associato allo stream `cout`,

```
" x = 5"
```

Per la stringa " `x =` ", coerentemente col fatto che `string` non è un tipo base ma un container definito nella STL, verrà usata una *operator function* definita in `<string>`.

Similmente, il nostro obiettivo è inserire i nostri tipi `ScaledHandPosition`, `RawHandPosition` e `GloveCalibration` in questo sistema standard di gestione dell'I/O, in modo che si possa scrivere codice come

```
ScaledHandPosition *current = g->getScaledHandPosition();
cout << "il gesto corrente è: " << *current;
```

e ottenere sullo standard output una rappresentazione sensata di un oggetto `ScaledHandPosition`, quella stessa rappresentazione che sarà utilizzata nel meccanismo di save/load e per la trasmissione su socket.

L'input formattato è gestito in maniera simmetrica all'output: esiste una classe `istream` che definisce l'operatore di input >> per un piccolo insieme di tipi standard, e una funzione `operator>>()` può essere definita per un tipo di dato *user-defined*.

Terminiamo mostrando l'implementazione del meccanismo in una delle tre classi, `ScaledHandPosition`:

```
// ** da ScaledHandPosition.h **************
friend ostream& operator<<(ostream& output, const ScaledHandPosition& p);
friend istream& operator>>(istream& input, ScaledHandPosition& p);

// ** da ScaledHandPosition.cpp **************
ostream& operator<<(ostream& output, const ScaledHandPosition& p) {
    output << "(";
    for (int i = 0; i < p.numSensors-1; i++ )
        output << p.scaledSensorValues[i] << "|";
    output << p.scaledSensorValues[p.numSensors-1];
    output << ")" ;
    return output;
}
istream& operator>>(istream& input, ScaledHandPosition& p) {
    char useless;
    input >> useless; //"("
    for (int i = 0; i < p.numSensors-1; i++ )
        input >> p.scaledSensorValues[i] >> useless;
    input >> p.scaledSensorValues[p.numSensors-1];
    input >> useless; //")"
```

```
    return input;
}
```

Ecco quindi un esempio di sequenza di caratteri che rappresentarà su stream un oggetto `ScaledHandPosition`:

```
(0.479846|0.357143|0.0298077|0.770341|1|0.859729|0.689552|0.781818|
0.847458|0.600897|0.926724|0.900404|0.783051|0.744108|1|1|0|0)
```

Si noti la simmetria delle operazioni di input/output e l'utilizzo degli *inserters/extractors* anche per i valori `float` contenuti in `ScaledHandPosition`.

Confronto e ordinamento di gesti

La classe fondamentale su cui si basa il riconscimento dei gesti è `ScaledHandPosition`. La variabile statica `cmpToll` contiene l'impostazione della tolleranza nel riconoscimento, è viene usata per stabilire i margini entro il quale una `ScaledHandPosition` si può considerare "uguale" a un'altra.

Ecco un estratto dal file `ScaledHandPosition.h` che mostra l'implementazione degli operatori di confronto.

```
1   // ...
2   static void setCompareThreshold(float toll) {
3       cmpToll = toll;
4   };
5   static float getCompareThreshold() {
6       return cmpToll;
7   };
8   int ScaledHandPosition::cmp(const ScaledHandPosition *other) const {
9     for (int i = 0; i < numSensors; i++)
10      if ((scaledSensorValues[i]-other->scaledSensorValues[i]) < -cmpToll)
11          return -1;  // <
12      else
13      if ((scaledSensorValues[i]-other->scaledSensorValues[i]) > cmpToll)
14          return 1;   // >
15    return 0;         // ==
16  }
17  bool operator== (const ScaledHandPosition &a ) const {
18      return (this->cmp(&a)==0);
19  };
20  bool operator!=( const ScaledHandPosition &a ) const {
```

```
21      return (this->cmp(&a)!=0);
22  }
23  bool operator<(const ScaledHandPosition &a) const {
24      return (this->cmp(&a)<0);
25  }
26  bool operator>(const ScaledHandPosition &a) const {
27      return (this->cmp(&a)>0);
28  }
29
30  // ...
31
32  private:
33      float *scaledSensorValues;
34      int numSensors;
35      static float cmpToll; //threshold value in compare
36
37  // ...
38
```

I metodi statici `setCompareThreshold` e `getCompareThreshold` permettono di manipolare `cmpToll`, mentre `cmp` implementa il meccanismo di confronto di due oggetti `ScaledHandPosition` e viene utilizzata nelle semplicissime *operator functions* che implementano gli operatori di confronto.

3.1.5 Un confronto tra librerie

Per rendere l'idea del vantaggio nell'utilizzare ISISgloveAPI al posto dell'API nativa fornita dalla 5DT, confrontiamo due frammenti di codice (puramente indicativo) che le utilizzano. Obiettivo del codice è accedere al dispositivo (collegato su una porta qualsiasi), lasciare 5 secondi di tempo per calibrare, poi salvare un gesto (saved), attendere un altro secondo e poi bloccarsi in un loop fin quando non si effettua nuovamente il gesto salvato.

```
1   // codice C di esempio utilizzante API 5DT
2   fdGlove *pGlove;
3   char szPort[5];
4   strcpy(szPort,"COMx");
5   char i;
6   for (i='1'; i<='8'; i++) {
7       szPort[3] = i;
```

```
8       if (NULL != (pGlove = fdOpen( szPort ))) break;
9   }
10  sleep(5000);
11  float cmpToll = 0.15;
12  int numsensors = fdGetNumSensors(pGlove);
13  float *saved = (float*) malloc(numsensors * sizeof(float));
14  float *current = (float*) malloc(numsensors * sizeof(float));
15  fdGetSensorScaledAll(pGlove, saved);
16  sleep(1000);
17  do {
18      fdGetSensorScaledAll(pGlove, current);
19      for (int i = 0; i < numsensors; i++)
20          if ( (current[i] - saved[i]) < -cmpToll )
21              break;  // <
22          else if ( (current[i] - saved[i]) > cmpToll )
23              break;  // >
24      if (i==numsensors) break; // == (match found)
25      sleep(20);
26  } while (1);
27  printf("ok!\n");
```

```
1   //codice C++ che utilizza ISISgloveAPI
2   Glove *g = new Glove();
3   sleep(5000);
4   ScaledHandPosition::setCompareThreshold(0.15);
5   ScaledHandPosition *saved = g->getScaledHandPosition();
6   ScaledHandPosition *current = new ScaledHandPosition(g->getNumSensors());
7   sleep(1000);
8   do {
9       g->updateScaledHandPosition(current);
10      sleep(20);
11  } while (*current != *saved);
12  cout << "ok!" << end;
```

Se si dovessero salvare/caricare dati (calibrazione/gesti) su file, la differenza in termini di linee di codice, come è facile immaginare, aumenterebbe ancora di più.

3.1.6 Sviluppi futuri

Potrebbe essere interessante cambiare l'implementazione della funzione `cmp` di `ScaledHandPosition`, introducendo un criterio di riconoscimento più sofisticato del semplice confronto di tutti i valori a meno di un margine di tolleranza. L'importante sarebbe mantenere valida la relazione d'ordine che permette di usare efficientemente i containers standard per la ricerca.

3.2 ISISgloveManager

ISISgloveManager è un'utility di gestione dei 5DT Data Glove che permette di:

- accedere al dispositivo e visualizzare in tempo reale
  - i valori *raw*/*scaled* dei sensori
  - gli estremi del range di valori *raw* rilevati con la mano di un determinato utente (ovvero i valori di cui consiste la calibrazione del guanto)
- salvare/caricare la calibrazione
- salvare dei gesti, associandovi una stringa identificativa
- riconoscere i gesti salvati, regolando la tolleranza a runtime
- agire da server TCP che effettua lo streaming dei valori *raw*/*scaled* e delle stringhe identificative dei gesti riconosciuti

L'utility è pensata per essere avviata e poi rimanere attiva in background, richiamabile tramite un'icona nella *taskbar*.

3.2.1 Utilizzo dell'applicazione

Avviata l'applicazione, possiamo indicare la porta seriale dove è collegato il guanto o cliccare direttamente "Open Device" e lasciare che venga eseguito lo scanning automatico delle porte. Se l'accesso al dispositivo ha successo, la griglia sulla destra del tab si popola dei valori ottenuti dai sensori. Attivato il guanto, diventa possibile gestirne la calibrazione (save/load/reset), e si attivano le funzionalità relative al riconoscimento dei gesti.

L'applicazione salva i gesti in files con estensione ".hand" ospitati in una subdirectory "data" (creata alla prima esecuzione, se necessario). All'avvio del programma tali files vengono letti e i gesti caricati in memoria, per permettere il riconoscimento in tempo reale. Il nome del file è la stringa identificativa del gesto.

Il bottone "Add to known...", abilitato solo se non si sta tenendo la mano in una posizione già riconosciuta, per evitare conflitti, permette di salvare il gesto corrente e di associarvi una stringa identificativa. E' poi presente una label che indica se la posizione corrente della mano corrisponde a un gesto noto, permettendo il testing del riconoscimento (la quale tolleranza si regola

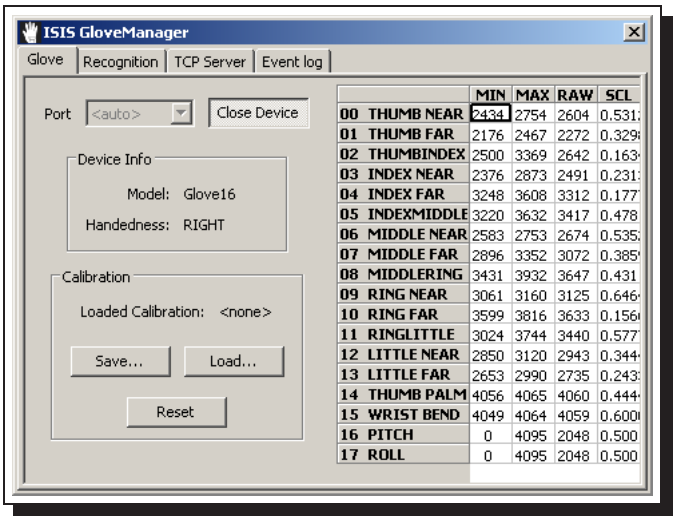


Figura 3.2: ISISgloveManager: gestione del guanto

con uno slider). Sulla destra la lista dei gesti "noti", con la possibilità di eliminarne.

Il terzo tab permette di gestire il server TCP. E' possibile configurare l'interfaccia di rete e la porta TCP sul quale metterlo in ascolto, ed è presente una lista degli eventuali client connessi, con indicazione di indirizzo IP e porta remota.

Infine, è presente un tab contenente un log degli eventi salienti verificatisi nell'esecuzione dell'applicazione, utile soprattutto in caso di problemi.

Chiudere la finestra di ISISgloveManager manda il software in background, e lo si può richiamare in primo piano, oppure terminarlo, dal menù della sua icona nella TaskBar.

In genere, per quanto riguarda le dipendenze funzionali del programma:

- terminare l'applicazione causa la disattivazione del guanto
- disattivare il guanto comporta l'arresto del server TCP
- arrestare il server TCP chiude tutte le connessioni con i clients

Introdurre questo meccanismo di chiusura a cascata ha evitato di dover gestire alcune situazioni anormali (server arrestato ma connessioni ancora attive, server attivo ma niente dati da inviare...).

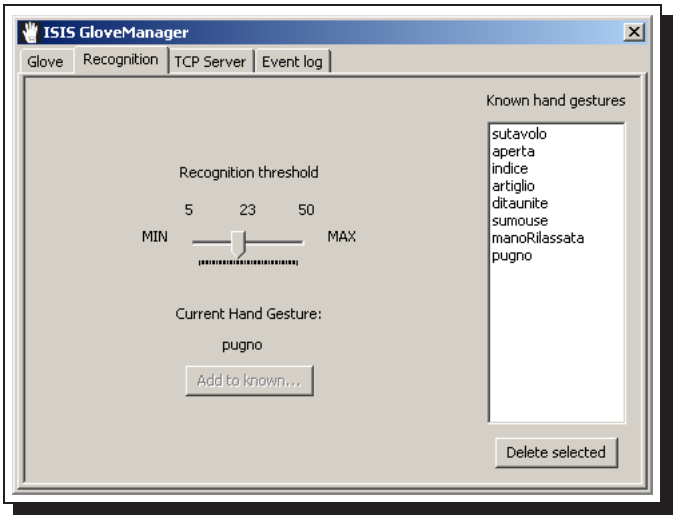


Figura 3.3: ISISgloveManager: riconoscimento dei gesti

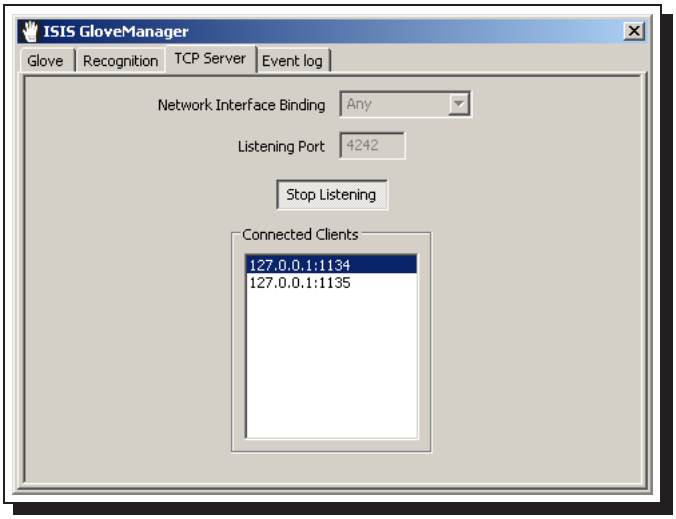


Figura 3.4: ISISgloveManager: server TCP

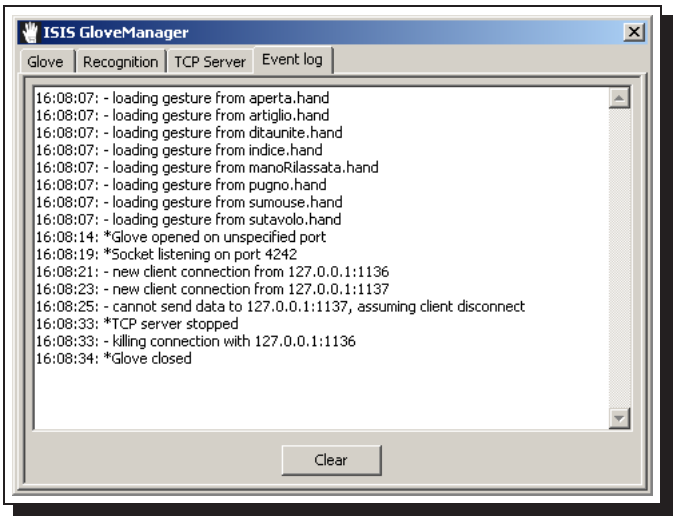


Figura 3.5: ISISgloveManager: log esecuzione

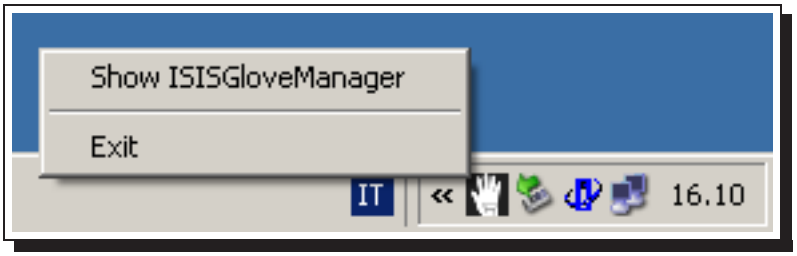


Figura 3.6: ISISgloveManager: l'icona nella *taskbar*

3.2.2 Architettura

L'applicazione è stata così strutturata:

- il thread principale è quello della GUI, tab-based: tale thread si preoccupa di gestire l'interfaccia utente, abilitando/disabilitando i controlli sui vari tab coerentemente allo stato del sistema (ad esempio: se il guanto non è attivo, non è possibile salvare un gesto o avviare il server);
- un thread secondario (ma non per importanza, in quanto implementa le funzionalità chiave dell'applicazione) preleva di continuo i valori dai sensori del guanto, controlla eventuali corrispondenze con i gesti salvati, e poi passa valori ed eventuali match al thread della GUI. Se il server è attivo e ci sono client connessi, effettua anche il broadcast di tali informazioni;
- un altro thread secondario è quello del server TCP, e si occupa essenzialmente di accettare le connessioni dai client.

Considerando che i clients sono passivi, e non è quindi necessario attendere input sui sockets, ma solo inviare dati a intervalli regolari, ci è sembrato sensato non complicare l'architettura dell'applicazione avviando un thread per ogni client connesso: il broadcast viene effettuato dal thread che si interfaccia col guanto. Tra l'altro, dato il tipo di applicazione, viene difficile immaginare contesti in cui siano presenti molte applicazioni client connesse contemporaneamente (mentre è ragionevole pensare, ad esempio, a un client grafico principale, come il nostro demo Quest3D, e ad altri che indipendentemente eseguano il logging dei dati di una sessione d'uso, o calcolino statistiche di qualche genere...).

Il class diagram dell'applicazione (figura 3.7) dovrebbe essere un buon punto di riferimento per passare da questa parziale descrizione architetturale all'implementazione vera e propria:

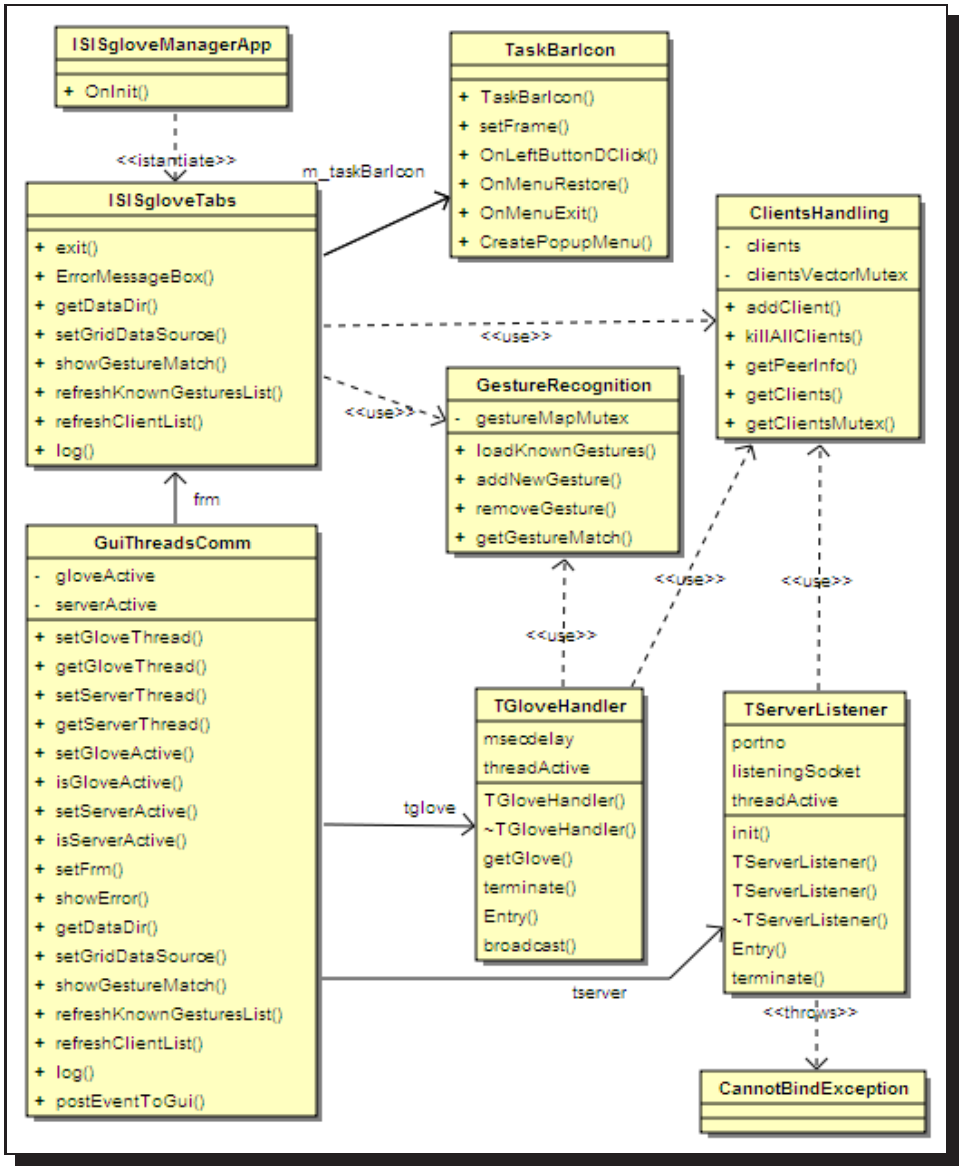


Figura 3.7: ISISgloveManager: class diagram - i metodi privati di ISISgloveTabs sono omessi per ragioni di spazio

3.2.3 Implementazione

Descriviamo alcuni aspetti interessanti dell'implementazione del programma, una componente alla volta.

ISISgloveTabs: la GUI

L'implementazione della GUI, che costituisce anche il thread principale dell'applicazione, è stata realizzata partendo da una classe generata con *DialogBlocks*, un editor visuale per wxWidgets. DialogBlocks permette di lavorare al layout delle finestre e di associare degli *stub* ad alcuni eventi, come la pressione di un bottone o il trascinamento di uno slider. Il codice generato è stato poi modificato ed integrato, con l'obiettivo di fornire una GUI semplice ed efficace che gestisse il flusso di esecuzione dell'applicazione, compresa l'aggiunta di una semplice classe (`TaskBarIcon`) che si occupi di gestire l'icona nella *taskbar*.

Il risultato è una quantità di codice piuttosto elevata, ma non particolarmente significativa nel descrivere il funzionamento del programma: si è fatto in modo di evitare gli input errati e di attivare/disattivare i controlli dell'interfaccia a seconda dello stato dell'applicazione.

Unici aspetti interessanti sono quelli correlati alla comunicazione con i threads secondari, discussi in seguito.

TGloveHandler: prelievo e broadcast dei dati dall'hardware

La classe `TGloveHandler` implementa un thread (deriva da `wxThread`) che si occupa di interagire, grazie a ISISgloveAPI, col guanto. Il costruttore istanzia un oggetto `Glove`, mentre il metodo `Entry` (che viene eseguito quando si avvia l'esecuzione del thread) merita di essere analizzato in dettaglio.

Innanzitutto vengono istanziati tre oggetti: `sh`, `rh` e `gc`. Questi sono rispettivamente oggetti `ScaledHandPosition`, `RawHandPosition` e `GloveCalibration` ottenuti da `g`, puntatore all'oggetto `Glove` istanziato nel costruttore del thread. Tali oggetti vengono impostati come "sorgente dati" per la GUI, in modo da permettere l'aggiornamento della griglia di valori sul "Glove tab".

```
ScaledHandPosition *sh = g->getScaledHandPosition();
RawHandPosition *rh = g->getRawHandPosition();
GloveCalibration *gc = g->getCalibration();

GuiThreadsComm::setGridDataSource(sh,rh,gc);
```

Parte poi il loop fondamentale dell'applicazione, che ogni 20 millisecondi (intervallo scelto considerando che, da manuale, l'hardware permette 52 letture al secondo) effettua sostanzialmente queste operazioni:

- aggiorna gli oggetti `sh`, `rh` e `gc` con i valori letti dall'hardware

  ```
  g->updateScaledHandPosition(sh);
  g->updateRawHandPosition(rh);
  g->updateCalibration(gc);
  ```

- controlla se tali valori corrispondono a qualche gesto "noto"

  ```
  std::string posMatch = GestureRecognition::getGestureMatch(sh);
  ```

  (e in caso positivo, segnala alla GUI il match perché questa lo visualizzi)

  ```
  GuiThreadsComm::postEventToGui(T_EVT_POSMATCH);
  ```

- prepara la stringa da inviare ai clients, sfruttando elegantemente l'overloading degli operatori di I/O effettuato in ISISgloveAPI e la classe `ostringstream`, che permette di scrivere in una stringa come se fosse uno stream

  ```
  std::ostringstream fromGlove;  // output string stream
  fromGlove << "SCL:" << *sh << std::endl;
  fromGlove << "RAW:" << *rh << std::endl;
  if (posMatch!="")
      fromGlove << "POS:" << posMatch << std::endl;
  toClients = fromGlove.str();
  ```

- effettua il broadcast di tale stringa ai clients

  ```
  broadcast(toClients);
  ```

Il metodo broadcast merita ugualmente una descrizione dettagliata:

- richiediamo l'accesso esclusivo al `Vector` di puntatori a `wxSocketBase` che mantiene gli oggetti `wxSocketBase` relativi alle connessioni con i clients, in quanto durante il broadcast non vogliamo interferenze col thread che accetta le nuove connessioni.

```
ClientsHandling::getClientsMutex()->Lock();
std::vector<wxSocketBase*> *v = ClientsHandling::getClients();
std::vector<wxSocketBase*>::iterator i;
```

- iteriamo sul vector e per ogni client effettuiamo una write della stringa sul socket Se tale write non ha successo, marchiamo la posizione corrente del vector come "cancellabile", considerando il client come disconnesso

```
bool anyClientDisconnect = false;
const char *buf = forClients.c_str();

for (i = v->begin(); i != v->end(); ++i) {
    wxSocketBase *sock = *i;
    wxSocketOutputStream SocketOutputStream(*sock);
    wxDataOutputStream out(SocketOutputStream);

    out.Write8((wxUint8*)buf,strlen(buf));
    if (!out.IsOk()) {
        GuiThreadsComm::log(wxString::Format(
            _("- cannot send data to %s, assuming client disconnect\n"),
            ClientsHandling::getPeerInfo(sock)));
        *i = NULL; //mark but don't delete (or iterator gets BROKEN!)
        anyClientDisconnect = true;
    }
}
```

- terminata l'iterazione sul vector, se qualche `Write8` è fallita, effettuiamo le cancellazioni degli elementi precedentemente marcati, e indichiamo alla GUI di aggiornare la lista dei client connessi

```
if (anyClientDisconnect) {
    v->erase(remove_if(v->begin(), v->end(), &is_NULL), v->end() );
    GuiThreadsComm::refreshClientList(*v);
}
ClientsHandling::getClientsMutex()->Unlock();
```

Nota: `is_NULL` è un predicato, definito precedentemente, che indica la condizione di cancellazione alla combinazione delle funzioni `erase`/`remove_if`, tecnica standard da usare per la rimozione condizionale di elementi da un container della STL.

```
inline bool is_NULL(wxSocketBase* s) {
    return (s==NULL);
};
```

TServerListener: gestione del server TCP

Il thread TServerListener è molto semplice: viene creato quando dalla GUI, dopo aver attivato il guanto, si seleziona un'interfaccia di rete e una porta TCP valida e si sceglie di avviare il server TCP. Il costruttore della classe tenta di aprire un socket TCP in ascolto sulla porta e sull'interfaccia di rete specificate nella GUI.

```
listeningSocket = new wxSocketServer(addr);
if (!listeningSocket->IsOk()) {
    GuiThreadsComm::log(_("! Socket error! Cannot bind!\n"));
    throw  CannotBindException();
}
```

Il metodo Entry consiste essenzialmente in un loop di accettazione delle connessioni terminato quando, alla chiusura del server, si chiama terminate(), che setta il flag threadActive a false e distrugge listeningSocket, in modo da sbloccare l'Accept(). In alcuni casi, Accept fallisce per cause diverse dalla terminazione volontaria, ed ecco perché si rende necessario il flag threadActive.

```
while (threadActive) {
    wxSocketBase *sock = listeningSocket->Accept();

    // if Accept failed or interrupted by termination
    if ((sock == NULL) || (!sock->IsOk())) continue;

    // non blocking writes to clients
    // this prevents the broadcast being stopped by "bad" clients
    sock->SetFlags(wxSOCKET_NOWAIT);

    // store socket in the "connected clients" array
    ClientsHandling::addClient(sock);
}
```

GuiThreadsComm: comunicazione GUI/threads secondari

La classe statica `GuiThreadsComm` fa da tramite tra il main thread (la GUI) e i thread secondari dell'applicazione, tenendo traccia del loro stato e gestendo la comunicazione tra di essi. Tralasciando i metodi ovvi della classe, analizziamo brevemente l'uso degli eventi che è stato fatto per evitare alcune situazioni d'errore e la non responsività della GUI.

```
//da GuiThreadsComm.h
enum ThreadEvent {
    T_EVT_GLOVESTARTED, //glove thread started
    T_EVT_GLOVESTOPPED, //glove thread terminated
    T_EVT_SERVERSTARTED,    //server thread started
    T_EVT_SERVERSTOPPED,    //server thread terminated
    T_EVT_POSMATCH  //position match from glove thread
};
//da GuiThreadsComm.cpp
void GuiThreadsComm::postEventToGui(ThreadEvent eventId) {
    wxCommandEvent myevent( wxEVT_COMMAND_MENU_SELECTED, THREAD_EVENT );
    myevent.SetInt(eventId);
    wxPostEvent( frm, myevent );
}
```

Essendo per noi sufficiente inviare un intero come meccanismo di notifica, ricicliamo un evento predefinito di wxWidgets non utilizzato altrove (`wxEVT_COMMAND_MENU_SELECTED`), e lo aggiungiamo alla coda processata dal thread principale con `wxPostEvent`.

In situazioni dove è necessario uno scambio di dati più sofisticato, è possibile implementare una propria classe evento, ma per un semplice meccanismo di notifica sarebbe stato eccessivo.

Vediamo ora come avviene la ricezione e il processing degli eventi, nella classe che implementa la GUI:

```
//da ISISgloveTabs.cpp
BEGIN_EVENT_TABLE( ISISgloveTabs, wxFrame )
    // ... qui altro codice
    EVT_MENU(THREAD_EVENT, ISISgloveTabs::OnThreadEvent)
    EVT_IDLE(ISISgloveTabs::OnIdle)
END_EVENT_TABLE()

//...
```

```
void ISISgloveTabs::OnIdle(wxIdleEvent& event) {
    idleUpdateGrid();
    //idleUpdatePosMatch(); //too slow!
}
void ISISgloveTabs::OnThreadEvent(wxCommandEvent& event) {
    int n = event.GetInt();
    switch(n) {
        case T_EVT_GLOVESTARTED: //glove thread started
            onGloveStarted(); onPositionMatch(); break;
        case T_EVT_GLOVESTOPPED: //glove thread terminated
            onGloveStopped(); break;
        case T_EVT_SERVERSTARTED: //server thread started
            onServerStarted(); break;
        case T_EVT_SERVERSTOPPED: //server thread terminated
            onServerStopped(); break;
        case T_EVT_POSMATCH: //position match from glove thread
            onPositionMatch(); break;
    }
}
```

Il metodo `OnThreadEvent` viene associata al processing del nostro evento, e a seconda dell'intero ricevuto esegue un'operazione adeguata. E' a questo punto necessario inserire nel discorso anche l'implementazione dell'*idle event handler*, `OnIdle`. L'*idle event* è un evento predefinito che viene generato se, completato un normale ciclo di esecuzione e processati tutti gli eventi in coda, alla GUI rimane del "tempo di inattività" (*idle time*), che può quindi essere usato per eseguire operazioni non critiche (tipicamente aggiornamenti nella visualizzazione).

Tralasciando quindi le operazioni occasionali che non danno problemi, come quelle di aggiornamento della lista dei client connessi e dei gesti riconosciuti, ci sono due elementi di cui l'update è critico: la griglia che visualizza i valori dei sensori, sul tab "Glove", e la label che visualizza in tempo reale la stringa corrispondente a eventuali gesti riconosciuti, nel tab "Recognition". Forzare l'aggiornamento di entrambe le componenti della GUI a ogni lettura dei sensori rendeva il programma fastidiosamente lento, tanto che la lettura e la trasmissione dei valori dei sensori veniva sensibilmente rallentata quando il tab con la griglia dati era visualizzato.

Si è quindi lasciato in *idle time* l'aggiornamento della griglia, effettuato dal metodo `idleUpdateGrid()`:

```
void ISISgloveTabs::idleUpdateGrid() {
```

```
    if (!GuiThreadsComm::isGloveActive()) return;
    wxString val;
    for (int i= 0; i< 18; i++){
        val = wxString::Format(_("%4d"),
                c_gc->getRawMin((EfdSensors)i));
        gloveGrid->SetCellValue(i,0,val);
        val = wxString::Format(_("%4d"),
                c_gc->getRawMax((EfdSensors)i));
        gloveGrid->SetCellValue(i,1,val);
        val = wxString::Format(_("%4d"),
                c_rhp->getSensorValue((EfdSensors)i));
        gloveGrid->SetCellValue(i,2,val);
        val = wxString::Format(_("%1.2f"),
                c_shp->getSensorValue((EfdSensors)i));
        gloveGrid->SetCellValue(i,3,val);
    }
}
```

Invocare questo metodo dall'*idle handler* ha consentito di mantenere fluido l'aggiornamento della griglia senza per questo rallentare la (più importante) lettura dei dati dall'hardware.

Tale tipo di soluzione non si è però rivelata adatta quando si è trattato di aggiornare la label indicanta il "gesto corrente" nel tab *Recognition*: relegarla al processing in "idle time" introduceva un ritardo fastidioso per l'utilizzatore intento a verificare il corretto riconoscimento dei gesti da lui definiti e nell'aggiungerne di nuovi.

Aggiornare di continuo la stringa, d'altro canto, sebbene in questo caso non desse problemi di prestazioni percettibili come nel caso della griglia, causava una sorta di "flickering" del testo.

La soluzione è stata adottare il meccanismo degli eventi descritto prima per forzare un'aggiornamento intelligente della stringa e della sua visualizzazione nell'interfaccia grafica:

```
//dal loop principale di TGloveHandler:
    std::string posMatch = GestureRecognition::getGestureMatch(sh);
    if (posMatch != prevPosMatch) {
        GuiThreadsComm::setGestureMatchString(posMatch);
        prevPosMatch = posMatch;
        GuiThreadsComm::postEventToGui(T_EVT_POSMATCH);
    }
```

Se da getGestureMatch si ottiene una stringa diversa da quella attualmente impostata, la nuova stringa si passa alla GUI (setGestureMatchString(posMatch)) e poi si notifica l'evento (postEventToGui(T_EVT_POSMATCH)).

L'event handler mostrato poco sopra, ricevuto tale evento, chiamerà onPositionMatch(), che effettuerà nel thread principale l'aggiornamento vero e proprio della GUI.

Per quanto riguarda gli altri eventi, sono utilizzati per aggiornare lo stato della GUI relativamente allo stato dei due thread secondari, in modo da evitare condizioni d'errore: quando ad esempio si richiede l'accesso al guanto, non si possono considerare immediatamente disponibili i valori dei sensori, ma va atteso l'evento che indica la corretta inizializzazione del dispositivo e l'inizio del loop di prelievo dei dati dall'hardware.

ClientsHandling: controllare le connessioni dei clients

ClientsHandling, che raggruppa delle funzioni relative alla gestione dei clients, è una semplice classe statica così definita:

```
class ClientsHandling {
    public:
        static void addClient(wxSocketBase *s);
        static void killAllClients();
        static wxString getPeerInfo(wxSocketBase *s);
        static std::vector<wxSocketBase*> *getClients() { return &clients; };
        static wxMutex* getClientsMutex() { return clientsVectorMutex; };

    private:
        static std::vector<wxSocketBase*> clients;
        static wxMutex *clientsVectorMutex;
};
```

Una connessione con un client è gestita tramite l'oggetto wxSocketBase ottenuto all'accettazione della sua richiesta di comunicazione, operata nel thread TServerListener, ed è quindi ragionevole avere un vector di puntatori a tali oggetti come struttura dati su cui basare la gestione dei clients:

```
static std::vector<wxSocketBase*> clients;
```

A tale vector accederanno sia il thread TServerListener (per accettare nuove connessioni, o per terminare le esistenti quando viene chiuso il server), sia il thread TGloveHandler, che si occupa del broadcast dei dati (anche

lui però manipola in scrittura il `vector`, rimuovendo gli oggetti a cui non riesce ad inviare dati). Per rendere l'accesso alla struttura dati *thread-safe* utilizziamo quindi il `wxMutex clientsVectorMutex`.

I metodi sono molto semplici e non è necessario esaminarne l'implementazione:

- `addClient` viene utilizzato da `TServerListener` quando accetta una nuova connessione, per inserire un puntatore al nuovo oggetto `wxSocketBase` nel `vector`
- `killAllClients`, usata quando il server viene chiuso, termina tutte le connessioni e svuota il `vector`
- `getPeerInfo`, dato un puntatore a `wxSocketBase`, restituisce una stringa composta da indirizzo IP e porta remota del client relativo, utilizzata per popolare la lista dei *client connessi* nella GUI

GestureRecognition: un container di oggetti ScaledHandPosition

Grazie all'overloading degli operatori eseguito nella classe `ScaledHandPosition`, descritto precedentemente, possiamo usare una `std::map` per implementare in maniera efficiente il riconoscimento dei gesti.

Ecco la definizione della classe statica `GestureRecognition`:

```
class GestureRecognition {
  public:
    static void loadKnownGestures();

    static void addNewGesture(std::string shpname, ScaledHandPosition *shp);
    static void removeGesture(std::string name);

    static std::string getGestureMatch(ScaledHandPosition *p);

  private:
    static std::map<ScaledHandPosition, std::string> gestureMap;
    static wxMutex *gestureMapMutex;
};
```

Elemento fondamentale della classe è la `gestureMap`. Come vediamo nella definizione della classe, la `map` ha come chiave oggetti `ScaledHandPosition`

e come valori le relative stringhe identificative, definite quando si aggiunge un gesto tramite la GUI oppure quando lo si carica da file in fase di inizializzazione.

Il `wxMutex`, `gestureMapMutex`, si occupa di gestire l'accesso concorrente (da parte di GUI e del thread `TGloveHandler` che controlla i *match*) alla `map`. Sebbene l'accesso da parte di TGloveHandler sia in sola lettura, è consigliato, non essendo il multithreading parte della definizione standard del linguaggio, adottare una strategia di programmazione "difensiva".

Analizziamo brevemente i metodi forniti dalla classe:

- `loadKnownGestures`, chiamato in fase di inizializzazione dell'applicazione, si occupa di riempire la `gestureMap` caricando i files ".hand", gestendo le possibilità di errore del caso (conflitti tra gesti, files malformati, creazione della "data directory" alla prima esecuzione dell'applicazione)

- `addNewGesture`, utilizzato quando si aggiunge un nuovo gesto dalla GUI, permette piuttosto ovviamente di aggiungere una nuova coppia key/value alla `map`

- `removeGesture` serve a eliminare una coppia key/value dalla `map` in base al valore stringa accettato come parametro: si vuole infatti cancellare un gesto dalla map scegliendone la stringa identificativa tramite la GUI. Dovendo effettuare una cancellazione per valore e non per chiave è necessaria una semplice ricerca:

```
void GestureRecognition::removeGesture(std::string name) {
    gestureMapMutex->Lock();
    std::map<ScaledHandPosition, std::string>::iterator gestureMapIt;
    for (gestureMapIt = gestureMap.begin();
         gestureMapIt != gestureMap.end();
         ++gestureMapIt) {
            if (gestureMapIt->second == name) {
                gestureMap.erase(gestureMapIt);
                break;
            }
    }
    gestureMapMutex->Unlock();
}
```

- `getGestureMap`, già incontrata analizzando `TGloveHandler`, è l'operazione chiave di cui ci interessava avere un'implementazione efficiente,

essendo effettuata a ogni lettura dei valori dal guanto. Dato in input il gesto, sotto forma di oggetto `ScaledHandPosition`, la funzione sfrutta l'algoritmo `find` della STL:

```
std::string GestureRecognition::getGestureMatch(ScaledHandPosition *p) {
    std::string res("");
    gestureMapMutex->Lock();
    std::map<ScaledHandPosition, std::string>::iterator gestureMapIt;
    gestureMapIt = gestureMap.find(*p);
    if ((gestureMapIt != gestureMap.end())&&(gestureMapIt->first == *p))
        res = gestureMapIt->second;
    gestureMapMutex->Unlock();
    return res;
}
```

La `std::map` garantisce l'accesso ai valori in tempo *O(log(n))*, ed è di solito implementata con un albero binario *self-balanced*. Ricordiamo, per l'ultima volta, che sfruttare la `std::map` e i relativi algoritmi (come `find`) è stato possibile grazie all'implementazione degli operatori di confronto nella classe `ScaledHandPosition`: il compilatore stesso avrebbe impedito di utilizzare tale tipo come chiave della `std::map` se non avesse saputo come stabilirne l'ordinamento (di default gli algoritmi della STL usano l'operatore <).

3.2.4 Sviluppi futuri

Potrebbe essere interessante espandere la logica di riconoscimento dei gesti del programma in un sistema di riconoscimento dei movimenti: un'idea potrebbe essere creare una finestra di salvataggio dei movimenti basata su keyframes, ai quali viene assegnato un gesto (riutilizzando quindi la logica di gesture-matching già implementata). Fatto ciò si dovrebbe trovare un modo di effettuare il riconoscimento dei keyframes nel tempo, con un certo grado di tolleranza, e prevedendo che ci possano essere dei movimenti che, ad esempio, condividano alcuni keyframes (un automa a stati finiti?). Sarebbe ottimo anche incorporare alla GUI del programma una zona che visualizzi un modello animato della mano, cosa che potrebbe anche tornare utile nel sistema di salvataggio dei movimenti (associata alla visualizzazione del modello potrebbe esserci una timeline con le consuete funzionalità in stile VCR). Il modello 3D della mano si potrebbe incorporare senza rinunciare alla portabilità usando *OpenGL* e la classe `wxGLCanvas` di wxWidgets, che

permette di creare una zona, all'interno di una container window, al quale inviare comandi OpenGL.

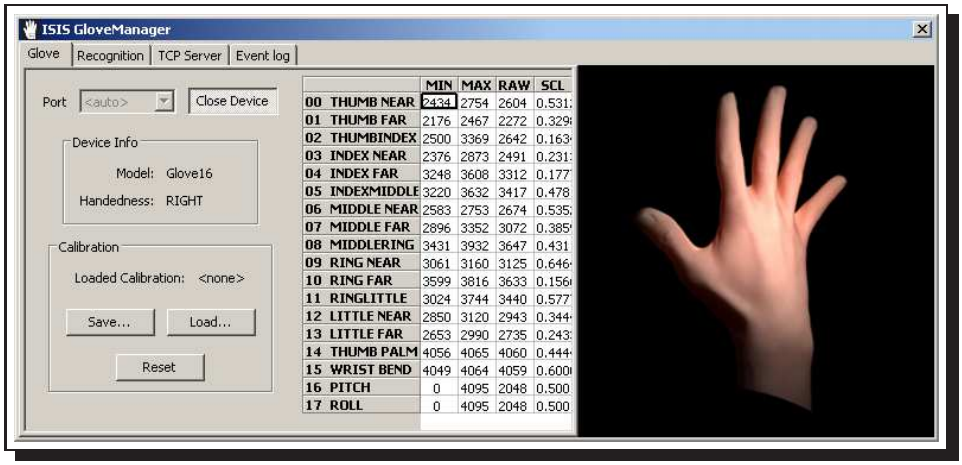


Figura 3.8: un GUI mockup di un possibile ISISgloveManager 2.0: sarà mai implementato?

3.3 ISISgloveManagerInput

Descriveremo ora come il supporto di Quest3D al 5DT Data Glove è stato migliorato grazie all'interfacciamento con ISISgloveManager, e presenteremo un esempio di schema utilizzabile per associare al riconoscimento di un gesto delle azioni all'interno di Quest3D.

3.3.1 Il supporto nativo al 5DT Data Glove, e perché sostituirlo

Il supporto nativo di Quest3D al 5DT Data Glove, parte delle caratteristiche offerte dall'edizione VR, consiste nei generici channels `Glove Source` e `Glove Value` e nello specifico `Glove Driver 5DT` (da collegare a `Glove Source` per indicare il tipo di sorgente). Questa ragionevole generalizzazione ai vari modelli di guanti supportati obbliga a impostare manualmente un channel `Glove Value` per ogni sensore che si desidera utilizzare. Fatto ciò, i channels `Glove Value` si aggiornano continuamente con i valori scalati prelevati dal dispositivo, che va inizializzato chiamando una volta il channel `Glove Source`.

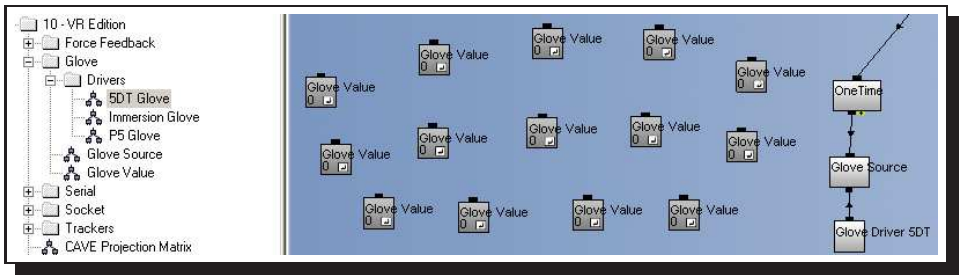


Figura 3.9: Esempio d'uso del supporto nativo al guanto.

Quest3D accede direttamente all'hardware, mentre interfacciando l'engine con ISISgloveManager evitiamo la necessità di calibrare il guanto all'avvio di ogni sessione 3D e, soprattutto, sfruttiamo la funzionalità di riconoscimento dei gesti implementata nell'utility di gestione. Se infatti i valori scalati ottenuti dai sensori sono direttamente utili se si tratta di animare il modello di una mano, è molto più ragionevole utilizzare come trigger di azioni nell'ambiente virtuale dei gesti, potenzialmente combinati con la posizione della mano (fornita dal dispositivo di motion tracking).

Un vantaggio collaterale del fatto che non è Quest3D a controllare direttamente l'hardware è il fatto che questo può essere usato contemporanea-

mente da più applicazioni client, come discusso descrivendo l'architettura di ISISgloveManager.

3.3.2 Sockets in Quest3D

Il supporto ai sockets TCP è considerata una feature avanzata in Quest3D, ed è presente solo a partire dalla versione 3.5 dell'edizione VR del software. Non va confuso col gruppo di channel (con prefisso `Network`) basati sulle DirectX e pensati per l'utilizzo in giochi multiplayer.

L'implementazione attuale dei channels per l'utilizzo dei sockets è abbastanza limitata: non è possibile agire da server (mettendo una porta TCP in ascolto ed accettando connessioni) e non si può effettuare più di una connessione come client. Tutto quello che si può fare è, in pratica, stabilire un'unica connessione e su di essa inviare/ricevere dati. Fortunatamente, ciò è esattamente quello che ci serve per comunicare con ISISgloveManager.

I channels relativi al supporto dei sockets sono questi quattro:

- SocketInfoValue
- SocketAction
- SocketString
- SocketReceiver

Al momento, il channel `SocketReceiver` è totalmente privo di documentazione (compreso il WIKI) ed è facile immaginare che servirà a implementare in futuro una componente server che riceva connessioni. Descriviamo brevemente gli altri.

- `SocketInfoValue` è un channel, di *BaseType* `Value`, che indica se la connessione è attiva o meno, assumendo rispettivamente valore 1 o 0.
- `SocketString` è un channel, di *BaseType* `Text`, che si aggiorna quando una nuova stringa è ricevuta dalla connessione su socket.
- `SocketAction` è il channel fondamentale per l'uso dei socket, e permette di eseguire diverse azioni a seconda di come lo si configura nell'editor: *Connect, Disconnect, Send Data, Start Logging, Stop Logging*. Ognuna di queste usa diversamente i due figli del channel, etichettati *Action Arguments*:

- `Connect (Text,Value)`: tenta di stabilire una connessione TCP con l'host indicato nel figlio `Text`, alla porta indicata nel figlio `Value`.
- `Disconnect`: effettua la disconnessione. Non necessita figli, poiché si può aprire solo una connessione per volta e, per forza di cose, è quella a venire chiusa.
- `Send Data (Text)`: invia la stringa contenuta nel figlio `Text` sulla connessione.
- `Start Logging (Text)`: inizia il logging delle stringhe inviate e ricevute su un file di testo di cui si indica il path nel figlio `Text`.
- `Stop Logging`: interrompe il logging.

3.3.3 Lua scripting: canali con logica personalizzata

Lua è un linguaggio di scripting general-purpose, progettato pensando alla semplicità e all'estendibilità. Supporta solo pochi tipi di dati primitivi, e un'unica struttura dati composta, la *table* (una sorta di array associativo eterogeneo), che viene conseguentemente utilizzata estensivamente. Lua sceglie di fornire un piccolo insieme di funzionalità generiche - che possono essere estese per adeguarsi a diversi tipi di problemi - invece di definire una specifica più complessa e rigida, orientata verso uno specifico paradigma di programmazione. Queste scelte di design hanno fatto sì che l'implementazione di riferimento dell'interprete del bytecode Lua sia molto compatta (attualmente circa 150KB), una delle caratteristiche che lo rende interessante, insieme al tipo di licenza con cui è rilasciato e, soprattutto, alla semplice e robusta C API che lo accompagna. Questa ha reso infatti popolare l'*embedding* di Lua in altre applicazioni, che lo utilizzano tipicamente per fornire un meccanismo di estensibilità/programmabilità agli utenti.

E questo è anche il caso di Quest3D, che permette l'utilizzo dello scripting Lua tramite il channel `Lua script`, al quale si può collegare un numero illimitato di figli (senza restrizioni di tipo). La finestra delle proprietà di tale channel presenta un editor di testo pronto a ospitare il nostro codice Lua che, attraverso l'implementazione delle due funzioni `CallChannel` e `GetValue` (o almeno di una delle due), definisce la logica del nostro channel "personalizzato" via scripting.

La funzione `CallChannel` viene eseguita quando il channel viene chiamato, mentre `GetValue` permette di restituire un valore quando il channel è usato come figlio: il channel `Lua script` ha infatti `Value` come *BaseType*.

Va precisato che è disponibile solo un sottoinsieme delle funzioni standard di Lua, ovvero quello dei gruppi "Base", "String" e "Math", a cui si aggiungono alcune funzioni di un gruppo "q" di Quest3D.

L'utilizzo dello scripting Lua in Quest3D si rivela particolarmente utile per descrivere concisamente dei blocchi di logica non banali, soprattutto se comprendono strutture iterative, oltre che per processare blocchi di testo, per gestire il caricamento dinamico dei "channel groups" (anche da remoto, via HTTP) e altro.

Nel nostro caso, utilizzeremo uno script Lua per effettuare il parsing delle stringhe ricevute via socket da ISISgloveManager, e successivamente per realizzare uno "switch" con selettore basato stringhe.

Senza voler discutere delle caratteristiche generali di Lua, e tralasciando le funzioni relative al caricamento dinamico dei *channel groups*, vediamo le funzioni essenziali per far comunicare un channel `Lua script` con i figli.

La struttura predefinita `channel` permette di accedere al figlio collegato a una data posizione con `GetChild`:

```
local pf = channel.GetChild(0)
```

ad esempio, inizializza una variabile `pf`, che permetterà di interagire col primo figlio del channel usando funzioni getters/setters adeguate:

- `pf:GetValue()`
- `pf:SetValue(value)`

se si tratta di un figlio di *BaseType* `Value`;

- `pf:GetText()`
- `pf:SetText(text)`

se si tratta di un figlio di *BaseType* `Text`.

Similmente, si può invocare la funzione `CallChannel`:

- `pf:CallChannel()`

3.3.4 L'interfacciamento con ISISgloveManager

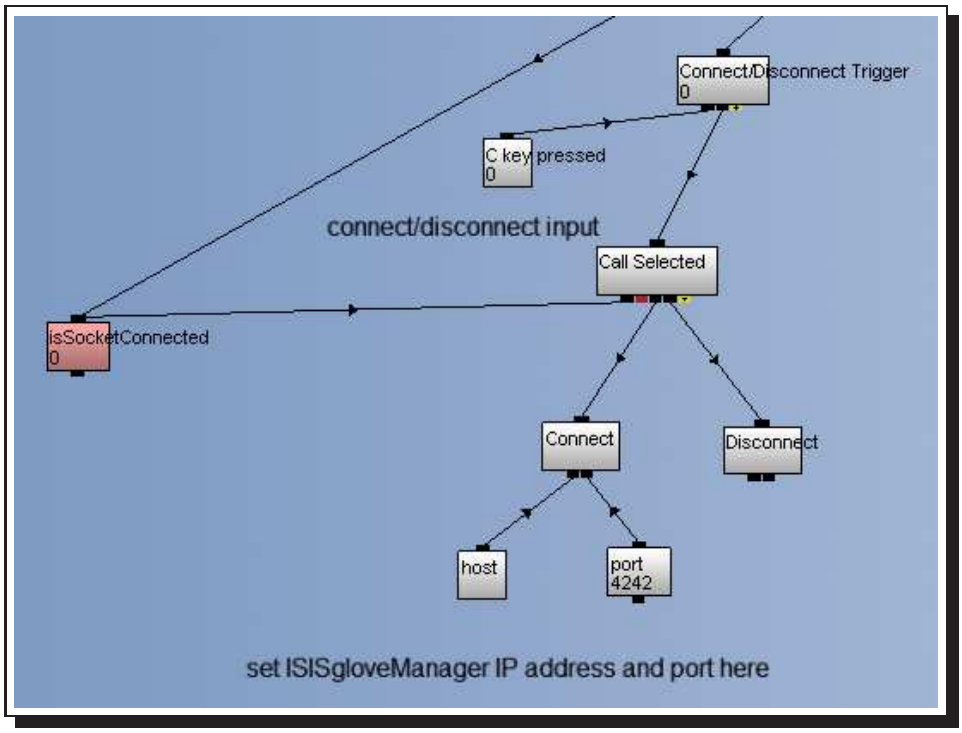
Il template Quest3D per l'interfacciamento con ISISgloveManager si può analizzare considerando, dall'alto verso il basso, tra aree distinte:

- connessione/disconnessione a ISISgloveManager

- conversione dei dati ricevuti in informazioni utilizzabili in Quest3D (valori scaled+eventuale stringa che indichi il riconoscimento di un gesto)
- chiamata di channels in funzione del riconoscimento di gesti

Andiamo a descrivere più in dettaglio l'implementazione del tutto, una sezione alla volta.

Connessione e disconnessione

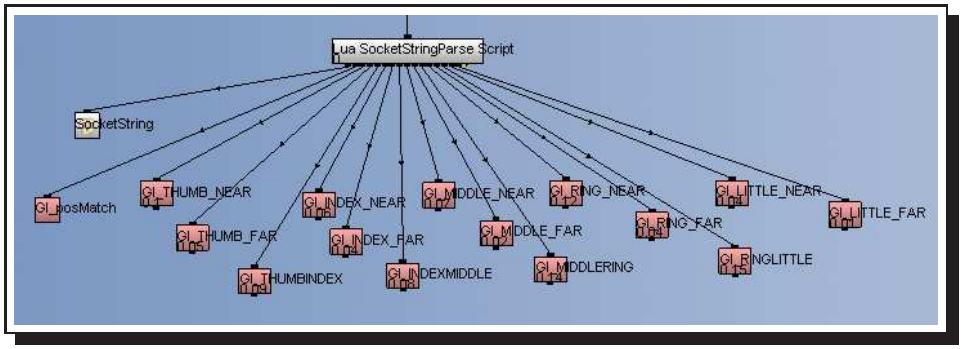


Il `Connect/Disconnect Trigger`, dato un input (di default la pressione del tasto 'C'), attiva/disattiva la connessione con ISISgloveManager.

Il channel `isSocketConnected`, pubblico (accessibile da altri *channel groups*), è un `SocketInfoValue` che indica se la connessione è attiva o no.

Nei figli del channel `Connect` vanno impostati host e porta di ISISgloveManager. I valori di default impostati nel template sono "127.0.0.1" e "4242".

Il parsing dei dati ricevuti via socket



E' questa la parte cruciale del template, basata sul channel `Lua Script` che effettua il parsing dei dati ottenuti dalla connessione TCP, `Lua SocketStringParse Script`. Il primo figlio costituisce l'input allo script, sotto forma delle stringhe che arrivano sulla connessione, tramite un channel `SocketString`, descritto in precedenza. Tutti gli altri figli del channel sono di output: il secondo, `GI_posMatch`, è un channel di tipo `Text` in cui vengono inserite le stringhe identificative dei gesti rilevati, impostate in ISISgloveManager come descritto in precedenza. Tutti gli altri figli sono di tipo `Value`, e ospitano i valori scalati in arrivo via socket. Se si è precedentemente utilizzato il supporto nativo di Quest3D al guanto, è sufficiente sostituire i channels `Glove Value` di cui si è accennato prima con questi `Value`, aggiornati continuamente dallo script Lua. Si noti che se mai dovesse servire avere in Quest3D i valori *raw*, si potrebbe immediatamente modificare lo script per averli in sostituzione o in aggiunta a quelli *scaled*;

Analizziamo le parti salienti dello script Lua che implementa la logica del channel, riportato per intero in appendice.

Lo script estrae dal channel di input la stringa in arrivo dal socket, che è nel formato:

```
SCL:(0.100515|0.0507042|0.0905612|0.0623053|0.0356436|0.0806846|
0.0664962|0.0186722|0.141553|0.118785|0.0377358|0.145349|0.042394|
0.0139721|0.2|0.3125|0.500122|0.500122)
RAW:(2436|2070|2591|2439|3220|3264|2586|2866|3527|3088|3609|3172|
2776|2607|4057|4053|2048|2048)
POS:manoRilassata
```

Ricordiamo che ISISgloveManager manda sempre una linea contenente i valori scaled (con prefisso "`SCL:`") e una contenente i valori raw (con prefisso

"RAW:", ignorati in questa implementazione). Quando viene riconosciuto un gesto, invia un'ulteriore linea, con prefisso "POS:".

Essendo le tre linee a lunghezza variabile, `string.find` si occupa di individuare le posizioni in cui iniziano, sfruttando i prefissi.

```
local sclStart = string.find (socketString, "SCL:",1,true)
local rawStart = string.find (socketString, "RAW:",1,true)
local posStart = string.find (socketString, "POS:",1,true)
```

Se non viene trovata l'occorrenza di "SCL:", vuol dire che la connessione non è attiva, e quindi l'esecuzione del channel termina.

```
if (sclStart == nil) then
    return -1
end
```

La linea "POS:" invece potrebbe esserci o meno, e il channel agisce di conseguenza: se non c'è, azzera il channel di output `GI_posMatch` (a cui si accede tramite la variabile `PositionMatchChannel`), altrimenti lo aggiorna al valore ricevuto.

```
if (posStart == nil) then
    posMatchString = ""
else
    local posEnd =  string.find(socketString,"\n",posStart+4,true)
    posMatchString = string.sub(socketString,posStart+4,posEnd-1)
end
PositionMatchChannel:SetText(posMatchString)
```

A questo punto la linea contenente i valori scalati viene divisa in token, che vengono convertiti in `float` e assegnati ai figli `Value`:

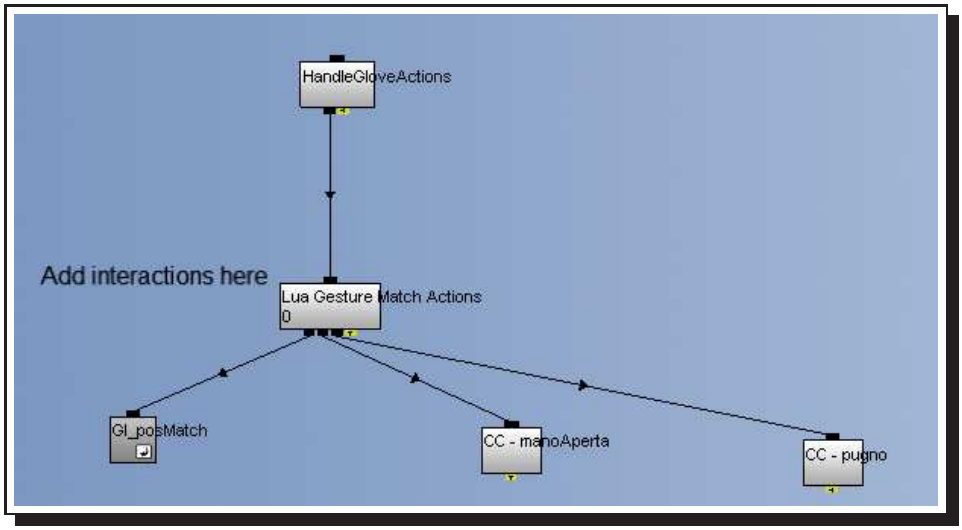
```
scaledValuesString = string.sub(socketString,sclStart,rawStart-1)
sclStart = sclStart + 5;  -- salta "SCL:("
local nextSep = string.find(scaledValuesString,"|",sclStart,true)
sens = 0
while (sens < 14) do
    str = string.sub(scaledValuesString,sclStart, nextSep-1)
    channel.GetChild(sens+2):SetValue(tonumber(str))

    sclStart = nextSep+1
```

```
    nextSep = string.find(scaledValuesString,"|",sclStart,true)

    sens = sens + 1
end
```

Definire azioni correlate al match con un gesto



Il blocco inferiore implementa una sorta di "switch" string-based: l'obiettivo è "chiamare" un determinato channel figlio quando viene rilevato un certo gesto. Tra i channels predefiniti di Quest3D, una funzionalità simile è implementata dal `ChannelSwitch`, che però obbliga ad usare come selettore un intero che indica il figlio da chiamare: abbiamo allora realizzato in Lua un breve script che, invece, consente di utilizzare la stringa nel channel `GI_posMatch` come selettore.

```
function CallChannel()
    local InputChannel = channel.GetChild(0)
    local posMatch = InputChannel:GetText()
    if posMatch == nil then
        return
    end
    posMatch_callChannel_map = {
        ["manoAperta"] = 1,
        ["pugno"]      = 2,
```

```
    }
    channelToCall = posMatch_callChannel_map[posMatch]
    if channelToCall ~= nil then
        channel.GetChild(channelToCall):CallChannel()
    end
end
```

Lo script sfrutta una *table* Lua per stabilire un mapping tra stringa identificativa del gesto e posizione del figlio da chiamare: nell'utilizzo del template è sufficiente collegare al channel dei figli (tipicamente `ChannelCaller`) a piacimento e aggiornare di conseguenza `posMatch_callChannel_map`.

Si noti che questo approccio mantiene tutta la generalità possibile: i `ChannelCaller` usati come figli potrebbero implementare direttamente delle azioni, ma anche essere usati a loro volta come input a strutture di channels di complessità arbitraria che, unendo l'informazione sul gesto riconosciuto ad altri dati (ad esempio lo spostamento della mano, rilevato col dispositivo di tracking, o la prossimità di un oggetto nell'ambiente virtuale), implementano azioni complesse non direttamente legate al riconoscimento di un gesto.

Inserire nell'editor il template sviluppato

Per capire come funzioni il meccanismo dei *template*, va spiegato che l'entità base persistente trattata da Quest3D è il "channel group". Un progetto è costituito da uno o più channel groups collegati tra loro e in cui c'è un unico punto di partenza, da cui parte la visita del grafo. Un "channel group" viene salvato in una coppia di files ".cgr" e ".igr", dei quali uno contiene i dati veri e propri del gruppo di canali e l'altro (il file ".igr") contiene informazioni utilizzate unicamente dall'editor, come le posizioni dei "rettangolini", i testi di commento e altre informazioni.

La directory "Template", contenuta nella directory di installazione di Quest3D, contiene proprio dei *channel groups*, organizzati in directory e subdirectory a piacimento: l'editor riprodurrà fedelmente la struttura ad albero del file system nel tab "template" (figura 3.10), da cui si trascinano i *channel groups* nella composizione della scena.

Nel nostro caso abbiamo quindi aggiunto una categoria "ISISlab" per ospitare i channel groups relativi alle componenti da noi sviluppate (quella di interfacciamento a ISISgloveManager, di cui si è appena parlato, e quella per l'utilizzo dell'InterSense PCTracker, che coinvolge anche un channel sviluppato ad-hoc, descritta a breve).

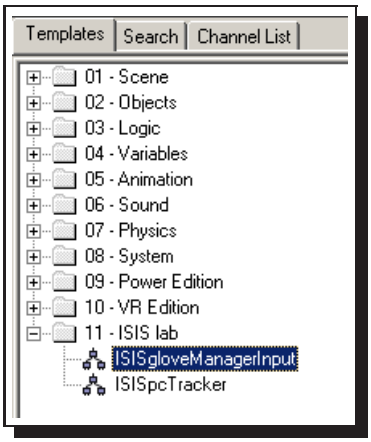


Figura 3.10: L'albero dei template

3.3.5 Sviluppi futuri

La realizzazione di questa componente è strettamente collegata a quella di ISISgloveManager, e nello specifico nella definizione del protocollo di trasmissione dei dati su socket. Nonostante il parsing (un po' macchinoso) implementato via scripting, non sono stati rilevati problemi di prestazioni: in ogni caso, se servisse, si potrebbe realizzare un'implementazione più efficiente creando un *custom channel* in C++, similmente a quanto verrà fatto nel capitolo seguente.

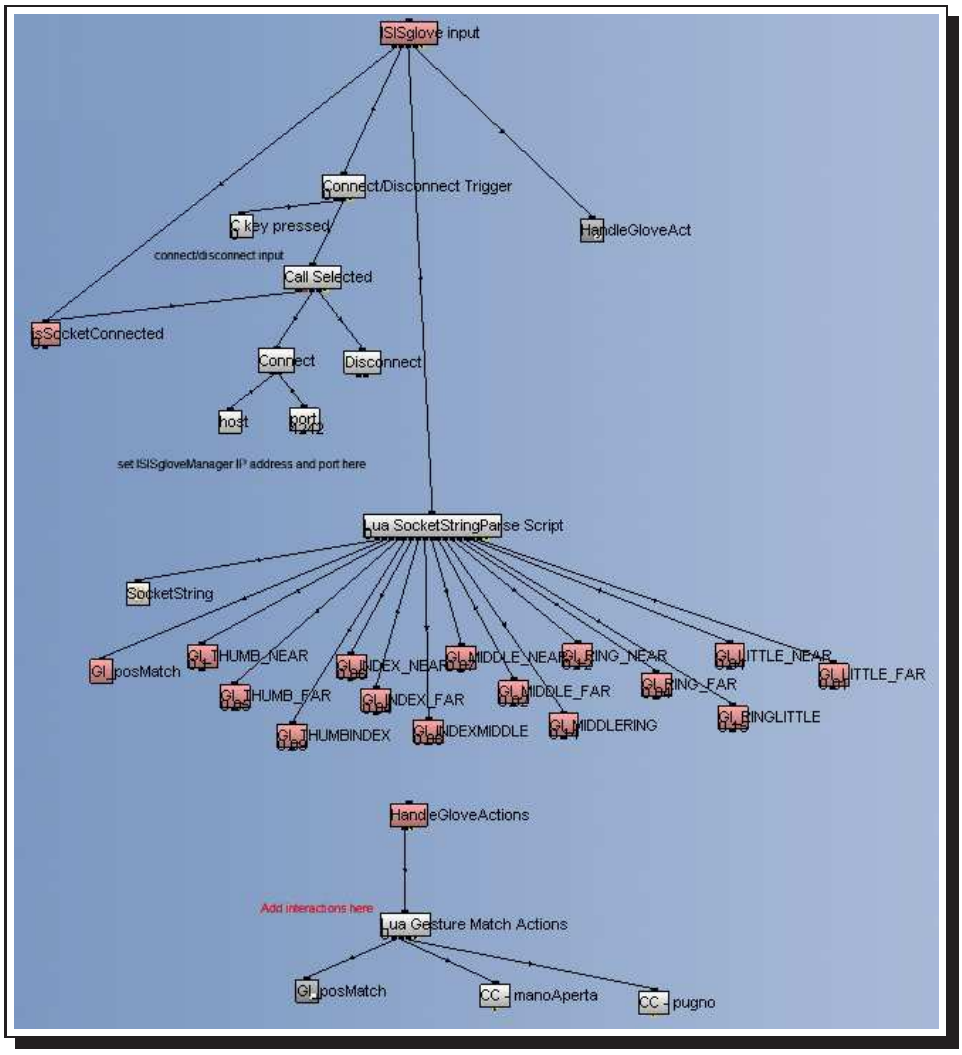


Figura 3.11: Il template di interfacciamento con ISISgloveManager: visione d'insieme.

3.4 ISISpcTracker

Descriviamo come e perché è stato implementato il supporto al nostro dispositivo di motion tracking, spaziando dall'SDK fornita con l'hardware alle tecniche relative all'implementazione di *custom channel* Quest3D.

3.4.1 Il supporto nativo ai trackers *InterSense*, e perché sostituirlo

L'edizione *VR* di *Quest3D* include il supporto ad alcuni dei più popolari dispositivi di *motion tracking*. Tra questi, anche un trio di channels per il supporto generico ai trackers della *InterSense*.

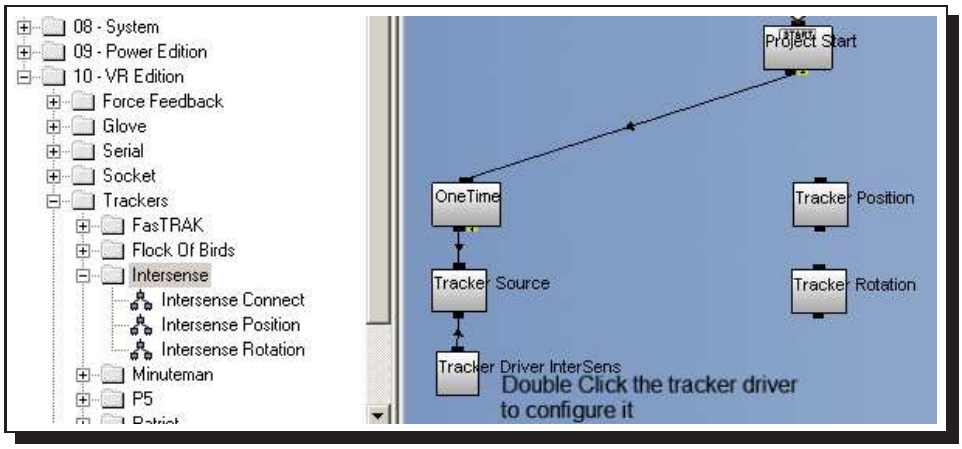


Figura 3.12: I channels per il supporto dei trackers InterSense.

In una prima fase, questi sono stati usati per ottenere i dati della stazione *MiniTrax* "Head" allacciata al Data Glove, e i vettori posizione/orientamento ottenuti dal dispositivo (tramite gli appositi channels `Intersense Position` e `Intersense Rotation`) sono stati usati per muovere nello spazio il "polso" della mano scheletrica 3D usata, raggiungendo un obiettivo fondamentale per l'effettiva usabilità del sistema (provate a compiere una qualsiasi azione nell'ambiente che vi circonda col braccio legato al busto, potendo muovere solo le dita...).

Il passo successivo da compiere era fornire un modo comodo di spostarsi nell'ambiente virtuale con una *walkthrough camera*. Con a disposizione mouse e tastiera, lo standard di fatto per questo tipo di navigazione, reso popolare dai videogiochi "first person shooter", è la combinazione dei tasti W-

A-S-D per spostarsi avanti/indietro e a sinistra/destra, usati con la mano sinistra, e del mouse per dirigere la visuale (e quindi fissare la direzione "avanti"), usato con la mano destra.

Nella nostra interfaccia, però, la mano destra è impegnata col guanto, e con a disposizione solo la sinistra il problema si potrebbe risolvere solo parzialmente (ad esempio: usando il mouse per spostare la visuale come di consueto, e usando i bottoni sinistro/destro in sostituzione dei tasti W/S, per camminare avanti/indietro). Ma oltre al fatto che in questo modo l'interfaccia si impoverisce (non c'e' modo di effettuare lo "strafe"), va considerato che non siamo abituati ad usare il mouse con la mano sinistra e che, in caso si usi il guanto all'impiedi di fronte a un megaschermo, avendo massima liberta di movimento per il braccio, accedere comodamente a un mouse con la sinistra potrebbe non essere immediato.

Fortunatamente la seconda stazione *MiniTrax* a nostra disposizione, la "Wand" dotata di joystick e bottoni, si presta decisamente a risolvere il problema. Orientare la stazione nello spazio è qualcosa che viene molto naturale per spostare lo "sguardo", eliminando la necessità del mouse, mentre il joystick va a sostituire i tasti W-A-S-D. Trattandosi di un input analogico su due assi, abbiamo anche potenzialmente qualcosa in più: mentre i tasti possono essere solo premuti o meno, abbiamo qui un valore nell'intervallo 0-255 per ogni asse, utilizzabile per cambiare la velocità con cui ci si sposta. Inoltre i 5 bottoni disponibili possono essere usati, a seconda dell'applicazione, come modificatori (correre, volare), come trigger di azioni relative sempre allo spostamento (saltare, chinarsi), o, se si tratta di un gioco, sparare (sarà un caso che uno dei 5 bottoni della stazione è posizionato come un grilletto?).

Non c'è stato modo, però, complice la mancanza di documentazione del channel `Intersense Connect`, di utilizzare due "terzetti" di canali separati per riuscire ad accedere ai dati della seconda stazione collegata all'*hub PC-Tracker*. Se anche fosse stato possibile, tra l'altro, le informazioni prelevabili dai channels "ufficiali" sono solo quelle relative a posizione e orientamento, mentre era essenziale poter utilizzare il joystick della "Wand": l'unica strada percorribile era quella della creazione di un "channel" ad-hoc che provvedesse a supportare adeguatamente il nostro hardware.

Nota: prima di affrontare lo sviluppo di un channel, si è provato ad utilizzare un particolare driver fornito dalla InterSense, che permette di configurare nella sezione "Periferiche di gioco", sotto piattaforma Windows, un joystick "virtuale" associato a una stazione di tracking. Il driver, che comunque avrebbe fornito una soluzione solo parziale, ha reso inutilizzabili (a causa di frequenti reboot) i due sistemi su cui è stato provato, e si è per

forza di cose deciso di ignorarne l'esistenza.

3.4.2 Cenni sull'SDK dell'Intersense PCTracker

L'API fornisce un vasto set di funzioni per utilizzare i dispositivi *InterSense*. L'accesso ai trackers è gestito tramite la libreria a collegamento dinamico *isense.dll*, che implementa un'interfaccia generica a tutti modelli prodotti dall'azienda, supportando fino ad 8 trackers e permettendo di gestirne anche la configurazione.

Non abbiamo bisogno di agire sulla configurazione del tracker dall'interno di *Quest3D*, quindi ci limiteremo ad analizzare le funzioni e le strutture dati necessarie a prelevare gli input dalle due stazioni *MiniTrax*.

Un ottimo punto di partenza è il programma di esempio fornito dalla *InterSense*:

```
void main() {
    ISD_TRACKER_HANDLE handle;
    ISD_TRACKER_INFO_TYPE tracker;
    ISD_TRACKER_DATA_TYPE data;
    handle = ISD_OpenTracker( NULL, 0, FALSE, FALSE );
    if( handle > 0 )
        printf( "\n Az El Rl X Y Z \n" );
    else
        printf( "Tracker not found. Press any key to exit" );
    while( !kbhit() ) {
        if( handle > 0 ) {
            ISD_GetData( handle, &data );
            printf( "%7.2f %7.2f %7.2f %7.3f %7.3f %7.3f ",
                data.Station[0].Orientation[0],
                data.Station[0].Orientation[1],
                data.Station[0].Orientation[2],
                data.Station[0].Position[0],
                data.Station[0].Position[1],
                data.Station[0].Position[2] );
                ISD_GetCommInfo( handle, &tracker );
            printf( "%5.2fKbps %d Records/s \r",
                tracker.KBitsPerSec, tracker.RecordsPerSec );
        }
        Sleep( 6 );
    }
```

```
    ISD_CloseTracker( handle );
}
```

Questo semplice codice inizializza il dispositivo e preleva in loop, visualizzandole su schermo, le informazioni di orientamento e posizionamento dalla prima stazione (le stesse fornite dai channels "base" *Quest3D*).

Per la realizzazione del nostro channel sarà quindi necessario fare qualcosa di simile, adattato al flusso di esecuzione di *Quest3D* e per entrambe le stazioni.

Esaminiamo quindi le funzioni e le strutture dati coinvolte.

- `ISD_TRACKER_HANDLE ISD_OpenTracker(`
  `HWND hParent,`
  `DWORD commPort,`
  `Bool infoScreen,`
  `Bool verbose )`

  `hParent`: handle della finestra "padre". Parametro opzionale e pensato per sviluppi futuri del software *InterSense*. Attualmente va quindi passato NULL.

  `commPort`: se diverso da 0, forza l'individuazione di un tracker sulla porta RS232 specificata. Altrimenti, più semplicemente, esegue uno scan prima per dispositivi USB e poi su tutte le porte seriali, a diversi baud rate. Se non c'è motivo di fare altrimenti, conviene quindi passare 0 per avere flessibilità massima.

  `infoScreen`: passare TRUE dovrebbe visualizzare una finestra di informazioni sullo stato del dispositivo, ma si tratta di una feature non ancora implementata: attualmente tali info vengono scritte dalla DLL in una console. Normalmente conviene quindi passare FALSE.

  `verbose`: passare TRUE aumenta i dettagli riportati in console dalla DLL.

  Nella sua forma più semplice l'accesso al device (sotto forma di una struttura `ISD_TRACKER_HANDLE`) si ottiene quindi con una chiamata

  ```
  handle = ISD_OpenTracker( NULL, 0, FALSE, FALSE );
  ```

- `Bool ISD_CloseTracker( ISD_TRACKER_HANDLE handle )`

  De-inizializza il tracker, chiude la comunicazione e libera le risorse associate al dispositivo. Il parametro `handle` è quello ottenuto da una precedente chiamata di `ISD_OpenTracker`. Se c'è più di un tracker aperto, passare 0 come parametro `handle` li chiude tutti. Restituisce FALSE in caso di fallimento.

- ```
  Bool ISD_GetData(
      ISD_TRACKER_HANDLE handle,
      ISD_TRACKER_DATA_TYPE *data )
  ```

  Preleva i dati dalle stazioni del tracker `handle`, scrivendoli nella struttura `data` passata.

Il tipo `ISD_TRACKER_HANDLE` non è che un intero, mentre analizzare la struttura `ISD_TRACKER_DATA_TYPE` è essenziale per prelevare correttamente le informazioni di cui abbiamo bisogno.

`ISD_TRACKER_DATA_TYPE` contiene un array (`Station`) di strutture `ISD_STATION_STATE_TYPE`, ognuna delle quali rappresenta le informazioni relative a una stazione *MiniTrax*.

- ```
  typedef struct  {
      ISD_STATION_STATE_TYPE Station[ISD_MAX_STATIONS];
  } ISD_TRACKER_DATA_TYPE;
  ```

- ```
  typedef struct  {
      BYTE TrackingStatus;
      BYTE NewData;
      BYTE CommIntegrity;
      BYTE bReserved3;
      float Orientation[4];
      float Position[3];
      float TimeStamp;
      Bool ButtonState[MAX_NUM_BUTTONS];
      short AnalogData[MAX_ANALOG_CHANNELS];
      BYTE AuxInputs[ISD_MAX_AUX_INPUTS];
      LONG lReserved2;
      LONG lReserved3;
      LONG lReserved4;
      DWORD dwReserved1;
      DWORD dwReserved2;
  ```
  ```

```
    DWORD dwReserved3;
    DWORD dwReserved4;
    float fReserved1;
    float fReserved2;
    float fReserved3;
    float fReserved4;
} ISD_STATION_STATE_TYPE;
```

`TrackingStatus`: E' un valore da 0 a 255 che rappresenta la qualità corrente del tracking.

`NewData`: Un flag TRUE se si tratta di nuovi dati, modificato dalle chiamate `ISD_GetData`

`CommIntegrity`: Integrità della comunicazione, in caso di dispositivi con collegamento wireless.

`Orientation`: Orientamento in coordinate sferiche (*"Euler form"*, in gradi) oppure in forma di quaternione (Per consentire questa seconda rappresentazione è un array di 4 `float` e non di 3), a seconda dell'impostazione del flag `AngleFormat` nella struttura `ISD_STATION_INFO_TYPE`. Tale struttura, che evitiamo di descrivere in dettaglio, contiene le informazioni sulla configurazione di una stazione *MiniTrax*.

`Position`: La posizione della stazione *MiniTrax*, in metri.

`TimeStamp`: Se richiesto, in secondi.

`ButtonState`: Un array contenente lo stato dei bottoni, se presenti.

`AnalogData`: Un array per gli input analogici dalle stazioni. Attualmente, sono in uso solo due valori, per gli assi x/y del joystick presente sulla stazione di tipo "Wand". Il range dei valori è 0-255 (a joystick libero si hanno in input i valori 127,127).

`AuxInputs`: Relativo a input ausiliari connessi alle porte I2C di alcune stazioni *MiniTrax* (poco documentato).

3.4.3 Estendere *Quest3D*: la creazione di un channel

Un channel, "dietro le scene", è una DLL. Installato l'SDK di *Quest3D* e configurato adeguatamente l'ambiente di sviluppo, nel nostro caso *Visual C++ 2005 Express Edition*, è possibile ottenere dei channels personalizzati perfettamente integrabili nel modello di editing di *Quest3D*.

Per integrare correttamente i nostri channel in Quest3D, è importante capire come l'editor accede ai channels, cosa che coinvolge due subdirectory della directory di installazione del programma: "Channels" e "3rd".

La directory "Channels" contiene, abbastanza ovviamente, le DLL che implementano i channels. L'editor prepara la lista dei channels disponibili scorrendo, all'avvio, questa directory. La directory "3rd" è pensata per ospitare DLL di terze parti. Le DLL dei channels possono dipendere da DLL presenti in questa directory, come spiegheremo a breve.

Per far sì che *Quest3D* possa inizializzare e collegare correttamente un channel, bisogna che questo sia strutturato in un determinato modo: si deriva quindi la propria classe da `A3d_Channel`, classe che definisce le funzioni base per soddisfare i requisiti dell'engine.

Coerentemente col meccanismo *Type/BaseType* illustrato precedentemente, si può anche creare un channel derivando da un altro channel preesistente (fig. 3.13).

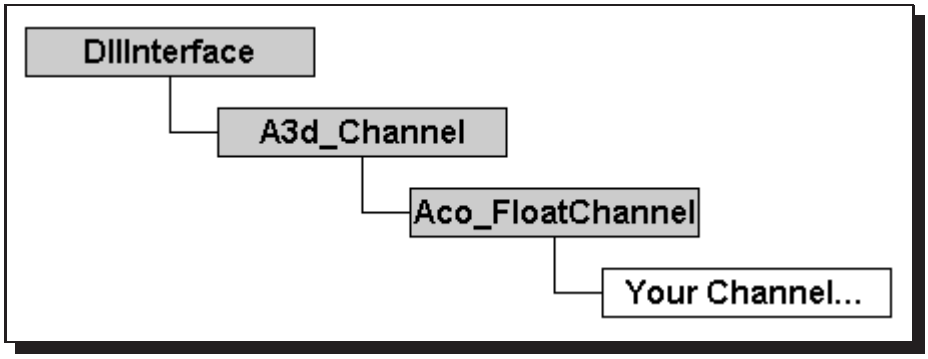


Figura 3.13: la classe `Aco_FloatChannel` corrisponde nell'editor al BaseType "Value".

Va accennato che ogni channel, per essere gestito correttamente dall'editor, deve avere un GUID univoco, ma questo e altri dettagli sul linking della DLL possono essere ignorati usando i template forniti con l'SDK, preccupandosi soltanto di sostituire nell'header dell'esempio preso come punto di partenza un GUID da noi generato (col tool `guidgen.exe` della *Microsoft*).

Essenziale è invece comprendere come il nostro channel si integra nel flusso di esecuzione di *Quest3D*.

Il costruttore della classe viene eseguito quando si trascina il channel dalla lista dei componenti alla regione di editing, ed è lui che deve definire le caratteristiche del componente, specificando ad esempio quali *link square*

avrà il rettangolino che rappresenta il channel nell'editor. Tale costruttore viene anche utilizzato in fase di inizializzazione dei files eseguibili pubblicati, e deve quindi provvedere all'allocazione di eventuali risorse (che andranno poi rilasciate nel distruttore, pena il possibile crash dell'engine).

La logica vera e propria del channel, ovvero il codice che viene eseguito ogni volta che il channel viene chiamato, è quello della funzione `CallChannel`, di cui va fatto l'overloading.

Va inoltre dichiarata la lista di dipendenze del channel, tramite la funzione `DoDependencyInit`. Tipicamente in tale funzione sono presenti una serie di chiamate alle funzioni `AddChannelGuidDepend` e `AddDLLDepend`, che permettono rispettivamente di definire dipendenze da altri channels e da DLL di terze parti.

Altre funzioni non utilizzate nell'implementazione del nostro channel, ma che meritano di essere accennate, sono `SaveChannel` e `LoadChannel`, che permettono di salvare e caricare dei dati (tipicamente delle impostazioni di configurazione) con un meccanismo ben integrato nell'engine, che incorporerà i dati nei files ".cgr" stessi. Se infatti si prova ad utilizzare un proprio meccanismo di persistenza alternativo, basato su files esterni, va considerato che l'operazione di publish del progetto non saprà come gestirli, e quindi tali dati non verranno incorporati e caricati dai channels come ci si potrebbe aspettare.

All'interno dell'editor, alcuni channels hanno una finestra di "proprietà" per manipolarne la configurazione: anche queste "channel dialog", come sono chiamate nella documentazione dell'SDK, si possono realizzare ed associare ai propri custom channel.

3.4.4 Implementare il supporto all'*Intersense PC Tracker*

Studiate le funzioni basilari dell'API dell'*InterSense* e dell'SDK di *Quest3D*, possimo passare alla realizzazione del channel `IntersensePCTracker` e del template su esso basato, **ISISpcTracker**.

Ecco un estratto del file `IntersensePCtracker.h`:

```
#define MYCHANNEL_NAME  "IntersensePCtracker"
#define CHILDS_N    19
#define TRACKER_HAND    0
#define TRACKER_WAND    1

class MYCHANNEL_API IntersensePCtracker: public A3d_Channel {
    public:
```

```
        IntersensePCtracker();
        virtual ~IntersensePCtracker();
        virtual void DoDependencyInit(A3d_List* currentDependList);
        virtual void CallChannel();

    protected:
        ISD_TRACKER_HANDLE       handle;
        ISD_TRACKER_DATA_TYPE    data;
};
```

Variabili di istanza del channel (con scope `protected`, come suggerito nelle coding guidelines di *Quest3d*) sono le strutture necessarie a prelevare i dati dalle stazioni *MiniTrax*, come discusso in precedenza. Quando "chiamato", il channel che andiamo a creare aggiornerà i valori dei suoi figli con tali dati. L'input dalle stazioni *MiniTrax* consiste in quattro vettori di tre `float` (posizione e orientamento), due valori interi nel range 0-255, relativi agli assi del joystick, e una serie di 5 valori booleani che indicano lo stato dei bottoni. In *Quest3D* la tipizzazione dei valori non è così articolata, e usiamo in genere dei channels di tipo `Value`. Un channel di tipo `Value` incapsula un `float`, ed è implementato dalla classe `Aco_FloatChannel`, caratterizzata delle funzioni `GetFloat/GetOldFloat/SetFloat`.

Il channel `IntersePCtracker` comunica quindi con un totale di 19 channel figli (3*4 + 2 + 5) di tipo `Value`. Tale numero, un pò alto rispetto all'usuale, avrebbe potuto essere abbassato a 11 creando 4 figli di tipo `Vector` al posto dei 12 `Value` separati. Ciò non è stato fatto per due motivi ben precisi:

1. spesso è capitato di dover applicare delle trasformazioni solo ad alcune componenti dei vettori (ad esempio invertendo un asse), e in tal caso avere un oggetto `Vector` e non le componenti separate obbliga a un passaggio in più.

2. il `Vector` di Quest3D è basato sul tipo `D3DXVECTOR3`, quindi per utilizzarlo è necessario installare e aggiungere alle dipendenze del progetto anche l'SDK delle *DirectX*.

Inoltre, le tre componenti dei vettori possono essere accorpate facilmente a livello di Template (infatti è stato fatto), e quindi, a scapito dell'ordine, otteniamo praticità nell'editing e nel mantenimento del codice.

Analizziamo quindi costruttore, distruttore e funzione `CallChannel` implementate.

```
IntersensePCtracker::IntersensePCtracker() {
    SetChannelName(MYCHANNEL_NAME);

    handle = ISD_OpenTracker( NULL, 0, FALSE, FALSE );
    if(handle <= 0) {
        ueMsg("Cannot open Intersense PCTracker");
        for (int c=0; c<3; c++) {
            data.Station[TRACKER_HAND].Position[c] = 0;
            data.Station[TRACKER_HAND].Orientation[c] = 0;
            data.Station[TRACKER_WAND].Position[c] = 0;
            data.Station[TRACKER_WAND].Orientation[c] = 0;
        }
        data.Station[TRACKER_WAND].AnalogData[0] = 127;
        data.Station[TRACKER_WAND].AnalogData[1] = 127;
        for (int c=0; c<6; c++)
            data.Station[TRACKER_WAND].ButtonState[c] = -1;
    }

    SetChildCreationCount(0);
    ChildCreation child;
    child.channelType.guid = FLOAT_CHANNEL_GUID;
    strcpy(child.channelType.name, FLOAT_CHANNEL_NAME);
    child.initialize = true; // crea automaticamente i channel figli

    char *channelnames[] = {
        "Hand Pos X", "Hand Pos Y", "Hand Pos Z",
        "Hand Rot X", "Hand Rot Y", "Hand Rot Z",
        "Wand Pos X", "Wand Pos Y", "Wand Pos Z",
        "Wand Rot X", "Wand Rot Y", "Wand Rot Z",
        "Joystick X", "Joystick Y",
        "button 1", "button 2", "button 3", "button 4", "button 5"
    };

    for (int cnum =  0; cnum < CHILDS_N; cnum++) {
        child.requestLink = cnum;
        strcpy(child.name,channelnames[cnum]);
        SetChildCreateType(child, cnum);
    }
}
```

Essenzialmente il costruttore ottiene l'handle necessario ad accedere al tracker, e poi crea i 19 link square di tipo Value e i relativi figli. Se ci sono problemi nell'accesso al tracker (ad esempio se questo è scollegato) il costruttore si preoccupa a segnalare il problema nell'editor, con la funzione di debugging `ueMsg`, e riempie la struttura "`data`" di valori che permettono di rilevare la condizione di errore all'interno flusso di esecuzione del progetto: si sono scelti valori neutri per posizione, orientamento e assi del joystick, mentre assegnare "-1" allo stato dei bottoni, che può normalmente essere soltanto 0 o 1, indica esplicitamente la condizione di errore. Nel template ISISpcTracker è inclusa un'espressione `isTrackerActive` che controlla tale valore e fornisce agli utilizzatori un valore 0/1 che indica lo stato del dispositivo.

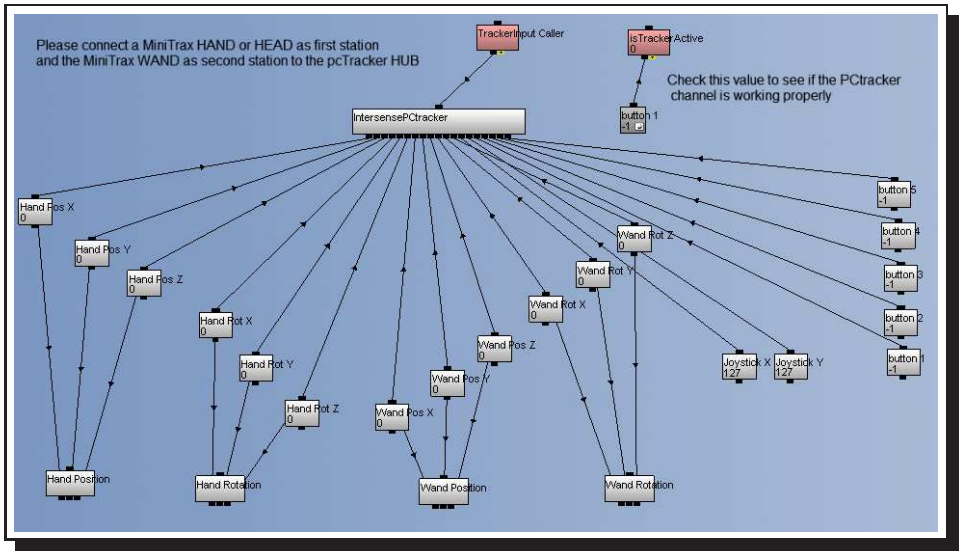


Figura 3.14: Un'immagine del template ISISpcTracker, contenente il nostro channel.

La creazione dei *link square*, aspetto fondamentale della caratterizzazione del channel, merita di essere discussa in dettaglio.

Nella definizione dei *link squares* sono coinvolte la struttura `ChildCreation` e le funzioni `SetChildCreationCount` e `SetChildCreateType`.

`SetChildCreationCount(0)` va usato prima di aggiungere i link squares con `SetChildCreateType`, e va omesso se si sta creando un channel derivato da un altro, a meno che non si voglia evitare di ereditare i link squares dal channel "padre".

Più interessanti `SetChildCreateType` e la la relativa struttura `ChildCreation`:

- `void SetChildCreateType(ChildCreation child, int createNr)`

  crea un link square in posizione `createNr`, in base alla descrizione passata nella struttura `ChildCreation` "child", che è quindi quella che permette di definirne le proprietà.

- ```
  struct ChildCreation {
       ChannelType channelType;
       int requestLink;
       bool linked;
       bool initialize;
       char name[100];
       GUID channelUpgradeGUID;
       ChildInfo* childInfo;
  };
  ```

  `channelType`: è una struttura che definisce il tipo di channel che si potrà collegare al link square. Tra le altre cose, contiene il nome del tipo e il GUID univoco che definisce il tipo del canale, oltre che, eventualmente, il GUID di un BaseType e informazioni sul versioning.

  ```
  struct ChannelType {
       char name[80];
       GUID guid;
       GUID baseguid;
       int mversion;
       int miversion;
       int version;
       DWORD pluginType;
  };
  ```

  `requestLink`: posizione del link, o -1 se si tratta di un child link "growing" e, quindi, potenzialmente multiplo.

  `linked`: attualmente non usato.

  `initialize`: se settato a TRUE, crea anche un channel figlio (e non solo il *link square*) di tipo adeguato.

  `name`: una NULL-terminated string dove inserire il nome del *link square*. La documentazione raccomanda che sia di lunghezza minore o uguale a 60 caratteri (99?)
  ```

`channelUpgradeGUID`: se initialize è TRUE, si può impostare qui un GUID per indicare il tipo del figlio da creare, se questo deve essere diverso da quello impostato in `channelType.guid`.

`childInfo`: un puntatore a una struttura `ChildInfo` contenente informazioni addizionali sul channel collegato come figlio in un determinato momento. La struttura è gestita da *Quest3D* e normalmente non dovrebbe essere necessario accedervi.

Nel nostro caso, i *link square* si differenziano solo per nome e posizione, quindi definiamo una volta per tutte le proprietà comuni

```
ChildCreation child;
child.channelType.guid = FLOAT_CHANNEL_GUID;
strcpy(child.channelType.name, FLOAT_CHANNEL_NAME);
child.initialize = true; // crea automaticamente i figli
```

...definiamo in `channelnames` i nomi che dovranno avere nell'editor, e poi li creiamo (insieme ai relativi figli, grazie a "`initialize`") con un semplice ciclo.

```
for (int cnum =  0; cnum < CHILDS_N; cnum++) {
    child.requestLink = cnum;
    strcpy(child.name,channelnames[cnum]);
    SetChildCreateType(child, cnum);
}
```

E' da dire che il nostro caso è decisamente semplice, poichè creiamo 19 link squares dello stesso tipo e che permettono di collegarvi un unico figlio. Va infatti almeno accennato che si possono definire anche *link square* senza restrizione di tipo e/o che consentono di collegare un numero indeterminato di channels. Ad esempio l'essenziale channel `Render` ha un link square singolo che accetta il tipo `Camera`, e due link square "growing" che permettono rispettivamente di collegare un numero indeterminato di `3D Object` e `Light` (figura 3.15).

Un channel in cui numero e tipo di figli collegati non è noto a priori va trattato con maggiore attenzione, utilizzando nella funzione `CallChannel` apposite funzioni per recuperare dinamicamente le informazioni necessarie a interagire correttamente con i figli (`GetChildLinkPositionCount`, `GetChildFromLinkPosition`...).

Il distruttore non ha bisogno di commenti:

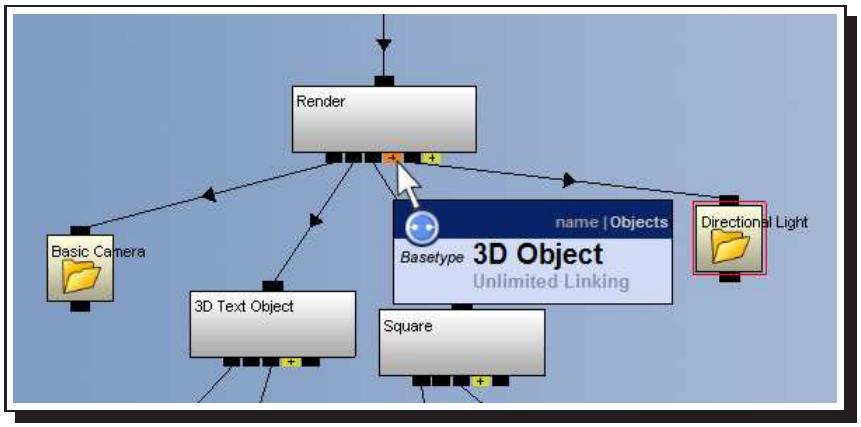


Figura 3.15: un channel `Render`: notare la dicitura "Unlimited linking" e il *link square* col "+": collegandovi un channel, sarà al suo posto inserito un nuovo *link square*, e quello col "+" slitterà a destra per il prossimo collegamento...

```
IntersensePCtracker::~IntersensePCtracker() {
    if (handle>0) ISD_CloseTracker( handle );
}
```

Molto semplice anche la funzione `CallChannel`, che preleva i dati dalle stazioni con `ISD_GetData`, come descritto precedentemente, e imposta di conseguenza i valori dei figli creati dal costruttore. Notare che se l'accesso al tracker è fallito (`handle<=0`) `ISD_GetData` non viene chiamata, e nei figli vengono assegnati i valori che indicano la condizione di errore, impostati nel costruttore. Altra piccola nota va fatta per quanto riguarda i valori che indicano l'orientamento delle stazioni: mentre il tracker fornisce dei valori in gradi (range -180/180), Quest3D lavora col range 3.14/-3.14, quindi va effettuata la semplice conversione visibile alle righe 16 e 23.

```
1   void IntersensePCtracker::CallChannel() {
2     Aco_FloatChannel* floatChannel=NULL;
3
4     if (handle > 0 )
5       ISD_GetData( handle, &data );
6
7     for (int cnum =  0; cnum < CHILDS_N; cnum++) {
8       floatChannel = (Aco_FloatChannel*)GetChild(cnum);
9       if (floatChannel != NULL){
```

```
10          switch(cnum) {
11           case 0: case 1: case 2:
12            floatChannel->SetFloat(data.Station[TRACKER_HAND].Position[cnum] );
13            break;
14           case 3: case 4: case 5:
15            floatChannel->SetFloat(
16                a.Station[TRACKER_HAND].Orientation[cnum-3]/180*3.14 );
17            break;
18           case 6: case 7: case 8:
19            floatChannel->SetFloat( data.Station[TRACKER_WAND].Position[cnum-6] );
20            break;
21           case 9: case 10: case 11:
22            floatChannel->SetFloat(
23                data.Station[TRACKER_WAND].Orientation[cnum-9]/180*3.14 );
24            break;
25           case 12:
26            floatChannel->SetFloat( data.Station[TRACKER_WAND].AnalogData[0] );
27            break;
28           case 13:
29            floatChannel->SetFloat( data.Station[TRACKER_WAND].AnalogData[1] );
30            break;
31           case 14: floatChannel->SetFloat(data.Station[TRACKER_WAND].ButtonState[1]);
32            break;
33           case 15: floatChannel->SetFloat(data.Station[TRACKER_WAND].ButtonState[0]);
34            break;
35           case 16: floatChannel->SetFloat(data.Station[TRACKER_WAND].ButtonState[2]);
36            break;
37           case 17: floatChannel->SetFloat(data.Station[TRACKER_WAND].ButtonState[3]);
38            break;
39           case 18: floatChannel->SetFloat(data.Station[TRACKER_WAND].ButtonState[5]);
40            break;
41         } // switch
42        }  // if
43      }    // for
44    }
```

Ricordiamo che l'accesso ai dispositivi di tracking avviene tramite *isense.dll*, quindi questa va dichiarata in `DoDependencyInit` e va copiata nella directory "3rd".

```
void IntersensePCtracker::DoDependencyInit(A3d_List* currentDependList) {
    AddChannelGuidDepend(MYCHANNEL_GUID, currentDependList);
    AddDLLDepend(``isense.dll'', currentDependList); //dipendenza InterSense
}
```

Capitolo 4

SkeleTronDemo

On the other side of the screen, it all looks so easy.
da *Tron*

SkeletronDemo ha l'obiettivo di presentare l'uso combinato delle componenti sviluppate all'interno di applicazione dimostrativa Quest3D.

- il guanto viene utilizzato per animare il modello 3D di una mano scheletrica
- le stazioni di tracking vengono utilizzate per la gestione della telecamera nell'esplorazione dell'ambiente e per il posizionamento della mano
- il riconoscimento di un gesto combinato all'orientamento della mano vengono usati per implementare un'interazione d'esempio

4.1 La mano virtuale

Il primo passo da compiere nel preparare il nostro demo è proiettare nella scena la mano che indossa il wired glove (con allacciata la prima stazione di tracking).

Occupiamoci prima dell'animazione della mano e poi del suo posizionamento nella scena.

4.1.1 Animare un modello di mano scheletrica

Uno degli esempi forniti con Quest3D VR Edition, allo scopo di mostrare come utilizzare il supporto built-in ai 5DT Data Glove, consiste proprio in

un modello 3d dello scheletro di una mano animato tramite il guanto. Per renderlo utilizzabile col maggior numero possibile di modelli, il demo fornito utilizza però soltanto 5 sensori (un valore di flessione per ogni dito), quindi lo si è dovuto modificare ed espandere in modo da sfruttare tutti i 14 sensori a nostra disposizione.

Ovviamente il supporto built-in è stato sostituito col template ISISgloveInput descritto in precedenza.

Animare un'articolazione consiste, concretamente, nell'utilizzare il valore ottenuto in input da un sensore del guanto per manipolare un asse del vettore rotazione dell'elemento da muovere.

Il modello fornito è composto di 15 elementi i cui *Motion Vector* sono collegati gerarchicamente, a partire dal polso/corpo della mano (elemento "wrist") e fino alle estremità delle dita:

- corpo della mano (1 elemento), a cui sono collegate le falangi delle dita
- indice, medio, anulare e mignolo, che sono composti da tre elementi ciascuno, etichettati "meta", "proxy" e "midd" (falange, falangina e falangetta), per un totale di 12 elementi, collegati tra loro "dito per dito" come in natura
- pollice, costituito dai rimanenti 2 elementi (il pollice manca della falange intermedia)

Il mapping tra sensori e operazioni di rotazione degli assi è illustrato in figura 4.1:

- i numeri indicano i sensori presenti sul guanto
- i cerchi indicano un'operazione di flessione o di rotazione di una componente del modello
  - i cerchi attorno ai numeri rappresentano una flessione direttamente ricavata dal valore del sensore cerchiato
  - i quattro cerchi verdi indicano le flessioni tra falangina e falangetta, punti che, non essendo dotati di sensori, vengono flessi in base al valore del sensore più vicino (quello tra falange e falangina, come indicato dalle frecce)
  - i cinque cerchi rossi indicano le operazioni di rotazione delle dita su diverso asse, quelle per ottenere la divaricazione tra le dita. Il guanto ha un sensore tra ogni coppia di dita, quindi si devono usare quattro valori per gestire cinque rotazioni, come suggerito dalle frecce.

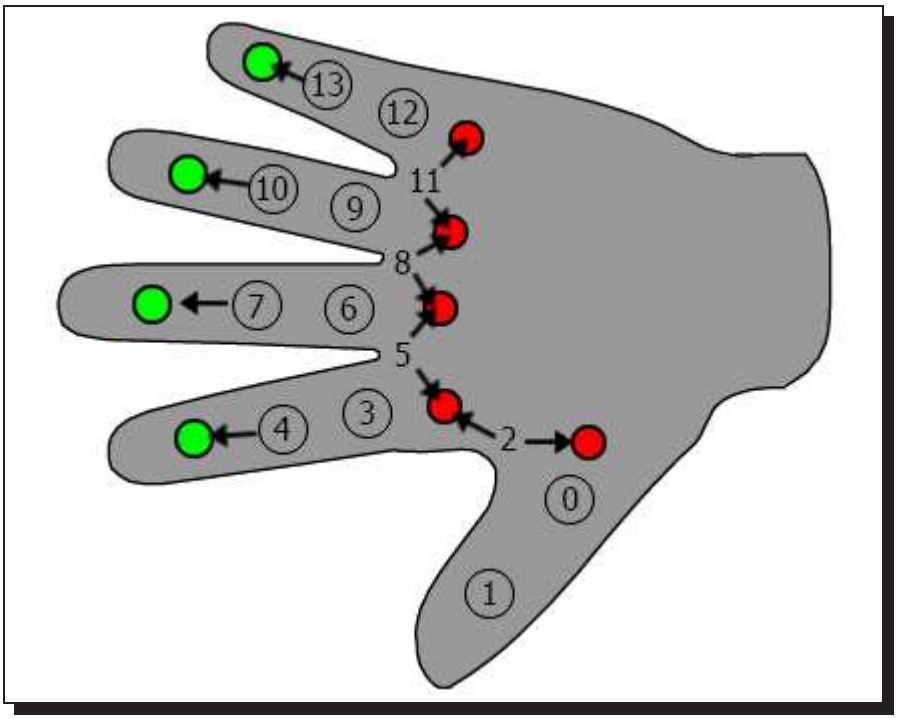


Figura 4.1:

La conversione da quattro a cinque valori viene effettuata secondo lo schema in figura 4.2, considerando uno o due valori adiacenti al dito da muovere, ottenendo un'approssimazione soddisfacente.

Spiegato con che criterio è stato organizzato il mapping dei valori, analizziamo un esempio concreto estrapolato dall'implementazione del demo. Consideriamo l'animazione di una falange, quella dell'indice, per mostrare sia la flessione che la divaricazione: tutte le animazioni delle articolazioni sono simili strutturalmente, ma quelle degli elementi "meta" delle dita, connessi direttamente al "wrist", devono implementare anche la divaricazione.

L'immagine 4.3 ritrae il `Motion Vector` dell'elemento "Index Meta":

- notiamo l'impostazione della "parent matrix", che definisce la dipendenza posizionale dell'elemento nei confronti del padre, in questo caso il corpo della mano, sotto forma del link al channel `Wrist Motion`, visibile sulla destra. Questo significa, in parole povere, che spostare o ruotare in un certo modo l'oggetto `Wrist Motion` comporterà un mo-

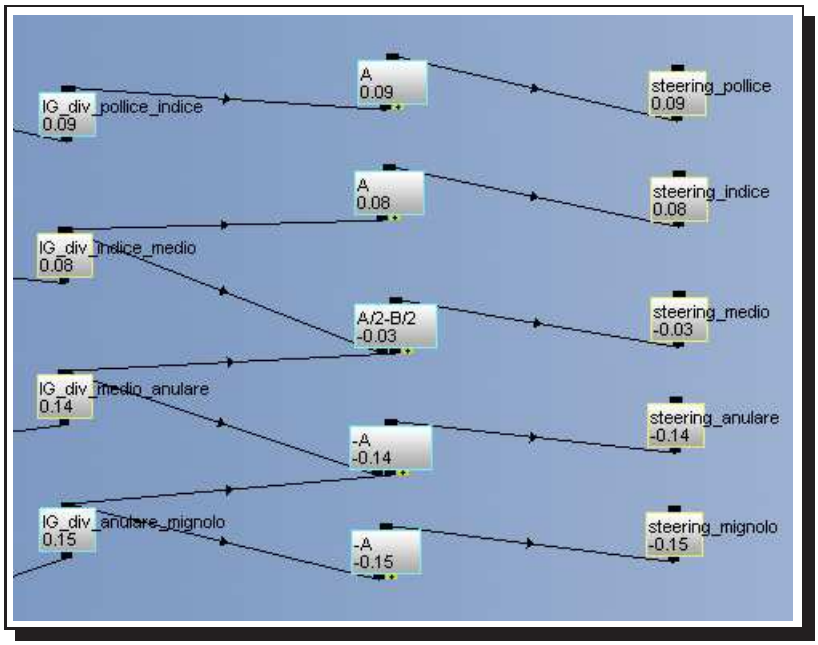


Figura 4.2: da 4 valori di divaricazione a 5 di *steering* delle dita

vimento in cascata anche di questo elemento, riflettendo il fatto che il dito è attaccato al corpo della mano;

- notiamo in basso i due input per divaricazione e flessione del dito:
  - il valore `steering_indice`, ricavato come indicato prima, fa da input a un channel *Envelope*, il quale definisce una componente del vettore di rotazione del dito, sull'asse della divaricazione;
  - il valore `IG_indice_f1` è il valore scaled di flessione ottenuto da ISISgloveInput, relativo al sensore 3, che fa da input a un channel Envelope, il quale definisce un'altra componente del vettore rotazione del dito, quella relativa all'asse su cui il dito si flette.

I channels `Envelope` servono a stabilire un mapping tra l'insieme di valori compresi tra 0 e 1, forniti in input, e dei coefficienti di rotazione adeguati, in modo da avere un movimento credibile delle dita del modello in base al movimento della mano nel guanto.

I valori in input ai channels `Envelope`, per essere precisi, non sono direttamente gli input ottenuti dal guanto, ma dei channel `Inertia` su di essi basati. L'impostazione dell'inerzia permette di regolare la *smoothness* dei

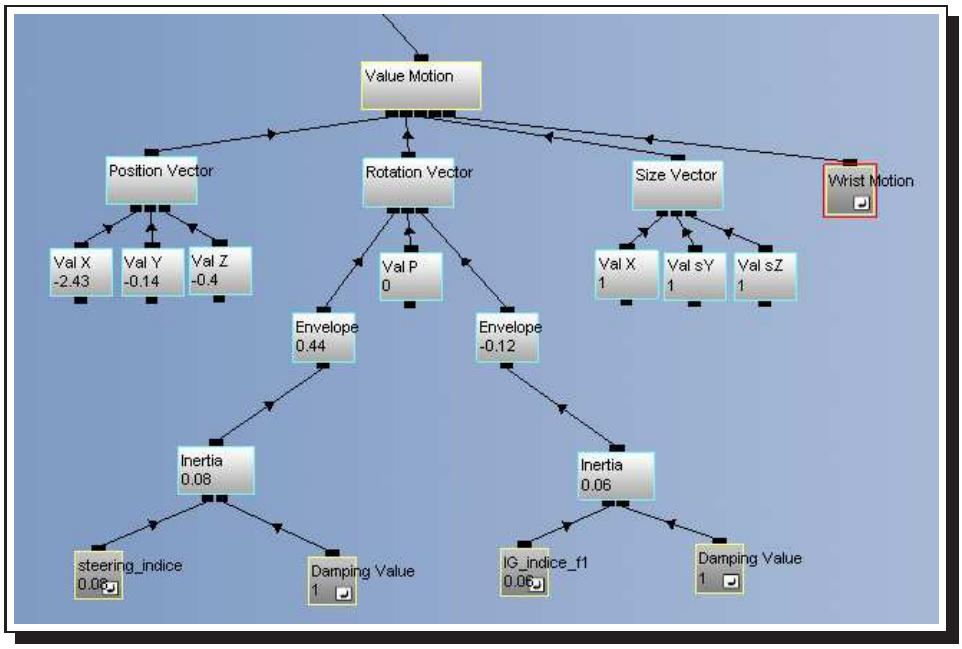


Figura 4.3:

movimenti: nell'esempio in figura, in cui ha valore 1, è come se fosse assente, ma si può configurare a livello globale nel progetto.

4.1.2 Motion tracking del polso

Una volta ottenuto accesso al PCtracker, il posizionamento della mano nello spazio in prossimità della camera è stato abbastanza immediato: si è trattato di utilizzare i vettori posizione/orientamento prelevati dalla stazione di tracking allacciata al guanto per modificare il *Motion Vector* dell'oggetto Wrist (il corpo della mano), di cui si è già accennato.

Esaminiamo le due strutture di channels in figura 4.4. Nella prima, osserviamo sull'estrema sinistra i due vettori direttamente ottenuti dal template ISISpcTracker: `Hand PositionVector` e `Hand RotationVector`.

Il vettore posizione, in alto, viene moltiplicato per uno scalare (5) per ottenere dei valori che causino uno spostamento della mano proporziale all'ampiezza della visuale della telecamera (che descriveremo a breve). Il vettore rotazione, eccetto uno scambio di assi che dipende dall'orientamento del modello 3D della mano, non necessita variazioni. Nella seconda immagine, vediamo come i vettori ottenuti dalle trasformazioni precedenti,

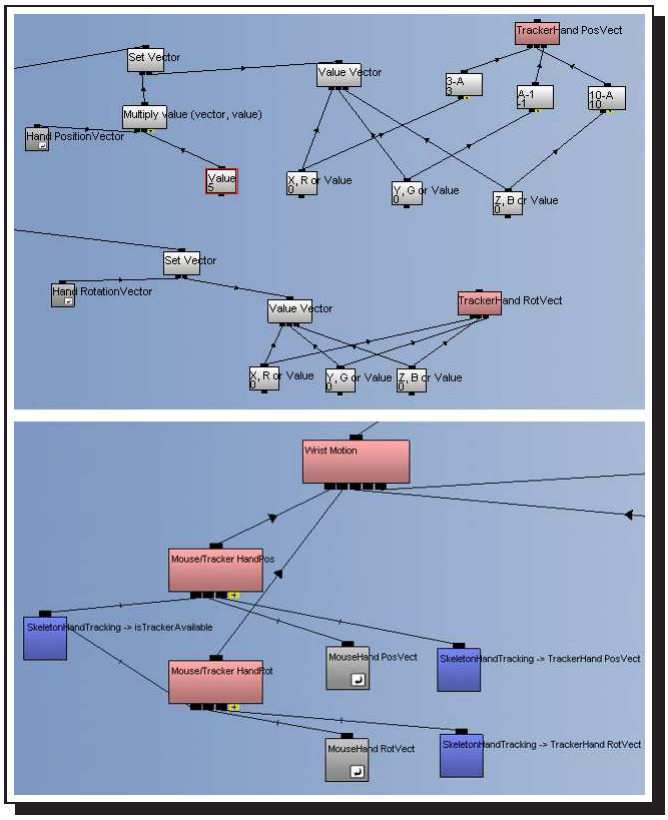


Figura 4.4:

`TrackerHand PosVect` e `TrackerHand RotVect`, vengono collegati al *Motion Vector* del polso. Avremmo potuto collegarli direttamente ai link square di `Wrist Motion`, ma per poter effettuare dei test senza hardware a disposizione è stato implementato un meccanismo sostitutivo di input (mouse+tastiera) che non ci interessa descrivere: la struttura è stata lasciata intatta in questa immagine d'esempio solo per mostrare un possibile utilizzo di `isTrackerActive`, citato nel capitolo precedente, ovvero quello di far ripiegare l'applicazione su meccanismi di input alternativi se il tracker non è disponibile.

4.2 Una walkthrough camera tracker-based

Come discusso parlando dell'esigenza di implementare un custom channel per supportare entrambe le stazioni di tracking, la stazione MiniTrax "Wand" si presta particolarmente ad essere utilizzata per controllare una *walkthrough camera*. Nella nostra implementazione d'esempio

- l'orientamento della stazione (Wand Rotation Vector) definisce la direzione della camera
- il joystick viene utilizzato per spostarsi avanti, indietro e lateralmente
- il posizionamento della stazione (Wand Position Vector) non si rivela utile in questo contesto, e non viene usato

Ai bottoni si possono facilmente associare comandi o modificatori utili alla specifica applicazione che si sta sviluppando. A puro scopo dimostrativo, nel nostro demo:

- il bottone 1 viene usato per abilitare/disabilitare la gravità sulla camera, permettendo il "fly mode"
- il bottone 2 fa il reset della posizione della camera a quella iniziale
- il bottone 5 (il grilletto) fa da modificatore "corsa", raddoppiando la velocità di spostamento quando è premuto

Sono state volutamente implementate tre azioni che fanno un uso diverso del bottone (switch on/off, comando e modificatore).

Esaminiamo l'implementazione del movimento, mostrata in figura 4.5:

Sulla sinistra, osserviamo come viene impostata la direzione (viene lasciato disponibile, sempre per comodità di testing, anche l'input da tastiera): i due channels blu in basso sono i valori sui due assi del joystick, che definiscono la direzione del vettore di spostamento. Questo subisce delle trasformazioni (calcolo dello spostamento e di eventuali collisioni con l'ambiente) per poi essere impostato come *Position Vector* della camera, e senza scendere troppo nei dettagli osserviamo come influiscono i valori dei bottoni della stazione di tracking, ovvero gli altri due channel in blu. Il primo è il modificatore della corsa, che se è attivo (quando si tiene premuto lo shift sinistro della tastiera, oppure il grilletto della stazione MiniTrax) raddoppia la velocità di spostamento. Il secondo (F come "fly mode", oppure il primo bottone della stazione) attiva/disattiva uno switch che seleziona il vettore

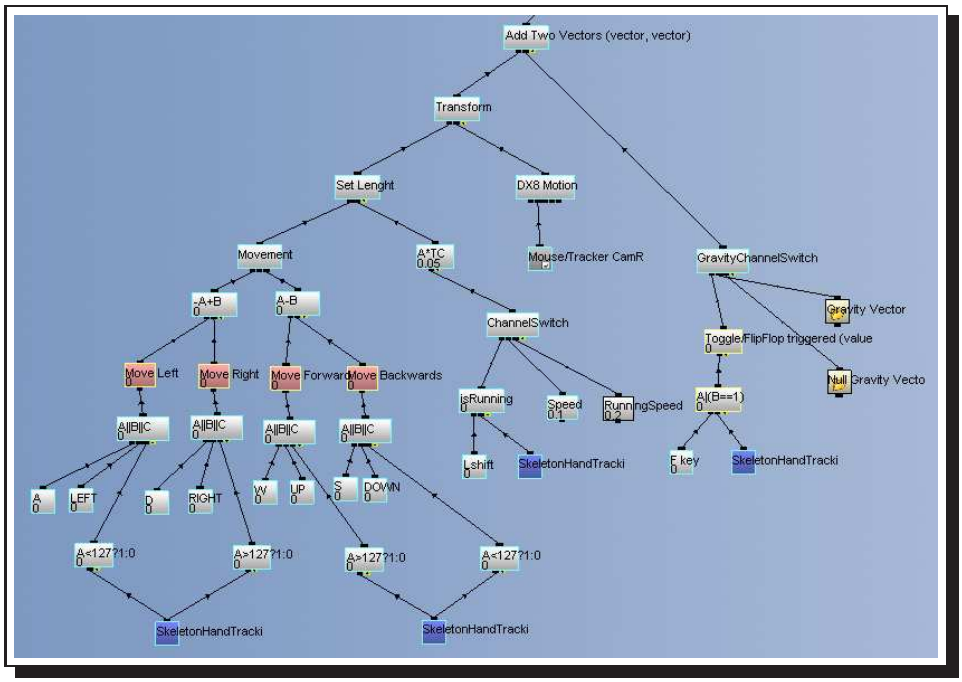


Figura 4.5: la parte del calcolo del vettore posizione della camera che coinvolge l'input dell'utente

gravità da utilizzare nel calcolo della posizione della camera: il primo simula la caduta verso il basso, il secondo è un vettore nullo che permette di "volare" nella scena.

Molto più semplice l'impostazione della rotazione della camera, visibile in figura 4.6.

Non c'è niente di particolare da notare: anche in questo caso viene utilizzato un input alternativo di test se il tracker non è disponibile, ed è poi stato necessario invertire un asse.

4.3 Un esempio di interazione gesture-based

Descriviamo infine una semplice interazione attivata da una combinazione di input di motion tracking e guanto: evidenziamo un oggetto in scena indicandolo con la mano.

Usando il channel `CollisionRayCheck`, controlliamo se sulla direzione indicata dalla mano (ottenuta dalla stazione di tracking) c'è un oggetto

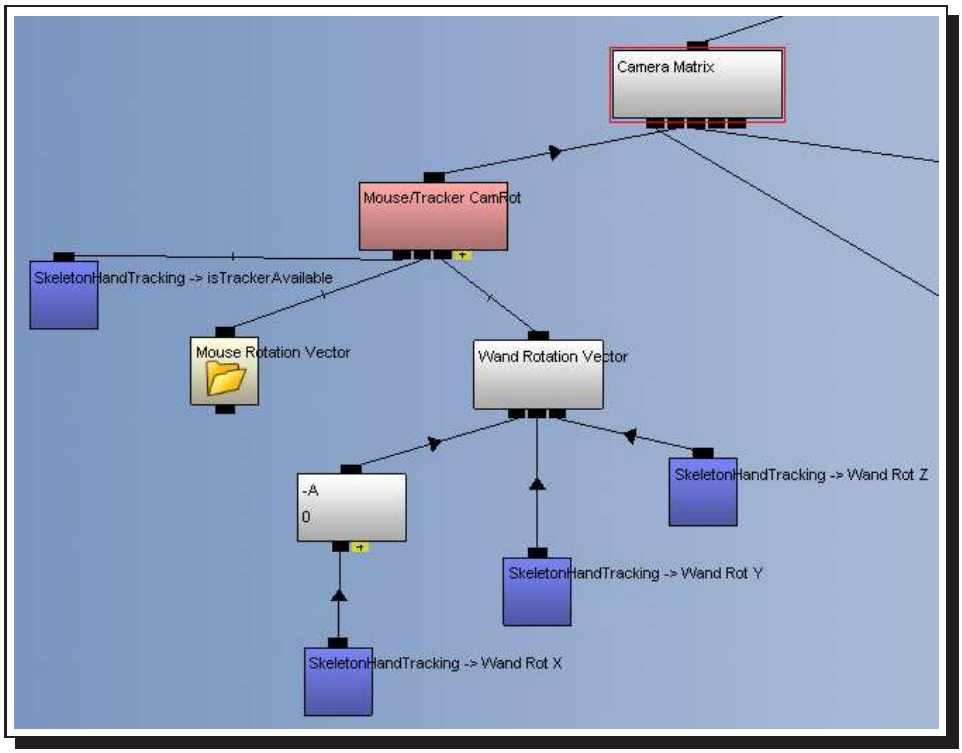


Figura 4.6: orientamento della camera

predisposto per il rilevamento delle collisioni, e in tal caso, se viene rilevato anche il riconoscimento di un gesto definito in ISISgloveManager come "indicePuntato", alteriamo l'illuminazione emissiva dell'oggetto per evidenziarlo, illuminandolo.

In figura 4.7 l'espressione che controlla le due condizioni e uno screenshot di SkeleTronDemo che mostra l'interazione.

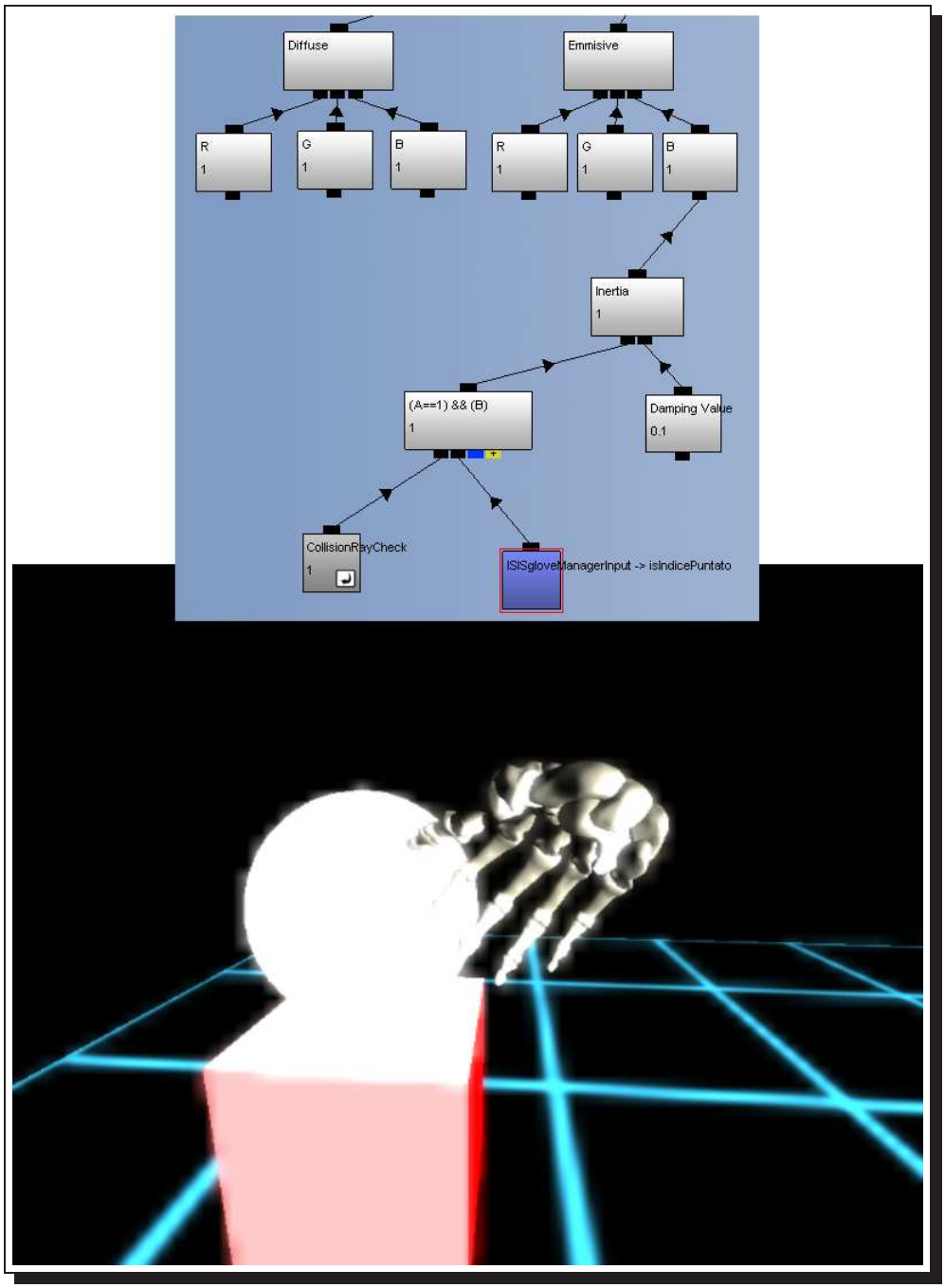


Figura 4.7: un'azione determinata dall'input combinato di tracker e guanto - la sfera illuminata è l'oggetto indicato

Capitolo 5

Conclusioni

Hey! It compiles! Ship it!
anonimo, online

L'obiettivo di costruire un'infrastruttura per l'interazione hand-based in Quest3D può dirsi raggiunto. Si è cercato di sfruttare al massimo l'hardware a nostra disposizione e di curare lo sviluppo di ogni componente in modo che, se necessario, possa essere utilizzata indipendentemente dalle altre, sebbene alcune caratteristiche siano state progettate pensando all'utilizzo combinato della libreria.

Quest3D, superata una prima fase di diffidenza dovuta alla metodologia atipica di sviluppo, si è rivelato un ottimo strumento di editing. Se infatti in alcuni casi, all'interno di un progetto, l'uso dei *channels* può apparire poco indicato o limitante, abbiamo visto come sia relativamente semplice adoperare lo scripting Lua o l'SDK per creare dei componenti ad hoc perfettamente integrati nel modello di sviluppo dell'engine.

Il naturale proseguimento del lavoro svolto, considerando l'utilizzo combinato di quanto sviluppato, è la progettazione di un'applicazione pensata per sfruttare estensivamente, all'interno dell'engine Quest3D, l'input combinato di stazioni di tracking e guanto: si spera che SkeleTronDemo possa essere una solida base di partenza per la creazione di sistemi più vasti e sofisticati. Sarebbe ottimo dedicare energie alla creazione di una serie di template Quest3D che implementino (in maniera facilmente riutilizzabile) il riconoscimento di una vasta gamma di azioni hand-based, o ancora meglio la realizzazione di un meccanismo di interazione generico. A tale scopo va tenuto d'occhio il supporto di Quest3D alla simulazione della fisica, at-

tualmente in fase di totale rivoluzione: correntemente l'engine si basa sulla libreria *ODE* (Open Dynamics Engine), ma dalla prossima versione passerà al sistema *Newton Game Dynamics*, propagandato come decisamente superiore.

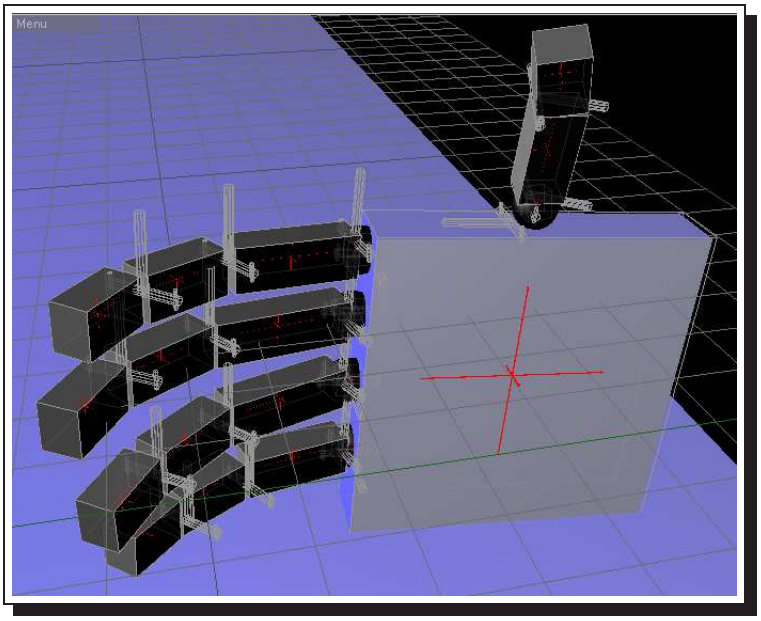


Figura 5.1: il modello creato per il testing di ODE

Il tentativo di ottenere con ODE un modello *physics-enabled* della mano (ovvero costruito con primitive che supportassero la simulazione della fisica, come visibile in figura 5.1), effettuato nella fase di testing delle potenzialità del guanto, aveva dato risultati interessanti ma non concretamente utilizzabili, a causa di oscillazioni e tremolio del modello, probabilmente dovuti a conflitti tra le forze in gioco. Un problema di fondo dell'approccio all'interazione basata sulla fisica è nel fatto che il modello è sottoposto a forze originate dall'ambiente virtuale oltre che a quelle calcolate in base all'input del guanto, e quindi (trattandosi di un dispositivo non-aptico, che quindi non può riflettere le forze dell'ambiente sulla mano dell'utilizzatore) non si potrà mai avere una piena, immediata corrispondenza tra posizione di mano reale e mano virtuale. Non è però da escludere che il cambio di engine fisico possa aiutare a raggiungere risultati migliori o la creazione di un modello più semplice e meglio strutturato che possa, se non superare, almeno sostituire in maniera generica l'approccio gesture-driven attuale. Osserviamo con un

esempio come un sistema basato sulla fisica sia drasticamente diverso da uno basato sul riconoscimento di gesti: consideriamo di voler raccogliere una pallina. Con l'infrastruttura gesture-driven creata, se si volesse implementare tale interazione, si potrebbero seguire questi passi:

- stabilire un area dello spazio in prossimità della pallina in cui è possibile avviare l'interazione
- quando la mano entra in quest'area (coordinate ottenute dal dispositivo di tracking)
  - se si passa dallo stato "manoAperta" a quello "pugno" (gesti riconosciuti da ISISgloveManager), agganciare l'oggetto pallina alla mano (ovvero impostare come sue coordinate quelle della mano, più un certo offset per non avere una totale sovrapposizione)

Se invece si riuscisse a implementare un generico sistema basato sulla fisica, bisognerebbe soltanto definire la pallina come oggetto parte della simulazione fisica, dotato di una certa massa: sarebbero poi le forze operate dalle dita del modello *physics-enabled* a far sì che si possa interagire con la palla in qualsiasi modo (raccogliendola, facendola rotolare sul palmo, spingendola con un dito... tutte azioni che con l'altro approccio vanno implementate una per una).

Una soluzione intermedia da valutare è quella basata sui channels di *collision detection* basata su sferoidi, anche se normalmente non sono utilizzati per operazioni "di precisione" come quelle operabili con le dita di una mano.

E' da dire che gli esperimenti in questa direzione sono stati abbandonati in favore dell'approccio *gesture-driven* anche per una piccola, grande limitazione del Data Glove 16: l'impossibilità di rilevare la traslazione del pollice rispetto al palmo. Questa carenza si avverte molto di meno utilizzando il riconoscimento dei gesti che tentando la riproduzione delle forze esercitate dalle dita: il peggio che può accadere è che gesti differenti solo per la traslazione del pollice vengano considerati erroneamente come uguali.

Per far sì che questa mancanza non si notasse a livello visivo, è stato introdotta qualche piccola alterazione nell'input all'animazione del modello, facendo sì che il pollice della mano scheletrica assumesse pose "di solito" adeguate anche in assenza del dato sulla traslazione.

Per il resto, il Data Glove 16 si è rivelato uno strumento adeguato, così come il dispositivo di tracking della InterSense che, superata una problematica fase di setup hardware e software, si è comportato egregiamente

fornendo un tracking preciso e pulito. Anche l'API dell'InterSense, sebbene sia stata utilizzata minimamente e non analizzata in dettaglio come quella del guanto, è apparsa ben strutturata e semplicemente utilizzabile.

Riguardo lo sviluppo di ISISgloveManager, va fatto un appunto sulle difficoltà insorte nella compilazione in ambiente Linux. Il driver presente sul sito 5DT al momento della stesura, risalente al 2000 o giù di lì, è praticamente inutilizzabile con una *toolchain* di compilazione odierna, ma la 5DT, contattata via e-mail al riguardo, è stata molto disponibile ad inviare un *build* aggiornato del driver, che ha aiutato a risolvere il problema. Essendo tutto il resto della libreria sviluppata strettamente legato a Quest3D e quindi a Windows, va comunque detto che la versione Linux di ISISgloveManager è stata degnata di ben poca attenzione e testing, anche se si spera che non dovrebbe dare problemi di rilievo.

Lavorare con queste tecnologie di input alternativo è indubbiamente interessante, che siano destinate a rimanere di nicchia o meno. Secondo una prospettiva ottimistica, col progressivo abbassamento dei costi coinvolti, non è infatti da escludere che - a quasi vent'anni dal *Nintendo Power Glove* - i wired-gloves possano avere una seconda chance di diventare una periferica comune, questa volta senza le drastiche limitazioni dell'epoca. Come spesso accade in questi casi, quel che occorre per rendere di successo una periferica è probabilmente una *killer-app* che la renda ambita ed insostituibile, e il fatto che tale applicazione non sia ancora stata creata non fa che rendere più intrigante l'avvicinarsi allo sviluppo di software in questo campo.

Bibliografia

[1] SA Mehdi, YN Khan. "Sign language recognition using sensor gloves". Neural Information Processing, 2002. ICONIP'02.

[2] T. Ullmann, J. Sauer. "Intuitive virtual grasping for non haptic environments". Computer Graphics and Applications, 2000.

[3] P. Hong, M. Turk, T. S. Huang. "Gesture modeling and recognition using finite state machines". Automatic Face and Gesture Recognition, 2000.

[4] S. S. Fels, G. E. Hinton. "Glove-Talk: a neural network interface between a data-glove and a speech synthesizer". Neural Networks, IEEE Transactions on, 1993.

[5] D. J. Sturman, D. Zeltzer. "A survey of glove-based input". Computer Graphics and Applications, IEEE, 1994.

[6] J. Davis, M. Shah. "Recognizing hand gestures". Proc. European Conf. Computer Vision.

[7] M. Kallmann, D. Thalmann. "Direct 3D interaction with smart objects". Proceedings of the ACM symposium on Virtual Reality Software and Technology (VRST), 1999.

[8] B. Dorner. "Chasing the colour glove: visual hand tracking". Master of Science degree Thesis, Simon Fraser University, 1994.

[9] B. Stroustrup, "The C++ Programming Language (Special Edition)".

[10] "Bjarne Stroustrup's FAQ"
http://www.research.att.com/~bs/bs_faq.html

[11] "wxWidgets Overview"
http://www.wxwidgets.org/about/datasheets/wxWidgetsOverview.pdf

[12] J. Smart, K. Hock. "Cross-Platform GUI Programming with wxWidgets". Prentice Hall, 2005.

[13] "Quest3D Reference Manual"

[14] "Quest3D Tutorial Manual"

[15] "Quest3D SDK Documentation"

[16] "Wikipedia: Lua (programming language)"
http://en.wikipedia.org/wiki/Lua_%28programming_language%29

[17] "Lua 5.1 Reference Manual"
http://www.lua.org/manual/5.1/

[18] "InterSense PCTracker Manual"

[19] "5DT Data Glove 16 Manual"

[20] "Brief Biography of Jaron Lanier"
http://www.jaronlanier.com/general.html

[21] "Wikipedia: Jaron Lanier"
http://en.wikipedia.org/wiki/Jaron_Lanier

[22] "Jaron's World: Virtual Horizon"
http://discovermagazine.com/2007/may/jaron2019s-world

[23] "The Future of Virtual Reality: Part Two of a Conversation with Jaron Lanier"
http://java.sun.com/features/2003/02/lanier_qa2.html

[24] "A Minority within the Minority"
http://www.21cmagazine.com/minority.html

[25] "Wikipedia: Virtual Reality"
http://en.wikipedia.org/wiki/Virtual_Reality

[26] "Wikipedia: Motion Capture"
http://en.wikipedia.org/wiki/Motion_capture

[27] "Wikipedia: Match Moving"
http://en.wikipedia.org/wiki/Match_moving

[28] "Wikipedia: Sensor fusion"
http://en.wikipedia.org/wiki/Sensor_fusion

[29] "Wikipedia: Kalman filter"
http://en.wikipedia.org/wiki/Kalman_filter

[30] "Wikipedia: Wired Glove"
http://en.wikipedia.org/wiki/Wired_glove

[31] "Wikipedia: Power Glove"
http://en.wikipedia.org/wiki/Power_Glove

[32] "The Angry Nintendo Nerd's review of the Powerglove"
http://www.youtube.com/watch?v=MYDuy7wM8Gk

[33] "The Tao Of Programming"
http://www.canonical.org/~kragen/tao-of-programming.html

[34] "Tron", dir. Steven Lisberger, 1982

[35] "The Lawnmower Man", dir. Brett Leonard, 1992

[36] "Johnny Mnemonic", dir. Robert Longo, 1995

[37] "Minority Report", dir. Steven Spielberg, 2002

[38] W. Gibson, "Neuromancer" (Neuromante), 1984

[39] W. Gibson, "Burning Chrome" (La notte che bruciammo Chrome), 1986

Appendice A

Appendice

A.1 Listati

A.1.1 ISISgloveAPI

La classe Glove

```
1  #ifndef GLOVE_H
2  #define GLOVE_H
3
4  #include <fglove.h>
5
6  #include <string>
7
8  #include "GloveCalibration.h"
9  #include "ScaledHandPosition.h"
10 #include "RawHandPosition.h"
11
12 class GloveNotAvailableException {};
13
14 class Glove {
15
16     private:
17         bool init(char *); //constructor helper
18
19     public:
20         Glove(void);        // scans serial ports
21         Glove(std::string); // open on specified port ("COM[1-8]"|"/dev/glove")
22         ~Glove(void);
23
24         bool isConnected();
25
26         bool isRightHand();
27         bool isLeftHand();
28         int getNumSensors();
29         char *getType();
30         unsigned char *getDriverInfo();
31
```

```
32          ScaledHandPosition *getScaledHandPosition();
33          void updateScaledHandPosition(ScaledHandPosition *hp);
34
35          RawHandPosition *getRawHandPosition();
36          void updateRawHandPosition(RawHandPosition *hp);
37
38          GloveCalibration *getCalibration();
39          void updateCalibration(GloveCalibration *gc);
40
41          void setCalibration(GloveCalibration*);
42          void resetCalibration();
43
44      private:
45          fdGlove *pGlove;            //pointer to underlying glove resource
46          bool connected;
47
48          enum EfdGloveHand handedness;    // left or right?
49          enum EfdGloveTypes type;         // glove model?
50
51          int numSensors;              // for future dev, now hardcoded to 18
52          unsigned char driverInfo[32];// as specified in the docs
53  };
54
55  #endif
```

```
1   #include "Glove.h"
2
3   #include <stdio.h>
4
5   bool Glove::init(char *szPort) {
6       if (NULL != (pGlove = fdOpen( szPort ))) {
7           // non-changing info gathering
8           handedness = (EfdGloveHand) fdGetGloveHand(pGlove);
9           numSensors = fdGetNumSensors(pGlove);
10          type = (EfdGloveTypes)fdGetGloveType(pGlove);
11          return true;
12      }
13      return false;
14  }
15
16  Glove::Glove(void) {
17      // on win if serial port not specified, try to find it
18      char szPort[5];
19      strcpy(szPort,"COMx");
20      char i;
21      for (i='1'; i<='8'; i++) {
22          szPort[3] = i;
23          if (init(szPort)) return;   // ok
24      }
25      throw GloveNotAvailableException(); // failed
26  }
27
28  Glove::Glove(std::string serialPort) {
29      char szPort[10]; // fdOpen needs a non-const char array
30      strncpy(szPort, serialPort.c_str(),9); szPort [9] = '\0';
31
32      if (init(szPort)) return;               //ok
```

```
33      else throw GloveNotAvailableException(); //failed
34  }
35
36  Glove::~Glove(void) {
37      fdClose( pGlove );
38  }
39
40
41  int Glove::getNumSensors()  { return numSensors;}
42
43  char *Glove::getType() {
44      char *szType = "?";
45      switch (type)   {
46          case FD_GLOVENONE: szType = "None";     break;
47          case FD_GLOVE7:    szType = "Glove7";   break;
48          case FD_GLOVE7W:   szType = "Glove7W";  break;
49          case FD_GLOVE16:   szType = "Glove16";  break;
50          case FD_GLOVE16W:  szType = "Glove16W"; break;
51      }
52      return  szType;
53  }
54
55  unsigned char *Glove::getDriverInfo() {
56      unsigned char *driverdata = (unsigned char*)malloc(32*sizeof(unsigned char));
57      fdGetDriverInfo(pGlove,driverdata);
58      return driverdata;
59  }
60
61  bool Glove::isRightHand() {
62      return (handedness==FD_HAND_RIGHT);
63  }
64
65  bool Glove::isLeftHand() {
66      return (handedness==FD_HAND_LEFT);
67  }
68
69  ScaledHandPosition *Glove::getScaledHandPosition() {
70      ScaledHandPosition *hp = new ScaledHandPosition(getNumSensors());
71      fdGetSensorScaledAll(pGlove, hp->scaledSensorValues);
72      return hp;
73  }
74
75  void Glove::updateScaledHandPosition(ScaledHandPosition *hp) {
76      fdGetSensorScaledAll(pGlove, hp->scaledSensorValues);
77  }
78
79  RawHandPosition *Glove::getRawHandPosition() {
80      RawHandPosition *hp = new RawHandPosition(getNumSensors());
81      fdGetSensorRawAll(pGlove, hp->rawSensorValues);
82      return hp;
83  }
84
85  void Glove::updateRawHandPosition(RawHandPosition *hp) {
86      fdGetSensorRawAll(pGlove, hp->rawSensorValues);
87  }
88
```

```
89  void Glove::setCalibration(GloveCalibration* gc) {
90      fdSetCalibrationAll(pGlove, gc->pUpper, gc->pLower);
91  }
92  void Glove::resetCalibration() {
93      fdResetCalibration(pGlove);
94  }
95  void Glove::updateCalibration(GloveCalibration *gc) {
96      fdGetCalibrationAll(pGlove, gc->pUpper, gc->pLower);
97  }
98
99  GloveCalibration *Glove::getCalibration() {
100     GloveCalibration *gc = new GloveCalibration(getNumSensors());
101     fdGetCalibrationAll(pGlove, gc->pUpper, gc->pLower);
102     return gc;
103 }
```

La classe astratta SaveLoad

```
1   #ifndef SAVELOAD_H
2   #define SAVELOAD_H
3
4   #include <exception>
5   #include <fstream>
6   #include <iostream>
7
8   class CannotLoadException : public std::exception {};
9   class CannotSaveException : public std::exception {};
10
11  using namespace std;
12
13  class SaveLoad {
14      friend ostream& operator<<(ostream& s, const SaveLoad& sl) {
15          return sl.put(s); //uses the right put()
16      };
17
18      friend istream& operator>>(istream& s, SaveLoad& sl) {
19          return sl.get(s); //uses the right get()
20      }
21
22      public:
23          virtual ostream& put(ostream& s) const = 0; //write *this to s
24          virtual istream& get(istream& s) = 0; //read *this from s
25
26          void saveToFile(std::string filename);
27          void loadFromFile(std::string filename);
28  };
29  #endif
```

La classe GloveCalibration

```
1   #ifndef GLOVECALIBRATION_H
2   #define GLOVECALIBRATION_H
3
4   #include "SaveLoad.h"
5
6   class GloveCalibration : public SaveLoad {
```

```

    friend std::ostream& operator<<(std::ostream& output, const GloveCalibration& p);
    friend std::istream& operator>>(std::istream& input, GloveCalibration& p);

    public:
        GloveCalibration(int);
        ~GloveCalibration(void);

        GloveCalibration& operator=(const GloveCalibration& gc); // assignment
        GloveCalibration (const GloveCalibration& gc);           // copy constructor

        unsigned short getRawMax(EfdSensors sensorId) {
            return pUpper[sensorId];
        };

        unsigned short getRawMin(EfdSensors sensorId) {
            return pLower[sensorId];
        };

        void setRawMax(EfdSensors sensorId, unsigned short max) {
            pUpper[sensorId] = max;
        }
        void setRawMin(EfdSensors sensorId, unsigned short min) {
            pLower[sensorId] = min;
        }

    private:
        std::ostream& put(ostream& s) const { return s << *this; };
        std::istream& get(istream& s) { return s >> *this; };

        unsigned short *pUpper;
        unsigned short *pLower;
        int numSensors;

    friend class Glove;
};

#endif


#include "Glove.h"
#include "GloveCalibration.h"

GloveCalibration::GloveCalibration(int numSens) {
    numSensors = numSens;
    pUpper = new unsigned short[numSensors];
    pLower = new unsigned short[numSensors];
}

GloveCalibration::~GloveCalibration(void){
    delete[] pUpper;
    delete[] pLower;
}

GloveCalibration& GloveCalibration::operator=(const GloveCalibration& gc) {
    if (&gc != this) {
        if (numSensors != gc.numSensors) {
```

```
18              delete[] pUpper;
19              delete[] pLower;
20              numSensors = gc.numSensors;
21              pUpper = new unsigned short[numSensors];
22              pLower = new unsigned short[numSensors];
23          }
24          for (int i = 0 ;  i < numSensors; i++ ) {
25              pUpper[i] = gc.pUpper[i];
26              pLower[i] = gc.pLower[i];
27          }
28      }
29      return *this;
30  }
31
32  GloveCalibration::GloveCalibration (const GloveCalibration& gc) {
33      numSensors = gc.numSensors;
34      pUpper = new unsigned short[numSensors];
35      pLower = new unsigned short[numSensors];
36      for (int i = 0 ;  i < numSensors; i++ ) {
37          pUpper[i] = gc.pUpper[i];
38          pLower[i] = gc.pLower[i];
39      }
40  }
41
42  std::ostream& operator<<(std::ostream& output, const GloveCalibration& p) {
43      output << "(";
44      for (int i = 0; i < p.numSensors-1; i++ )
45          output << p.pLower[i] << "-" << p.pUpper[i] << "|";
46
47      output << p.pLower[p.numSensors-1] << "-" << p.pUpper[p.numSensors-1];
48      output << ")" ;
49      return output;
50  }
51
52  std::istream& operator>>(std::istream& input, GloveCalibration& p) {
53      char useless;
54      input >> useless; //"("
55      for (int i = 0; i < p.numSensors-1; i++ )
56          input >> p.pLower[i] >> useless >> p.pUpper[i] >> useless;
57      input >> p.pLower[p.numSensors-1] >> useless >> p.pUpper[p.numSensors-1];
58      input >> useless; //")"
59      return input;
60  }
```

La classe ScaledHandPosition

```
1   #ifndef SCALEDHANDPOSITION_H
2   #define SCALEDHANDPOSITION_H
3
4   #include <fglove.h>
5
6   #include "SaveLoad.h"
7
8   class ScaledHandPosition : public SaveLoad {
9
10      friend std::ostream& operator<<(std::ostream& output, const ScaledHandPosition& p);
11      friend std::istream& operator>>(std::istream& input, ScaledHandPosition& p);
```

```
12
13      public:
14          ScaledHandPosition(int);
15          ~ScaledHandPosition(void);
16
17          ScaledHandPosition& operator=(const ScaledHandPosition& sh); // assignment
18          ScaledHandPosition (const ScaledHandPosition& sh);           // copy constructor
19
20          static void setCompareThreshold(float toll) {
21              cmpToll = toll;
22          };
23          static float getCompareThreshold() {
24              return cmpToll;
25          };
26
27          int cmp(const ScaledHandPosition *other) const {
28              for (int i = 0; i < numSensors; i++)
29                  if ( (scaledSensorValues[i] - other->scaledSensorValues[i]) < -cmpToll )
30                      return -1;  // <
31                  else if ( (scaledSensorValues[i] - other->scaledSensorValues[i]) > cmpToll )
32                      return 1;   // >
33              return 0;           // ==
34          }
35
36          bool operator== (const ScaledHandPosition &a ) const {
37              return (this->cmp(&a)==0);
38          };
39          bool operator!=( const ScaledHandPosition &a ) const {
40              return (this->cmp(&a)!=0);
41          }
42          bool operator<(const ScaledHandPosition &a) const {
43              return (this->cmp(&a)<0);
44          }
45          bool operator>(const ScaledHandPosition &a) const {
46              return (this->cmp(&a)>0);
47          }
48
49          float getSensorValue(EfdSensors sensorId) {
50              return scaledSensorValues[sensorId];
51          };
52
53          void setSensorValue(EfdSensors sensorId, float val) {
54              scaledSensorValues[sensorId] = val;
55          };
56
57      private:
58          std::ostream& put(std::ostream& s) const { return s << *this; };
59          std::istream& get(std::istream& s) {    return s >> *this; };
60
61          float *scaledSensorValues;
62          int numSensors;
63          static float cmpToll; //threshold value in compare
64
65      friend class Glove;
66  };
67
```

```
68  #endif

 1  #include "ScaledHandPosition.h"
 2
 3  float ScaledHandPosition::cmpToll = 0.15;
 4
 5  ScaledHandPosition::ScaledHandPosition(int numSens){
 6      numSensors = numSens;
 7      scaledSensorValues = new float[numSensors];
 8  }
 9
10  ScaledHandPosition::~ScaledHandPosition(void) {
11      delete[] scaledSensorValues;
12  }
13
14  ScaledHandPosition& ScaledHandPosition::operator=(const ScaledHandPosition& sh) {
15      if (&sh != this) {
16          if (numSensors != sh.numSensors) {
17              delete[] scaledSensorValues;
18              numSensors = sh.numSensors;
19              scaledSensorValues = new float[numSensors];
20          }
21          for (int i = 0 ;  i < numSensors; i++ )
22              scaledSensorValues[i] = sh.scaledSensorValues[i];
23      }
24      return *this;
25  }
26
27  ScaledHandPosition::ScaledHandPosition (const ScaledHandPosition& sh) {
28      numSensors = sh.numSensors;
29      scaledSensorValues = new float[numSensors];
30      for (int i = 0 ;  i < numSensors; i++ )
31          scaledSensorValues[i] = sh.scaledSensorValues[i];
32  }
33
34  std::ostream& operator<<(std::ostream& output, const ScaledHandPosition& p) {
35      output << "(";
36      for (int i = 0; i < p.numSensors-1; i++ )
37          output << p.scaledSensorValues[i] << "|";
38      output << p.scaledSensorValues[p.numSensors-1];
39      output << ")" ;
40      return output;
41  }
42
43  std::istream& operator>>(std::istream& input, ScaledHandPosition& p) {
44      char useless;
45      input >> useless; //"("
46      for (int i = 0; i < p.numSensors-1; i++ )
47          input >> p.scaledSensorValues[i] >> useless;
48      input >> p.scaledSensorValues[p.numSensors-1];
49      input >> useless; //")"
50      return input;
51  }
```

La classe RawHandPosition

```
1  #ifndef RAWHANDPOSITION_H
```

```
#define RAWHANDPOSITION_H

#include <fglove.h>

#include "SaveLoad.h"

class RawHandPosition : public SaveLoad {

    friend std::ostream& operator<<(std::ostream& output, const RawHandPosition& p);
    friend std::istream& operator>>(std::istream& input, RawHandPosition& p);

    public:
        RawHandPosition(int);
        ~RawHandPosition(void);

        RawHandPosition& operator=(const RawHandPosition& sh); // assignment
        RawHandPosition (const RawHandPosition& sh);           // copy constructor

        unsigned short getSensorValue(EfdSensors sensorId) {
            return rawSensorValues[sensorId];
        };

        void setSensorValue(EfdSensors sensorId, unsigned short val) {
            rawSensorValues[sensorId] = val;
        };

    private:
        std::ostream& put(std::ostream& s) const { return s << *this; };
        std::istream& get(std::istream& s) { return s >> *this; };

        unsigned short *rawSensorValues;
        int numSensors;

    friend class Glove;
};

#endif
```

```
#include "RawHandPosition.h"

RawHandPosition::RawHandPosition(int numSens) {
    numSensors = numSens;
    rawSensorValues = new unsigned short[numSensors];
}

RawHandPosition::~RawHandPosition(void) {
    delete[] rawSensorValues;
}

RawHandPosition& RawHandPosition::operator=(const RawHandPosition& sh) {
    if (&sh != this) {
        if (numSensors != sh.numSensors) {
            delete[] rawSensorValues;
            numSensors = sh.numSensors;
            rawSensorValues = new unsigned short[numSensors];
        }
        for (int i = 0 ;  i < numSensors; i++ )
```

```
20              rawSensorValues[i] = sh.rawSensorValues[i];
21      }
22      return *this;
23  }
24
25  RawHandPosition::RawHandPosition (const RawHandPosition& sh) {
26      numSensors = sh.numSensors;
27      rawSensorValues = new unsigned short[numSensors];
28      for (int i = 0 ;  i < numSensors; i++ )
29          rawSensorValues[i] = sh.rawSensorValues[i];
30  }
31
32  std::ostream& operator<<(std::ostream& output, const RawHandPosition& p) {
33      output << "(";
34      for (int i = 0; i < p.numSensors-1; i++ )
35          output << p.rawSensorValues[i] << "|";
36      output << p.rawSensorValues[p.numSensors-1];
37      output << ")" ;
38      return output;
39  }
40
41  std::istream& operator>>(std::istream& input, RawHandPosition& p) {
42      char useless;
43      input >> useless; //"("
44      for (int i = 0; i < p.numSensors-1; i++ )
45          input >> p.rawSensorValues[i] >> useless;
46      input >> p.rawSensorValues[p.numSensors-1];
47      input >> useless; //")"
48      return input;
49  }
```

A.1.2 ISISgloveManager

La *main* class: ISISgloveManagerApp

```
1   /*************************************************************
2    * Name:      ISISgloveManagerApp.h
3    * Author:    Dario Scarpa (dario.scarpa@duskzone.it)
4    *************************************************************/
5
6   #ifndef ISISGLOVEMANAGERAPP_H
7   #define ISISGLOVEMANAGERAPP_H
8
9   #include <wx/wxprec.h>
10
11  #ifdef __BORLANDC__
12      #pragma hdrstop
13  #endif
14
15  #ifndef WX_PRECOMP
16      #include <wx/wx.h>
17  #endif
18
19  class ISISgloveManagerApp : public wxApp {
20      public:
21          virtual bool OnInit();
```

```
};

#endif // ISISGLOVEMANAGERAPP_H_H
```

```
#ifdef WX_PRECOMP //
#include "wx_pch.h"
#endif

#ifdef __BORLANDC__
#pragma hdrstop
#endif //__BORLANDC__

#include "ISISgloveManagerApp.h"
#include "GUI/ISISgloveTabs.h"

IMPLEMENT_APP(ISISgloveManagerApp);

bool ISISgloveManagerApp::OnInit() {

    ISISgloveTabs* mainWindow = new ISISgloveTabs( NULL, ID_ISISGLOVEMANAGER );
    mainWindow->Show(true);

    return true;
}
```

Implementazione GUI: la classe ISISgloveTabs

```
/////////////////////////////////////////////////////////////////////////////
// Name:        ISISgloveTabs
// Purpose:     ISISgloveManager tab-based GUI
// Author:      Dario 'Dusk' Scarpa
// Created:     26/06/2007 02:51:32
/////////////////////////////////////////////////////////////////////////////

// Generated by DialogBlocks (unregistered), 26/06/2007 02:51:32
//...and hand-edited in the following months :)

#ifndef _ISISGLOVETABS_H_
#define _ISISGLOVETABS_H_

#if defined(__GNUG__) && !defined(NO_GCC_PRAGMA)
#pragma interface "ISISgloveTabs.h"
#endif

/*!
 * Includes
 */
#include "wx/wx.h"

////@begin includes
#include "wx/frame.h"
#include "wx/notebook.h"
#include "wx/tglbtn.h"
#include "wx/grid.h"
////@end includes

```

```
30  //***** MY INCLUDES*******
31  #include <vector>
32  #include <map>
33  #include <wx/socket.h>
34  #include "TaskBarIcon.h"
35  #include "../gloveAPI/ScaledHandPosition.h"
36  #include "../gloveAPI/RawHandPosition.h"
37  #include "../gloveAPI/GloveCalibration.h"
38  #include "../gloveAPI/Glove.h"
39  //************************
40
41
42  /*!
43   * Forward declarations
44   */
45
46  ////@begin forward declarations
47  class wxToggleButton;
48  class wxGrid;
49  ////@end forward declarations
50  class TServerHandler;
51  class TGloveHandler;
52
53  /*!
54   * Control identifiers
55   */
56
57  ////@begin control identifiers
58  #define ID_ISISGLOVEMANAGER 10007
59  #define ID_NOTEBOOK_GLOVEMANAGER 10008
60  #define ID_PANEL_GLOVE 10009
61  #define ID_STATICTEXT 10000
62  #define ID_CHOICE_GLOVE_SERIALPORT 10001
63  #define ID_TOGGLEBUTTON_GLOVE_OPENCLOSE 10011
64  #define ID_STATICTEXT1 10013
65  #define ID_BUTTON_GLOVE_CALIBSAVE 10014
66  #define ID_BUTTON_GLOVE_CALIBLOAD 10016
67  #define ID_BUTTON_GLOVE_CALIBRESET 10017
68  #define ID_GRID 10025
69  #define ID_PANEL_RECOGNITION 10005
70  #define ID_SLIDER_RECOGNITION_THRESHOLD 10002
71  #define ID_STATICTEXT_RECOGNITION_CURRENTGESTURE 10003
72  #define ID_BUTTON_RECOGNITION_ADDGESTURE 10006
73  #define ID_LISTBOX_RECOGNITION_KNOWNGESTURES 10023
74  #define ID_BUTTON_RECOGNITION_DELETEGESTURES 10024
75  #define ID_PANEL_TCPSERVER 10004
76  #define ID_COMBOBOX_TCPSERVER_LIF 10010
77  #define ID_TEXTCTRL_TCPSERVER_LPORT 10015
78  #define ID_TOGGLEBUTTON_TCPSERVER_STARTSTOP 10019
79  #define ID_LISTBOX_TCPSERVER_CLIENTS 10020
80  #define ID_PANEL_EVENTLOG 10012
81  #define ID_TEXTCTRL_EVENTLOG_TXTLOG 10021
82  #define ID_BUTTON_EVENTLOG_CLEAR 10022
83  #define SYMBOL_ISISGLOVETABS_STYLE wxCAPTION|wxRESIZE_BORDER|wxSYSTEM_MENU|wxCLOSE_BOX
84  #define SYMBOL_ISISGLOVETABS_TITLE _("ISIS GloveManager")
85  #define SYMBOL_ISISGLOVETABS_IDNAME ID_ISISGLOVEMANAGER
```

```
86  #define SYMBOL_ISISGLOVETABS_SIZE wxSize(480, 360)
87  #define SYMBOL_ISISGLOVETABS_POSITION wxDefaultPosition
88
89  #define THREAD_EVENT 500
90  ////@end control identifiers
91
92  /*!
93   * Compatibility
94   */
95
96  #ifndef wxCLOSE_BOX
97  #define wxCLOSE_BOX 0x1000
98  #endif
99
100 /*!
101  * ISISgloveTabs class declaration
102  */
103
104 class ISISgloveTabs: public wxFrame
105 {
106     DECLARE_CLASS( ISISgloveTabs )
107     DECLARE_EVENT_TABLE()
108
109 public:
110     /// Constructors
111     ISISgloveTabs();
112     ISISgloveTabs( wxWindow* parent,
113         wxWindowID id = SYMBOL_ISISGLOVETABS_IDNAME,
114         const wxString& caption = SYMBOL_ISISGLOVETABS_TITLE,
115         const wxPoint& pos = SYMBOL_ISISGLOVETABS_POSITION,
116         const wxSize& size = SYMBOL_ISISGLOVETABS_SIZE,
117         long style = SYMBOL_ISISGLOVETABS_STYLE );
118
119     bool Create( wxWindow* parent,
120         wxWindowID id = SYMBOL_ISISGLOVETABS_IDNAME,
121         const wxString& caption = SYMBOL_ISISGLOVETABS_TITLE,
122         const wxPoint& pos = SYMBOL_ISISGLOVETABS_POSITION,
123         const wxSize& size = SYMBOL_ISISGLOVETABS_SIZE,
124         long style = SYMBOL_ISISGLOVETABS_STYLE );
125
126     /// Destructor
127     ~ISISgloveTabs();
128
129 private:
130     /// Initialises member variables
131     void Init();
132
133     /// Creates the controls and sizers
134     void CreateControls();
135
136 ////@begin ISISgloveTabs event handler declarations
137
138     /// wxEVT_COMMAND_CHECKBOX_CLICKED event handler for ID_TOGGLEBUTTON_GLOVE_OPENCLOSE
139     void OnTogglebuttonGloveOpencloseClick( wxCommandEvent& event );
140
141     /// wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_GLOVE_CALIBSAVE
```

```
142       void OnButtonGloveCalibsaveClick( wxCommandEvent& event );
143
144       /// wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_GLOVE_CALIBLOAD
145       void OnButtonGloveCalibloadClick( wxCommandEvent& event );
146
147       /// wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_GLOVE_CALIBRESET
148       void OnButtonGloveCalibresetClick( wxCommandEvent& event );
149
150       /// wxEVT_COMMAND_SLIDER_UPDATED event handler for ID_SLIDER_RECOGNITION_THRESHOLD
151       void OnSliderRecognitionThresholdUpdated( wxCommandEvent& event );
152
153       /// wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_RECOGNITION_ADDGESTURE
154       void OnButtonRecognitionAddgestureClick( wxCommandEvent& event );
155
156       /// wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_RECOGNITION_DELETEGESTURES
157       void OnButtonRecognitionDeletegesturesClick( wxCommandEvent& event );
158
159       /// wxEVT_COMMAND_CHECKBOX_CLICKED event handler for ID_TOGGLEBUTTON_TCPSERVER_STARTSTOP
160       void OnTogglebuttonTcpserverStartstopClick( wxCommandEvent& event );
161
162       /// wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_EVENTLOG_CLEAR
163       void OnButtonEventlogClearClick( wxCommandEvent& event );
164
165   ////@end ISISgloveTabs event handler declarations
166
167
168       //event handler used in thread communication and sync
169       void OnThreadEvent(wxCommandEvent& event);
170
171
172   ////@begin ISISgloveTabs member function declarations
173
174       /// Retrieves bitmap resources
175       wxBitmap GetBitmapResource( const wxString& name );
176
177       /// Retrieves icon resources
178       wxIcon GetIconResource( const wxString& name );
179   ////@end ISISgloveTabs member function declarations
180
181       /// Should we show tooltips?
182       static bool ShowToolTips();
183
184   ////@begin ISISgloveTabs member variables
185       wxChoice* choiceSerialPort;
186       wxToggleButton* btnOpenCloseGlove;
187       wxStaticBox* sboxDeviceInfo;
188       wxStaticText* txtDImodelLabel;
189       wxStaticText* txtDImodel;
190       wxStaticText* txtDIhandednessLabel;
191       wxStaticText* txtDIhandedness;
192       wxStaticText* txtDIdriverinfoLabel;
193       wxStaticText* txtDIdriverinfo;
194       wxStaticBox* sboxCalibration;
195       wxStaticText* txtLoadedCalibrationLabel;
196       wxStaticText* txtLoadedCalibration;
197       wxButton* btnCalibrationSave;
```

```
198       wxButton* btnCalibrationLoad;
199       wxButton* btnCalibrationReset;
200       wxGrid* gloveGrid;
201       wxSlider* sliderRecognition;
202       wxStaticText* txtRecognizedGesture;
203       wxButton* btnAddGesture;
204       wxListBox* listBoxKnownGestures;
205       wxComboBox* comboListIF;
206       wxTextCtrl* txtListPort;
207       wxToggleButton* btnServerStartStop;
208       wxListBox* listBoxClients;
209       wxTextCtrl* txtlog;
210   ////@end ISISgloveTabs member variables
211
212       //********** BEGIN HAND-CODED STUFF ************
213
214       // global
215       private:
216           void ConfigureControls();
217           void initTaskBarIcon();
218           void destroyTaskBarIcon();
219           void OnClose(wxCloseEvent& event);
220       public:
221           void exit();
222           void ErrorMessageBox(wxString msg);
223           wxString getDataDir() { return dataDir;};
224
225       //idle update stuff
226       private:
227           void OnIdle(wxIdleEvent& event);
228           void idleUpdateGrid();
229           //void idleUpdatePosMatch();
230
231       // glove tab
232       private:
233           void startStopGlove();
234           void startGlove();
235           void stopGlove();
236           void onGloveStarted();
237           void onGloveStopped();
238
239           void saveCalibration();
240           void loadCalibration();
241           void resetCalibration();
242
243           void EnableCalibrationBox(bool);
244           void EnableDeviceInfoBox(bool);
245       public:
246           void setGridDataSource(ScaledHandPosition *shp, RawHandPosition *rhp, GloveCalibration *gc);
247
248
249       //recognition tab
250       private:
251           void EnableGestureControls(bool status);
252           void setCompareThresholdBySliderValue();
253           void onPositionMatch();
```

```
254
255          void deleteSelectedGesture();
256          void addCurrentGesture();
257
258      public:
259          void setGestureMatchString(std::string &s);
260          void refreshKnownGesturesList(std::map<ScaledHandPosition,std::string>);
261
262
263      //server tab
264      private:
265          void startStopServer();
266          void startServer();
267          void stopServer();
268          void onServerStarted();
269          void onServerStopped();
270          void EnableServerControls(bool);
271
272      public:
273          void refreshClientList(std::vector<wxSocketBase*> clients);
274
275      // log tab
276      public:
277          void log (const wxString& text);
278
279
280     protected:
281          TaskBarIcon   *m_taskBarIcon;
282      #if defined(__WXCOCOA__)
283          TaskBarIcon   *m_dockIcon;
284      #endif
285
286      private:
287          bool closingApp;
288          bool serverStoppedByGloveStop;
289
290          ScaledHandPosition *c_shp;
291          RawHandPosition *c_rhp;
292          GloveCalibration *c_gc;
293
294          std::string posMatch;
295          wxString dataDir;
296      //********** END HAND-CODED STUFF **************
297  };
298
299  #endif
300      // _ISISGLOVETABS_H_
```

```
1   #if defined(__GNUG__) && !defined(NO_GCC_PRAGMA)
2   #pragma implementation "ISISgloveTabs.h"
3   #endif
4
5   // For compilers that support precompilation, includes "wx/wx.h".
6   #include "wx/wxprec.h"
7
8   #ifdef __BORLANDC__
9   #pragma hdrstop
```

```
10  #endif
11
12  #ifndef WX_PRECOMP
13  #include "wx/wx.h"
14  #endif
15
16  ////@begin includes
17  ////@end includes
18
19  #include "ISISgloveTabs.h"
20
21  ////@begin XPM images
22  #include "gloveIcon32.xpm"
23  ////@end XPM images
24
25  //begin myStuff
26  #include <map>
27  #include <string>
28  #include <sstream>
29  #include <fstream>
30
31  #include <wx/socket.h>
32  #include <wx/thread.h>
33  #include <wx/wfstream.h>
34  #include <wx/txtstrm.h>
35  #include <wx/filename.h>
36  #include <wx/datetime.h>
37
38  #include "../GuiThreadsComm.h"
39  #include "../TServerListener.h"
40  #include "../TGloveHandler.h"
41  #include "../GestureRecognition.h"
42  #include "../ClientsHandling.h"
43
44  TServerListener *serverThread;
45  TGloveHandler *gloveThread;
46
47  //end myStuff
48
49  /*!
50   * ISISgloveTabs type definition
51   */
52
53  IMPLEMENT_CLASS( ISISgloveTabs, wxFrame )
54
55  /*!
56   * ISISgloveTabs event table definition
57   */
58
59  BEGIN_EVENT_TABLE( ISISgloveTabs, wxFrame )
60
61  ////@begin ISISgloveTabs event table entries
62      EVT_TOGGLEBUTTON( ID_TOGGLEBUTTON_GLOVE_OPENCLOSE, ISISgloveTabs::OnTogglebuttonGloveOpencloseClick )
63
64      EVT_BUTTON( ID_BUTTON_GLOVE_CALIBSAVE, ISISgloveTabs::OnButtonGloveCalibsaveClick )
65
```

```
66      EVT_BUTTON( ID_BUTTON_GLOVE_CALIBLOAD, ISISgloveTabs::OnButtonGloveCalibloadClick )
67
68      EVT_BUTTON( ID_BUTTON_GLOVE_CALIBRESET, ISISgloveTabs::OnButtonGloveCalibresetClick )
69
70      EVT_SLIDER( ID_SLIDER_RECOGNITION_THRESHOLD, ISISgloveTabs::OnSliderRecognitionThresholdUpdated )
71
72      EVT_BUTTON( ID_BUTTON_RECOGNITION_ADDGESTURE, ISISgloveTabs::OnButtonRecognitionAddgestureClick )
73
74      EVT_BUTTON( ID_BUTTON_RECOGNITION_DELETEGESTURES, ISISgloveTabs::OnButtonRecognitionDeletegesturesClick )
75
76      EVT_TOGGLEBUTTON(ID_TOGGLEBUTTON_TCPSERVER_STARTSTOP, ISISgloveTabs::OnTogglebuttonTcpserverStartstopClick)
77
78      EVT_BUTTON( ID_BUTTON_EVENTLOG_CLEAR, ISISgloveTabs::OnButtonEventlogClearClick )
79
80  ////@end ISISgloveTabs event table entries
81      EVT_CLOSE(ISISgloveTabs::OnClose)
82
83      EVT_MENU(THREAD_EVENT, ISISgloveTabs::OnThreadEvent)
84
85      EVT_IDLE(ISISgloveTabs::OnIdle)
86
87  END_EVENT_TABLE()
88
89  /*!
90   * ISISgloveTabs constructors
91   */
92
93
94  void ISISgloveTabs::ErrorMessageBox(wxString msg) {
95      wxMessageBox(msg,  _("Error!"), wxICON_ERROR);
96  }
97
98
99  ISISgloveTabs::ISISgloveTabs()
100 {
101     Init();
102 }
103
104 ISISgloveTabs::ISISgloveTabs(wxWindow* parent,
105                              wxWindowID id,
106                              const wxString& caption,
107                              const wxPoint& pos,
108                              const wxSize& size,
109                              long style ) {
110     Init();
111     Create( parent, id, caption, pos, size, style );
112 }
113
114 /*!
115  * ISISgloveTabs creator
116  */
117
118 bool ISISgloveTabs::Create( wxWindow* parent,
119                             wxWindowID id,
120                             const wxString& caption,
121                             const wxPoint& pos,
```

```
122                             const wxSize& size,
123                             long style ){
124     ////@begin ISISgloveTabs creation
125         wxFrame::Create( parent, id, caption, pos, size, style );
126
127         CreateControls();
128         //SetIcon(GetIconResource(wxT("gloveIcon32.xpm")));
129         SetIcon(image_xpm);
130         Centre();
131     ////@end ISISgloveTabs creation
132         return true;
133     }
134
135     /*!
136      * ISISgloveTabs destructor
137      */
138
139     ISISgloveTabs::~ISISgloveTabs()
140     {
141     ////@begin ISISgloveTabs destruction
142     ////@end ISISgloveTabs destruction
143         destroyTaskBarIcon();
144     }
145
146     /*!
147      * Member initialisation
148      */
149
150     void ISISgloveTabs::Init()
151     {
152     ////@begin ISISgloveTabs member initialisation
153         choiceSerialPort = NULL;
154         btnOpenCloseGlove = NULL;
155         sboxDeviceInfo = NULL;
156         txtDImodelLabel = NULL;
157         txtDImodel = NULL;
158         txtDIhandednessLabel = NULL;
159         txtDIhandedness = NULL;
160         txtDIdriverinfoLabel = NULL;
161         txtDIdriverinfo = NULL;
162         sboxCalibration = NULL;
163         txtLoadedCalibrationLabel = NULL;
164         txtLoadedCalibration = NULL;
165         btnCalibrationSave = NULL;
166         btnCalibrationLoad = NULL;
167         btnCalibrationReset = NULL;
168         gloveGrid = NULL;
169         sliderRecognition = NULL;
170         txtRecognizedGesture = NULL;
171         btnAddGesture = NULL;
172         listBoxKnownGestures = NULL;
173         comboListIF = NULL;
174         txtListPort = NULL;
175         btnServerStartStop = NULL;
176         listBoxClients = NULL;
177         txtlog = NULL;
```

```
178  ////@end ISISgloveTabs member initialisation
179      dataDir = _("./data/");
180  }
181
182  /*!
183   * Control creation for ISISgloveTabs
184   */
185
186  void ISISgloveTabs::CreateControls()
187  {
188  ////@begin ISISgloveTabs content construction
189
190      ISISgloveTabs* itemFrame1 = this;
191
192      wxNotebook* itemNotebook2 = new wxNotebook( itemFrame1, ID_NOTEBOOK_GLOVEMANAGER,
193          wxDefaultPosition, wxDefaultSize, wxBK_DEFAULT );
194
195      wxPanel* itemPanel3 = new wxPanel( itemNotebook2, ID_PANEL_GLOVE,
196          wxDefaultPosition, wxDefaultSize, wxSUNKEN_BORDER|wxTAB_TRAVERSAL );
197      wxBoxSizer* itemBoxSizer4 = new wxBoxSizer(wxHORIZONTAL);
198      itemPanel3->SetSizer(itemBoxSizer4);
199
200      wxBoxSizer* itemBoxSizer5 = new wxBoxSizer(wxVERTICAL);
201      itemBoxSizer4->Add(itemBoxSizer5, 0, wxGROW|wxALL, 5);
202      wxBoxSizer* itemBoxSizer6 = new wxBoxSizer(wxHORIZONTAL);
203      itemBoxSizer5->Add(itemBoxSizer6, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
204      wxStaticText* itemStaticText7 = new wxStaticText( itemPanel3, ID_STATICTEXT,
205          _("Port"), wxDefaultPosition, wxDefaultSize, 0 );
206      itemBoxSizer6->Add(itemStaticText7, 0, wxALIGN_CENTER_VERTICAL|wxALL|wxADJUST_MINSIZE, 5);
207
208      wxArrayString choiceSerialPortStrings;
209
210  //    #ifdef WIN32
211      choiceSerialPortStrings.Add(_("<auto>"));
212      choiceSerialPortStrings.Add(_("COM1"));
213      choiceSerialPortStrings.Add(_("COM2"));
214      choiceSerialPortStrings.Add(_("COM3"));
215      choiceSerialPortStrings.Add(_("COM4"));
216      choiceSerialPortStrings.Add(_("COM5"));
217      choiceSerialPortStrings.Add(_("COM6"));
218      choiceSerialPortStrings.Add(_("COM7"));
219      choiceSerialPortStrings.Add(_("COM8"));
220  //    #else
221      choiceSerialPortStrings.Add(_("/dev/glove"));
222      choiceSerialPortStrings.Add(_("/dev/ttyS0"));
223  //    #endif
224      choiceSerialPort = new wxChoice( itemPanel3, ID_CHOICE_GLOVE_SERIALPORT,
225          wxDefaultPosition, wxDefaultSize, choiceSerialPortStrings, 0 );
226      choiceSerialPort->SetStringSelection(_("<auto>"));
227      itemBoxSizer6->Add(choiceSerialPort, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
228
229      btnOpenCloseGlove = new wxToggleButton( itemPanel3, ID_TOGGLEBUTTON_GLOVE_OPENCLOSE,
230          _("Open Device"), wxDefaultPosition, wxDefaultSize, 0 );
231      btnOpenCloseGlove->SetValue(false);
232      itemBoxSizer6->Add(btnOpenCloseGlove, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
233
```

```
234       sboxDeviceInfo = new wxStaticBox(itemPanel3, wxID_ANY, _("Device Info"));
235       wxStaticBoxSizer* itemStaticBoxSizer10 = new wxStaticBoxSizer(sboxDeviceInfo, wxVERTICAL);
236       itemBoxSizer5->Add(itemStaticBoxSizer10, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
237       wxGridSizer* itemGridSizer11 = new wxGridSizer(3, 2, 0, 0);
238       itemStaticBoxSizer10->Add(itemGridSizer11, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
239       txtDImodelLabel = new wxStaticText( itemPanel3, wxID_STATIC,
240           _("Model:"), wxDefaultPosition, wxDefaultSize, 0 );
241       itemGridSizer11->Add(txtDImodelLabel, 0,
242           wxALIGN_RIGHT|wxALIGN_CENTER_VERTICAL|wxALL|wxADJUST_MINSIZE, 5);
243
244       txtDImodel = new wxStaticText( itemPanel3, wxID_STATIC,
245           _("<unknown>"), wxDefaultPosition, wxDefaultSize, 0 );
246       itemGridSizer11->Add(txtDImodel, 0,
247           wxALIGN_LEFT|wxALIGN_CENTER_VERTICAL|wxALL|wxADJUST_MINSIZE, 5);
248
249       txtDIhandednessLabel = new wxStaticText( itemPanel3, wxID_STATIC,
250           _("Handedness:"), wxDefaultPosition, wxDefaultSize, 0 );
251       itemGridSizer11->Add(txtDIhandednessLabel, 0, wxALIGN_RIGHT|wxALIGN_CENTER_VERTICAL|wxALL|wxADJUST_MINSIZE,
252
253       txtDIhandedness = new wxStaticText( itemPanel3, wxID_STATIC,
254           _("<unknown>"), wxDefaultPosition, wxDefaultSize, 0 );
255       itemGridSizer11->Add(txtDIhandedness, 0, wxALIGN_LEFT|wxALIGN_CENTER_VERTICAL|wxALL|wxADJUST_MINSIZE, 5);
256
257       /*txtDIdriverinfoLabel = new wxStaticText( itemPanel3, wxID_STATIC,
258           _("Driver Info:"), wxDefaultPosition, wxDefaultSize, 0 );
259       itemGridSizer11->Add(txtDIdriverinfoLabel, 0, wxALIGN_RIGHT|wxALIGN_CENTER_VERTICAL|wxALL|wxADJUST_MINSIZE,
260
261       txtDIdriverinfo = new wxStaticText( itemPanel3, wxID_STATIC,
262           _("<unknown>"), wxDefaultPosition, wxDefaultSize, 0 );
263       itemGridSizer11->Add(txtDIdriverinfo, 0, wxALIGN_LEFT|wxALIGN_CENTER_VERTICAL|wxALL|wxADJUST_MINSIZE, 5);
264       */
265
266       sboxCalibration = new wxStaticBox(itemPanel3, wxID_ANY, _("Calibration"));
267       wxStaticBoxSizer* itemStaticBoxSizer18 = new wxStaticBoxSizer(sboxCalibration, wxVERTICAL);
268       itemBoxSizer5->Add(itemStaticBoxSizer18, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
269       wxBoxSizer* itemBoxSizer19 = new wxBoxSizer(wxHORIZONTAL);
270       itemStaticBoxSizer18->Add(itemBoxSizer19, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
271       txtLoadedCalibrationLabel = new wxStaticText( itemPanel3, ID_STATICTEXT1,
272           _("Loaded Calibration:"), wxDefaultPosition, wxDefaultSize, 0 );
273       itemBoxSizer19->Add(txtLoadedCalibrationLabel, 0, wxALIGN_CENTER_VERTICAL|wxALL|wxADJUST_MINSIZE, 5);
274
275       txtLoadedCalibration = new wxStaticText( itemPanel3, wxID_STATIC,
276           _("<none>"), wxDefaultPosition, wxDefaultSize, 0 );
277       itemBoxSizer19->Add(txtLoadedCalibration, 0, wxALIGN_CENTER_VERTICAL|wxALL|wxADJUST_MINSIZE, 5);
278
279       wxBoxSizer* itemBoxSizer22 = new wxBoxSizer(wxHORIZONTAL);
280       itemStaticBoxSizer18->Add(itemBoxSizer22, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
281       btnCalibrationSave = new wxButton( itemPanel3, ID_BUTTON_GLOVE_CALIBSAVE,
282           _("Save..."), wxDefaultPosition, wxDefaultSize, 0 );
283       itemBoxSizer22->Add(btnCalibrationSave, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
284
285       btnCalibrationLoad = new wxButton( itemPanel3, ID_BUTTON_GLOVE_CALIBLOAD,
286           _("Load..."), wxDefaultPosition, wxDefaultSize, 0 );
287       itemBoxSizer22->Add(btnCalibrationLoad, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
288
289       btnCalibrationReset = new wxButton( itemPanel3, ID_BUTTON_GLOVE_CALIBRESET,
```

```
290         _("Reset"), wxDefaultPosition, wxDefaultSize, 0 );
291     itemStaticBoxSizer18->Add(btnCalibrationReset, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
292
293     wxBoxSizer* itemBoxSizer26 = new wxBoxSizer(wxVERTICAL);
294     itemBoxSizer4->Add(itemBoxSizer26, 0, wxGROW|wxALL, 5);
295
296     /* begin grid hand-modified */
297     gloveGrid = new wxGrid( itemPanel3, ID_GRID, wxDefaultPosition, wxSize(240, 320), 0 );
298
299     gloveGrid->SetDefaultColSize(32);
300     gloveGrid->SetDefaultRowSize(10);
301     gloveGrid->SetColLabelSize(14);
302     gloveGrid->SetRowLabelSize(100);
303     /* end grid hand-modified */
304
305     gloveGrid->CreateGrid(18, 4, wxGrid::wxGridSelectCells);
306     itemBoxSizer26->Add(gloveGrid, 0, wxGROW, 5);
307
308     itemNotebook2->AddPage(itemPanel3, _("Glove"));
309
310     wxPanel* itemPanel28 = new wxPanel( itemNotebook2, ID_PANEL_RECOGNITION,
311         wxDefaultPosition, wxDefaultSize, wxSUNKEN_BORDER|wxTAB_TRAVERSAL );
312     wxBoxSizer* itemBoxSizer29 = new wxBoxSizer(wxHORIZONTAL);
313     itemPanel28->SetSizer(itemBoxSizer29);
314
315     wxBoxSizer* itemBoxSizer30 = new wxBoxSizer(wxVERTICAL);
316     itemBoxSizer29->Add(itemBoxSizer30, 1, wxALIGN_CENTER_VERTICAL|wxALL, 5);
317     wxBoxSizer* itemBoxSizer31 = new wxBoxSizer(wxVERTICAL);
318     itemBoxSizer30->Add(itemBoxSizer31, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
319     wxStaticText* itemStaticText32 = new wxStaticText( itemPanel28, wxID_STATIC,
320         _("Recognition threshold"), wxDefaultPosition, wxDefaultSize, 0 );
321     itemBoxSizer31->Add(itemStaticText32, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
322
323     wxBoxSizer* itemBoxSizer33 = new wxBoxSizer(wxHORIZONTAL);
324     itemBoxSizer31->Add(itemBoxSizer33, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
325     wxStaticText* itemStaticText34 = new wxStaticText( itemPanel28, wxID_STATIC,
326         _("MIN"), wxDefaultPosition, wxDefaultSize, 0 );
327     itemBoxSizer33->Add(itemStaticText34, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
328
329     sliderRecognition = new wxSlider( itemPanel28, ID_SLIDER_RECOGNITION_THRESHOLD,
330         25, 5, 50, wxDefaultPosition, wxDefaultSize, wxSL_HORIZONTAL | wxSL_AUTOTICKS | wxSL_LABELS);
331     itemBoxSizer33->Add(sliderRecognition, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
332
333     wxStaticText* itemStaticText36 = new wxStaticText( itemPanel28, wxID_STATIC,
334         _("MAX"), wxDefaultPosition, wxDefaultSize, 0 );
335     itemBoxSizer33->Add(itemStaticText36, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
336
337     wxBoxSizer* itemBoxSizer37 = new wxBoxSizer(wxVERTICAL);
338     itemBoxSizer30->Add(itemBoxSizer37, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
339     wxStaticText* itemStaticText38 = new wxStaticText( itemPanel28, wxID_STATIC,
340         _("Current Hand Gesture:"), wxDefaultPosition, wxDefaultSize, 0 );
341     itemBoxSizer37->Add(itemStaticText38, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
342
343     txtRecognizedGesture = new wxStaticText( itemPanel28, ID_STATICTEXT_RECOGNITION_CURRENTGESTURE,
344         _("<not recognized>"), wxDefaultPosition, wxDefaultSize, 0 );
345     itemBoxSizer37->Add(txtRecognizedGesture, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
```

```
346
347       btnAddGesture = new wxButton( itemPanel28, ID_BUTTON_RECOGNITION_ADDGESTURE,
348           _("Add to known..."), wxDefaultPosition, wxDefaultSize, 0 );
349       itemBoxSizer37->Add(btnAddGesture, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
350
351       wxBoxSizer* itemBoxSizer41 = new wxBoxSizer(wxVERTICAL);
352       itemBoxSizer29->Add(itemBoxSizer41, 0, wxGROW|wxALL, 5);
353       wxStaticText* itemStaticText42 = new wxStaticText( itemPanel28, wxID_STATIC,
354           _("Known hand gestures"), wxDefaultPosition, wxDefaultSize, 0 );
355       itemBoxSizer41->Add(itemStaticText42, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
356
357       wxArrayString listBoxKnownGesturesStrings;
358       listBoxKnownGestures = new wxListBox( itemPanel28, ID_LISTBOX_RECOGNITION_KNOWNGESTURES,
359           wxDefaultPosition, wxDefaultSize, listBoxKnownGesturesStrings, wxLB_SINGLE );
360       itemBoxSizer41->Add(listBoxKnownGestures, 1, wxGROW|wxALL, 5);
361
362       wxButton* itemButton44 = new wxButton( itemPanel28, ID_BUTTON_RECOGNITION_DELETEGESTURES,
363           _("Delete selected"), wxDefaultPosition, wxDefaultSize, 0 );
364       itemBoxSizer41->Add(itemButton44, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
365
366       itemNotebook2->AddPage(itemPanel28, _("Recognition"));
367
368       wxPanel* itemPanel45 = new wxPanel( itemNotebook2, ID_PANEL_TCPSERVER,
369           wxDefaultPosition, wxDefaultSize, wxSUNKEN_BORDER|wxTAB_TRAVERSAL );
370       wxBoxSizer* itemBoxSizer46 = new wxBoxSizer(wxVERTICAL);
371       itemPanel45->SetSizer(itemBoxSizer46);
372
373       wxGridSizer* itemGridSizer47 = new wxGridSizer(2, 2, 0, 0);
374       itemBoxSizer46->Add(itemGridSizer47, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
375       wxStaticText* itemStaticText48 = new wxStaticText( itemPanel45, wxID_STATIC,
376           _("Network Interface Binding"), wxDefaultPosition, wxDefaultSize, 0 );
377       itemGridSizer47->Add(itemStaticText48, 0, wxALIGN_RIGHT|wxALIGN_CENTER_VERTICAL|wxALL, 5);
378
379       wxArrayString comboListIFStrings;
380       comboListIFStrings.Add(_("Any"));
381       comboListIFStrings.Add(_("127.0.0.1"));
382       comboListIF = new wxComboBox( itemPanel45, ID_COMBOBOX_TCPSERVER_LIF,
383           _("Any"), wxDefaultPosition, wxSize(100, -1), comboListIFStrings, wxCB_DROPDOWN );
384       comboListIF->SetStringSelection(_("Any"));
385       itemGridSizer47->Add(comboListIF, 0, wxALIGN_LEFT|wxALIGN_CENTER_VERTICAL|wxALL, 5);
386
387       wxStaticText* itemStaticText50 = new wxStaticText( itemPanel45, wxID_STATIC,
388           _("Listening Port"), wxDefaultPosition, wxDefaultSize, 0 );
389       itemGridSizer47->Add(itemStaticText50, 0, wxALIGN_RIGHT|wxALIGN_CENTER_VERTICAL|wxALL, 5);
390
391       txtListPort = new wxTextCtrl( itemPanel45, ID_TEXTCTRL_TCPSERVER_LPORT,
392           _("4242"), wxDefaultPosition, wxSize(50, -1), 0 );
393       txtListPort->SetMaxLength(5);
394       itemGridSizer47->Add(txtListPort, 0, wxALIGN_LEFT|wxALIGN_CENTER_VERTICAL|wxALL, 5);
395
396       btnServerStartStop = new wxToggleButton( itemPanel45, ID_TOGGLEBUTTON_TCPSERVER_STARTSTOP,
397           _("Start Listening"), wxDefaultPosition, wxDefaultSize, 0 );
398       btnServerStartStop->SetValue(false);
399       itemBoxSizer46->Add(btnServerStartStop, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
400
401       wxStaticBox* itemStaticBoxSizer53Static = new wxStaticBox(itemPanel45, wxID_ANY, _("Connected Clients"));
```

```
402       wxStaticBoxSizer* itemStaticBoxSizer53 = new wxStaticBoxSizer(itemStaticBoxSizer53Static, wxVERTICAL);
403       itemBoxSizer46->Add(itemStaticBoxSizer53, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
404       wxArrayString listBoxClientsStrings;
405       listBoxClients = new wxListBox( itemPanel45, ID_LISTBOX_TCPSERVER_CLIENTS,
406           wxDefaultPosition, wxSize(-1, 140), listBoxClientsStrings, wxLB_SINGLE );
407       itemStaticBoxSizer53->Add(listBoxClients, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
408
409       itemNotebook2->AddPage(itemPanel45, _("TCP Server"));
410
411       wxPanel* itemPanel55 = new wxPanel( itemNotebook2, ID_PANEL_EVENTLOG,
412           wxDefaultPosition, wxDefaultSize, wxSUNKEN_BORDER|wxTAB_TRAVERSAL );
413       wxBoxSizer* itemBoxSizer56 = new wxBoxSizer(wxVERTICAL);
414       itemPanel55->SetSizer(itemBoxSizer56);
415
416       txtlog = new wxTextCtrl( itemPanel55, ID_TEXTCTRL_EVENTLOG_TXTLOG,
417           _T(""), wxDefaultPosition, wxDefaultSize, wxTE_MULTILINE );
418       itemBoxSizer56->Add(txtlog, 1, wxGROW|wxALL, 5);
419
420       wxButton* itemButton58 = new wxButton( itemPanel55, ID_BUTTON_EVENTLOG_CLEAR,
421           _("Clear"), wxDefaultPosition, wxDefaultSize, 0 );
422       itemBoxSizer56->Add(itemButton58, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
423
424       itemNotebook2->AddPage(itemPanel55, _("Event log"));
425
426   ////@end ISISgloveTabs content construction
427       ConfigureControls();
428   }
429
430
431   /*!
432    * wxEVT_COMMAND_CHECKBOX_CLICKED event handler for ID_TOGGLEBUTTON_GLOVE_OPENCLOSE
433    */
434   void ISISgloveTabs::OnTogglebuttonGloveOpencloseClick( wxCommandEvent& event ) {
435       startStopGlove();
436   }
437
438   /*!
439    * wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_GLOVE_CALIBSAVE
440    */
441   void ISISgloveTabs::OnButtonGloveCalibsaveClick( wxCommandEvent& event ) {
442       saveCalibration();
443   }
444
445   /*!
446    * wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_GLOVE_CALIBLOAD
447    */
448   void ISISgloveTabs::OnButtonGloveCalibloadClick( wxCommandEvent& event ) {
449       loadCalibration();
450   }
451
452   /*!
453    * wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_GLOVE_CALIBRESET
454    */
455   void ISISgloveTabs::OnButtonGloveCalibresetClick( wxCommandEvent& event ) {
456       resetCalibration();
457   }
```

```
458
459  /*!
460   * wxEVT_COMMAND_SLIDER_UPDATED event handler for ID_SLIDER_RECOGNITION_THRESHOLD
461   */
462  void ISISgloveTabs::OnSliderRecognitionThresholdUpdated( wxCommandEvent& event ) {
463      setCompareThresholdBySliderValue();
464  }
465
466  void ISISgloveTabs::setCompareThresholdBySliderValue() {
467      int val = sliderRecognition->GetValue();
468      float fval = ((float) val) / 100;
469      ScaledHandPosition::setCompareThreshold(fval);
470  }
471
472  /*!
473   * wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_RECOGNITION_ADDGESTURE
474   */
475  void ISISgloveTabs::OnButtonRecognitionAddgestureClick( wxCommandEvent& event ) {
476      addCurrentGesture();
477  }
478
479  /*!
480   * wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_RECOGNITION_DELETEGESTURES
481   */
482  void ISISgloveTabs::OnButtonRecognitionDeletegesturesClick( wxCommandEvent& event ) {
483      deleteSelectedGesture();
484  }
485
486  /*!
487   * wxEVT_COMMAND_CHECKBOX_CLICKED event handler for ID_TOGGLEBUTTON_TCPSERVER_STARTSTOP
488   */
489  void ISISgloveTabs::OnTogglebuttonTcpserverStartstopClick( wxCommandEvent& event ) {
490      startStopServer();
491  }
492
493  /*!
494   * wxEVT_COMMAND_BUTTON_CLICKED event handler for ID_BUTTON_EVENTLOG_CLEAR
495   */
496  void ISISgloveTabs::OnButtonEventlogClearClick( wxCommandEvent& event ) {
497      txtlog->Clear();
498  }
499
500  /*!
501   * Should we show tooltips?
502   */
503  bool ISISgloveTabs::ShowToolTips() {
504      return true;
505  }
506
507  /*!
508   * Get bitmap resources
509   */
510  wxBitmap ISISgloveTabs::GetBitmapResource( const wxString& name ) {
511      // Bitmap retrieval
512  ////@begin ISISgloveTabs bitmap retrieval
513      wxUnusedVar(name);
```

```
514       return wxNullBitmap;
515   ////@end ISISgloveTabs bitmap retrieval
516   }
517
518   /*!
519    * Get icon resources
520    */
521   wxIcon ISISgloveTabs::GetIconResource( const wxString& name ) {
522       // Icon retrieval
523   ////@begin ISISgloveTabs icon retrieval
524       wxUnusedVar(name);
525       if (name == _T("gloveIcon32.xpm"))
526       {
527           wxIcon icon(image_xpm);
528           return icon;
529       }
530       return wxNullIcon;
531   ////@end ISISgloveTabs icon retrieval
532   }
533
534   void ISISgloveTabs::EnableCalibrationBox(bool status) {
535       txtDImodel->Enable(status);
536       txtDImodelLabel->Enable(status);
537       //txtDIdriverinfo->Enable(status);
538       //txtDIdriverinfoLabel->Enable(status);
539       txtDIhandedness->Enable(status);
540       txtDIhandednessLabel->Enable(status);
541       sboxDeviceInfo->Enable(status);
542
543       if (status) {
544           Glove *g = GuiThreadsComm::getGloveThread()->getGlove();
545
546           wxString handedness;
547           if (g->isRightHand()) handedness = _("RIGHT"); else handedness = _("LEFT");
548           wxString model(g->getType(),wxConvUTF8);
549   //      wxString driverinfo((wxChar*)g->getDriverInfo(),wxConvUTF8);
550   //      txtDIdriverinfo->SetLabel(driverinfo);
551           txtDImodel->SetLabel(model);
552           txtDIhandedness->SetLabel(handedness);
553       } else {
554   //      txtDIdriverinfo->SetLabel(_("<unknown>"));
555           txtDImodel->SetLabel(_("<unknown>"));
556           txtDIhandedness->SetLabel(_("<unknown>"));
557       }
558   }
559
560   void ISISgloveTabs::EnableDeviceInfoBox(bool status) {
561       sboxCalibration->Enable(status);
562       txtLoadedCalibrationLabel->Enable(status);
563       txtLoadedCalibration->Enable(status);
564       btnCalibrationSave->Enable(status);
565       btnCalibrationLoad->Enable(status);
566       btnCalibrationReset->Enable(status);
567       txtLoadedCalibration->SetLabel(_("<none>"));
568   }
569
```

```
570 void ISISgloveTabs::EnableGestureControls(bool status) {
571     btnAddGesture->Enable(status);
572     txtRecognizedGesture->SetLabel( _("<none>"));
573 }
574
575 void ISISgloveTabs::EnableServerControls(bool status) {
576     if (status)
577         btnServerStartStop->SetLabel(_("Stop Listening"));
578     else
579         btnServerStartStop->SetLabel(_("Start Listening"));
580     choiceSerialPort->Enable(status);
581     txtListPort->Enable(status);
582     comboListIF->Enable(status);
583 }
584
585 void ISISgloveTabs::ConfigureControls() {
586     GuiThreadsComm::setFrm(this);
587
588     GestureRecognition::loadKnownGestures();
589     gloveGrid->SetColLabelValue(0,_("MIN"));
590     gloveGrid->SetColLabelValue(1,_("MAX"));
591     gloveGrid->SetColLabelValue(2,_("RAW"));
592     gloveGrid->SetColLabelValue(3,_("SCL"));
593
594     gloveGrid->SetRowLabelAlignment(wxALIGN_LEFT, wxALIGN_CENTRE);
595     gloveGrid->DisableDragGridSize();
596     gloveGrid->DisableDragColMove();
597     gloveGrid->DisableDragColSize();
598     gloveGrid->DisableDragRowSize();
599
600     wxArrayString sl(18);
601     sl.Add(_("00_THUMB NEAR"));
602     sl.Add(_("01_THUMB FAR"));
603     sl.Add(_("02_THUMBINDEX"));
604     sl.Add(_("03_INDEX NEAR"));
605     sl.Add(_("04_INDEX FAR"));
606     sl.Add(_("05_INDEXMIDDLE"));
607     sl.Add(_("06_MIDDLE NEAR"));
608     sl.Add(_("07_MIDDLE FAR"));
609     sl.Add(_("08_MIDDLERING"));
610     sl.Add(_("09_RING NEAR"));
611     sl.Add(_("10_RING FAR"));
612     sl.Add(_("11_RINGLITTLE"));
613     sl.Add(_("12_LITTLE NEAR"));
614     sl.Add(_("13_LITTLE FAR"));
615     sl.Add(_("14_THUMB PALM"));
616     sl.Add(_("15_WRIST BEND"));
617     sl.Add(_("16_PITCH"));
618     sl.Add(_("17_ROLL"));
619
620     for(int i = 0; i < 18; i++)
621         gloveGrid->SetRowLabelValue(i, sl[i]);
622
623     gloveGrid->EnableEditing(false);
624     txtlog->SetEditable(false);
625     initTaskBarIcon();
```

```
626
627         EnableCalibrationBox(false);
628         EnableDeviceInfoBox(false);
629         EnableGestureControls(false);
630
631         setCompareThresholdBySliderValue();
632
633         closingApp = false;
634         serverStoppedByGloveStop = false;
635     }
636
637     void ISISgloveTabs::initTaskBarIcon() {
638         m_taskBarIcon = new TaskBarIcon();
639         #if defined(__WXCOCOA__)
640         m_dockIcon = new TaskBarIcon(wxTaskBarIcon::DOCK);
641         #endif
642         m_taskBarIcon->SetIcon(image_xpm);
643         m_taskBarIcon->setFrame(this);
644     }
645
646     void ISISgloveTabs::destroyTaskBarIcon() {
647         delete m_taskBarIcon;
648         #if defined(__WXCOCOA__)
649         delete m_dockIcon;
650         #endif
651     }
652
653     void ISISgloveTabs::OnClose(wxCloseEvent &event) {
654         this->Show(false);
655     }
656
657     void ISISgloveTabs::exit() {
658         this->Show(false);
659         m_taskBarIcon->RemoveIcon();
660         closingApp = true;
661         if (GuiThreadsComm::isGloveActive())
662             stopGlove();
663         else
664             Destroy();
665     }
666
667     void ISISgloveTabs::log (const wxString& text) {
668         if (!wxThread::IsMain()) wxMutexGuiEnter();
669         wxString txt;
670         txt << wxDateTime::Now().FormatTime() << _(": ");
671         txt << text;
672         txtlog->AppendText(txt);
673         txtlog->ShowPosition(txtlog->GetLastPosition()); //autoscroll
674         if (!wxThread::IsMain()) wxMutexGuiLeave();
675     }
676
677     void ISISgloveTabs::idleUpdateGrid() {
678         if (!GuiThreadsComm::isGloveActive()) return;
679         wxString val;
680         for (int i= 0; i< 18; i++){
681             val =  wxString::Format(_("%4d"), c_gc->getRawMin((EfdSensors)i));
```

```
682             gloveGrid->SetCellValue(i,0,val);
683             val =   wxString::Format(_("%4d"), c_gc->getRawMax((EfdSensors)i));
684             gloveGrid->SetCellValue(i,1,val);
685             val =  wxString::Format(_("%4d"),  c_rhp->getSensorValue((EfdSensors)i));
686             gloveGrid->SetCellValue(i,2,val);
687             val =  wxString::Format(_("%1.2f"), c_shp->getSensorValue((EfdSensors)i));
688             gloveGrid->SetCellValue(i,3,val);
689         }
690     }
691
692     void ISISgloveTabs::setGridDataSource(ScaledHandPosition *shp, RawHandPosition *rhp, GloveCalibration *gc) {
693         c_shp = shp;
694         c_rhp = rhp;
695         c_gc = gc;
696     }
697
698     void ISISgloveTabs::startStopGlove() {
699          if  (btnOpenCloseGlove->GetValue()) { //if not pressed
700             startGlove();
701          } else {                              //if pressed
702             stopGlove();
703          }
704     }
705
706     void ISISgloveTabs::startGlove() {
707         try {
708             wxString selPort = choiceSerialPort->GetStringSelection();
709             std::string selectedPort = (const char*)selPort.mb_str();
710             gloveThread = new TGloveHandler(selectedPort);
711         } catch (GloveNotAvailableException) {
712             ErrorMessageBox(_("Cannot open glove on this port!"));
713             btnOpenCloseGlove->SetValue(false);
714             return;
715         }
716
717         if (gloveThread->Create() != wxTHREAD_NO_ERROR) {
718             ErrorMessageBox(_("Cannot create glove handler thread!"));
719             btnOpenCloseGlove->SetValue(false);
720             return;
721         }
722         gloveThread->Run();
723     }
724
725     void ISISgloveTabs::stopGlove() {
726         if (GuiThreadsComm::isServerActive()) {
727             serverStoppedByGloveStop = true;
728             stopServer();
729         } else {
730             gloveThread->terminate();
731         }
732     }
733
734     void ISISgloveTabs::onGloveStarted() {
735         btnOpenCloseGlove->SetLabel(_("Close Device"));
736         choiceSerialPort->Disable();
737
```

```
738         EnableCalibrationBox(true);
739         EnableDeviceInfoBox(true);
740         EnableGestureControls(true);
741     }
742
743     void ISISgloveTabs::onGloveStopped() {
744         btnOpenCloseGlove->SetLabel(_("Open Device"));
745         choiceSerialPort->Enable();
746
747         EnableCalibrationBox(false);
748         EnableDeviceInfoBox(false);
749         EnableGestureControls(false);
750
751         gloveGrid->ClearGrid();
752
753         //if onGloveStopped called by app close
754         if (closingApp) Destroy();
755     }
756
757     void ISISgloveTabs::startServer() {
758         wxSocketBase::Initialize(); //mandatory under win32 (prevents documented CRASH)
759         wxIPV4address listeningAddr;
760         wxString listeningInterface = comboListIF->GetValue();
761         if (listeningInterface == _("Any"))
762             listeningAddr.AnyAddress(); //INET_ANY
763         else {
764             if (!listeningAddr.Hostname(listeningInterface)) {
765                 ErrorMessageBox(_("Invalid listening interface!"));
766                 btnServerStartStop->SetValue(false);
767                 return;
768             }
769         }
770
771         wxString strport;
772         unsigned long longPort;
773
774         strport = txtListPort->GetValue();
775
776         if ((!strport.ToULong(&longPort))
777            || (longPort < 1 || longPort > 65535)
778            || (!listeningAddr.Service((unsigned short)longPort))) {
779             ErrorMessageBox(_("Invalid TCP port!"));
780             txtListPort->Clear();
781             txtListPort->SetFocus();
782             btnServerStartStop->SetValue(false);
783             return;
784         }
785
786         if (!GuiThreadsComm::isGloveActive()) {
787             ErrorMessageBox(_("No data to stream!\nPlease activate the glove before starting the server..."));
788             btnServerStartStop->SetValue(false);
789             return;
790         }
791
792         try {
793             serverThread = new TServerListener(listeningAddr);
```

```
794         } catch (CannotBindException) {
795             ErrorMessageBox(_("Cannot start server!\nInvalid interface or port in use by another socket."));
796             btnServerStartStop->SetValue(false);
797             return;
798         }
799 
800         if (serverThread->Create() != wxTHREAD_NO_ERROR) {
801             ErrorMessageBox(_("Cannot create TCP server thread!"));
802             btnServerStartStop->SetValue(false);
803             return;
804         }
805         serverThread->Run();
806     }
807 
808     void ISISgloveTabs::stopServer() {
809          serverThread->terminate();
810     }
811 
812     void ISISgloveTabs::onServerStarted() {
813         btnServerStartStop->SetLabel(_("Stop Listening"));
814         txtListPort->Disable();
815         comboListIF->Disable();
816     }
817     void ISISgloveTabs::onServerStopped() {
818         btnServerStartStop->SetLabel(_("Start Listening"));
819         choiceSerialPort->Enable();
820         txtListPort->Enable();
821         comboListIF->Enable();
822         if (btnServerStartStop->GetValue())
823             btnServerStartStop->SetValue(false);
824 
825         if (serverStoppedByGloveStop) {
826             gloveThread->terminate();
827             serverStoppedByGloveStop = false;
828         }
829     }
830 
831     void ISISgloveTabs::startStopServer() {
832         if  (btnServerStartStop->GetValue()) { //if not pressed
833             startServer();
834         } else { // if pressed
835             stopServer();
836         }
837     }
838 
839     void ISISgloveTabs::loadCalibration() {
840         wxString caption = _("Choose a calibration file...");
841         wxString wildcard = _("CAL files (*.cal)|*.cal");
842         wxString defaultDir = dataDir;
843         wxString defaultFilename = wxEmptyString;
844 
845         wxFileDialog dialog(this, caption, defaultDir, defaultFilename, wildcard, wxOPEN);
846         if (dialog.ShowModal() == wxID_OK) {
847             wxString path = dialog.GetPath();
848 
849             GloveCalibration *gc = new GloveCalibration((gloveThread->getGlove())->getNumSensors());
```

```
850
851         try{
852             gc->loadFromFile((const char*)path.mb_str());
853             (gloveThread->getGlove())->setCalibration(gc);
854             txtLoadedCalibration->SetLabel(dialog.GetFilename());
855             wxMessageBox(_("Glove calibration loaded and set"));
856         } catch (CannotLoadException) {
857             GuiThreadsComm::showError(_("Error in loading calibration!\nBad permissions/corrupted file?"));
858         }
859     } else {
860         wxMessageBox(_("Glove calibration NOT loaded"));
861     }
862 }
863
864 void ISISgloveTabs::saveCalibration() {
865     wxString caption = _("Save calibration to file...");
866     wxString wildcard = _("CAL files (*.cal)|*.cal");
867     wxString defaultDir = dataDir;
868     wxString defaultFilename = _("myCalibration.cal");
869
870     wxFileDialog dialog(this, caption, defaultDir, defaultFilename, wildcard, wxSAVE);
871     if (dialog.ShowModal() == wxID_OK) {
872         wxString path = dialog.GetPath();
873
874         GloveCalibration *gc = (gloveThread->getGlove())->getCalibration();
875         std::string filename = (const char*)path.mb_str();
876         try {
877             gc->saveToFile(filename);
878             txtLoadedCalibration->SetLabel(dialog.GetFilename());
879             wxMessageBox(_("Glove calibration saved"));
880         } catch (CannotSaveException) {
881             wxMessageBox(_("Error in saving calibration! Bad permissions?"));
882         }
883     } else {
884         wxMessageBox(_("Glove calibration NOT saved"));
885     }
886 }
887
888 void ISISgloveTabs::resetCalibration() {
889     (gloveThread->getGlove())->resetCalibration();
890     txtLoadedCalibration->SetLabel(_("<none>"));
891     wxMessageBox(_("Glove calibration reset"));
892 }
893
894 void ISISgloveTabs::refreshKnownGesturesList(std::map<ScaledHandPosition,std::string> knowns) {
895     listBoxKnownGestures->Clear();
896
897     std::map<ScaledHandPosition,std::string>::const_iterator it;
898     int i=0;
899     for (it=knowns.begin(); it!= knowns.end(); ++it, ++i){
900         wxString s((it->second).c_str(),wxConvUTF8);
901         listBoxKnownGestures->Insert(s,i);
902     }
903 }
904
905 void ISISgloveTabs::deleteSelectedGesture() {
```

```
906       unsigned int n = listBoxKnownGestures->GetSelection();
907       if (n==wxNOT_FOUND) return;
908       wxString selectedGestureName = listBoxKnownGestures->GetStringSelection();
909
910       std::string selectedGestureNameS(selectedGestureName.mb_str());
911       GestureRecognition::removeGesture(selectedGestureNameS);
912       listBoxKnownGestures->Delete(n);
913
914       wxRemoveFile( dataDir + wxString::Format(_("%s.hand"),selectedGestureName));
915   }
916
917   void ISISgloveTabs::addCurrentGesture() {
918       if (!GuiThreadsComm::isGloveActive()) {
919           ErrorMessageBox(_("No data to save!\nPlease activate the glove first..."));
920           return;
921       }
922       ScaledHandPosition *tosave = (gloveThread->getGlove())->getScaledHandPosition();
923
924       wxTextEntryDialog dialog(this,
925                                _("Insert the gesture name\n"),
926                                _("Gesture name selection"),
927                                _(""),
928                                wxOK | wxCANCEL);
929       if (dialog.ShowModal() == wxID_CANCEL) return;
930       std::string gestureNameS = (const char*) dialog.GetValue().mb_str();
931
932       if (gestureNameS.length()==0) {
933           ErrorMessageBox(_("No gesture name specified!\nAborting save..."));
934           return;
935       }
936       for (unsigned int i = 0; i < gestureNameS.length(); i++)
937           if (isspace(gestureNameS[i])) {
938               ErrorMessageBox(_("Gesture name not valid, no spacing character allowed!\nAborting save..."));
939               return;
940           }
941
942       if ( (gestureNameS.length()<6) || (gestureNameS.substr(gestureNameS.length()-5) != ".hand") )
943           gestureNameS += ".hand";
944
945       wxString gestureName(gestureNameS.c_str(), wxConvUTF8);
946
947       //avoid writing outside the data directory...
948       wxFileName toCutPath(gestureName);
949       wxFileName gfn( dataDir + toCutPath.GetFullName() ) ;
950
951       if (!gfn.IsOk()) {
952           ErrorMessageBox(_("Invalid characters!\nAborting save..."));
953           return;
954       }
955
956       if (wxFileExists( gfn.GetFullPath() )) {
957           ErrorMessageBox(_("File exists! Gesture name not available!\nAborting save..."));
958           return;
959       }
960
961       try {
```

```
962             tosave->saveToFile((const char*)(gfn.GetFullPath()).mb_str());
963         } catch (CannotSaveException) {
964             ErrorMessageBox(_("Cannot save! Bad filename/permissions?\nAborting save..."));
965             return;
966         }
967
968         std::string strippedName = (const char*)(gfn.GetName()).mb_str();
969         GestureRecognition::addNewGesture( strippedName, tosave );
970         listBoxKnownGestures->Insert( gfn.GetName(), listBoxKnownGestures->GetCount() );
971     }
972
973     void ISISgloveTabs::OnIdle(wxIdleEvent& event) {
974         idleUpdateGrid();
975         //idleUpdatePosMatch(); //too slow!
976     }
977
978     void ISISgloveTabs::setGestureMatchString(std::string &s) {
979         posMatch = s;
980     }
981
982     void ISISgloveTabs::onPositionMatch() {
983         wxString str(posMatch.c_str(), wxConvUTF8);
984         if (!wxThread::IsMain()) wxMutexGuiEnter();
985         if (str == _("")) {
986             if ( txtRecognizedGesture->GetLabel().Cmp(_("<not recognized>"))==0 ) return;
987             txtRecognizedGesture->SetLabel( _("<not recognized>"));
988             btnAddGesture->Enable();
989         } else {
990             if ( txtRecognizedGesture->GetLabel().Cmp(str)==0 ) return;
991             txtRecognizedGesture->SetLabel(str);
992             btnAddGesture->Disable();
993         }
994         if (!wxThread::IsMain()) wxMutexGuiLeave();
995     }
996
997     void ISISgloveTabs::OnThreadEvent(wxCommandEvent& event) {
998         int n = event.GetInt();
999         switch(n) {
1000            case T_EVT_GLOVESTARTED: //glove thread started
1001                onGloveStarted(); onPositionMatch(); break;
1002            case T_EVT_GLOVESTOPPED: //glove thread terminated
1003                onGloveStopped(); break;
1004            case T_EVT_SERVERSTARTED: //server thread started
1005                onServerStarted(); break;
1006            case T_EVT_SERVERSTOPPED: //server thread terminated
1007                onServerStopped(); break;
1008            case T_EVT_POSMATCH: //position match from glove thread
1009                onPositionMatch(); break;
1010        }
1011    }
1012
1013    void ISISgloveTabs::refreshClientList(std::vector<wxSocketBase*> clients) {
1014         if (!wxThread::IsMain()) wxMutexGuiEnter();
1015         listBoxClients->Clear();
1016         for (unsigned int i=0; i<clients.size(); i++)
1017            listBoxClients->Insert(ClientsHandling::getPeerInfo(clients[i]),i);
```

```
1018        if (!wxThread::IsMain()) wxMutexGuiLeave();
1019    }
```

L'accesso alla *taskbar*: la classe TaskBarIcon

```
 1  #ifndef TASKBARICON_H
 2  #define TASKBARICON_H
 3
 4  #include "wx/taskbar.h"
 5  #include "wx/menu.h"
 6
 7  class ISISgloveTabs; //fwd ref
 8
 9  class TaskBarIcon: public wxTaskBarIcon {
10      public:
11      #if defined(__WXCOCOA__)
12          TaskBarIcon(wxTaskBarIconType iconType = DEFAULT_TYPE)
13              :   wxTaskBarIcon(iconType)
14      #else
15          TaskBarIcon()
16      #endif
17          {}
18
19          void setFrame(ISISgloveTabs* x) { dialog = x; };
20          void OnLeftButtonDClick(wxTaskBarIconEvent&);
21          void OnMenuRestore(wxCommandEvent&);
22          void OnMenuExit(wxCommandEvent&);
23          virtual wxMenu *CreatePopupMenu();
24
25      DECLARE_EVENT_TABLE()
26
27      private:
28          ISISgloveTabs *dialog;
29  };
30
31  #endif // TASKBARICON_H
```

```
 1  #include "TaskBarIcon.h"
 2  #include "ISISgloveTabs.h"
 3
 4  enum {
 5      PU_RESTORE = 10001,
 6      PU_EXIT,
 7  };
 8  BEGIN_EVENT_TABLE(TaskBarIcon, wxTaskBarIcon)
 9      EVT_MENU(PU_RESTORE, TaskBarIcon::OnMenuRestore)
10      EVT_MENU(PU_EXIT,    TaskBarIcon::OnMenuExit)
11      EVT_TASKBAR_LEFT_DCLICK(TaskBarIcon::OnLeftButtonDClick)
12  END_EVENT_TABLE()
13
14  void TaskBarIcon::OnMenuRestore(wxCommandEvent& ) {
15      dialog->Show(true);
16  }
17
18  void TaskBarIcon::OnMenuExit(wxCommandEvent& ) {
19      dialog->exit();
```

```
20  }
21
22  void TaskBarIcon::OnLeftButtonDClick(wxTaskBarIconEvent&) {
23      dialog->Show(true);
24      dialog->Raise();
25  }
26
27  // Overridables
28  wxMenu *TaskBarIcon::CreatePopupMenu() {
29      wxMenu *menu = new wxMenu;
30      menu->Append(PU_RESTORE, _T("&Show ISISGloveManager"));
31
32  #ifndef __WXMAC_OSX__ /*Mac has built-in quit menu*/
33      menu->AppendSeparator();
34      menu->Append(PU_EXIT,    _T("E&xit"));
35  #endif
36      return menu;
37  }
```

Comunicazione GUI/threads: la classe GuiThreadsComm

```
1   #ifndef GUITHREADSCOMM_H
2   #define GUITHREADSCOMM_H
3
4   #include <wx/thread.h>
5   #include <wx/socket.h>
6
7   #include "gloveAPI/ScaledHandPosition.h"
8   #include "gloveAPI/GloveCalibration.h"
9   #include "GUI/ISISgloveTabs.h"
10
11  class TGloveHandler;
12  class TServerListener;
13
14  enum ThreadEvent {
15          T_EVT_GLOVESTARTED, //glove thread started
16          T_EVT_GLOVESTOPPED, //glove thread terminated
17          T_EVT_SERVERSTARTED,    //server thread started
18          T_EVT_SERVERSTOPPED,    //server thread terminated
19          T_EVT_POSMATCH  //position match from glove thread
20  };
21
22  class GuiThreadsComm {
23      public:
24
25          //glove/server threads/activity
26          static void setGloveThread(TGloveHandler *tg);
27          static TGloveHandler *getGloveThread();
28
29          static void setServerThread(TServerListener *ts);
30          static TServerListener *getServerThread();
31
32          static void setGloveActive(bool status);
33          static bool isGloveActive();
34
35          static void setServerActive(bool status);
36          static bool isServerActive();
```

```
37
38          // GUI ***************
39          // generic stuff
40          static void setFrm(ISISgloveTabs *f);
41          static void showError(const wxString& text);
42          static wxString getDataDir() { return frm->getDataDir();}
43          //glove tab
44          static void setGridDataSource(ScaledHandPosition *shp, RawHandPosition *rhp, GloveCalibration *gc);
45          //recognition tab
46          static void setGestureMatchString( std::string &s );
47          static void refreshKnownGesturesList(std::map<ScaledHandPosition,std::string>);
48          //server tab
49          static void refreshClientList(std::vector<wxSocketBase*> clients);
50          //log tab
51          static void log (const wxString& text);
52
53          //to send events from threads
54          static void postEventToGui(ThreadEvent e);
55
56      private:
57      //GUI
58          static ISISgloveTabs *frm;
59
60      //glove/server activity
61          static TGloveHandler *tglove;
62          static TServerListener *tserver;
63
64          static bool gloveActive;
65          static bool serverActive;
66  };
67  #endif // GUITHREADSCOMM_H
```

```
1   #include "GuiThreadsComm.h"
2
3   TGloveHandler *GuiThreadsComm::tglove;
4   TServerListener *GuiThreadsComm::tserver;
5
6   bool GuiThreadsComm::gloveActive = false;
7   bool GuiThreadsComm::serverActive = false;
8
9   ISISgloveTabs *GuiThreadsComm::frm = NULL;
10
11  //*****************************************
12
13  void GuiThreadsComm::setServerThread(TServerListener *ts) { tserver = ts; }
14  TServerListener *GuiThreadsComm::getServerThread() { return tserver; }
15
16  void GuiThreadsComm::setGloveThread(TGloveHandler *tg) { tglove = tg; }
17  TGloveHandler *GuiThreadsComm::getGloveThread() { return tglove; }
18
19  void GuiThreadsComm::setGloveActive(bool status) { gloveActive = status; }
20  bool GuiThreadsComm::isGloveActive() { return gloveActive; }
21
22  void GuiThreadsComm::setServerActive(bool status) { serverActive = status; }
23  bool GuiThreadsComm::isServerActive() { return serverActive; }
24
25  //*****************************************
```

```
26
27  void GuiThreadsComm::setFrm(ISISgloveTabs *f) { frm = f;}
28
29  void GuiThreadsComm::showError(const wxString& text) {
30      if (frm) frm->ErrorMessageBox(text);
31  }
32
33  //glove tab
34  void GuiThreadsComm::setGridDataSource(
35    ScaledHandPosition *shp, RawHandPosition *rhp, GloveCalibration *gc) {
36      if (frm) frm->setGridDataSource(shp, rhp, gc);
37  }
38
39  //recognition tab
40  void GuiThreadsComm::setGestureMatchString(std::string &s ){
41      if (frm) frm->setGestureMatchString(s);
42  }
43
44  void GuiThreadsComm::refreshKnownGesturesList(std::map<ScaledHandPosition, std::string> m) {
45      if (frm) frm->refreshKnownGesturesList(m);
46  }
47
48  //server tab
49  void GuiThreadsComm::refreshClientList(std::vector<wxSocketBase*> clients) {
50      if (frm) frm->refreshClientList(clients);
51  }
52
53  //log tab
54  void GuiThreadsComm::log (const wxString& text) {
55      if (frm) frm->log(text);
56  }
57
58  //*****************************************
59
60  void GuiThreadsComm::postEventToGui(ThreadEvent eventId) {
61      wxCommandEvent myevent( wxEVT_COMMAND_MENU_SELECTED, THREAD_EVENT );
62      myevent.SetInt(eventId);
63      wxPostEvent( frm, myevent );
64  }
```

Riconoscimento gesti: la classe GestureRecognition

```
1   #ifndef GESTURERECOGNITION_H
2   #define GESTURERECOGNITION_H
3
4   #include <map>
5
6   #include "gloveAPI/ScaledHandPosition.h"
7
8   class GestureRecognition {
9       public:
10
11          static void loadKnownGestures();
12
13          static void addNewGesture(std::string shpname, ScaledHandPosition *shp);
14          static void removeGesture(std::string name);
15
```

```
16          static std::string getGestureMatch(ScaledHandPosition *p);
17
18      private:
19          static std::map<ScaledHandPosition, std::string> gestureMap;
20          static wxMutex *gestureMapMutex;
21  };
22  #endif

 1  #include <string>
 2  #include <algorithm>
 3
 4  #include <wx/wx.h>
 5  #include <wx/dir.h>
 6
 7  #include "GestureRecognition.h"
 8  #include "GuiThreadsComm.h"
 9
10
11  std::map<ScaledHandPosition, std::string> GestureRecognition::gestureMap;
12  wxMutex *GestureRecognition::gestureMapMutex = new wxMutex();
13
14
15  void GestureRecognition::loadKnownGestures() {
16      ScaledHandPosition *shp;
17
18      wxString datadir =  GuiThreadsComm::getDataDir();
19      if (!wxDir::Exists(datadir))  //first run?
20          wxMkdir( datadir, 777 );
21
22      wxDir dir(datadir);
23      if (!dir.IsOpened()) {
24          wxString warning = wxString::Format(_("! warning: cannot open data directory!\nBad permissions?") );
25          GuiThreadsComm::log(warning);
26          return;
27      }
28
29      wxString filename;
30      wxString filespec = _("*.hand");
31      int flags = wxDIR_FILES;
32      bool cont = dir.GetFirst(&filename, filespec, flags);
33
34      float savedCmpThr = ScaledHandPosition::getCompareThreshold();
35      ScaledHandPosition::setCompareThreshold(0.0);
36      while (cont) {
37          wxString tolog = wxString::Format(_("- loading gesture from %s\n"), filename);
38          GuiThreadsComm::log(tolog);
39
40          wxString loadPath = datadir + filename;
41          std::string handfile = (const char*) filename.mb_str();
42          shp = new ScaledHandPosition(18);
43          try {
44              shp->loadFromFile((const char*)loadPath.mb_str());
45              std::string shpname = handfile.substr(0,handfile.length()-5);
46              addNewGesture(shpname, shp);
47          } catch (CannotLoadException) {
48              GuiThreadsComm::showError(  wxString::Format(
49                  _("Error in loading gesture from %s!\nBad permissions/corrupted file?"), loadPath));
```

```
50          }
51          cont = dir.GetNext(&filename);
52      }
53      ScaledHandPosition::setCompareThreshold(savedCmpThr);
54      GuiThreadsComm::refreshKnownGesturesList(gestureMap);
55  }
56
57
58  void GestureRecognition::addNewGesture(std::string shpname, ScaledHandPosition *shp) {
59      gestureMapMutex->Lock();
60      int mapsize = gestureMap.size();
61      gestureMap[*shp] = shpname;
62      gestureMapMutex->Unlock();
63      if (mapsize == gestureMap.size()) {
64          wxString warning = wxString::Format(
65              _("! warning: position (%s) overwrites a previously known position (gesture conflict)\n"),
66              wxString(shpname.c_str(),wxConvUTF8) );
67          GuiThreadsComm::log(warning);
68      }
69  }
70
71  void GestureRecognition::removeGesture(std::string name) {
72      gestureMapMutex->Lock();
73      std::map<ScaledHandPosition, std::string>::iterator gestureMapIt;
74      for (gestureMapIt = gestureMap.begin(); gestureMapIt != gestureMap.end(); ++gestureMapIt) {
75          if (gestureMapIt->second == name) {
76              gestureMap.erase(gestureMapIt);
77              break;
78          }
79      }
80      gestureMapMutex->Unlock();
81  }
82
83  std::string GestureRecognition::getGestureMatch(ScaledHandPosition *p) {
84      std::string res("");
85      gestureMapMutex->Lock();
86      std::map<ScaledHandPosition, std::string>::iterator gestureMapIt;
87      gestureMapIt = gestureMap.find(*p);
88      if ( (gestureMapIt != gestureMap.end()) && (gestureMapIt->first == *p) )
89          res = gestureMapIt->second;
90      gestureMapMutex->Unlock();
91      return res;
92  }
```

Gestione dei clients: la classe ClientsHandling

```
1   #ifndef CLIENTSHANDLING_H
2   #define CLIENTSHANDLING_H
3
4   #include <vector>
5   #include <wx/socket.h>
6
7   class ClientsHandling {
8       public:
9           static void addClient(wxSocketBase *s);
10          static void killAllClients();
11          static wxString getPeerInfo(wxSocketBase *s);
```

```
12          static std::vector<wxSocketBase*> *getClients() { return &clients; };
13          static wxMutex* getClientsMutex() { return clientsVectorMutex; };
14
15      private:
16          static std::vector<wxSocketBase*> clients;
17          static wxMutex *clientsVectorMutex;
18  };
19
20  #endif // CLIENTSHANDLING_H
```

```
1   #include "ClientsHandling.h"
2   #include "GuiThreadsComm.h"
3   #include <wx/wx.h>
4
5   wxMutex *ClientsHandling::clientsVectorMutex = new wxMutex();
6   std::vector<wxSocketBase*> ClientsHandling::clients;
7
8   void ClientsHandling::addClient(wxSocketBase *s) {
9       clientsVectorMutex->Lock();
10
11      clients.push_back(s);
12
13      GuiThreadsComm::log(wxString::Format(_("- new client connection from %s\n"),getPeerInfo(s)));
14      GuiThreadsComm::refreshClientList(clients);
15
16      clientsVectorMutex->Unlock();
17  }
18
19  void ClientsHandling::killAllClients() {
20      clientsVectorMutex->Lock();
21
22      std::vector<wxSocketBase*>::iterator i;
23
24      for (i = clients.begin(); i != clients.end(); ++i) {
25          wxSocketBase *sock = *i;
26          sock->Destroy();
27          GuiThreadsComm::log(wxString::Format(_("- killing connection with %s\n"),getPeerInfo(sock)));
28      }
29
30      clients.clear();
31      GuiThreadsComm::refreshClientList(clients);
32
33      clientsVectorMutex->Unlock();
34  }
35
36  wxString ClientsHandling::getPeerInfo(wxSocketBase *sock) {
37      wxIPV4address addr;
38      sock->GetPeer(addr);
39      return (wxString::Format(_("%s:%d"), (addr.IPAddress()).c_str(), addr.Service()));
40  }
```

Il thread di gestione del guanto: la classe TGloveHandler

```
1   #ifndef TGLOVEHANDLER_H
2   #define TGLOVEHANDLER_H
3
```

```
4   #include <sstream>
5
6   #include <wx/thread.h>
7
8   #include <wx/datstrm.h>
9   #include <wx/mstream.h>
10  #include <wx/sckstrm.h>
11
12  #include "GuiThreadsComm.h"
13  #include "gloveAPI/Glove.h"
14  #include "gloveAPI/ScaledHandPosition.h"
15
16  class TGloveHandler : public wxThread {
17      public:
18          //glove maximum sampling: 52 readings per second
19          TGloveHandler(std::string glovePort = "COM1", int msec = 20)  {
20              //GuiThreadsComm::log(_("*constructor: TGloveHandler\n"));
21              if (glovePort == "<auto>")
22                  g = new Glove();
23              else
24                  g = new Glove(glovePort);
25              msecdelay = msec;
26              threadActive = true;
27
28              wxString s;
29              if (glovePort == "<auto>")
30                  s = _("*Glove opened on unspecified port\n");
31              else
32                  s =wxString::Format(_("*Glove opened on port %s\n"),wxString(glovePort.c_str(),wxConvUTF8));
33              GuiThreadsComm::log(s);
34              GuiThreadsComm::postEventToGui(T_EVT_GLOVESTARTED);
35              GuiThreadsComm::setGloveThread(this);
36          };
37
38          ~TGloveHandler() {
39              //GuiThreadsComm::log(_("*distructor: TGloveHandler\n"));
40              GuiThreadsComm::log(_("*Glove closed\n"));
41
42              GuiThreadsComm::setGloveThread(NULL);
43              delete g;
44              GuiThreadsComm::postEventToGui(T_EVT_GLOVESTOPPED);
45          }
46
47          Glove* getGlove() { return g;}
48
49          void terminate();
50
51      private:
52          Glove *g;
53
54          int msecdelay;
55          bool threadActive;
56
57          virtual void *Entry();
58          void broadcast(std::string&);
59  };
```

```
60
61  #endif // TGLOVEHANDLER_H
62

 1  #include "TGloveHandler.h"
 2  #include "GuiThreadsComm.h"
 3  #include "GestureRecognition.h"
 4  #include "ClientsHandling.h"
 5
 6  #include <vector>
 7  #include <algorithm>
 8
 9  void *TGloveHandler::Entry() {
10      //GuiThreadsComm::log(_("**TGloveHandler->Entry...\n"));
11
12      ScaledHandPosition *sh = g->getScaledHandPosition();
13      RawHandPosition *rh = g->getRawHandPosition();
14      GloveCalibration *gc = g->getCalibration();
15
16      GuiThreadsComm::setGridDataSource(sh,rh,gc);
17      GuiThreadsComm::setGloveActive(true);
18
19      std::string toClients;
20      std::string prevPosMatch = "";
21
22      while (threadActive) {
23          g->updateScaledHandPosition(sh);
24          g->updateRawHandPosition(rh);
25          g->updateCalibration(gc);
26
27          std::string posMatch = GestureRecognition::getGestureMatch(sh);
28
29          if (posMatch != prevPosMatch) {
30              GuiThreadsComm::setGestureMatchString(posMatch);
31              prevPosMatch = posMatch;
32              GuiThreadsComm::postEventToGui(T_EVT_POSMATCH);
33          }
34
35          std::ostringstream fromGlove;  // output string stream
36          fromGlove << "SCL:" << *sh << std::endl;
37          fromGlove << "RAW:" << *rh << std::endl;
38          if (posMatch!="")
39              fromGlove << "POS:" << posMatch << std::endl;
40
41          toClients = fromGlove.str();
42          broadcast(toClients);
43
44          Sleep(msecdelay);
45      }
46      GuiThreadsComm::setGloveActive(false);
47      return NULL;
48  }
49
50  void TGloveHandler::terminate() {
51      //GuiThreadsComm::log(_("**TGloveHandler->terminate...\n"));
52      threadActive = false;
53  }
```

```
54
55  inline bool is_NULL(wxSocketBase* s) {
56          return (s==NULL);
57  };
58  void TGloveHandler::broadcast(std::string& forClients) {
59      ClientsHandling::getClientsMutex()->Lock();
60      std::vector<wxSocketBase*> *v = ClientsHandling::getClients();
61      std::vector<wxSocketBase*>::iterator i;
62
63      bool anyClientDisconnect = false;
64      const char *buf = forClients.c_str();
65
66      for (i = v->begin(); i != v->end(); ++i) {
67          wxSocketBase *sock = *i;
68          wxSocketOutputStream SocketOutputStream(*sock);
69          wxDataOutputStream out(SocketOutputStream);
70
71          out.Write8((wxUint8*)buf,strlen(buf));
72          if (!out.IsOk()) {
73              GuiThreadsComm::log(wxString::Format(
74                  _("- cannot send data to %s, assuming client disconnect\n"),
75                  ClientsHandling::getPeerInfo(sock)));
76              *i = NULL; //mark for delete (deleting here would BREAK the iterator!)
77              anyClientDisconnect = true;
78          }
79      }
80      if (anyClientDisconnect) {
81          v->erase(remove_if(v->begin(), v->end(), &is_NULL), v->end() );
82          GuiThreadsComm::refreshClientList(*v);
83      }
84      ClientsHandling::getClientsMutex()->Unlock();
85  }
```

Il thread di gestione del server: la classe TServerListener

```
1   #ifndef TSERVERLISTENER_H
2   #define TSERVERLISTENER_H
3
4   #include <wx/wx.h>
5   #include <wx/thread.h>
6   #include <wx/socket.h>
7
8   #include "GuiThreadsComm.h"
9   #include "ClientsHandling.h"
10
11  class CannotBindException {};
12
13  class TServerListener : public wxThread {
14      private:
15          unsigned short portno;
16          wxSocketServer *listeningSocket;
17
18          bool threadActive;
19
20          void init(wxIPV4address addr);
21
22      public:
```

```
23
24          TServerListener(wxIPV4address addr)  {
25              init(addr);
26          };
27
28          TServerListener(unsigned short p)  {
29              wxIPV4address listeningAddr;
30              listeningAddr.AnyAddress();
31              listeningAddr.Service(p);
32              init(listeningAddr);
33          };
34
35          ~TServerListener() {
36              // GuiThreadsComm::log(_("*distructor: TServerListener\n"));
37
38              GuiThreadsComm::log(_("*TCP server stopped\n"));
39              ClientsHandling::killAllClients();
40              GuiThreadsComm::setServerThread(NULL);
41              GuiThreadsComm::postEventToGui(T_EVT_SERVERSTOPPED);
42          };
43
44          virtual void *Entry();
45          void terminate();
46  };
47
48  #endif
```

```
 1  #include "TServerListener.h"
 2  #include "GuiThreadsComm.h"
 3
 4  void TServerListener::init(wxIPV4address addr) {
 5      portno = addr.Service();
 6
 7      //GuiThreadsComm::log(_("*constructor: TServerListener\n"));
 8      listeningSocket = new wxSocketServer(addr);
 9
10      if (!listeningSocket->IsOk()) {
11          GuiThreadsComm::log(_("! Socket error! Cannot bind!\n"));
12          throw  CannotBindException();
13      }
14
15      wxString s = wxString::Format(_("*Socket listening on port %d\n"),portno);
16      GuiThreadsComm::log(s);
17
18      threadActive = true;
19
20      GuiThreadsComm::postEventToGui(T_EVT_SERVERSTARTED);
21      GuiThreadsComm::setServerThread(this);
22  }
23
24  void *TServerListener::Entry() {
25      GuiThreadsComm::setServerActive(true);
26      //GuiThreadsComm::log(_("**TServerListener->Entry...\n"));
27      wxSocketBase *sock;
28      while (threadActive) {
29          //wxSocketBase *sock = listeningSocket->Accept();
30          sock = listeningSocket->Accept();
```

```
31
32          // if Accept failed or interrupted by termination
33          if ((sock == NULL) || (!sock->IsOk())) continue;
34
35          // non blocking writes to clients
36          // this prevents the broadcast being stopped by "bad" clients
37          sock->SetFlags(wxSOCKET_NOWAIT);
38
39          // store socket in the "connected clients" array
40          ClientsHandling::addClient(sock);
41      }
42      //GuiThreadsComm::setServerActive(false);
43      return NULL;
44  }
45
46  void TServerListener::terminate() {
47      threadActive = false;
48      GuiThreadsComm::setServerActive(false);
49      //GuiThreadsComm::log(_("**TServerListener->terminate...\n"));
50      listeningSocket->Destroy();
51  }
```

A.1.3 ISISgloveInput

Ricezione dati da ISISgloveManager: lo script...

```
1   -- CallChannel function is called when channel is called
2   function CallChannel()
3
4
5   -- useless/not implemented
6
7   -- "_14_THUMB PALM"
8   -- "_15_WRIST BEND"
9   -- "_16_PITCH"
10  -- "_17_ROLL"
11
12
13
14  local InputChannel = channel.GetChild(0)
15  local PositionMatchChannel = channel.GetChild(1)
16
17
18  -- get socket data
19  local socketString = InputChannel:GetText()
20
21
22  -- parse socket data
23
24  local scaledValuesString
25  local rawValuesString
26  local posMatchString
27
28  local sclStart = string.find (socketString, "SCL:",1,true)
29  local rawStart = string.find (socketString, "RAW:",1,true)
30  local posStart = string.find (socketString, "POS:",1,true)
```

```

if (sclStart == nil) then return -1 end

if (posStart == nil) then
    posMatchString = ""
else
    local posEnd =  string.find(socketString,"\n",posStart+4,true)
    posMatchString = string.sub(socketString,posStart+4,posEnd-1)
end

PositionMatchChannel:SetText(posMatchString)

scaledValuesString = string.sub(socketString,sclStart,rawStart-1)
sclStart = sclStart + 5;  -- salta "SCL:("
local nextSep = string.find(scaledValuesString,"|",sclStart,true)

sens = 0
while (sens < 14) do
    str = string.sub(scaledValuesString,sclStart, nextSep-1)
    channel.GetChild(sens+2):SetValue(tonumber(str))

    sclStart = nextSep+1
    nextSep = string.find(scaledValuesString,"|",sclStart,true)

    sens = sens + 1
end


end

-- GetValue function is called when this channels is used as Value
function GetValue()
    return -1
end

```

ChannelCaller da posMatch: lo script...

```
-- CallChannel function is called when channel is called
function CallChannel()

    local InputChannel = channel.GetChild(0)
    local posMatch = InputChannel:GetText()

    if posMatch == nil then
        return
    end


    posMatch_callChannel_map = {
        ["manoAperta"] = 1,
        ["pugno"]      = 2,
    }

    channelToCall = posMatch_callChannel_map[posMatch]

    if channelToCall ~= nil then
```

```
20          channel.GetChild(channelToCall):CallChannel()
21      end
22
23  end
24
25  -- GetValue function is called when this channels is used as Value
26  function GetValue()
27      return -1
28  end
29
```

A.1.4 ISISpcTracker

Nota: per compilare il channel vanno aggiunti i files types.h, isense.h e isense.c presenti nel codice di esempio dell'SDK InterSense

Implementazione del channel

```
1   #pragma once
2
3   #include "isense.h"
4
5   #ifdef MYCHANNEL_EXPORTS
6   #define MYCHANNEL_API __declspec(dllexport)
7   #else
8   #define MYCHANNEL_API __declspec(dllimport)
9   #endif
10
11  #define MYCHANNEL_NAME      "IntersensePCtracker"
12  #define MYCHANNEL_VERSION       1
13
14  // {D6F8E85C-517F-47c4-9F15-7CF2194F5E4C}
15  static const GUID MYCHANNEL_GUID =
16  { 0xd6f8e85c, 0x517f, 0x47c4, { 0x9f, 0x15, 0x7c, 0xf2, 0x19, 0x4f, 0x5e, 0x4c } };
17
18
19  #define CHILDS_N 19
20
21  /* the channels works with the MiniTrax Hand/Head station connected as first
22     station and the MiniTrax Wand (the one with buttons and joystick) connected as second.
23     If for any reason you can't connect the stations this way, swap the defined values */
24
25  #define TRACKER_HAND 0
26  #define TRACKER_WAND 1
27
28
29  class MYCHANNEL_API IntersensePCtracker: public A3d_Channel
30  {
31  public:
32      IntersensePCtracker();
33      virtual ~IntersensePCtracker();
34
35      // Add needed dependencies for publishing
36      virtual void            DoDependencyInit(A3d_List* currentDependList);
37
38      //CallChannel function invoked by the engine
```

```
39      virtual void            CallChannel();
40
41  protected:
42
43      ISD_TRACKER_HANDLE       handle;
44      ISD_TRACKER_DATA_TYPE    data;
45  };
46
47  // Leave rest of header file intact!
48  #define MYCHANNELDLL_EXPORTS extern "C" { \
49  __declspec(dllexport) DllInterface * __cdecl InitDLL() \
50  { \
51      return new IntersensePCtracker; \
52  } \
53  }
54
```

```
 1
 2  #include "stdafx.h"
 3  #include "IntersensePCtracker.h"
 4
 5  #include "Aco_Float.h"
 6  #include "A3d_EngineInterface.h"
 7
 8
 9  ChannelType channelType;
10
11  extern "C" __declspec( dllexport ) ChannelType* __cdecl GetType() {
12      ZeroMemory(channelType.name, 80);
13      StringCbCopy(channelType.name,  79, MYCHANNEL_NAME);
14      channelType.version         = MYCHANNEL_VERSION;
15      channelType.guid            = MYCHANNEL_GUID;
16      channelType.baseguid        = MYCHANNEL_GUID;
17      channelType.minimumEdition  = EDITION_LEVEL_ALL;        // All Quest3D Editions
18
19      return &channelType;
20  }
21
22  MYCHANNELDLL_EXPORTS
23
24
25  IntersensePCtracker::IntersensePCtracker() {
26      SetChannelName(MYCHANNEL_NAME);
27
28      handle = ISD_OpenTracker( NULL, 0, FALSE, FALSE );
29      if(handle <= 0) {
30          ueMsg("Cannot open Intersense PCTracker");
31          for (int c=0; c<3; c++) {
32              data.Station[TRACKER_HAND].Position[c] = 0;
33              data.Station[TRACKER_HAND].Orientation[c] = 0;
34              data.Station[TRACKER_WAND].Position[c] = 0;
35              data.Station[TRACKER_WAND].Orientation[c] = 0;
36          }
37
38          data.Station[TRACKER_WAND].AnalogData[0] = 127;
39          data.Station[TRACKER_WAND].AnalogData[1] = 127;
40
```

```
41          for (int c=0; c<6; c++)
42              data.Station[TRACKER_WAND].ButtonState[c] = -1;
43      }
44
45      SetChildCreationCount(0);
46
47      ChildCreation child;
48      child.channelType.guid = FLOAT_CHANNEL_GUID;
49      strcpy(child.channelType.name, FLOAT_CHANNEL_NAME);
50      child.initialize = true; // crea automaticamente i channel figli
51
52      char *channelnames[] = {
53                  "Hand Pos X", "Hand Pos Y", "Hand Pos Z",
54                  "Hand Rot X", "Hand Rot Y", "Hand Rot Z",
55                  "Wand Pos X", "Wand Pos Y", "Wand Pos Z",
56                  "Wand Rot X", "Wand Rot Y", "Wand Rot Z",
57                  "Joystick X", "Joystick Y",
58                  "button 1", "button 2", "button 3", "button 4", "button 5"
59      };
60
61      for (int cnum =  0; cnum < CHILDS_N; cnum++) {
62          child.requestLink = cnum;
63          strcpy(child.name,channelnames[cnum]);
64          SetChildCreateType(child, cnum);
65      }
66  }
67
68  IntersensePCtracker::~IntersensePCtracker() {
69      if (handle>0) ISD_CloseTracker( handle );
70  }
71
72  void IntersensePCtracker::DoDependencyInit(A3d_List* currentDependList) {
73      AddChannelGuidDepend(MYCHANNEL_GUID, currentDependList);
74      AddDLLDepend("isense.dll", currentDependList); //segnalata la dipendenza dalla dll InterSense
75  }
76
77  //channel logic
78  void IntersensePCtracker::CallChannel() {
79      Aco_FloatChannel* floatChannel=NULL;
80
81      if (handle > 0 )
82          ISD_GetData( handle, &data );
83
84      for (int cnum =  0; cnum < CHILDS_N; cnum++) {
85          floatChannel = (Aco_FloatChannel*)GetChild(cnum);
86          if (floatChannel != NULL){
87              switch(cnum) {
88                  case 0: case 1: case 2:
89                      floatChannel->SetFloat( data.Station[TRACKER_HAND].Position[cnum] ); break;
90                  case 3: case 4: case 5:
91                      floatChannel->SetFloat( data.Station[TRACKER_HAND].Orientation[cnum-3]/180*3.14 ); break;
92
93                  case 6: case 7: case 8:
94                      floatChannel->SetFloat( data.Station[TRACKER_WAND].Position[cnum-6] ); break;
95                  case 9: case 10: case 11:
96                      floatChannel->SetFloat( data.Station[TRACKER_WAND].Orientation[cnum-9]/180*3.14 ); break;
```

```
97
98                  case 12: floatChannel->SetFloat( data.Station[TRACKER_WAND].AnalogData[0] ); break;
99                  case 13: floatChannel->SetFloat( data.Station[TRACKER_WAND].AnalogData[1] ); break;
100
101                 case 14: floatChannel->SetFloat( data.Station[TRACKER_WAND].ButtonState[1]); break;
102                 case 15: floatChannel->SetFloat( data.Station[TRACKER_WAND].ButtonState[0]); break;
103                 case 16: floatChannel->SetFloat( data.Station[TRACKER_WAND].ButtonState[2]); break;
104                 case 17: floatChannel->SetFloat( data.Station[TRACKER_WAND].ButtonState[3]); break;
105                 case 18: floatChannel->SetFloat( data.Station[TRACKER_WAND].ButtonState[5]); break;
106             }
107         }
108     }
109 }
110
111
112
```

```
1  // stdafx.h : include file for standard system include files,
2  //  or project specific include files that are used frequently, but
3  //      are changed infrequently
4  //
5
6  #if !defined(AFX_STDAFX_H__80495E63_2DA6_11D4_A351_0050DAD61B65__INCLUDED_)
7  #define AFX_STDAFX_H__80495E63_2DA6_11D4_A351_0050DAD61B65__INCLUDED_
8
9  #if _MSC_VER > 1000
10 #pragma once
11 #endif // _MSC_VER > 1000
12
13
14 // Insert your headers here
15 #define WIN32_LEAN_AND_MEAN     // Exclude rarely-used stuff from Windows headers
16
17 #include <windows.h>
18 #include "A3d_List.h"
19 #include "A3d_Channels.h"
20 // TODO: reference additional headers your program requires here
21
22 //{{AFX_INSERT_LOCATION}}
23 // Microsoft Visual C++ will insert additional declarations immediately before the previous line.
24
25 #endif // !defined(AFX_STDAFX_H__80495E63_2DA6_11D4_A351_0050DAD61B65__INCLUDED_)
```

```
1  // stdafx.cpp : source file that includes just the standard includes
2  //  Float.pch will be the pre-compiled header
3  //  stdafx.obj will contain the pre-compiled type information
4
5  #include "stdafx.h"
6
7  // TODO: reference any additional headers you need in STDAFX.H
8  // and not in this file
```