ALMA MATER STUDIORUM

UNIVERSITA' DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
Sede di Forlì

Corso di Laurea in
INGEGNERIA AEROSPAZIALE
Classe L-9

ELABORATO FINALE DI LAUREA

in

MECCANICA RAZIONALE

PROGETTAZIONE DI UN'APPLICAZIONE DI TIPO "GESTURE-BASED" PER IL CONTROLLO DI VELIVOLI SENZA PILOTA A BORDO

CANDIDATO
NORMAN GABRIELE ANTONIO SANTINI

RELATORE
Prof. LEONARDO SECCIA

CORRELATORI
Ing. SARA BAGASSI
Ing. FRANCESCA LUCCHI

Anno Accademico 2013/2014
Sessione II

Indice

Capitolo 1

Introduzione

In ambito Aeronautico, un importante tema di ricerca riguarda lo studio di sistemi innovativi di interazione uomo - macchina, mediante la computer grafica e dispositivi di visualizzazione tridimensionale: infatti, già da anni controllori di volo e piloti vengono addestrati all'interno di specifici simulatori.
Anche altri settori, in cui l'innovazione tecnologica riveste un ruolo chiave, trovano nello sviluppo di sistemi innovativi di interazione un elemento strategico al fine di ottenere risultati sempre più performanti: ad esempio, astronauti, piloti di formula 1 e persino meccanici rientrano tra quelle figure che traggono un grande beneficio dall'utilizzo di sistemi integrati di "addestramento virtuale".
La capacità di rendere la simulazione quanto più possibile vicino alla realtà, la facilità e la naturalezza di interazione con macchine e simulatori sono stati sin dall'inizio l'obiettivo principale di ingegneri, informatici e ricercatori.
Questi sistemi devono rispettare delle specifiche ben precise al fine di poter essere certificati ed utilizzati.
In primo luogo devono rispondere ai requisiti dettati dalla particolare esigenza per la quale sono progettati, poi devono essere impiegati con naturalezza dall'utente, riducendo al minimo le possibilità di errore umano ed infine devono essere stimolanti, cioè creare curiosità, interesse e desiderio di essere usati.
Attualmente, l'evoluzione tecnologica sta portando all'abbandono di strumenti classici come la tastiera o il mouse; si pensi, ad esempio, ai computer con *touch screen*, al *Wiimote* ed a tutti quei dispositivi in grado di catturare i movimenti del corpo e trasferirli ad un computer tramite l'impiego di opportuni sensori,

telecamere e software specifici. Questi sistemi sono anche chiamati di *motion capture*, ma, pur essendo concepiti per favorire l'utente finale, non di rado accade che quest'ultimo si trovi a dover imparare l'utilizzo di sistemi complicati e non ergonomici. In figura 1.0.1 è mostrato un esempio di sistema di *motion capture* per la registrazione di movimenti delle mani.

Fig. 1.0.1
I guanti in lycra utilizzati dal MIT di Boston per il loro sistema di hand tracking. [foto tratta da www.tomshw.it].

In questo contesto alcune delle principali sfide per il futuro consisteranno nel consentire ad un potenziale utente di governare una macchina mediante comando vocale oppure con il movimento della mano o persino con la vista, senza che ci sia un contatto diretto con la macchina stessa e senza l'utilizzo di un *controller* esterno.

Figura 1.0.2
Sulla destra il controller Wii remote e sulla sinistra il Play Station Move. [foto tratta da www.games4all.it].

Sicuramente nell'ultimo periodo uno dei dispositivi che più si stanno imponendo per la notevole capacità di interagire in tempo reale con l'utilizatore è il *Kinect* prodotto da *Microsoft*.

Fig. 1.0.3
Microsoft Kinect for XBOX 360
[foto tratta da www.amazon.co.uk].

Il dispositivo si configura come un sistema di *motion capture*, consentendo di interagire con le applicazioni senza mouse, tastiere e telecomandi e con finalità che potrebbero riguardare svariati settori: ad esempio, la realizzazione di sistemi integrati con applicazioni per la robotica, in cui può essere impiegato per far muovere in modo naturale particolari componenti, per l'intrattenimento, per il campo della medicina, per far volare agevolmente un elicottero o per far muovere con semplicità un piccolo veicolo, evitando ostacoli mediante la creazione di una mappa 3D dell'ambiente.

1.1 Scopo del lavoro

Questa tesi ha avuto come obiettivo lo studio e sviluppo di un'interfaccia che permetta all'utente di controllare un simulatore di UAV mediante tecnica "gesturale", i cui movimenti sono rilevati attraverso il sistema *Microsoft Kinect* for *XBOX 360*.
Il lavoro in un primo momento ha riguardato una fase di progettazione e ideazione dell'interfaccia e del tipo di controllo "gesturale", pianificando i movimenti da associare ai comandi di volo.
Successivamente è iniziata la fase di programmazione, in cui come ambiente di lavoro è stato utilizzato il *Visual Studio 2010* e come linguaggio macchina il C++.
In questa fase si è studiato inoltre il sensore *Kinect* e si è visto come le variazioni delle coordinate dei punti del corpo, rilevate ed acquisite dal dispositivo durante i movimenti dell'utente, possano essere utilizzate per assegnare specifici comandi.

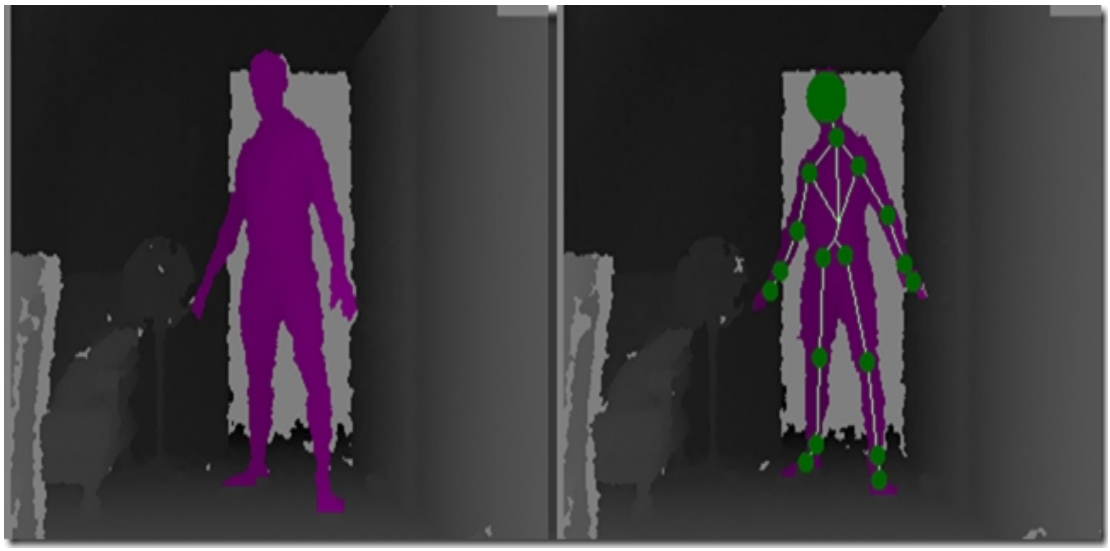

Fig.1.1.1
"Scheletrizzazione" dell'utente
[foto tratta da netscale.cse.nd.edu]

Ottenuti questi risultati, le informazioni lette dal sensore sono state trasferite al programma *Simulink*, linguaggio con cui il

simulatore di UAV qui impiegato è stato sviluppato, così da poter gestire i dati in ingresso in maniera semplice e intuitiva.
Infine, il risultato conclusivo di questo lavoro può essere apprezzato tramite un opportuno sistema di visualizzazione.

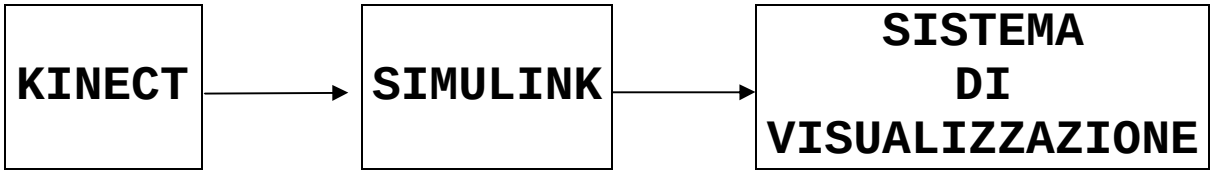

Fig. 1.1.2
Schema a blocchi del principio di funzionamento del sistema descritto.

Capitolo 2

Strumenti per l'interazione "gesture based"

In questo capitolo si intende fornire una introduzione alla tecnologia Kinect, valutandone il funzionamento da un punto di vista hardware.

2.1 Breve introduzione del progetto Kinect

Il Kinect è un dispositivo di input per la console di gioco Xbox 360.
Il progetto Kinect è stato sviluppato per la Microsoft dalla PrimeSense, azienda israeliana da tempo impiegata nella ricerca e sviluppo di dispositivi di controllo senza sensori da impugnare. Il software, invece, è stato sviluppato internamente nei Microsoft Game Studios e, più precisamente, dai programmatori della Rare.
PrimeSense è riuscita nell'intento di creare una macchina che registrasse i movimenti del corpo umano senza l'utilizzo di sensori da applicare al soggetto.
Kinect, infatti, grazie ad un sensore di movimento di nuova generazione, è in grado di far interagire l'utente con la console o col computer senza dover utilizzare nessun tipo di controller.

L'utente stesso diventa il controller, il movimento di braccia, gambe e testa sono gli input del sistema il quale li registra e li trasferisce al monitor in tempo reale, permettendo di visualizzare l'utente in un ambiente 3D (il limite massimo di utilizzatori simultanei assegnato da Microsoft alla periferica è pari a quattro, in piedi o seduti).

Inizialmente il Kinect è nato con la funzione di rilevare ed acquisire i movimenti e gesta del corpo per interfacciarsi con giochi per console, ma successivamente sono state sviluppate molte altre Applicazioni, che vanno oltre il semplice divertimento di un gioco per console.
Microsoft stessa, la quale inizialmente aveva dichiarato di essere contraria all'utilizzo del Kinect all'infuori dell'ambiente di gioco, intuì che il prodotto aveva grandi potenzialità per obiettivi differenti, sviluppandone una versione capace di interfacciarsi con il computer senza il bisogno di installare driver specifici.

Per tutti quei dispositivi che non nascevano per una diretta connessione si sono sviluppati nel tempo dei driver che comunque permettessero di utilizzare Kinect con il computer.
Questi driver consentono di accedere alle funzioni audio, video e sensori di profondità del Kinect e si basano su un API (*Application Programming Interface*) completa, la OpenNi (*Open Natural Interactions*), la quale contiene inoltre un "analizzatore di scena", che rileva le figure in primo piano e le separa dallo sfondo.

2.2 Analisi dell'hardware e principio di funzionamento

Il dispositivo dopo un disassemblaggio si presenta nel modo mostrato in Figura 2.2.1.

Fig. 2.2.1
Kinect disassemblato
[foto tratta da 360xkinect.wordpress.com]

L'apertura del dispositivo mostra a prima vista la presenza di:

- un array di microfoni orientati verso il basso: 3 sul lato destro e uno sul lato sinistro. Tale scelta è ritiene giustificata dal fatto che l'orientamento verso il basso è considerato ottimale per la registrazione del suono;

- tre apparati ottici utilizzati per il riconoscimento visuale del corpo in movimento;

- una ventola per la dissipazione del calore;

- 64MB di memoria flash;

- un accelerometro a tre assi;
- Prime Sense PS1080-A2, il chip che rappresenta il cuore della tecnologia di Kinect.

Più in particolare il sistema di apparati ottici si compone di una telecamera RGB e un sensore di profondità a raggi infrarossi.
Tale sensore è composto da un proiettore a infrarossi e da una telecamera sensibile alla stessa banda, che viene utilizzata per leggere quanto rilevato dai raggi infrarossi.

La telecamera RGB ha una risoluzione di 640 × 480 pixel, mentre quella a infrarossi usa una matrice di 320 × 240 pixel.
L'array di microfoni (che lavorano in parallelo per l'acquisizione del suono) è usato dal sistema per la calibrazione dell'ambiente in cui ci si trova, mediante l'analisi della riflessione del suono sulle pareti e sull'arredamento, così lavorando il sistema è in grado di riconoscere i comandi vocali distinguendoli dal rumore di fondo o da altri disturbi.
Pertanto per far si che i comandi vocali funzionino al meglio è necessario calibrare l'array di microfoni.
Infine la barra del Kinect è motorizzata e quindi in grado di muoversi intorno all'asse orizzontale seguendo i movimenti degli utenti, orientandosi nella posizione ottimale per il riconoscimento dei movimenti.

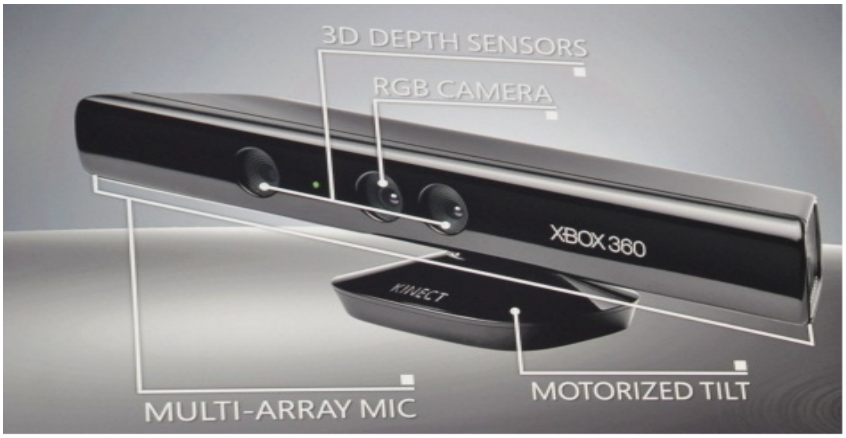

Fig. 2.2.2
[foto tratta da www.mrpt.org].

I risultati ottenuti da Kinect sono le mappe di profondità e le immagini a colori.
Dato che le immagini vengono elaborate sul Kinect e non sul PC, Microsoft ha inserito nella periferica due schede ed una barra di supporto metallico in parallelo, separati da quattro distanziatori metallici.
In figura 2.2.3 è possibile notare gli "occhi" della periferica: due telecamere (simili ad una webcam dotata di autofocus) ed un proiettore IR, inoltre tra le telecamere e l'IR è situato anche un piccolo LED di stato.

Fig. 2.2.3
I sensori che permettono l'acquisizione delle immagini e della profondità.
[foto tratta da www.mondogamesblog.com]

Il dispositivo presenta un sensore in grado di processare il flusso video ad un frame rate di 30 Hz, lo stream RGB usa una risoluzione VGA ad 8 bit di profondità con dimensione di 640x480 pixels con un filtro di Bayer per il colore, mentre il sensore monocromatico di profondità genera un flusso dati a risoluzione VGA (640 x 480 pixels) con 11 bit di profondità, quindi con 2048 valori possibili.
Il range di utilizzo del sensore va da un minimo di 0.4 metri ad un massimo di 4.0 metri.
Il campo di vista invece varia tra 57° orizzontalmente e 43° verticalmente, mentre l'intero dispositivo è in grado di ruotare fino a 27° lungo l'asse verticale.

La scheda madre ha 6 chip. Osservando la scheda A della figura 2.2.4, da sinistra a destra sono rispettivamente montati:

- Stereo ADC con microfono preamplificato (Wolfson Microelectronics WM8737G);
- N-Channel PowerTrench MOSFET (Fairchild Semiconductor FDS8984);
- Controller USB 2.0 hub (NEC uPD720114);
- Pacchetto SAP 6 mm x 4,9 mm non identificato forse SPI flash (H1026567 XBOX1001 X851716-005 Gepp);
- SoC per il controller dell’interfaccia della macchina fotografica (Marvell AP102);
- SDRAM DDR2 512 megabit (Hynix H5PS5162FF).

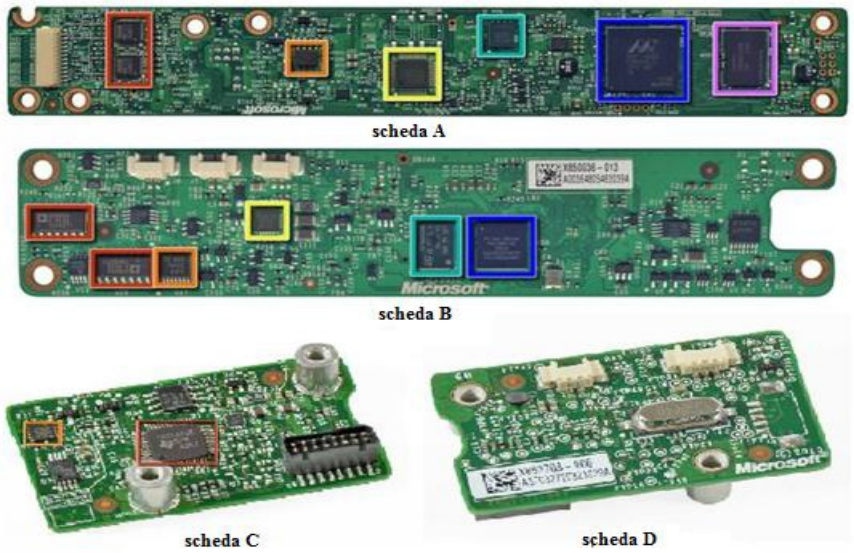

Figura 2.2.4
Componenti interni del sensore Kinect
[foto tratta da www.twch-nology.it]

Nella scheda B sono montati:

- Due CMOS Rail-to-Rail amplificatore d'uscita a basso costo (Analog Devices AD8694);
- Un campionatore e convertitore A/D 8 bit ad 8 canali, con interfaccia 12C (TI ADS7830I);
- Allegro Microsystems A3906;
- Una memoria Flash 1Mb x 8 oppure 512Kb x 16 (ST Microelectronics M29W800DB);
- Un processore d'immagini Soc Sensor (PrimeSense PS1080-A2).

Infine le schede C e D mostrano una scheda che dispone di un controller audio USB frontale e centrale (TI TAS1020B) mentre sul lato sinistro della scheda si può osservare un accelerometro (Kionix MEMS KXSD9), che probabilmente è utilizzato come stabilizzatore d'immagine.

Si analizza ora il **principio di funzionamento** del Microsoft Kinect for XBOX 360.

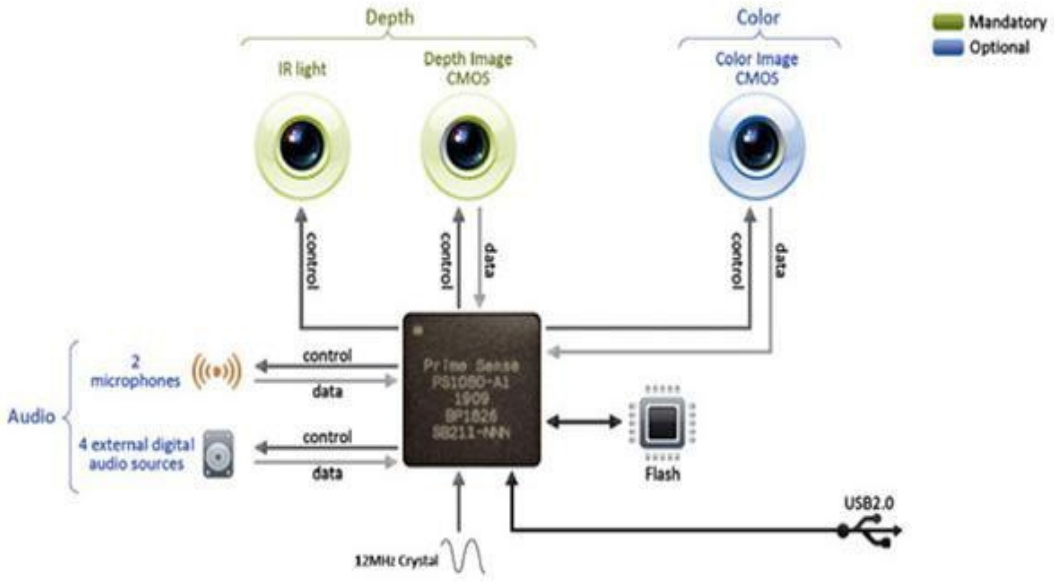

Figura 2.2.5
Schema generale di funzionamento del sensore Kinect
[foto tratta da www.webnews.it]

Il funzionamento di Kinect si può riassumere in due parti:

1. una prima parte in cui viene costruita una mappa di profondità (*Depth Image*) tramite l'analisi di luce strutturata creata dell'emettitore infrarossi;

2. una seconda parte in cui viene calcolata la posizione dell'utente ed assegnata "un'identità", utilizzando degli algoritmi di *tracking* implementati nel software.

2.3 La *Depth Image*

Le immagini di profondità semplificano molti problemi di computer-vision e di interazione come l'implementazione di interfacce basate su gesti, il *tracking* di oggetti e persone, la

ricostruzione 3D degli ambienti, la rimozione del background e la segmentazione della scena e il riconoscimento della posa del corpo.
La *Depth Image* (o mappa di profondità) è un'immagine in cui ogni pixel codifica la distanza (nella scena 3D) del punto di coordinate x,y dal sensore.
Esistono principalmente tre modi per calcolare suddetta distanza: il primo modo è dato dalla triangolazione stereo la quale consente di calcolare la profondità di un oggetto confrontando le immagini acquisite da due telecamere.
Il secondo modo è dato dalla così detta tecnica *time off light* che, al contrario della triangolazione stereo, utilizza una sola telecamera la quale calcola la distorsione che subisce un segnale luminoso proiettato sugli oggetti.
Infine il terzo modo, quello utilizzato da Kinect, è dato dalla proiezione di pattern luminosi, utilizzando un sistema di visione stereo costituito da un proiettore e da una telecamera.
Vengono proiettati nell'ambiente esaminato un'immensità di spot luminosi (infrarossi) la cui distribuzione a prima vista sembra casuale (si ottiene questo effetto grazie a una mascherina posta davanti al led).
Il pattern emesso è visibile in assenza di luci, puntando il Kinect verso una superficie piana e inquadrando tale superficie con una telecamera digitale.
La disposizione dei punti del pattern è nota a priori al software del Kinect, che al suo interno ha memorizzato come il pattern dovrebbe essere visto dal sensore IR se fosse proiettato su una superficie posta ad una distanza ben definita e perfettamente parallela al piano della Depth Camera.

Quest'ultimo quindi altro non è che un pattern di riferimento.

Per ognuno dei punti proiettati si può calcolare la distanza dal sensore triangolando la sua posizione attuale (espressa in pixel) con quella che avrebbe assunto nel pattern di riferimento.

Il risultato dell’elaborazione del pattern proiettato è una immagine di profondità grezza (Raw Depth Image), nel senso che i valori di profondità non sono espressi in una unità di misura precisa.

E' dunque necessaria una procedura di calibrazione che metta ad esempio in corrispondenza i valori grezzi con valori espressi in scala metrica.
Per poter riconoscere quale particolare punto si stia analizzando viene studiata (procedura di correlazione) la disposizione dei 64 punti vicini che per ogni punto del pattern è diversa e ben definita.
Le dimensione dei punti e la loro visibilità all'interno del pattern servono appunto a facilitare tale operazione di riconoscimento.
Per ogni punto è poi possibile ricavare la posizione 3D reale utilizzando la matrice di calibrazione della telecamera e associare il colore reale dell’oggetto colpito utilizzando l’immagine ottenuta mediante la telecamera RGB.
Tale procedura richiede un processo di calibrazione molto preciso per il quale viene utilizzata la Microsoft Calibration Card.

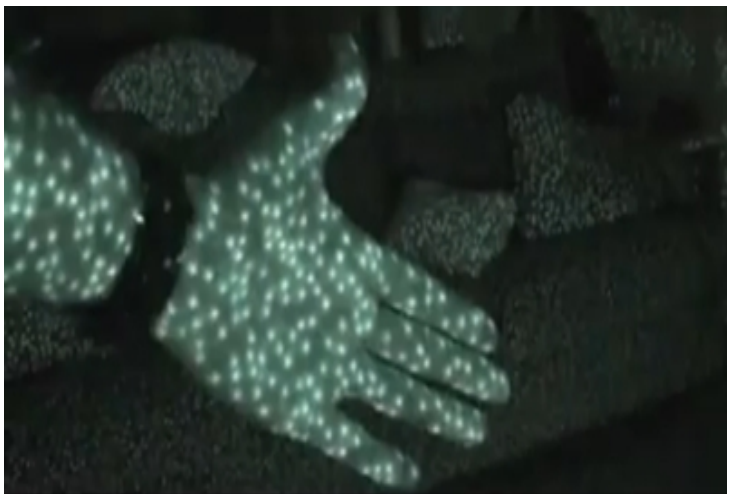

Figura 2.3.1
Esempio di proiezione di pattern 2D
[foto tratta da www.ingame.it]

Il risultato di questo tipo di lavoro è una deformazione della distanza tra i vari spot in funzione dell'angolazione delle superfici (per rendere intuitivo il meccanismo, si può pensare di stendere un lenzuolo a pois su un divano, sarebbe possibile farsi un'idea della forma dello stesso in base a come si dispongono i pallini), in quanto la fotocamera inquadra la proiezione bidimensionale della distribuzione tridimensionale degli spot.
Grazie alla densità degli spot luminosi si può intuire l'angolazione di una superficie inquadrata.

Questa tecnologia è stata brevettata nel 2005 da Zalevsky, Shpunt, Maizels e Garcia, sviluppata e integrata in un chip da PrimeSense.
Ogni passaggio di dati e informazioni avviene passando per l'unità di controllo costituita dal chip PS1080 che rappresenta un po' il cuore del sistema, infatti esso ha al suo interno molte funzionalità di processing per il tracciamento, per la ricostruzione della scena e per il riconoscimento di gesta.
Con questo sistema sarà sufficiente pertanto acquisire una singola immagine e quindi utilizzare un singolo algoritmo di matching per determinare la profondità degli oggetti (dato che l'altra immagine è costituita dal noto pattern originale).
Nota la struttura del pattern proiettato, l'unità di controllo è in grado di calcolare lo scostamento fra i punti proiettati e quelli ripresi dalla telecamera, ottenendo così la Depth Image.
E' evidente che l'orientamento, la forma e le dimensioni dei punti proiettati non sono costanti, ma dipendono dalla posizione del sensore ovvero dalla distanza tra esso e i punti stessi.
Nel brevetto di PrimeSense vengono espresse delle specifiche distanze in cui è possibile ottenere risoluzioni più o meno accurate:
regioni di distanze comprese tra gli 0.5m e gli 1.2m si ottiene la massima risoluzione, tra gli 1.2m e i 2m una risoluzione ancora accettabile mentre tra i 2m e i 4m la risoluzione è alquanto scarsa.

Secondo fonti Microsoft, il sistema riesce a determinare la posizione di un oggetto posto a 2 m dal Kinect, con un errore di solo 1 cm.
Inoltre a seconda della distanza tra sensore e punti, le figure compariranno sullo schermo con diverse colorazioni di grigio, viene assegnato infatti il grigio scuro per gli oggetti più lontani e il grigio chiaro per quelli più vicini.
Mentre alle figure animate (soggette a movimenti) riconosciute, vengono assegnati colori più sgargianti come il rosso, il verde, il blu e così via.

Per avere un'idea di una mappa di profondità si può pensare di osservare degli oggetti e scattare una foto.
Colorando poi i pixel dell’immagine con varie tonalità di grigio in base alla distanza di ogni pixel dalla telecamera.
Quello che si ottiene potrebbe essere un esempio di mappa di profondità, come mostrato il figura 2.3.2.

Figura 2.3.2
Esempio di mappa di profondità
[foto tratta da en.9jcg.com]

Figura 2.3.3
Il fascio di infrarossi proiettato dal Kinect
[foto tratta da appuntiit.forumfree.it]

La differenzazione tra figure animate e non, avviene attraverso dei filtri e delle linee guida compilate, ad esempio il sistema si aspetterà già che una figura animata (figura umana) avrà probabilmente una certa altezza, due gambe, due braccia e una testa.
Non tutte le persone hanno però la stessa conformazione fisica, inoltre spesso vengono utilizzati indumenti larghi o cappelli.
Per questo Microsoft ha inserito degli algoritmi che riconoscono ad esempio possibili cappelli o maglioni larghi.

2.4 Il Tracking

Il Tracking (o tracciamento) rappresenta la fase successiva alla costruzione della mappa di profondità.
Al fine di ottenere un risultato corretto e apprezzabile dal Tracking bisogna verificare che l'utente sia dentro il campo di vista del sensore e che non sia oscurato da altri utenti o oggetti.
Bisogna ricordare di rimanere ad una distanza compresa tra 1 e 3.5 metri senza mai uscire dal campo visivo del sensore onde evitare la perdita del controllo di Tracking.

Esistono due tipi di Tracking:
l'Hand Tracking (tracciamento della mano) e l'User Tracking o Skeleton Tracking (tracciamento dell'utente o dello scheletro).

Nell'Hand Tracking per iniziare ad assumere il controllo delle gesta bisogna prima compiere alcuni movimenti specifici chiamati "focus gesture" atti ad sincronizzare e performare la traccia dell'utente.

Le focus gesture si suddividono in:

- "click" (click);
- "wave" (ondulazione).

Affinché il sistema riconosca il gesto click, l'utente dopo essersi assicurato di stare di fronte al sensore (senza muovere altre parti del corpo) dovrà alzare la propria mano, tenere il palmo aperto, avanzarla verso il sensore quasi a simulare una spinta e ritirarla immediatamente verso di se, compiendo un movimento di almeno 20 cm.
Con il wave invece, l'utente dovrà alzare la mano e muoverla diverse volte (almeno 5) da sinistra verso destra.

Importante è ricordare di non compiere movimenti ne troppo veloci ne troppo lenti, se dovessero persistere ulteriori complicanze in questa fase bisognerebbe posizionarsi a 2 metri dal sensore (che rappresenta la distanza ottimale) e assicurarsi che la mano sia all'interno del campo visivo del Kinect.
Quando il sistema avrà riconosciuto queste due gesta da parte dell'utente, allora si sarà ottenuto il controllo e per "aiutare" il sistema a mantenerlo è buona norma mantenere la mano registrata ben separata dalle altre parti del corpo, renderla cioè ben visibile al sensore.

Per verificare che ogni operazione sia andata a buon fine bisogna constatare che nel computer appaia lo stato "gesture started" e una volta completata la registrazione, lo stato "gesture completed".

Oltre alle focus gesture ne esistono altre utili al sistema per aumentare il feedback con l'utente, esse sono:

- Swipe left;
- Swipe right;
- Raise hand candidate;
- Hand candidate moved.

Il sistema d'ora in avanti chiamerà "controlling hand" la mano utilizzata per le operazioni di click e wave, fornendo in output le sue coordinate nello spazio in millimetri.

Scopo dell'User Tracking invece è quello di identificare e tracciare l'utente nella scena, ad ogni utente infatti verrà assegnato una identificazione (ID) fornendo quindi in output una sorta di "etichetta identificativa".

L'algoritmo infine userà questa etichetta per generare lo scheletro.
Analogamente all'Hand Tracking prima di iniziare il tracciamento, al fine di ottenere un buon risultato, bisogna avere alcune accortezze come quella di non avere sedie, tavoli o altri oggetti accanto, che non ci siano altri utenti in movimento o quella di non muovere il sensore durante l'operazione.
Importante è ricordare che se l'utente per un qualsiasi motivo (come ad esempio l'uscita dal campo visivo del Kinect) perderà la visibilità col sensore per più di 10 secondi, il sistema automaticamente eliminerà l'ID creato e sarà pertanto necessario ripetere da capo l'operazione di Tracking.

Inoltre fondamentale è mantenere la parte superiore del corpo sempre dentro il campo di vista del sensore, ricordando che per lo Skeleton Tracking la distanza ottimale dal Kinect è di 2.5 metri.
Si inizia così la fase di calibrazione usata per regolare il modello "scheletrizzato" con le proporzioni del corpo dell'utente.
L'azione di calibrazione richiede di mantenere una specifica posizione detta "posizione a cactus" per circa 3 secondi in cui testa e braccia devono essere completamente visibili e l'utente non dovrà piegare le ginocchia o ruotare il dorso (figura 2.4.1).
L'output dello Skeleton Tracking sarà la posizione e l'orientamento delle giunture dello scheletro stesso.

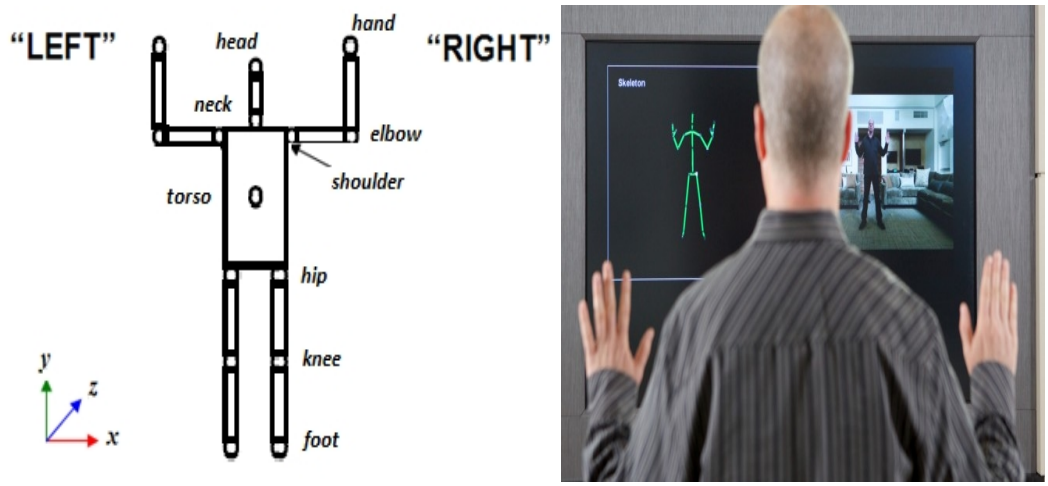

Fig 2.4.1
Posizione a "cactus"
[foto tratta da blog.msdn.com]

Nella figura 2.4.1 è possibile notare che il sistema è orientato nello spazio con tre assi ortonormali X,Y,Z che vengono assunti positivi rispettivamente da sinistra verso destra, dai piedi alla testa e dalle spalle verso il sensore.
Quindi per un utente posto frontalmente al sensore come in figura 2.4.1, verrà assunta come sinistra la sinistra dell'utente e come destra la destra dell'utente.

Questa terna di assi ortonormali rappresenta una rotazione tra le coordinate locali delle giunzioni e le coordinate terresti.
Per questo viene riconosciuta una posizione neutrale detta posizione a "T" in cui le orientazioni delle giunture sono allineate con le coordinate terrestri (figura 2.4.2).

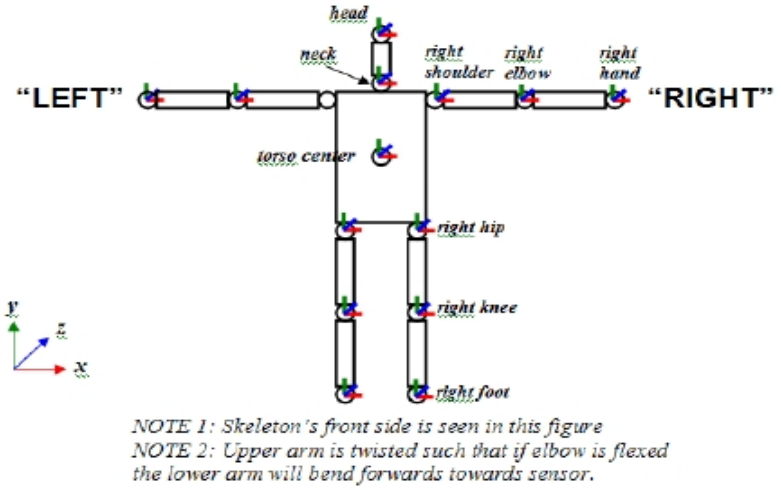

Fig 2.4.2
Posizione neutra a "T"
[foto tratta da blog.3dense.org]

Buona norma è ricordare che durante la fase di Skeleton Tracking bisogna evitare movimenti troppo veloci, tenere le gambe abbastanza divaricate così da permettere una giusta calibrazione ed evitare di indossare felpe con cappuccio e maglie larghe.

Il software lavora a 30 fps (frame per second) e riconosce ben 200 pose comuni che può assumere lo scheletro.
Nel caso in cui l'utente e quindi lo scheletro assegnato assuma una posizione non riconosciuta dal sistema, Kinect assegnerà ad esso una tra le pose riconosciute che più si adatta al caso, così da non perdere il tracciamento dell'utente.
Il tracking del Kinect si basa su circa 500.000 dati registrati, ottenuti dall'osservazione di altrettanti comportamenti umani. L'approccio utilizzato, si rifà alle moderne tecniche dell'*object*

recognition, tecniche che hanno dimostrato soddisfacenti livelli di efficienza computazionale e robustezza.
Queste tecniche si basano sul principio di suddivisione degli oggetti in parti (vedi figura 2.4.3).

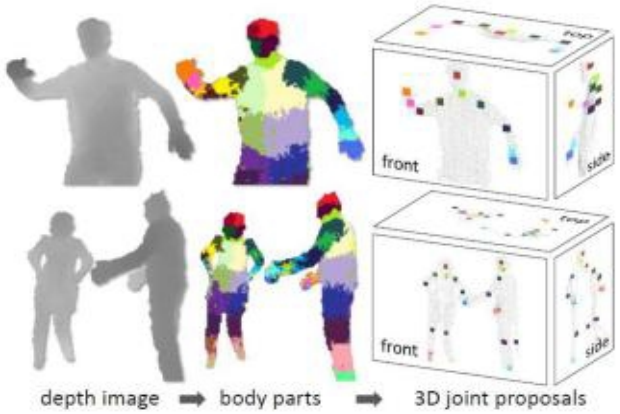

Figura 2.4.3
Evoluzione della fase di tracking
[foto tratta da www.gamasutra.com]

Con il passaggio da OpenNi 1.5 ad OpenNi 2.0 tutte le funzionalità descritte relative alla segmentazione, allo scheletro e alle pose sono racchiuse in una singola API, la User TrackerFrameRef.

Capitolo 3

"Gesture" e comandi di volo

In questo capitolo si intende spiegare nel dettaglio l'idea sviluppata per comandare un UAV col movimento della mano dell'utente.

3.1 Introduzione ai comandi di volo

Dalle ben comuni nozioni fisiche e matematiche della meccanica del volo è noto che per comandare un qualsiasi velivolo è necessario avere il controllo dello stesso lungo i suoi 3 assi principali: asse verticale, orizzontale e longitudinale.
Il controllo lungo questi 3 assi permette di effettuare manovre come l'imbardata, il rollio e il beccheggio(Figura 3.1.1).

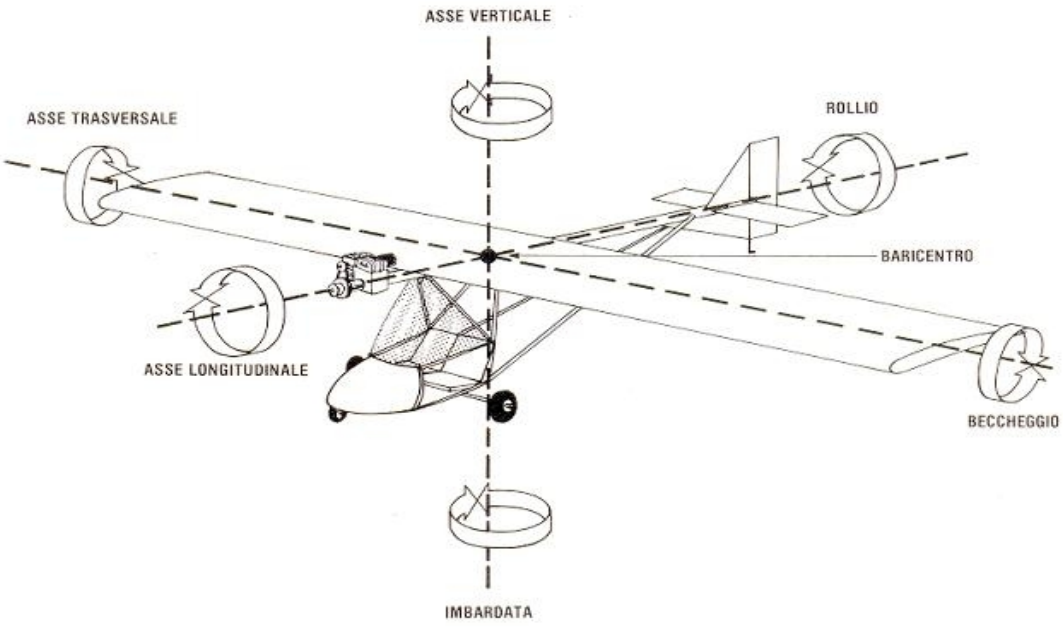

Fig. 3.1.1
Assi e comandi di volo
[foto tratta da www.alireggiane.com]

Bisogna poi aggiungere un quarto comando, la manetta, atto ad aumentare o diminuire il regime del motore.

3.2 Relazione movimento - comando

Utilizzando come base di lavoro gli esempi messi a disposizione dalle librerie di PrimeSense (analizzate nel successivo capitolo 4) e in particolare l'HandTracking, è stato analizzato come il Kinect fornisce in output le coordinate della mano dell'utente.
Utilizzando come ambiente di lavoro Visual Studio 2010 si potrà vedere (a seguito delle già citate procedure preliminari) una schermata in cui vengono mostrate tre colonne di numeri che rappresentano rispettivamente la coordinata X,Y,Z.
Le figure 3.2.1 e 3.2.2 mostrano proprio questo passaggio.

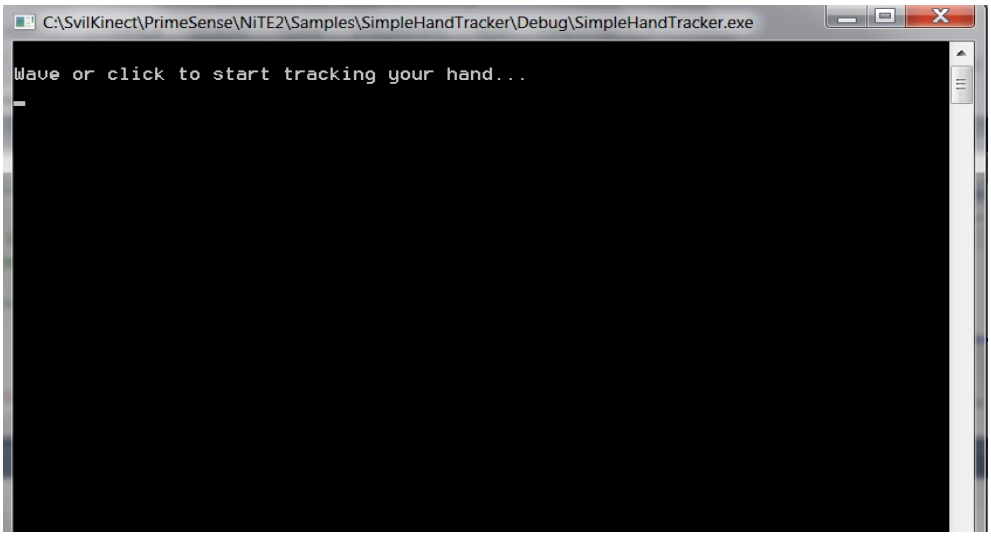

Fig. 3.2.1
Sistema in attesa che l'utente compia una delle due "Focus Gestures"

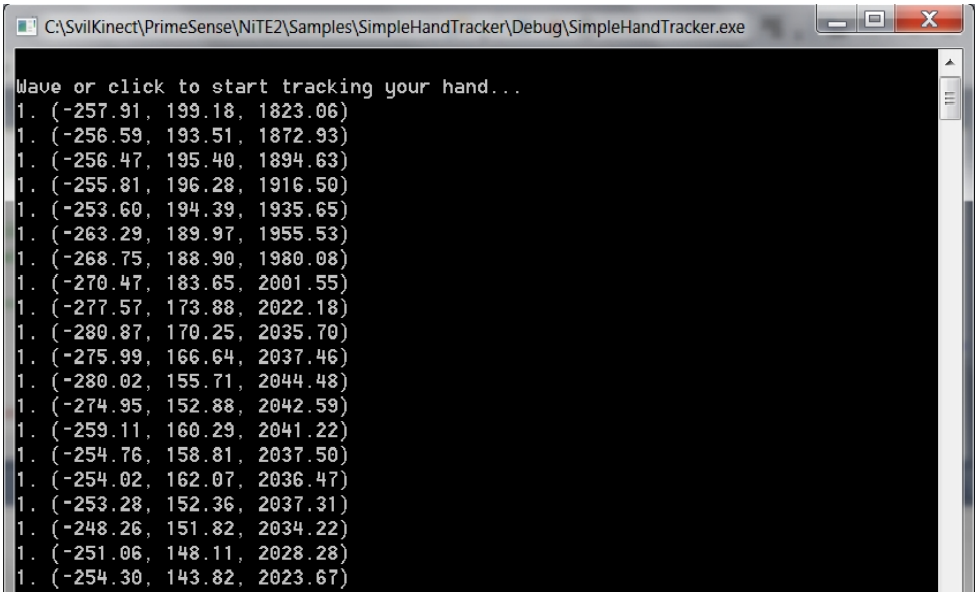

Fig. 3.2.2

Coordinate in output divise per colonne

Dalla figura 3.2.2 è possibile notare come Kinect restituisca all'utente le coordinate della sua mano con ben 5 cifre significative e con unità il millimetro.
Nella fattispecie questo esempio mostra l'output a seguito del movimento di "click", che porta al rilevamento di una serie di coordinate puntuali che descrivono il movimento compiuto.
Questo rappresenta un po' la chiave del sistema.
Infatti si è associato a certe variazioni Δ di coordinate un determinato comando.
In particolare al movimento della mano lungo l'asse Y del sensore verrà attribuito il beccheggio, al movimento lungo l'asse Z del sensore la manetta e infine un movimento lungo l'asse X del sensore la virata.
Per rendere più chiaro quanto esposto si faccia riferimento alle figure 3.2.3, 3.2.4 e 3.2.5.

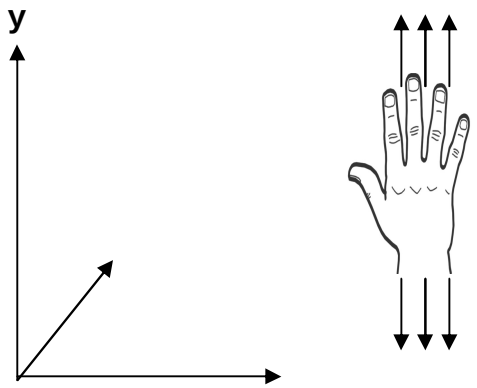

Fig. 3.2.3
Movimento di beccheggio (asse Y)

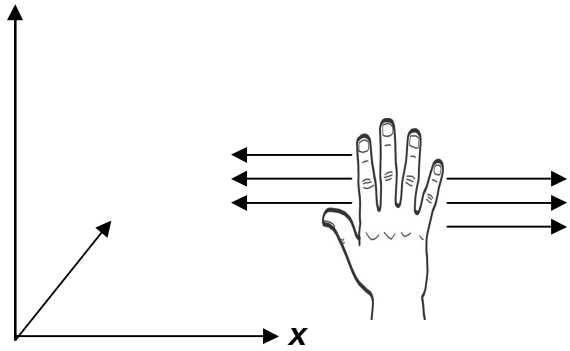

Fig 3.2.4
Movimento di virata (asse X)

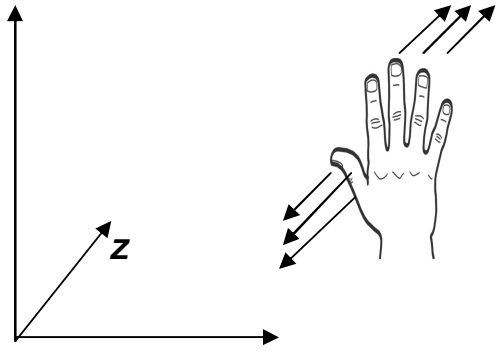

Fig. 3.2.5
Movimento di manetta (asse Z)

Si analizza di seguito i 3 comandi scritti:

- **Il beccheggio:**

Il comando di beccheggio si divide in altri due comandi:
la cabrata e la picchiata.
La cabrata avviene quando la mano si sposta lungo l'asse Y muovendosi dal basso verso l'alto, mentre la picchiata avviene quando la mano si sposta lungo lo stesso asse ma dall'alto verso il basso.
Inoltre si è deciso di assegnare al sistema tre diversi range di lavoro, in cui avere tre diversi comportamenti.
Infatti per spostamenti compresi tra i 10 e i 20 cm si ha come risposta una cabrata o una picchiata di 10 gradi, per spostamenti compresi tra i 20 e i 30 cm di 20 gradi e per spostamenti compresi tra i 30 e i 40 cm di 30 gradi.
In questo modo sarà semplice e intuitivo compiere ed annullare un comando di beccheggio, rendendo così l'UAV il più stabile possibile.
Infatti se ad esempio l'utente alzasse la propria mano di 25 cm l'UAV compierà una cabrata di 20 gradi, quando l'utente deciderà di stabilizzare l'UAV e quindi per annullare il comando appena chiamato basterà abbassare la mano di circa 25 cm, ovvero compiere una picchiata.

- **La virata:**

La virata si divide in due comandi: virata a destra e virata a sinistra.
Analogamente al beccheggio anche la virata è stata divisa in 3 range di lavoro.
Infatti se l'utente muove la mano lungo l'asse X da sinistra verso destra per uno spostamento compreso tra i 10 e i 20 cm si ha come risposta una virata a destra di 10 gradi, per spostamenti compresi tra i 20 e i 30 cm di 20 gradi e per spostamenti compresi tra i 30 e i 40 cm di 30 gradi.

La virata a sinistra ha esattamente gli stessi range ma questa volta l'utente deve spostare la mano da destra verso sinistra.

- **La manetta:**

Infine la manetta, come è stato detto, è data dallo spostamento della mano lungo l'asse Z del sensore.
Anch'essa è stata divisa i 3 range di lavoro: per uno spostamento in avanti compreso tra i 10 e i 20 cm, si ha che la manetta si trova al 40% del massimo raggiungibile; per uno spostamento compreso tra i 20 e i 30 cm si ottiene il 60% del massimo raggiungibile e per uno spostamento compreso tra i 30 e i 40 cm si ottiene il fondo corsa.
Analogamente uno spostamento all'indietro della mano si ottengono risultati opposti.

Capitolo 4

Sviluppo di un'applicazione per il controllo di velivoli senza pilota a bordo

Per poter sfruttare al meglio le caratteristiche del sensore si è fatto uno studio preventivo delle funzionalità offerte dalle librerie OpenNI e Nite. Le librerie mettono a disposizione molteplici funzioni sia per l'Hand Tracking che per lo Skeleton Tracking.
In questo capitolo verrà prima spiegato come impostare la comunicazione con il sensore e come ricavare i dati della mappa di profondità e poi verranno esposte tutte le funzionalità offerte sia dalla libreria OpenNI che dalla libreria NITE insieme agli esempi forniti.

4.1 OpenNi

OpenNI ovvero *Open Natural Interaction* rappresenta un'architettura di supporto, sotto licenza GNU GPL, indipendente dalle piattaforme di lavoro e multi-linguaggio che definisce le API per scrivere applicazioni che usano le Natural Interaction.
Obiettivo di OpenNI è quello di fornire uno standard API che consenta di comunicare con sensori visivi ed audio, per percepire figure, acquisire suoni e sviluppare funzionalità per analizzare ed elaborare dati video ed audio acquisiti in una scena.
La principale caratteristica del middleware è quella di scrivere applicazioni che elaborano dati senza doversi preoccupare del sensore che li ha prodotti.

Il livello più alto rappresenta il software che fa uso di OpenNI implementando le Natural Interaction.

Il livello centrale (middleware) rappresenta l'interfaccia OpenNI atta a comunicare con i sensori e le funzionalità disponibili.

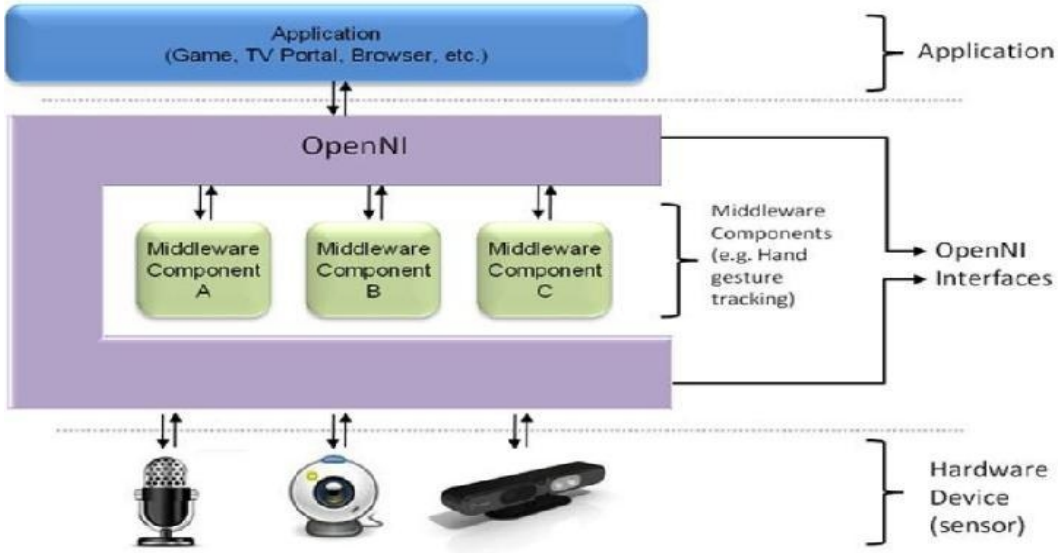

Fig. 4.1.1
Astrazione dei livelli di funzionamento di OpenNI
[foto tratta da www.vannickloriot.com]

Il livello più basso è il livello hardware rappresentato dai vari sensori di comunicazione visiva e audio.

Questi componenti sono indicati come moduli ed attualmente quelli supportati dalle API sono:

- sensore 3D;
- fotocamera RGB;
- dispositivo audio (un microfono o un array di microfoni).

Mentre i componenti del livello centrale sono:

- Analisi totale del corpo: è un componente che elabora i dati sensoriali e fornisce informazioni riguardanti l'utente visto come un sistema composto da articolazioni, il loro orientamento, il centro di massa del corpo e molto altro;

- Analisi della mano: è un componente software che elabora i dati sensoriali, fornendo una ricostruzione della mano e assegnando ad essa, in corrispondenza del palmo, un punto di noto;

- Rilevamento gesto: è un componente software che riconosce gesti predefiniti (ad esempio: Push, Wave, Circle) e li associa a determinati risultati;

- Analisi della scena: è un componente software che analizza l'immagine della scena così da permettere al sistema di riconoscere più utenti presenti nella scena, determinare le coordinate del piano e separare gli oggetti.

Un'altra importante funzionalità messa a disposizione da OpenNi è quella di riconoscere determinate pose assunte dal corpo dell'utente, permettendo così una calibrazione e una successiva creazione di uno scheletro basato sul suo corpo.
Tale scheletro tiene traccia dei movimenti dei vari arti del corpo in uno spazio in tre dimensioni.
Altre funzionalità permettono di individuare il movimento delle mani, di riconoscerle e di tenerne traccia per mezzo dell'assegnazione di un punto.
Tale punto, come è stato detto, identifica il palmo della mano dell'utente e verrà utilizzato per interfacciarsi con il sensore ed i programmi.

4.2 Nite

NITE a differenza di OpenNI lavora ad un livello superiore, infatti esso rappresenta una libreria, implementata sulle interfacce OpenNI, che fornisce funzionalità aggiuntive e facilita la creazione di applicazioni di controllo basato sul movimento delle mani dell'utente e sullo scheletro.
Attualmente NITE è disponibile per i sistemi Windows, Linux e Mac. Esso contiene implementazioni per tutti i moduli della libreria OpenNI.

Il funzionamento di NITE è basato sui livelli illustrati in figura 4.2.1.

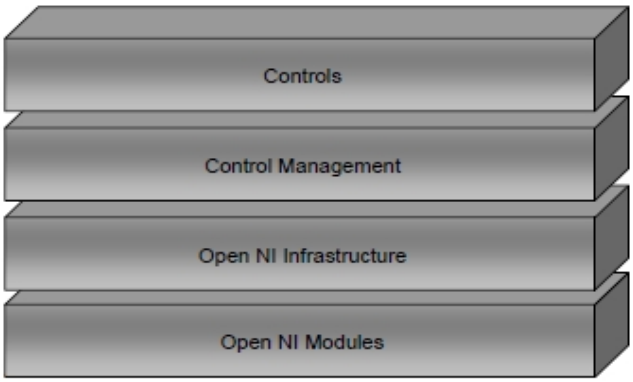

Figura 4.2.1
Schema dei livelli NITE
[[foto tratta da Prime Sensor NITE 1.3 Controls Programmer's Guide]

Si distinguono:

- OpenNi Modules: i moduli di OpenNI supportati da NITE sono Gesture Generator, Hands Generator, User Generator e Skeleton Tracking;

- OpenNi Infrastructure;

- Control Management: riceve un insieme di punti e li indirizza verso il livello di controllo;

- Controls: ogni controllo riceve un insieme di punti e li traduce in un'azione specifica di tale controllo. Successivamente viene richiamata una funzione (callback) dell'applicazione che può modificare il modulo di controllo attivo nel livello Control Management.
  Questo definisce il flusso dell'applicazione.

Un esempio di controllo è il controllo del punto, che permette di registrare quando un punto viene creato, quando scompare e quando si muove.
In realtà i controlli sono degli oggetti sempre in ascolto che attendono l'arrivo di nuovi dati ad ogni fotogramma.
Le funzioni di callback saranno invocate solo quando un determinato evento si verifica.

Di seguito verrà fatta una panoramica sul funzionamento della libreria e sugli esempi messi a disposizione.

Per prima cosa bisogna definire le sessioni: una sessione è uno stato in cui l'utente ha il controllo del sistema tramite il tracciamento della mano. Fin quando è nella sessione i punti della mano vengono tracciati e sono identificati da un ID.

Le sessioni possono essere in tre stati diversi:

- **not in session:** in questo stato il sistema è in attesa di iniziare una nuova sessione tramite un focus gesture. Quando questo gesto viene compiuto e riconosciuto la sessione passa allo stato in session;

- **in session:** è già stato riconosciuto il focus gesture ed è in corso il tracking della mano.

- **quick refocus:** la sessione passa in questo stato quando durante lo stato in session non vengono identificati più i punti della mano.

Quindi per evitare di terminare subito la sessione, il sistema si mette in attesa di un nuovo gesto definito come il *quick refocus gesture* (questo stato è opzionale).

In partenza il sistema si trova nello stato *not in session*, in attesa di un gesto.
Quando viene riconosciuto il focus gesture lo stato cambia in "*in session*" e inizia il tracking della mano.
Se poi il sistema non riconosce altre mani da tracciare si torna allo stato *not in session* se il quick refocus gesture è disabilitato oppure nello stato *quick refocus* se è abilitato.
In questo stato il sistema si aspetta che venga effettuato il focus gesture o il quick refocus gesture.
Se uno dei due è identificato prima di un certo timeout configurabile, lo stato diventa *in session*.
Altrimenti, se nessuno dei due viene riconosciuto lo stato cambia in *not in session*.
Le sessioni sono gestite dalla classe XnVSessionManager.

Si analizzano ora i tipi di controlli implementati in Nite.

I controlli sono oggetti che ricevono un particolare tipo di dati ed eseguono determinate azioni su tali dati.
In NITE i controlli più comuni sono i point control.

Questo tipo di controllo riceve gli hand point attivi da una sorgente e associa a questi punti delle specifiche azioni.
I controlli hanno eventi ai quali possono essere registrate delle callback, ovvero quando un controllo riconosce un determinato movimento chiama tutti gli eventi associati.

I point control permettono di associare eventi come la creazione di nuovi punti, il movimento dei punti e la scomparsa dei punti. Infatti quando un Point Control è connesso ad un Session Manager, riceve gli hand point attivi ad ogni frame ed esegue le sue azioni. Ogni controllo permette di registrare funzioni agli eventi specifici del controllo che vengono chiamate quando si verifica l'evento.

Ad esempio, il Swipe Detector ha l'evento Swipe, quindi tutte le callback registrate vengono chiamate quando il controllo riconosce il gesto Swipe.

I controlli supportati da NITE sono i seguenti:

- **Push Detector:**
  Questo controllo gestisce l'evento Push che consiste nel muovere la mano verso il sensore e tirarla indietro.
  Il gesto Push può essere usato, per esempio, per selezionare un oggetto o aprire una cartella.

- **Swipe Detector:**
  Rileva il gesto Swipe sia verso l'alto, sia verso il basso, a sinistra o a destra.
  Lo Swipe è un movimento breve in una specifica direzione dopo il quale la mano si ferma.
  Ad esempio, tale gesto potrebbe essere usato per sfogliare le pagine di un libro.

- **Steady Detector**:
  Il controllo cerca di riconoscere quando la mano è ferma per un determinato intervallo di tempo.
  Il gesto Steady avviene appunto quando la mano è completamente ferma o quasi.
  Questo gesto è utile agli altri controlli, per far sì che il successivo evento inizi da mano ferma.

- **Wave Detector:**

  Identifica il movimento ondulatorio della mano.
  Il gesto Wave consiste in un certo numero di cambiamenti di direzione della mano entro uno determinato lasso di tempo.

- **Circle Detector:**

  Questo controllo rileva movimenti circolari della mano. Affinché il gesto venga riconosciuto è necessario che la mano compia un giro completo sia in una direzione che nell'altra.
  La direzione positiva è considerata quella in senso orario, mentre quella in senso antiorario è considerata negativa.

- **SelectableSlider1D:**

  Il controllo riconosce uno scorrimento della mano lungo una delle tre direzioni degli assi X,Y,Z, cioè sinistra-destra, sopra-sotto, vicino-lontano.
  Può essere utilizzato per creare menu in cui ogni oggetto corrisponde ad una opzione del menu.

- **SelectableSlider2D:**

  Il controllo rileva uno scorrimento della mano in due direzioni sul piano X-Y.

Tutti questi controlli funzionano su un determinato hand point, non su tutti.
Questo punto (impostato dal Session Manager) viene chiamato Primary Point e inizialmente corrisponde al primo hand point riconosciuto.

Se il Primary Point per un qualche motivo non risulta più visibile al sensore ma altri handpoint sì, allora uno di questi diventerà il Primary Point.

Quindi ci sono altri quattro eventi associati al Primary Point: creazione del Primary Point, aggiornamento del Primary Point, distruzione del Primary Point e cambio del Primary Point, che avviene quando il Primary Point non è più disponibile ma esiste un altro hand point che prende il ruolo di Primary Point.
Se un controllo è connesso direttamente ad un Session Manager, significa che tutti i controlli sono sempre attivi e quindi rilevano gli eventi ogni volta che ci sono hand point attivi.

Nite inoltre permette di creare degli oggetti Flow, ovvero dei veri e propri controllori atti a determinare il flusso dei dati e configurabili per ricevere dati solo quando richiesto e quando i dati stessi sono significativi.
Ad esempio si può definire un intervallo di tempo all'interno del quale un gesto può essere riconosciuto, mentre tutto quello che avviene immediatamente prima o dopo viene ignorato.

Gli oggetti Flow disponibili in NITE sono:

- **flow router:** invia tutti i dati che riceve a un oggetto connesso;
- **broadcaster**: invia tutti i dati che riceve a tutti gli oggetti connessi;
- **point denoiser:** elimina piccoli movimenti causati dalla non perfetta stabilità della mano;
- **point area:** trasmette solo i punti che sono in una determinata area 3D;
- **virtual coordinates:** riguarda il piano dell'utente e trasforma tutti i punti in punti relativi a questo piano.

Per capire come operano i controlli e i vari oggetti NITE descritti fino ad ora si fa una panoramica sugli esempi forniti dalla libreria.

Single Control Sample

L'esempio mostra l'identificazione di un gesto Wave.
L'output è solamente rappresentato da una schermata.

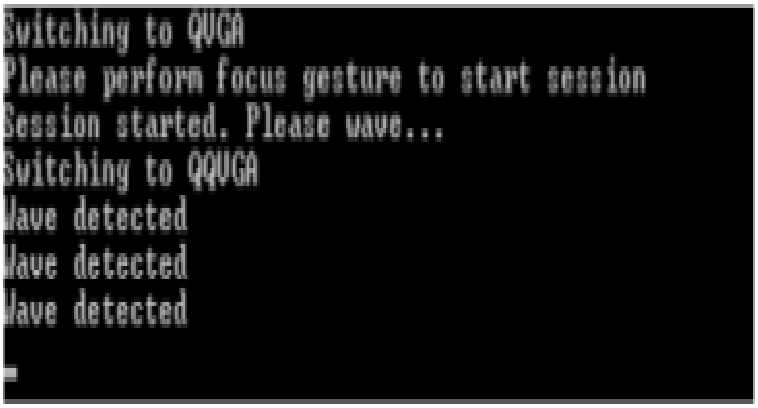

Fig. 4.2.2
Esempio di riconoscimento del gesto Wave.
[foto tratta da Prime Sensor NITE 1.3 Controls Programmer's Guide]

Circle Control Sample

In questo esempio viene identificato un movimento circolare della mano.
Viene disegnato un cerchio in base al movimento dell'handpoint.

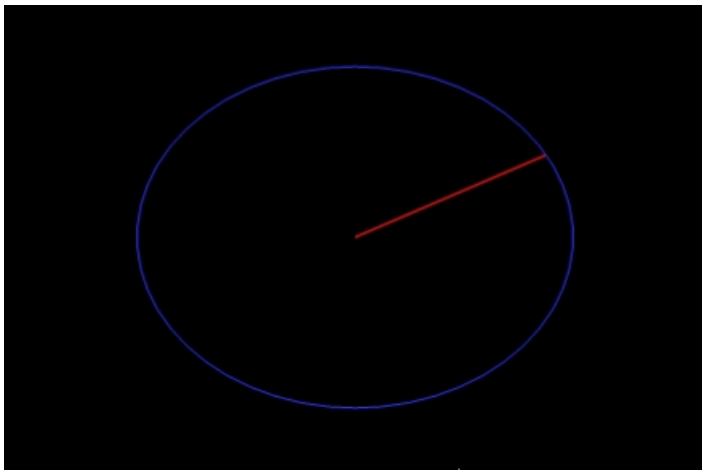

Fig. 4.2.3
Esempio di riconoscimento di un movimento circolare della mano.
[foto tratta da Prime Sensor NITE 1.3 Controls Programmer's Guide]

Point Viewer Sample

Nell'esempio si esegue il tracking delle mani.

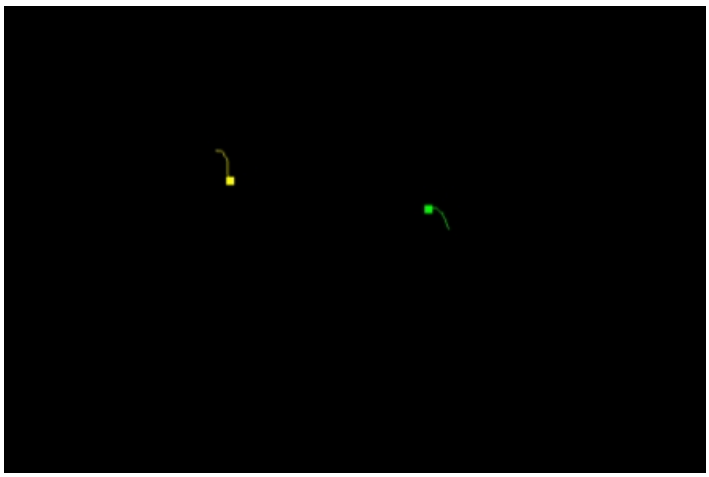

Fig. 4.2.4
Esempio di Hand Tracking.
[foto tratta da Prime Sensor NITE 1.3 Controls Programmer's Guide]

Boxes Sample

L'esempio mostra come vengono utilizzati gli oggetti Flow.
Vengono mostrati tre rettangoli e con il gesto Push è possibile selezionare uno di essi.
Inoltre una barra segue il movimento orizzontale della mano.

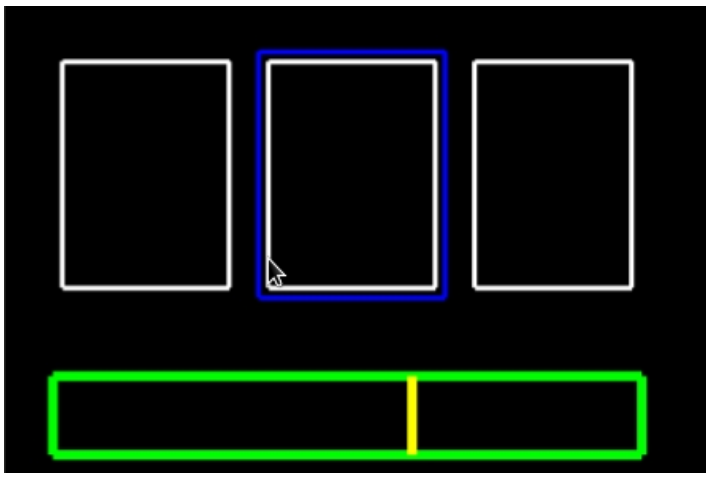

Figura 4.2.5
Esempio di selezione di oggetti.
[foto tratta da Prime Sensor NITE 1.3 Controls Programmer's Guide]

Stick Figure Sample

Nell'esempio viene riconosciuto il corpo dell'utente e gli viene assegnato il relativo scheletro.

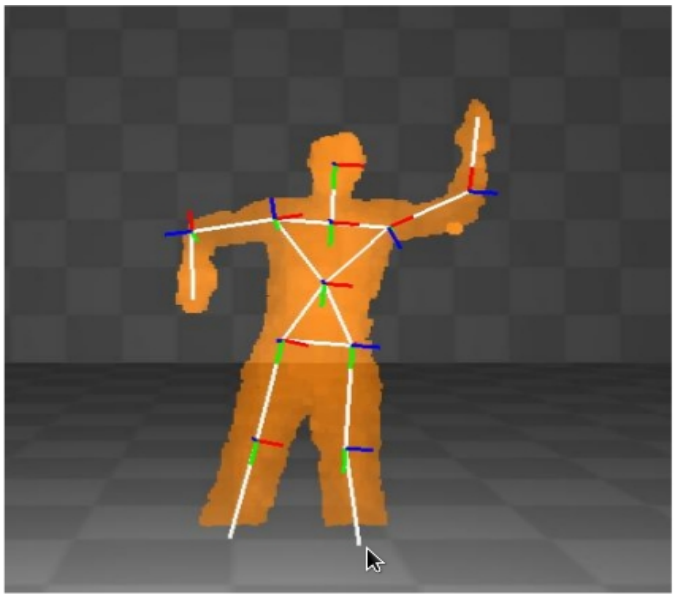

Fig. 4.2.6
Esempio di Skeleton Tracking.
[foto tratta da Prime Sensor NITE 1.3 Controls Programmer's Guide]

Capitolo 5

Connessione del modulo di interazione al modello velivolo

In questo capitolo si intende in un primo momento fornire una breve introduzione al calcolatore elettronico Matlab e all'interfaccia Simulink utilizzata nel corso del lavoro e successivamente si intende spiegare come è avvenuto il passaggio dei dati in output da Kinect a dati in input a Simulink.

5.1 Introduzione all'ambiente di calcolo

Come ambiente di calcolo, sviluppo e trasferimento dei dati rilevati col sensore Kinect si è scelto di utilizzare Matlab.
Matlab infatti, consente di manipolare matrici, visualizzare funzioni e dati, implementare algoritmi, creare interfacce utente e interfacciarsi con altri programmi.

All'interno di Matlab si è utilizzato Simulink, come piattaforma di simulazione in grado di modellare, simulare e analizzare sistemi dinamici di qualsiasi tipo (lineari e non).

La scelta è ricaduta su Simulink per primo perché il simulatore di UAV utilizzato è sviluppato in tale ambiente e poi per la sua semplicità di utilizzo.

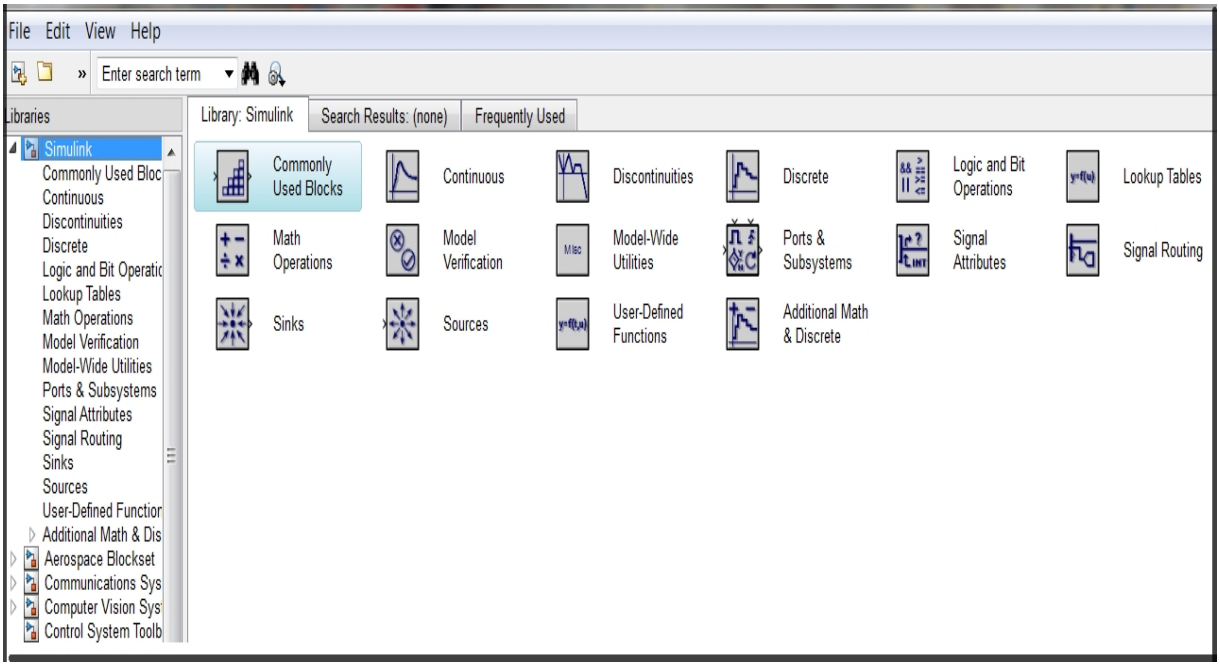

Fig. 5.1.1
Simulink Library

5.2 Operatori di trasferimento dati

Per il trasferimento dei dati in output da Kinect a Simulink ci si è appoggiati ad opportuni protocolli di trasporto dati: in particolare, il protocollo UDP (*User Datagram Protocol*) è impiegato per il trasporto semplice, senza connessione, che si basa sul trasferimento di pacchetti di dati, senza però impiegare connessioni di tipo client e server, contrariamente a TCP che è orientato alla connessione.
Inoltre, mentre TCP trasmette flussi di *byte*, il protocollo UDP invia pacchetti di taglia fissa, detti *datagram*.
L'assenza di connessione rende lo scambio via UDP più veloce di TCP, in particolare per trasferimenti di pochi byte.
Inoltre gli errori di trasmissione sono sporadici.
Per cui se i dati non sono critici, anche un protocollo che funziona quasi sempre bene può essere utile.

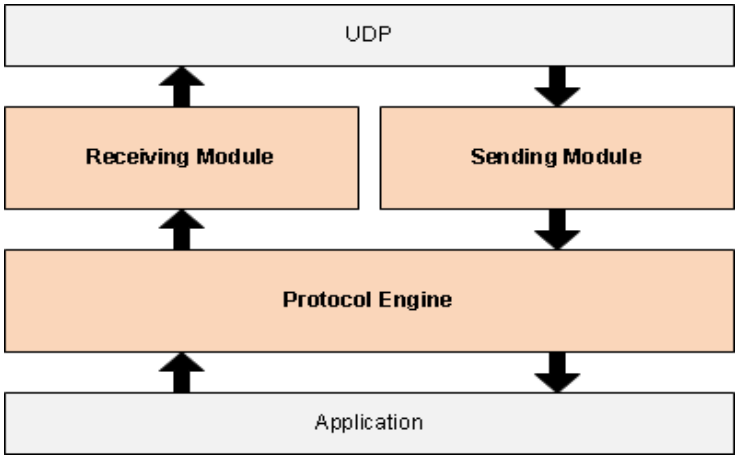

Fig. 5.2.1

Schema a blocchi del funzionamento dell'UDP
[foto tratta da www.codeproject.com]

5.3 Sistema Simulink sviluppato

Affinché il sensore Kinect invii correttamente i dati a Simulink è stato necessario studiare il simulatore, i dati richiesti in input e la sua architettura.
L'esempio di base che è stato utilizzato è rappresentato in figura 5.3.1.

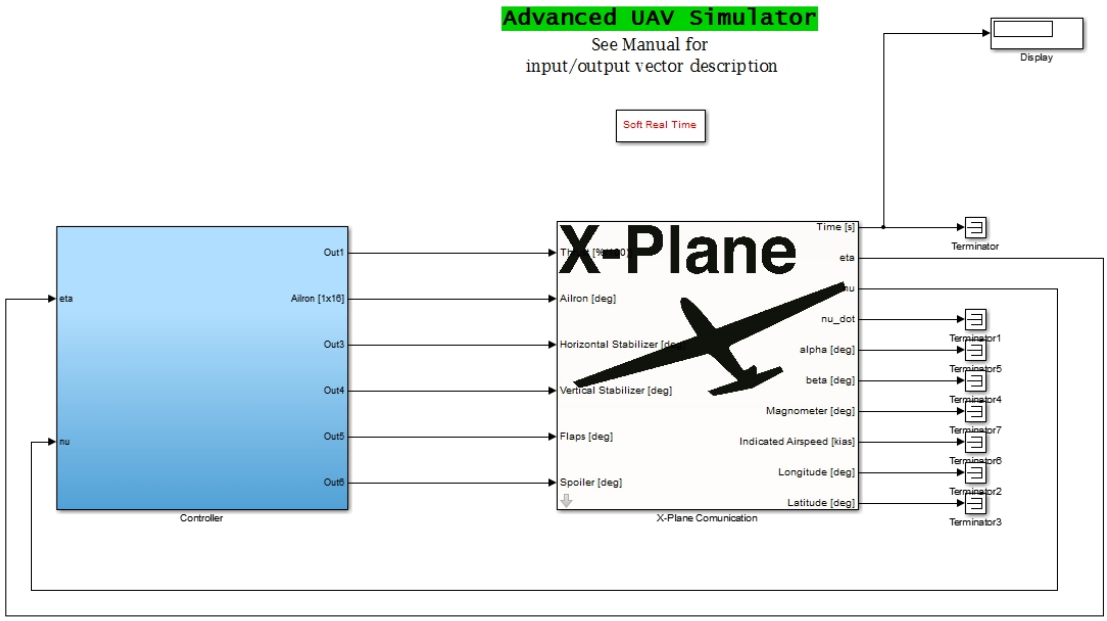

Fig. 5.3.1
Architettura del simulatore

Dalla figura 5.3.1 è possibile notare come i dati in input richiesti siano 6:

1. La spinta in percentuale di spinta massima;
2. La deflessione degli Alettoni in gradi;
3. La deflessione del timone orizzontale in gradi;
4. La deflessione del timone verticale in gradi;
5. La deflessione dei flaps in gradi;
6. La deflessione degli spoiler in gradi.

Per rendere il sistema il più semplice e intuitivo possibile è stato deciso di annullare la possibilità di controllare i flaps e gli spoiler e di unire in un unico comando la deflessione degli alettoni e del timone verticale.

Dall'analisi dei dati in input si è visto come essi in realtà non siano dei semplici numeri ma dei vettori, si faccia riferimento alle figure 5.3.2 e 5.3.3.

Table A.1: Input vector description

Variable	Description	Unit	Array Position	Dimension
ENGNThroRef[n]	Trust commanded from engine 1	Percent	0	[1x1]
ENGNThroRef[n]	Trust commanded from engine 2	Percent	1	[1x1]
ENGNThroRef[n]	Trust commanded from engine 3	Percent	2	[1x1]
ENGNThroRef[n]	Trust commanded from engine 4	Percent	3	[1x1]
ENGNThroRef[n]	Trust commanded from engine 5	Percent	4	[1x1]
ENGNThroRef[n]	Trust commanded from engine 6	Percent	5	[1x1]
ENGNThroRef[n]	Trust commanded from engine 7	Percent	6	[1x1]
ENGNThroRef[n]	Trust commanded from engine 8	Percent	7	[1x1]
AileronW1l1	Deflection Wing 1 Left Aileron 1	[deg]	8	[1x1]
AileronW1l2	Deflection Wing 1 Left Aileron 2	[deg]	9	[1x1]
AileronW1r1	Deflection Wing 1 Right Aileron 1	[deg]	10	[1x1]
AileronW1r2	Deflection Wing 1 Right Aileron 2	[deg]	11	[1x1]
AileronW2l1	Deflection Wing 2 Left Aileron 1	[deg]	12	[1x1]
AileronW2l2	Deflection Wing 2 Left Aileron 2	[deg]	13	[1x1]
AileronW2r1	Deflection Wing 2 Right Aileron 1	[deg]	14	[1x1]
AileronW2r2	Deflection Wing 2 Right Aileron 2	[deg]	15	[1x1]
AileronW3l1	Deflection Wing 3 Left Aileron 1	[deg]	16	[1x1]
AileronW3l2	Deflection Wing 3 Left Aileron 2	[deg]	17	[1x1]
AileronW3r1	Deflection Wing 3 Right Aileron 1	[deg]	18	[1x1]
AileronW3r2	Deflection Wing 3 Right Aileron 2	[deg]	19	[1x1]
AileronW4l1	Deflection Wing 4 Left Aileron 1	[deg]	20	[1x1]
AileronW4l2	Deflection Wing 4 Left Aileron 2	[deg]	21	[1x1]
AileronW4r1	Deflection Wing 4 Right Aileron 1	[deg]	22	[1x1]
AileronW4r2	Deflection Wing 4 Right Aileron 2	[deg]	23	[1x1]
HorizontalStabilizerW1E1	Deflection Horizontal Stabilizer Wing 1 Rudder 1	[deg]	24	[1x1]
HorizontalStabilizerW1E2	Deflection Horizontal Stabilizer Wing 1 Rudder 2	[deg]	25	[1x1]
HorizontalStabilizerW2E1	Deflection Horizontal Stabilizer Wing 2 Rudder 1	[deg]	26	[1x1]
HorizontalStabilizerW2E2	Deflection Horizontal Stabilizer Wing 2 Rudder 2	[deg]	27	[1x1]
VerticalStabilizerW1E1	Deflection Vertical Stabilizer Wing 1 Rudder 1	[deg]	28	[1x1]
VerticalStabilizerW1E2	Deflection Vertical Stabilizer Wing 1 Rudder 2	[deg]	29	[1x1]
VerticalStabilizerW2E1	Deflection Vertical Stabilizer Wing 2 Rudder 1	[deg]	30	[1x1]
VerticalStabilizerW2E2	Deflection Vertical Stabilizer Wing 2 Rudder 2	[deg]	31	[1x1]

Fig.5.3.2

[*foto tratta da UAV Simulink and X-Plane simulator, manual].*

]

Variable	Description	Unit	Array Position	Dimension
FlapsW1lF1	Deflection Wing 1 Left Flap 1	[deg]	32	[1x1]
FlapsW1lF2	Deflection Wing 1 Left Flap 2	[deg]	33	[1x1]
FlapsW1rF1	Deflection Wing 1 Right Flap 1	[deg]	34	[1x1]
FlapsW1rF2	Deflection Wing 1 Right Flap 2	[deg]	35	[1x1]
FlapsW2lF1	Deflection Wing 2 Left Flap 1	[deg]	36	[1x1]
FlapsW2lF2	Deflection Wing 2 Left Flap 2	[deg]	37	[1x1]
FlapsW2rF1	Deflection Wing 2 Right Flap 1	[deg]	38	[1x1]
FlapsW2rF2	Deflection Wing 2 Right Flap 2	[deg]	39	[1x1]
FlapsW3lF1	Deflection Wing 3 Left Flap 1	[deg]	40	[1x1]
FlapsW3lF2	Deflection Wing 3 Left Flap 2	[deg]	41	[1x1]
FlapsW3rF1	Deflection Wing 3 Right Flap 1	[deg]	42	[1x1]
FlapsW3rF2	Deflection Wing 3 Right Flap 2	[deg]	43	[1x1]
FlapsW4lF1	Deflection Wing 4 Left Flap 1	[deg]	44	[1x1]
FlapsW4lF2	Deflection Wing 4 Left Flap 2	[deg]	45	[1x1]
FlapsW4rF1	Deflection Wing 4 Right Flap 1	[deg]	46	[1x1]
FlapsW4rF2	Deflection Wing 4 Right Flap 2	[deg]	47	[1x1]
SpoilerW1lS1	Deflection Wing 1 Left Spoiler 1	[deg]	48	[1x1]
SpoilerW1lS2	Deflection Wing 1 Left Spoiler 2	[deg]	49	[1x1]
SpoilerW1rS1	Deflection Wing 1 Right Spoiler 1	[deg]	50	[1x1]
SpoilerW1rS2	Deflection Wing 1 Right Spoiler 2	[deg]	51	[1x1]
SpoilerW2lS1	Deflection Wing 2 Left Spoiler 1	[deg]	52	[1x1]
SpoilerW2lS2	Deflection Wing 2 Left Spoiler 2	[deg]	53	[1x1]
SpoilerW2rS1	Deflection Wing 2 Right Spoiler 1	[deg]	54	[1x1]
SpoilerW2rS2	Deflection Wing 2 Right Spoiler 2	[deg]	55	[1x1]
SpoilerW3lS1	Deflection Wing 3 Left Spoiler 1	[deg]	56	[1x1]
SpoilerW3lS2	Deflection Wing 3 Left Spoiler 2	[deg]	57	[1x1]
SpoilerW3rS1	Deflection Wing 3 Right Spoiler 1	[deg]	58	[1x1]
SpoilerW3rS2	Deflection Wing 3 Right Spoiler 2	[deg]	59	[1x1]
SpoilerW4lS1	Deflection Wing 4 Left Spoiler 1	[deg]	60	[1x1]
SpoilerW4lS2	Deflection Wing 4 Left Spoiler 2	[deg]	61	[1x1]
SpoilerW4rS1	Deflection Wing 4 Right Spoiler 1	[deg]	62	[1x1]
SpoilerW4rS2	Deflection Wing 4 Right Spoiler 2	[deg]	63	[1x1]

Fig. 5.3.3

[*foto tratta da UAV Simulink and X-Plane simulator, manual].*

E' possibile notare dalle figure 5.3.2 e 5.3.3 che in realtà i valori richiesti in ingresso siano molteplici e quindi non solamente 6.

Infatti per la spinta vengono richiesti 8 valori, per gli alettoni 16 e per il timone orizzontale e verticale 4.

Quindi i valori da passare da Kinect al simulatore dovranno essere ridimensionati tali da diventare un array di valori.

Inoltre il controllore dell'esempio mostrato in figura 5.3.1 è stato eliminato e sostituito dal blocco UDP.

Queste modifiche sono visibili nelle figure 5.3.4 e 5.3.5.

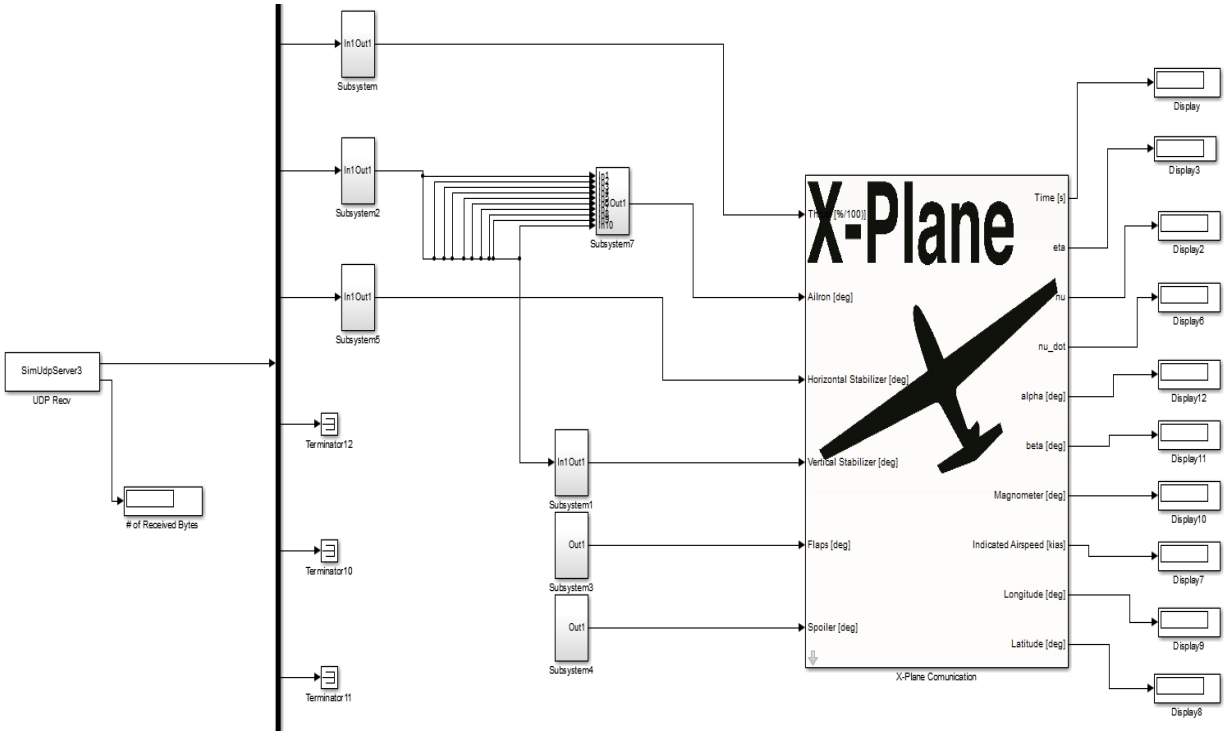

Fig. 5.3.4

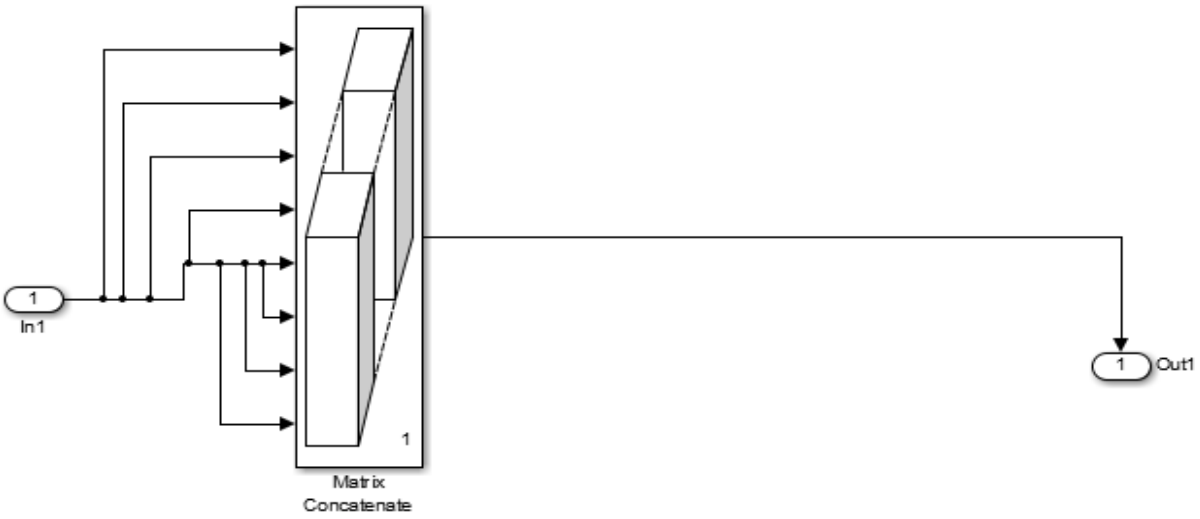

Fig. 5.3.5
Conversione dei valori di Kinect in matrici

Così modificato, l'esempio di partenza è ora pronto a mettersi in ascolto e ricevere dati dal sensore Kinect.

Il risultato sarà poi apprezzabile utilizzando un sistema di visualizzazione, in questo caso X-Plane.

Capitolo 6

Risultati e Conclusioni

Alla fine del lavoro si è riusciti ad ottenere dal sistema le azioni desiderate: infatti, i comportamenti richiesti nella fase di progettazione analizzata nel capitolo 3 sono poi stati implementati nel programma (si faccia riferimento all'appendice B), ottenendo così un controllo "gesturale" dei comandi di volo.
Il programma elaborato è stato poi collegato a *Simulink* ed è possibile pertanto apprezzare il risultato del lavoro osservando gli output dal simulatore.

Nel complesso le azioni progettuali architettate si sono dimostrate pertinenti al tipo di lavoro alla luce dei risultati ottenuti.

Infatti, l'interfaccia progettata consente all'utente di avere il pieno controllo dell'UAV col semplice movimento della mano: in particolare, la scelta di dividere il controllo in *range* di movimenti è risultata utile per mantenere la stabilità dell'UAV.
Ad esempio, considerando la manovra di beccheggio ottenuta dal movimento della mano lungo l'asse Y di *Kinect*, un movimento della mano verso l'alto di 15 cm comporta una cabrata di 10 gradi dell'UAV e per stabilizzarlo sarà sufficiente effettuare un movimento opposto di circa 15 cm (comunque non superiore a 20 e non inferiore a 10 cm), ovviando a problemi di annullamento di un comando che si potrebbero avere se si fosse implementato nel programma un algoritmo direttamente proporzionale tra azione e comando.

La tesi ha avuto sin dall'inizio l'obiettivo di creare un primo esempio di sistema "gesturale" per la movimentazione di un UAV simulato che sia il più semplice e intuitivo possibile.

Un importante risultato ottenuto è quello di utilizzare *hardware*, sensori e *software* commerciali e quindi facilmente accessibili a chiunque.
Riassumendo, il sistema proposto è frutto di un lavoro di progettazione, programmazione e conoscenza dei *software* utilizzati e si pone l'obiettivo di fornire all'operatore finale un'interfaccia innovativa e un nuovo modo di interagire con la propria postazione.
Il sistema rappresenta un tentativo originale di raccogliere alcune interessanti tecnologie che hanno trovato ancora limitate applicazioni nel settore aerospaziale.

Tra gli innumerevoli sviluppi futuri, citiamo la possibilità di progettare ulteriori *gesture* atte a controllare altri comandi tralasciati in questo lavoro, come ad esempio il controllo dei flaps, degli spoiler, del carrello ecc.
Inoltre, si potrebbe pensare di implementare un codice che consenta all'utente di avere a disposizione o un maggior numero di *range* di movimenti od anche un sistema che metta direttamente in correlazione i movimenti con i comandi, cioè senza ricorrere a *range* di spostamenti, prendendo però in considerazione gli inevitabili problemi di stabilità dell'UAV che si andranno a creare.

Appendice A - HandTracking

L'HandTracking è uno degli esempi utili ad utilizzare le funzionalità di Kinect messi a disposizione da PrimeSense.

La scrittura del programma è di seguito riportata:

```
#include "NiTE.h"
#include <NiteSampleUtilities.h>

int main(int argc, char** argv)
{
nite::HandTracker handTracker;
nite::Status niteRc;

niteRc = nite::NiTE::initialize();
if (niteRc != nite::STATUS_OK)
{
printf("NiTE initialization failed\n");
return 1;
}

niteRc = handTracker.create();
if (niteRc != nite::STATUS_OK)
{
printf("Couldn't create user tracker\n");
return 3;
}

handTracker.startGestureDetection(nite::GESTURE_WAVE);
handTracker.startGestureDetection(nite::GESTURE_CLICK);
printf("\nWave or click to start tracking your hand...\n");

nite::HandTrackerFrameRef handTrackerFrame;
while (!wasKeyboardHit())
{
niteRc = handTracker.readFrame(&handTrackerFrame);
if (niteRc != nite::STATUS_OK)
{
printf("Get next frame failed\n");
continue;
}
```

```
const nite::Array<nite::GestureData>& gestures =
handTrackerFrame.getGestures();

for (int i = 0; i < gestures.getSize(); ++i)
{
if (gestures[i].isComplete())
{
nite::HandId newId;
handTracker.startHandTracking(gestures[i].getCurrentPosition(), &newId);
}
}

const nite::Array<nite::HandData>& hands = handTrackerFrame.getHands();

for (int i = 0; i < hands.getSize(); ++i)
{
const nite::HandData& hand = hands[i];
if (hand.isTracking())
{
printf("%d. (%5.2f, %5.2f, %5.2f)\n", hand.getId(), hand.getPosition().x,
hand.getPosition().y, hand.getPosition().z);
}
}
}

nite::NiTE::shutdown();

}
```

E' possibile notare dall'analisi del programma come in un primo momento il sistema abbia bisogno di una inizializzazione.
Infatti prima ancora di iniziare la fase di rilevamento delle coordinate c'è una fase di calibrazione e quindi di tracciamento della mano.
Solo ora il sistema è pronto per iniziare a rilevare le coordinate, chiedendo all'utente (tramite una schermata a video) di compiere il gesto *Click* o *Wave* (si faccia riferimento alla figura 3.2.1).

A questo punto il sistema è posto in attesa di leggere un movimento da parte dell'utente e rimarrà in tale stato finché non verrà premuto un pulsante qualsiasi della tastiera.

Qualora l'utente compia uno dei due movimenti richiesti dal sistema, comparirà una nuova schermata video questa volta riportando, divise per colonne, il numero identificativo dell'utente, la coordinata X, Y, Z, della mano (come mostrato in figura 3.2.2).

Appendice B - Codice elaborato

Al fine di ottenere il comportamento desiderato dall'UAV l'*HandTracking* è stato così modificato:

```
#include "NiTE.h"

#include <NiteSampleUtilities.h>

#include "math.h"

# include "udp_lib.h"

#pragma comment(lib, "Ws2_32.lib")


int aggiorna_vettore_circolare(int index, int n_max, float vcirc[],
float el)
{
//assert(index < 0);

if (index >= n_max)
{
// vettore pieno! Devo liberare l'ultima pos.
for (int i = 1; i < n_max; i++)
{
vcirc[i - 1] = vcirc[i];
}
index = 0;
}
vcirc[index] = el;

return index + 1;
}
int main(int argc, char** argv)
{
nite::HandTracker handTracker;
nite::Status niteRc;

PTSocketUDP fromLoop;
int Port = 25000;
double buffer[6] = { 1, 2, 3, 4, 5, 6 };
```

```
InitSocket();

fromLoop = New_TSocketUDP(T_CLIENT);
int ris = Open_TSocketUDP(fromLoop, "137.204.201.64", Port, 0);
//UDP, noblocking stream on 1500
Connect_TSocketUDP(fromLoop);

// Send_TSocketUDP(fromLoop,(char*)buffer, sizeof(buffer));


FILE* fout = fopen("file_comandi.txt", "w");

float vector_xpos[30 + 1];
float vector_ypos[30 + 1];
float vector_zpos[30 + 1];
int   nxpos = 0;
int   nypos = 0;
int   nzpos = 0;

int punti_acquisiti = 0;

niteRc = nite::NiTE::initialize();
if (niteRc != nite::STATUS_OK)
{
printf("NiTE initialization failed\n");
return 1;
}

niteRc = handTracker.create();
if (niteRc != nite::STATUS_OK)
{
printf("Couldn't create user tracker\n");
return 3;
}

handTracker.startGestureDetection(nite::GESTURE_CLICK);
handTracker.startGestureDetection(nite::GESTURE_WAVE);
printf("\nWave or Click to start tracking your hand...\n");

nite::HandTrackerFrameRef handTrackerFrame;
while (!wasKeyboardHit())
{
niteRc = handTracker.readFrame(&handTrackerFrame);
if (niteRc != nite::STATUS_OK)
{
printf("Get next frame failed\n");
continue;
}
```

```
const nite::Array<nite::GestureData>& gestures =
handTrackerFrame.getGestures();
for (int i = 0; i < gestures.getSize(); ++i)
{
if (gestures[i].isComplete())
{
nite::HandId newId;
handTracker.startHandTracking(gestures[i].getCurrentPosition(), &newId);
}
}

const nite::Array<nite::HandData>& hands = handTrackerFrame.getHands();
for (int i = 0; i < hands.getSize(); ++i)
{
const nite::HandData& hand = hands[i];
if (hand.isTracking())
{
punti_acquisiti++;
printf("%d. (%5.2f, %5.2f, %5.2f)\n", hand.getId(), hand.getPosition().x,
hand.getPosition().y, hand.getPosition().z);

if (punti_acquisiti < 20) continue;
if (nxpos >= 15)
{

// il vettore circolare vector_xpos e' pieno! devo capire se sto dando
un comando!
double max_di_xpos = -1e10;
double min_di_xpos = 1e10;
double delta_x = 0;
for (int j = 0; j < 15; j++)
{
double xx = vector_xpos[j];
if (xx > max_di_xpos)
{
max_di_xpos = xx;
}
if (xx < min_di_xpos)
{
min_di_xpos = xx;
}
}
delta_x = max_di_xpos - min_di_xpos;
if (vector_xpos[0] < vector_xpos[14])
{
if (fabs(delta_x) > 100 && fabs(delta_x) < 200)
{
```

```
printf("VIRATA A DESTRA DI 10 GRADI");
buffer[0] = 10.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "VIRATA A DESTRA DI 10 GRADI\n");
fflush(fout);
}
}
if (fabs(delta_x) > 200 && fabs(delta_x) < 300)
{
printf("VIRATA A DESTRA DI 20 GRADI");
buffer[0] = 20.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "VIRATA A DESTRA DI 20 GRADI\n");
fflush(fout);
}
}
if (fabs(delta_x) > 300 && fabs(delta_x) < 400)
{
printf("VIRATA A DESTRA DI 30 GRADI");
buffer[0] = 30.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "VIRATA A DESTRA DI 30 GRADI\n");
fflush(fout);
}
}
}
if (vector_xpos[0] > vector_xpos[14])
{
if (fabs(delta_x) > 100 && fabs(delta_x) < 200)
{
printf("VIRATA A SINISTRA DI 10 GRADI");
buffer[0] = -10.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "VIRATA A SINISTRA DI 10 GRADI\n");
fflush(fout);
}
}
if (fabs(delta_x) > 200 && fabs(delta_x) < 300)
{
```

```
printf("VIRATA A SINISTRA DI 20 GRADI");
buffer[0] = -20.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "VIRATA A SINISTRA DI 20 GRADI\n");
fflush(fout);
}
}
if (fabs(delta_x) > 300 && fabs(delta_x) < 400)
{
printf("VIRATA A SINISTRA DI 30 GRADI");
buffer[0] = -30.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "VIRATA A SINISTRA DI 30 GRADI\n");
fflush(fout);
}
}
}
} // fine xpos
if (nypos >= 15)
{
// il vettore circolare vector_ypos e' pieno! devo capire se sto dando
un comando!
double max_di_ypos = -1e10;
double min_di_ypos = 1e10;
double delta_y = 0;
for (int j = 0; j < 15; j++)
{
double yy = vector_ypos[j];
if (yy > max_di_ypos)
{
max_di_ypos = yy;
}
if (yy < min_di_ypos)
{
min_di_ypos = yy;
}
}
delta_y = max_di_ypos - min_di_ypos;

if (vector_ypos[0] < vector_ypos[14])
{
if (fabs(delta_y) > 100 && fabs(delta_y) < 200)
{
```

```
printf("CABRATA DI 10 GRADI");
buffer[1] = 10.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "CABRATA DI 10 GRADI\n");
fflush(fout);
}
}
if (fabs(delta_y) > 200 && fabs(delta_y) < 300)
{
printf("CABRATA DI 20 GRADI");
buffer[1] = 20.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "CABRATA DI 20 GRADI\n");
fflush(fout);
}
}
if (fabs(delta_y) > 300 && fabs(delta_y) < 400)
{
printf("CABRATA DI 30 GRADI");
buffer[1] = 30.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "CABRATA DI 30 GRADI\n");
fflush(fout);
}
}
}
if (vector_ypos[0] > vector_ypos[14])
{
if (fabs(delta_y) > 100 && fabs(delta_y) < 200)
{
printf("PICCHIATA DI 10 GRADI");
buffer[1] = -10.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "PICCHIATA DI 10 GRADI\n");
fflush(fout);
}
}
if(fabs(delta_y) > 200 && fabs(delta_y) < 300)
{
```

```
printf("PICCHIATA DI 20 GRADI");
buffer[1] = -20.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "PICCHIATA DI 20 GRADI\n");
fflush(fout);
}
}
if(fabs(delta_y) > 300 && fabs(delta_y) < 400)
{
printf("PICCHIATA DI 30 GRADI");
buffer[1] = -30.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "PICCHIATA DI 30 GRADI\n");
fflush(fout);
}
}
}
} // fine ypos

if (nzpos >= 15)
{
// il vettore circolare vector_zpos e' pieno! devo capire se sto dando
un comando!
double max_di_zpos = -1e10;
double min_di_zpos = 1e10;
double delta_z = 0;
for (int j = 0; j < 15; j++)
{
double zz = vector_zpos[j];
if (zz > max_di_zpos)
{
max_di_zpos = zz;
}
if (zz < min_di_zpos)
{
min_di_zpos = zz;
}
}
delta_z = max_di_zpos - min_di_zpos;
if (vector_zpos[0] < vector_zpos[14])
{
if (fabs(delta_z) > 100 && fabs(delta_z) < 200)
{
```

```
printf("MANETTA INDIETRO DEL 40%%");
buffer[2] = -40.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "MANETTA INDIETRO DEL 40%%\n");
fflush(fout);
}
}
if (fabs(delta_z) > 200 && fabs(delta_z) < 300)
{
printf("MANETTA INDIETRO DEL 60%%");
buffer[2] = -60.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "MANETTA INDIETRO DEL 60%%\n");
fflush(fout);
}
}
if (fabs(delta_z) > 300 && fabs(delta_z) < 400)
{
printf("MANETTA INDIETRO A FONDO CORSA");
buffer[2] = -100.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));

if (fout != NULL)
{
fprintf(fout, "MANETTA INDIETRO A FONDO CORSA\n");
fflush(fout);
}
}
}
if (vector_zpos[0] > vector_zpos[14])
{
if (fabs(delta_z) > 100 && fabs(delta_z) < 200)
{
printf("MANETTA AVANTI DEL 40%%");
buffer[2] = 40.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "MANETTA AVANTI DEL 40%%\n");
fflush(fout);
}
}
if (fabs(delta_z) > 200 && fabs(delta_z) < 300)
```

```
{
printf("MANETTA AVANTI DEL 60%%");
buffer[2] = 60.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "MANETTA AVANTI DEL 60%%\n");
fflush(fout);
}
}
if (fabs(delta_z) > 300 && fabs(delta_z) < 400)
{
printf("MANETTA AVANTI A FONDO CORSA");
buffer[2] = 100.0;
Send_TSocketUDP(fromLoop, (char*)buffer, sizeof(buffer));
if (fout != NULL)
{
fprintf(fout, "MANETTA AVANTI A FONDO CORSA\n");
fflush(fout);
}
}
}
} // fine zpos
//printf("%d. (%5.2f, %5.2f, %5.2f)\n", hand.getId(),
hand.getPosition().x, hand.getPosition().y, hand.getPosition().z);
//vector_xpos[nxpos++] = x;
nxpos = aggiorna_vettore_circolare(nxpos, 30, vector_xpos,
hand.getPosition().x);
nypos = aggiorna_vettore_circolare(nypos, 30, vector_ypos,
hand.getPosition().y);
nzpos = aggiorna_vettore_circolare(nzpos, 30, vector_zpos,
hand.getPosition().z);
}
}
}

if (fout != NULL) fclose(fout);
nite::NiTE::shutdown();
}
```

Tra le principali modifiche attuate c'è la creazione di un vettore circolare capace di memorizzare trenta valori e di liberare poi la sua memoria una volta che è stato riempito per far spazio ad altri valori.

Questo vettore è necessario per capire in quale range di lavoro si trova la mano dell'utente, cioè se la mano si sta muovendo lungo un determinato asse e la sua direzione.
Infatti è grazie al confronto tra i valori del vettore che il sistema stabilisce che tipo di movimento sta avvenendo, e quindi che tipo di manovra assegnare all'UAV.
Inoltre, importante è notare la parte di trasmissione dei dati tramite canale UDP *Connect_TSocketUDP(fromLoop)*, che consente la comunicazione tra Kinect e Simulink.
La restante parte di programma rappresenta la traduzione in linguaggio C++ di quanto esposto nel capitolo 3.

Appendice C - Utilizzare Kinect come scanner 3D

Anche se non di competenza per il lavoro di questa tesi, a titolo informativo in questo capitolo viene brevemente illustrato come il dispositivo Microsoft Kinect possa all'occorrenza essere utilizzato come uno scanner 3D.

I programmi attualmente reperibili in rete e capaci di far funzionare il Kinect come scanner 3D sono ReconstructMe, Scenect e Skanect.
Le procedure di download sono semplici e libere.

Fig. A.1
Loghi dei software
[foto tratte da www.reconstructme.net, www.faro.com, www.skanect.occipital.com]

In figura A.2 è riportata l'interfaccia di Skanect mentre in figura A.3 è possibile apprezzare la qualità dell'immagine ottenuta dalla scannerizzazione.

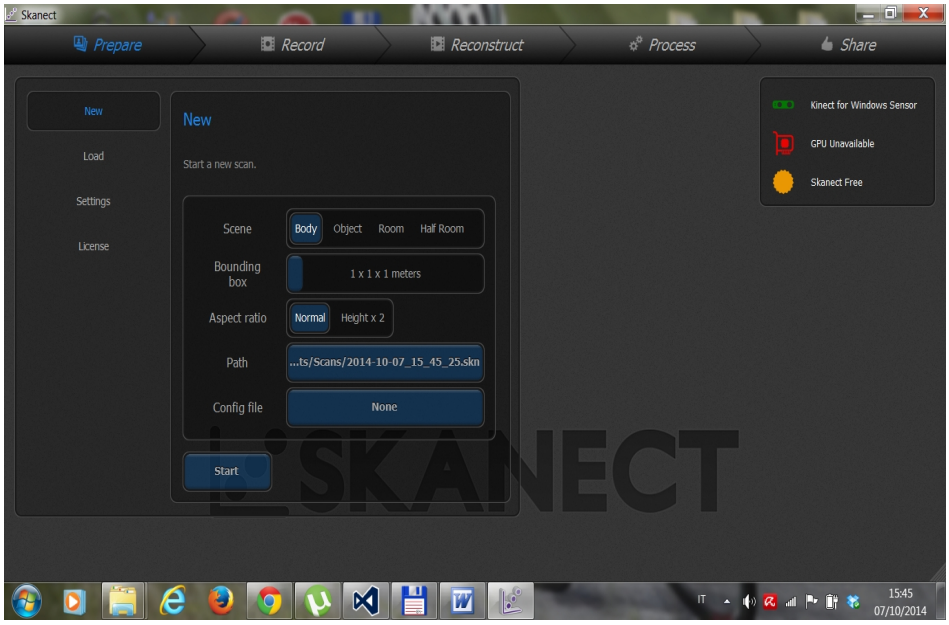

Fig. A.2

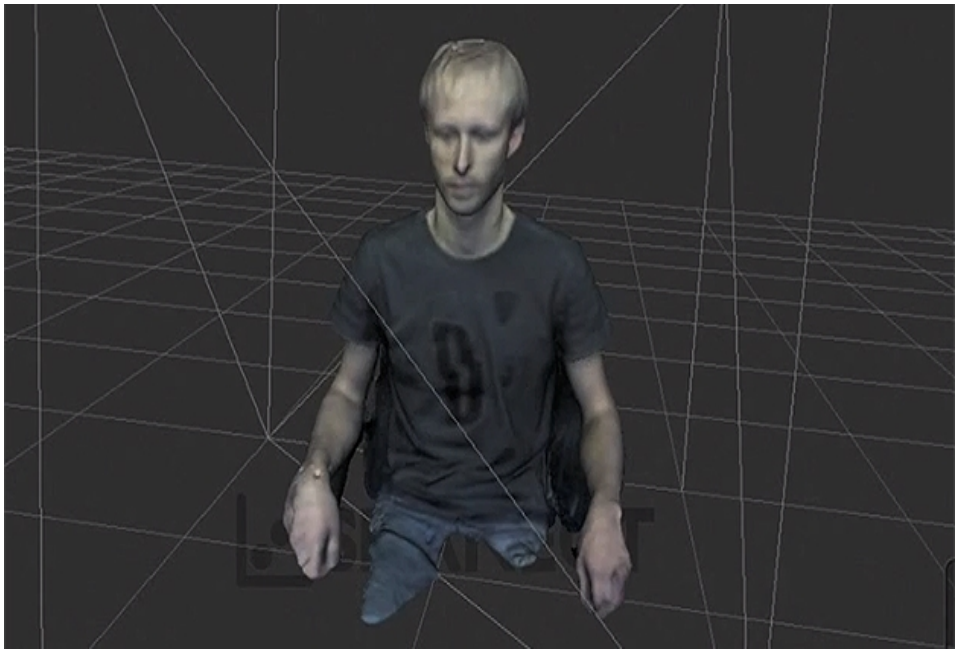

Fig. A.3

[foto tratta da www.solidsmack.com]

Bibliografia

1. Prime Sensor NITE 1.3 Framework Programmer's Guide, Versione 1.0, PrimeSense Inc.

2. Alan Dix, Janet Finlay, Gregory D. Abowd, Russel Beale - Interazione uomo-macchina - McGraw-Hill (2004)

3. PrimeSense - http://www.primesense.com -

4. Prime Sensor NITE 1.3 Algorithms notes

5. Harvey M. Deitel - C++ Fondamenti di programmazione

6. UAV Simulink and X-Plane simulator, manual

Ringraziamenti

Il mio GRAZIE più grande va d’obbligo ai miei genitori e alla mia famiglia.
Se oggi ho avuto la possibilità di raggiungere questo traguardo è grazie al sostegno costante e incondizionato che mi hanno sempre dato.
Sono proprio loro che mi hanno insegnato a non mollare mai e che le cose importanti vanno conquistate con il duro lavoro, la caparbietà e la determinazione.
Arrivare fin qui non è stato per niente facile.
Penso di non essere di natura un grande studioso ed è proprio per questo che per me quest’obiettivo vale davvero tanto.
Se sono riuscito a tagliare questo traguardo è proprio per la determinazione che ci ho messo nel raggiungere un obiettivo che mi ero prefissato e di dare soddisfazione a chi mi ha sostenuto e soprattutto a me stesso.
Certo è stata fondamentale la passione in questo ambito e la grande voglia di imparare sempre cose nuove.
Questa carriera universitaria non è stata formativa solo dal punto di vista didattico ma anche per altri aspetti: caratteriali, educativi e di maturazione.
Infine un grazie va a tutti coloro che ci sono stati durante il lavoro che mi ha visto impegnato questi mesi, un grazie particolare va all'Ing. Tiziano Bombardi per la sua preziosa collaborazione e pazienza dimostratami.
È stato un percorso lungo quindi sono tante le persone da ringraziare, chi c’è stato prima, chi dopo, chi più vicino, chi più lontano, chi comunque c’è sempre stato e soprattutto chi ci sarà!
«...in fondo oggi è il giorno che ti faceva paura ieri...»

Forlì, Dicembre 2014 Norman