ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

PROGETTAZIONE E SVILUPPO DI UN FRAMEWORK PER IL RICONOSCIMENTO REAL-TIME DI GESTURE SU KINECT

Elaborato in
BASI DI DATI

Relatore
Prof. ANNALISA FRANCO

Presentata da
LUCA GIULIANINI

Prima Sessione di Laurea
Anno Accademico 2017 – 2018

PAROLE CHIAVE

Gesture

Riconoscimento

Kinect

Machine Learning

Body Tracking

Alla mia famiglia e alla mia ragazza che mi sono stati accanto
nei momenti più difficili di questa avventura

Indice

Introduzione

Il gesto è uno degli strumenti di comunicazione più ancestrali e immediati che un essere umano possa utilizzare. Esso permette infatti di veicolare messaggi anche molto complessi in una frazione di secondo e garantisce ad entrambi gli interlocutori una comprensione ottimale. Per questo motivo esso è da sempre al centro di molte ricerche nell'ambito della computer science; lo scopo di queste ricerche è essenzialmente quello di garantire all'uomo un'interazione più naturale con un essere che di sua natura molto socievole non è, il computer.
Insegnare a un'ogetto, pressochè incapace di pensare, le basi della comunicazione non verbale, è un compito davvero complicato; basti pensare che ai nostri giorni esistono davvero pochissimi casi in cui un calcolatore sia in grado di comprendere e riconoscere un'emozione attraverso segnali non verbali. Per questo motivo molte persone rimangono scettiche sull'effettiva utilità e scalabilità di queste tecnologie nel lungo periodo.
E' innegabile però che in un momento storico come quello attuale in cui la tecnologia e il progresso fanno balzi da gigante (introducendo in sordina continue rivoluzioni come l'*industria 4.0*), la società si stia indirizzando verso un mondo sempre più automatizzato. Inoltre vedendo gli ultimi risultati ottenuti con l'utilizzo di *reti neurali* affiancate all'enorme quantitativo di dati a disposizione (*big data*), il passo per raggiungere le perfezione in questo campo è breve.

Obiettivi Lo scopo di questa tesi è quello di utilizzare alcune conoscenze di *machine learning* per realizzare un framework di *riconoscimento gesti*.

- La prima parte della tesi sarà rivolta a un'introduzione focalizzata sul concetto di **Interazione Uomo-Macchina** nella quale verranno sviscerati i capisaldi su cui si fonda tale disciplina.
  Successivamente verrà introdotto il concetto di **Gesture** fornendone da prima una visione astratta e poi una visione più tecnologica e implementativa dedicata allo stato dell'arte.
  Infine verrà introdotto il tema del riconoscimento di gesti (wearable e non wearable), in particolare verrano mostrate alcune tipologie di approc-

cio al riconoscimento seguite da una breve trattazione sulle tecnologie abilitanti questa disciplina.

- La parte centrale della tesi sarà rivolta a un'indagine (sintetica) esplorativa dei principali **algoritmi di machine learning** implicati nel riconoscimento gesti. In particolare grande enfasi verrà rivolta verso i due algoritmi utilizzati durante lo sviluppo del framework: l'algoritmo per il calcolo delle distanze fra sequenze temporali, **Dynamic Time Warping** e l'algoritmo di classificazione **K-Nearest Neighbor**.

- La seconda parte della tesi sarà riservata completamente all'elaborato. Inizieremo con un esposizione complessiva del sensore **Kinect** passando poi a una decrizione dettagliata dei processi di **analisi progettazione e sviluppo** del framework. Ogni processo verrà descritto nel dettaglio in sezioni dedicate, con un focus particolare alle strutture dati utilizzate per la gestione del dataset di gesture.
  In un secondo momento verrà discusso l'ambito applicativo dell'intero framework: in particolare verrano mostrati alcuni test sperimentali svolti su più utenti con lo scopo di analizzare le prestazioni generali del sistema.
  In calce seguiranno esempi di codice utili per l'utilizzo del framework.

Capitolo 1

Human-Computer Interaction

Sin dalla notte dei tempi l'uomo ha avvertito dentro di se l'impellente necessità di modificare il mondo intorno a lui.
Il metodo più semplice con cui ha realizzato questo impulso/desiderio è sempre stato attraverso la creazione di strumenti di volta in volta più complessi. In un lunghissimo viaggio lungo la storia umana possiamo notare come la complessità degli strumenti sia andata crescendo in modo esponenziale (siamo passati dai primi strumenti primordiali a veri e propri prodigi della meccanica in solo qualche millennio) [1]. Man mano che gli strumenti aumentavano di complessità essi guadagnavano una maggiore indipendenza dall'uomo e quindi pian piano si incominciava a scorgere la necessità di rendere più semplice l'interazione con essi. Questo processo è proseguito fino ai giorni nostri (con la nascita dei primi calcolatori) quando si è capito che era impensabile proseguire un tale processo di sviluppo dato che solo una piccola parte della popolazione poteva beneficiare di tale innovazione.

1.1 Definizione

Nasce cosi la human computer interaction, un campo di studio focalizzato sul design della tecnologia e, in particolare, sull'interazione uomo (l'utente) macchina (il computer). La Human-Computer Interaction (HCI) ha visto la luce per la prima volta agli inizi del 1980, inizialmente come area specifica della *computer science* che abbracciava le *Scienze Cognitive* e la *Human factors*. [1]

[1] La Human Factor, come disciplina, deriva dai problemi di progettazione delle attrezzature utilizzate dall'uomo durante la seconda guerra mondiale e quindi originariamente aveva forti legami militari. Il punto focale della disciplina erano gli aspetti senso-motori dell'interazione uomo-macchina (per esempio la progettazione di controlli di volo e altri dispositivi militari). L'interazione tra uomo e macchina non era originariamente vista in termini di aspetti comunicativi e cognitivi ma vista al livello tangibile, a livello muscolare.

La HCI si è espansa rapidamente e costantemente per tre decenni, attraendo professionisti di moltre altre discipline e incorporando diversi concetti e approcci. In larga misura, la HCI ora aggrega una raccolta di campi semi-autonomi di ricerca e pratica nell'informatica centrata sull'uomo. Tuttavia, la continua sintesi di concezioni nuove e approcci più disparati alla scienza ha prodotto un drammatico esempio di come paradigmi diversi possano essere riconcigliati e integrati in modo produttivo [2].

1.2 Nascita dell HCI

Fino alla fine degli anni '70, gli unici umani che interagivano con i computer erano professionisti della tecnologia dell'informazione e appassionati. Ciò cambiò in modo disordinato con l'emergere del *personal computing* negli anni '70. Il personal computing, che include sia software personale (applicazioni di produttività, come editor di testo e fogli di calcolo, giochi per computer interattivi) e piattaforme per personal computer (sistemi operativi, linguaggi di programmazione e hardware), ha reso chiunque un potenziale utente del computer, e ha evidenziato le carenze dei computer stessi rispetto all'usabilità che offrivano a coloro che volevano utilizzarli come semplici strumenti (fig 1.1).

Figura 1.1: Il personal computing spinge l'utilizzo del computer come prodotto di massa

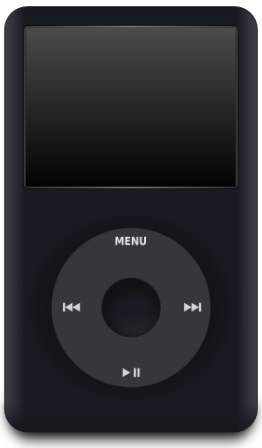
Il vasto calderone di *scienze cognitive*, che includeva la psicologia cognitiva, l'intelligenza artificiale, la linguistica, l'antropologia cognitiva e la filosofia della mente, si era formato alla fine degli anni '70.
Così, proprio nel momento in cui il personal computing ha presentato la necessità pratica di un'interazione con l'uomo, la scienza cognitiva ha fornito persone, concetti, abilità e una visione per affrontare tali bisogni attraverso una sintesi ambiziosa di scienza e ingegneria. Per questo motivo la HCI viene definita come uno dei primi esempi di *ingegneria cognitiva*.
L'ingegneria del software, bloccata in una complessità del software ingestibile negli anni '70 (la "crisi del software"), stava iniziando a concentrarsi su requisiti non funzionali, tra cui usabilità e manutenibilità.
La computer grafica era nata anch'essa negli anni '70 e si accorse rapidamente che i sistemi interattivi erano la chiave per progredire oltre i primi risultati.
Tutti questi filoni di sviluppo in informatica hanno indicato la stessa conclusione: **la strada da seguire per il computing implicava la comprensione e il potenziamento degli utenti**. Queste diverse forze di bisogno e opportunità si strinsero tutte intorno al 1980 creando un progetto interdisciplinare altamente visibile.

1.3 I Capisaldi della HCI

Il focus principale della Human Computer Interaction è sempre stato il concetto di **Usabilità**. [2] Questo concetto è stato inizialmente articolato in modo abbastanza ingenuo con lo slogan "facile da imparare, facile da usare", per questo motivo molti hanno iniziato ad associare alla HCI un'immagine molto semplificata incentrata solamente su un unico tema sminuendo cosi l'intera branca conoscitiva. Malgrado ciò, all'interno di HCI il concetto di usabilità è stato riarticolato e ricostruito quasi continuamente, ed è diventato sempre più ricco e complesso.
L'usabilità ora spesso include qualità come divertimento, benessere, efficacia collettiva, tensione estetica, maggiore creatività, flusso, supporto per lo sviluppo umano e altri [2].
Una menzione d'onore va a due temi fondamentali che hanno avuto la capacità di discostarsi sempre di più dal concetto più maturo di usabilità; l'**Accessibilità** e la **Semplicità**. Il primo ha garantito a chiunque una maggiore facilità nell'utilizzo dei servizi offerti mentre il secondo ha permesso di ridurre il gap che vi era, nell'uso della tecnologia, fra esperti del settore e semplici utenti.

[2] La norma ISO 9241-11:1998, poi aggiornata dalla ISO 9241-210:2010, definisce l'usabilità come: *Il grado in cui un prodotto può essere usato da particolari utenti per raggiungere certi obiettivi con efficacia, efficienza, soddisfazione in uno specifico contesto d'uso*

Grazie all'utilizzo di convenzioni e successivamente alla nascita di standard globalmente riconosciuti (vedi *ISO*) **Accessibilità e Semplicità hanno saputo imporsi diventando pian piano delle prerogative irrinunciabili alla usabilità**. (fig 1.2)



(a) L'ipod classic ha vinto la battaglia per l'usabilità in ambito mp3

(b) Spotify come esempio di streaming usabile

Figura 1.2: Spesso è l'usabilità a dettare il successo o l'insuccesso di un prodotto sul mercato.

1.3.1 Value Sensitive Design

E' interessante notare come **l'usabilità** sia una caratteristica profondamente connessa e dipendente dall'uomo; essa **ha senso solamente in presenza di un *utente* o di una *relazione d'uso*, e non esiste nel prodotto in sé**. Questo fatto non ci dovrebbe stupire, in fondo come abbiamo visto, tutta la HCI, da branca eclettica della computer science, fonda le proprie basi su discipline molto diverse.
Questa osservazione ha portato alla definizione della Value Sensitive Design (difficilmente traducibile in italiano). **Value Sensitive Design** è un approccio teoricamente fondato sul design della tecnologia che **tiene conto dei valori umani in modo completo durante tutto il processo di progettazione**. Impiega una metodologia tripartita integrativa e iterativa, costituita da indagini *concettuali*, *empiriche* e *tecniche*.[3]

1. Attraverso un indagine concettuale ci chiediamo su quali *valori* dell'utente il nostro design andrà a fare leva.

2. Un indagine empirica invece ci permetterà di capire *in che modo* è possibile raggiungere il nostro scopo e quindi prevedere il successo che il nostro design avrà sull'utente finale.

3. Infine un'indagine tecninca andrà a sondare pragmaticamente (in senso tecnologico) le possibilità realizzative del design.

1.3.2 User Experience

Mentre la Value Sensitive Design si riferisce al processo o alla strategia applicata per progettare esperienze, la **User Experience** [3] **si occupa dell'esperienza specifica che gli utenti hanno con i prodotti che usano**. È un riferimento al modo in cui un utente sperimenta e interagisce con un prodotto o servizio, **un concetto piuttosto che un processo**.
La User Experience, vista in questo senso, descrive la reazione dell'utente di fronte all'interazione con uno strumento in base a tre dimensioni: la dimensione *pragmatica*, quella *estetica/edonistica* e una *simbolica*.

1. La dimesione pragmatica è quella che si concentra maggiormente sui punti cardine, di basso livello, che rendono la nostra tecnologia *funzionale* e *usabile*. Questa dimensione è quella che più si avvicina al significato di Value Sensitive Design.

2. La dimesione edonistica è quella più apprezzata e vicina all'utente proprio perchè più personale ed emotiva. Paradossalmente è anche quella più complessa in campo progettuale proprio perchè spesso non dipende da standard ma solamente dai gusti personali degli utenti.

3. La dimesione simbolica rappresenta la parte più identificativa della tecnologia. In questo caso si entra nel campo del marketing e dell'importanza del brand.

1.4 Il futuro della HCI

Con il massiccio afflusso e avanzamento delle tecnologie, un sistema informatico è diventato una macchina molto potente che, pian piano, si è evoluta passando da un contesto *Activity-centered* (centrato sulle attività) a uno *Human-centered*. Come abbiamo visto, la la Human Computer Interaction è stata la prima e forse l'unica branca conoscitiva promotrice (e causa) di questo shift. Grazie alla propria presenza pervasiva è riuscita a cambiare totalmente

[3] L'ISO 9241-210[1] definisce l'esperienza d'uso come "*le percezioni e le reazioni di un utente che derivano dall'uso o dall'aspettativa d'uso di un prodotto, sistema o servizio*".

il nostro modo di vivere e ha definito nuove abitudini che col tempo hanno semplicato il nostro rapporto con la tecnologia.
Non ce ne rendiamo conto ma se diamo un piccolo sguardo intorno a noi ci accorgeremmo che siamo circondati da device intelligenti che oramai hanno acquisito un importanza prioritaria nelle nostre vite e dai quali saremo sempre più difficilmente portati ad allontanarci [4].
Prendendo atto che, secondo molte previsioni, questo rappresenti un trand in continua crescita, ci dovrebbero balzare alla mente due domande cardine:

1. **L'umanità è pronta a un'integrazione cosi fitta con la tecnologia?**

2. **La tecnologia riuscirà a garantire un'interazione sempre più naturale con l'uomo?**

La risposta alla prima domanda porta alla luce tematiche molto importanti, come la *singolarità tecnologia*[4] e il *transumanesimo*[5](fig 1.3), concetti molto futuristici e interessanti che però hanno ben poco a che fare con questa tesi.
La seconda domanda invece rappresenta il fulcro di questa trattazione e il nostro scopo sarà proprio provare a rispondere a questa domanda focalizzandoci su un particolare approccio di interazione HC, le **gesture**.

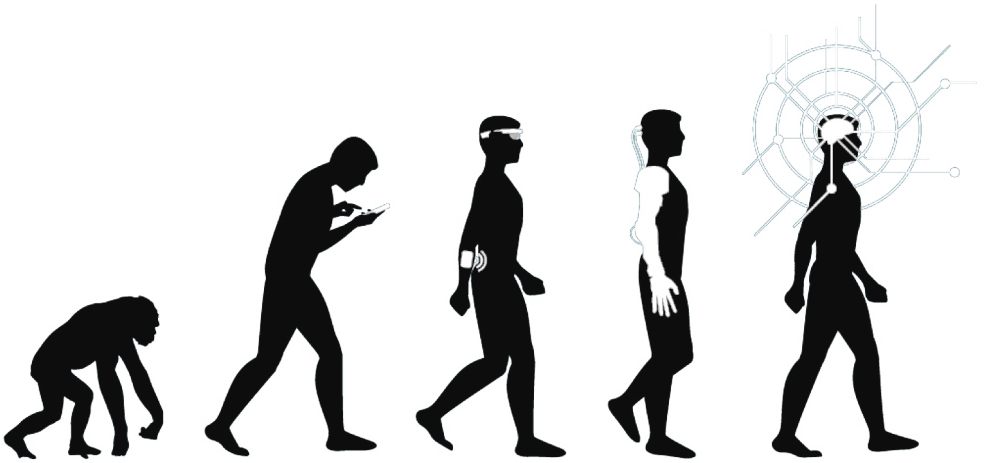


Figura 1.3: transumanesimo

[4]Nella futurologia, una singolarità tecnologica è un punto, congetturato nello sviluppo di una civiltà, in cui il progresso tecnologico accelera oltre la capacità di comprendere e prevedere degli esseri umani.

[5]Il transumanesimo (o transumanismo, a volte abbreviato con ¿H o H+ o H-plus) è un movimento culturale che sostiene l'uso delle scoperte scientifiche e tecnologiche per aumentare le capacità fisiche e cognitive e migliorare quegli aspetti della condizione umana che sono considerati indesiderabili, come la malattia e l'invecchiamento, in vista anche di una possibile trasformazione post umana.

Capitolo 2

Gesture

Come abbiamo visto nel capitolo precedente lo scopo principale della HCI è quello di garantire l'usabilità di una tecnologia. Per raggiungere tale obiettivo nel corso degli anni sono stati sviluppati un numero sempre maggiore di apparati di interfacciamento uomo-macchina.
Inizialmente questi avevano il solo e unico scopo di effettuare una mediazione fra *due protocolli comunicativi*: quello umano, molto più libero e slegato da convenzioni e quello della macchina, estremamente criptico e complicato. Tale divario nel processo comunicativo ha spinto i progettisti software, anche a causa di potenze di calcolo esigue e hardware ingombrante, a progettare interfacce sempre molto lontane dalle esigenze dell'utente, incentivando cosi la nascita di un alone di mistero intorno questo mondo.
Questa condizione ha portato gli esperti di HCI a ripensare totalmente il processo interattivo fra uomo e macchina. E' stato cosi definito un nuovo obiettivo a lungo termine che prende il nome di **Natural Migration**, ossia migrazione naturale, che ha lo scopo di **portare la comunicazione uomo-macchina a un livello completamente umano** (lo stesso utilizzato comunemente dalle persone per interagire l'una con l'altra)[4].
La Natural Migration ha rappresentato un punto di non ritorno a partire dal quale esperti di design e sviluppatori si sono cimentati nell'arduo compito di introdurre nella HCI nuove modalità di comunicazione uomo-uomo.
Fra queste modalità la comunicazione gestuale ha acquisito sempre più importanza tanto che la sua implementazione Human-Machine, **il riconoscimento di gesture**, attualmente rappresenta uno degli ambiti di ricerca più attivi e prolifici della computer science.

2.1 Definizione di Gesture

Gesto: I **gesti** sono movimenti elementari delle mani di una persona e rappresentano le comoponenti fondamentali della *comunicazione non verbale*.
Gli antropologi evoluzionisti considerano i gesti come il primo mezzo di comunicazione apparso nella storia dell'uomo, molto prima della nascita del lignuaggio.
Inoltre, i gesti sono parte naturale, ubiquitaria e significativa della lingua parlata. Secondo i ricercatori essi, oltre a essere un mezzo estremamente immediato di comuicare, assumono un ruolo fondamentale durante il processo di apprendimento della mente umana. Il gesto infatti accoppiato con il suono forma un sistema strettamente integrato capace a migliorare profondamente le prestazioni cognitive.
Ispirato all'interazione umana principalmente attraverso la visione e il suono, l'uso dei gesti è uno dei modi più potenti ed efficienti nell'interazione uomo-macchina.

Gesture: Una **gesture** è un componente della Human Computer Interaction che è diventata nel tempo ogetto di attenzione nei *sistemi multimodali*.[1]. Una definizione formale di gesture rimane però molto difficile da trovare. Se cercassimo nel vocabolario otterremmo un significato nel senso comune del termine quando in realtà nella comunità scientifica di definizioni ce ne sono e anche troppe.
Alcuni ricercatori definiscono una gesture come un **movimento tridimensionale del corpo o delle estremità**, altri come **il moto e l'orientamento delle mani e delle dita.**
Pochi fra questi considerano una gesture come i **segni inseriti da una mano attraverso un mouse, un polpastrello, un joystick o una penna**.
Solo in rarissimi casi i ricercatori hanno pensato il concetto di gesture con il significato di **espressione facciale** o **parlato** [5].

[1]Moller (2009): *multimodal dialogue systems are systems which enable human-machine interaction through a number of media, making use of different sensory channels. These channels may be used sequentially or in parallel, and they may provide complementary or redundant information to the user.*

2.1.1 Prossemica

Anche se nell'uso comune, i gesti sono caratterizzati da significati facilmente comprensibili, nella human computer interaction le gesture devono essere definite in modo esatto con l'obiettivo di renderle significative in ogni contesto di utilizzo. Non di meno la natura deterministica dell'input di un computer ci constringe a **definire un significato oggettivo per il concetto di gesture**. A questo scopo ci viene in aiuto la **Prossemica**, ossia la disciplina dedicata allo studio dei i gesti e dei metodi comunicativi. Introduciamo quindi qualche definizione:

Postura: *Una postura o posa è definita solamente dalla posizione della mano rispetto la posizione della testa. E' **statica** e quindi caratterizzata solo attraverso coordinate spaziali.*

HCI-Gesture: Definiamo con il nome di gesture una successione di Posture in un determinato intervallo di tempo.

Postura-Zero

In alcuni casi è possibile trovare una definizione ulteriore utilizzata per diversificare il significato delle posture. In questo caso viene aggiunta una **Postura Zero** che rappresenta un metodo per marcare l'avvenuta gesture (per esempio Kinect può essere messo in ascolto solo dopo aver alzato entrambe le mani). Solitamente il concetto di postura zero viene utilizzato quando è l'utente che definisce il punto di inizio e di fine della gesture.
La comunicazione tramite gesture diventa cosi un vero e proprio procollo comunicativo dove il messaggio inviato in modo asincrono (gesture) viene ricavato osservando opportuni delimitatori di start e stop (posture zero) (fig 2.1).

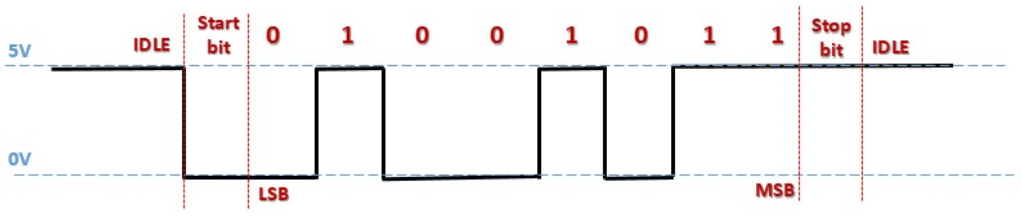


Figura 2.1: Il bus USB come parallelismo per comprendere la *Postura Zero*

2.1.2 Modelli Linguistici

Rispetto alle posture, le gesture contengono informazioni sul movimento molto più ricche, il che è importante per distinguerli fra loro, specialmente quelli più ambigui. La principale sfida del riconoscimento dei gesti risiede nella comprensione dei caratteri unici di quest'ultimi (fig 2.2). Esplorare e utilizzare questi caratteri nel riconoscimento dei gesti è fondamentale per ottenere le prestazioni desiderate.
Esistono due modelli linguistici per le gestures:

1. **Modello fonologico:** tratto dal **carattere concomitante del componente** [6]. Secondo questo modello **una gesture è individuata** in modo univoco **da** 3 caratteristiche simultanee e concomitanti che sono: **movimento, posizione e forma.**

2. **Modello movimento-attesa:** tratto dal **carattere dell'organizzazione sequenziale** [7]. Secondo questo modello **una gesture è rappresentata** come una sequenza lineare di **fasi di movimento alternate a fasi di attesa**. Sia le fasi di movimento che le fasi di attesa sono definite da **unità semantiche**.

Figura 2.2: Il comune gesto del "cosa vuoi", diventato famoso in tutto il mondo

2.2 Un Approccio formale al Riconoscimento

Come abbiamo visto nel capitolo precedente una gesture può essere definita attaverso due modelli linguistici, il modello Fonologico che si rifà al carattere concomitante del componente e il modello detto di Movimento-Attesa che fa riferimento al carattere dell'organizzazione sequenziale.
Il carattere concomitante del componente indica che componenti complementari, vale a dire movimento, posizione e componenti della forma, caratterizzano simultaneamente un gesto unico. Pertanto, **un metodo di riconoscimento dei gesti ideale dovrebbe avere la capacità di catturare, rappresentare e riconoscere questi componenti simultanei**. D'altra parte, **le fasi di movimento**, cioè le fasi di transizione, **sono definite come periodi durante i quali alcune componenti**, come il componente di forma, **sono in transizione**; **mentre le fasi di attesa sono definite come periodi durante i quali tutti i componenti sono statici**. Il carattere organizzativo sequenziale caratterizza un gesto come una disposizione sequenziale delle fasi di movimento e fasi di attesa. Si noti che il carattere concomitante del componente e il carattere di organizzazione sequenziale dimostrano rispettivamente le essenze dei gesti da aspetti spaziali e temporali. Il primo indica quali tipi di funzionalità devono essere estratti il successivo implica come l'utilizzo del ciclo di movimento e fasi di attesa in una sequenza di gesti possa rappresentare e modellare accuratamente il gesto [8] (fig 2.3).

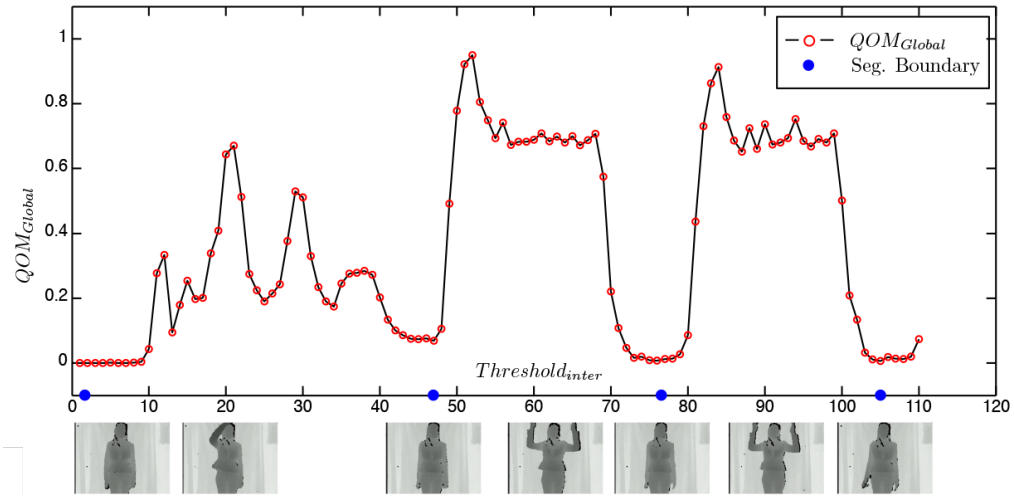


Figura 2.3: Un esempio di riconoscitore su paradigma linguistico. La modellazione del del ciclo movimento-attesa avviene in base alla quantità di moto registrata. La fase di attesa viene identificata tramite una Postura-Zero: le braccia sono rilassate, la quantità di moto minima

Il metodo appena descritto rappresenta l'approccio ideale al riconoscimento gesti poichè basandosi su un modello linguistico è capace di emulare molto bene il processo cognitivo umano.
E' bene però sottolieare che questa condizione di idealità avviene solo a livello teorico; nella pratica **l'implementazione di un riconoscitore** non è univoca, essa **dipende** infatti **sia dalla tipologia di sensore utilizzato sia dalla tipologia di riconoscimento richiesta**.

Capitolo 3

Riconoscimento di Gesture

Nell'ambito della computer science il concetto di riconoscimento non è sicuramente un ambito nuovo. Se da un lato bisognare dare grande merito alla HCI per aver spinto sin da subito in questa direzione, non si può dimenticare il grande apporto tecnico-conoscitivo fornito dalla robotica e dallo sviluppo dall'intelligenza artificiale di inizio anni 80. "Chi se non un robot necessitava maggiormente di un processo di riconoscimento"
A partire dagli anni '90 il tema inizia a diventare pervasivo e si diffonde in numerosi campi dell'informatica; le tecniche di classificazione che prima erano relegate ad ambiti di nicchia vengono riscoperte e nel giro di qualche anno vengono implementati con successo i primi riconoscitori.
Primi fra questi furono i riconoscitori vocali; la loro precocità in quest'ambito dipendeva dal semplice fatto che erano più facili da progettare dato che si basavano su dati di input più accessibili e generalmente più semplici da elaborare. Dovremo aspettare qualche anno, quando le tecnologie diventeranno più mature, prima di vedere implementati i primi riconoscitori in ambito gesture.

3.1 Applicazioni e Sistemi Tipici

Se dovessimo tracciare un grafico che rappresenta la percentuale di diffusione nel corso degli anni di dispositivi dotati di riconoscimento gesti, noteremmo subito un picco in corrispondenza dell'ultimo decennio. Questo enorme diffusione non è casuale ma è stata favorita da una serie di coincidenze molto particolari. L'ultimo decennio infatti ha visto, nell'ambito delle tecnologie informatiche e elettroniche, una serie di cambiamenti rivoluzionari che hanno modificato profondamente il rapporto che le persone avevano con la tecnologia.
Il vero innesco al cambiamento è partito dal comparto elettronico, che grazie a uno straordinario sviluppo nei processi di minimizzazione, ha potuto finalmen-

te produrre dispositivi compatti a un costo irrisorio. Lo sviluppo di software dedicati non si è fatto attendere e nel giro di qualche anno anche l'informatica ha vissuto la sua prima rivoluzione, la nascita dell'**industria 4.0**[1]. Sin dalla sua introduzione nel 2011, il concetto di industria 4.0 si è evoluto e adattato a nuove necessità, riuscendo così ad affermarsi rapidamente negli ambiti più disparati.
L'industria 4.0 deve gran parte della suo successo a un altro pilastro dei nostri giorni l'**Internet of things**[2], (internet delle cose) processo che ha portato allo sviluppo dei primi **dispositivi embedded** e quindi alla nascita di nuovi paradgmi di HCI come la **Ubiquous Computing**[3], detta anche **Pervasive Computing**.

3.1.1 Uno Sviluppo Pervasivo

Tutte queste rivoluzioni in campo tencologico ci mostrano come il riconoscimento di gesti sia estremamente legato all'aspetto indipendente e pervasivo della computazione. Recenti studi hanno riportato come il mercato dei dispositivi mobili e di riconoscimento siano in grande crescita.
Secondo una ricerca portata avanti da *Variant Market* dal 2016 al 2024 è prevista una crescita del 16% per il settore del riconoscimento nel mondo.
Un altro studio questa volta proveniente da *Industry ARC* riporta una crescita del 35% dal 2018 al 2022 per lo stesso settora relativamente all'elettronica di consumo.
Un'ultimo rapporto fornito da *Grand View Research*, anche se riferito solo al contesto cinese, ci riporta un quadro interessante. Secondo gli analisti di Grand View in China entro il 2025 ogni ambito elettronico concomitante al riconoscimento di gesti subirà uno sviluppo esponenziale, indipendentemente dalla fascia di mercato ricoperta (fig 3.1).

[1]Industria 4.0, prende il nome dall'iniziativa europea Industry 4.0, a sua volta ispirata ad un progetto fatto dal governo tedesco. Il termine indica una tendenza dell'automazione industriale che integra alcune nuove tecnologie produttive per migliorare le condizioni di lavoro e aumentare la produttività e la qualità produttiva degli impianti.

[2]L'Internet delle cose è una possibile evoluzione dell'uso della Rete: gli oggetti (le "cose") si rendono riconoscibili e acquisiscono intelligenza grazie al fatto di poter comunicare dati su se stessi e accedere ad informazioni aggregate da parte di altri

[3]Lo ubiquitous computing (in italiano: computazione ubiqua) o ubicomp è un modello post-desktop di interazione uomo-macchina, in cui l'elaborazione delle informazioni è stata interamente integrata all'interno di oggetti e attività di tutti i giorni.

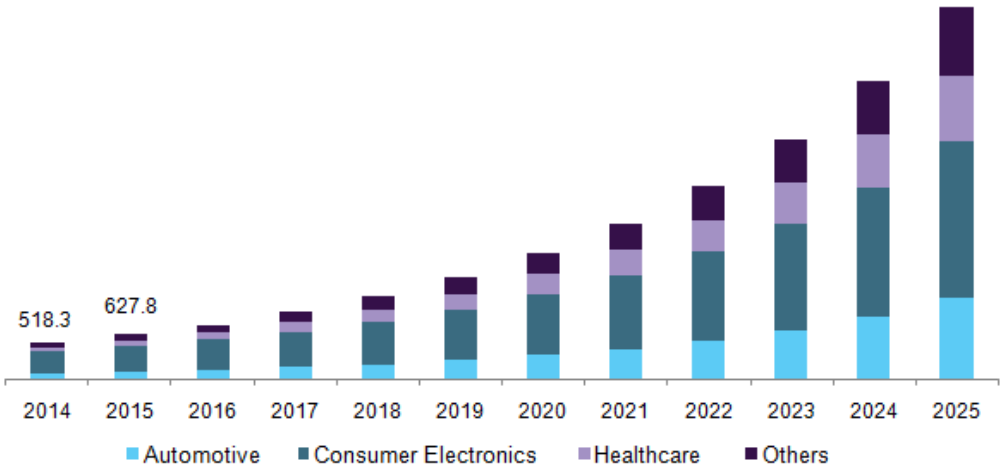


Figura 3.1: Lo studio riportato da Gran View

3.1.2 Ambiti di Applicazione

Al momento, per quanto riguarda i sistemi di riconoscimento gesture, le principali applicazioni sono distribuite attorno a 6 domini fondamentali[9]:

- **Riconoscimento del linguaggio dei segni:** Il riconoscimento della lingua dei segni attraverso i gesti delle mani è stato studiato per la prima volta per comprendere l' ASL, American Sign Language (lingguaggio americano dei segni). L'obbiettivo di questo studio e altri studi svolti fino ad ora consisteva nel fornire alle persone sordomute un strumento comunicativo capace di comprendere il linguaggio dei segni. Dato l'ambito di applicazione estremamente critico, lo sviluppo di applicazioni di questo genere hanno necessitato l'uso di tecnologie all'avanguardia. Fra i traguardi più importanti in questo campo troviamo **intAIRact**. Il vantaggio principale di questo framework rispetto agli applicativi esistenti risiede nell'algoritmo utilizzato. Quest'ultimo, infatti, è capace di inferire i gesti delle mani e le posizioni delle dita usando solamente immagini a profondità e RGB ottenute da fotocamere facilmente reperibili. Inoltre intAIRact è accessibile online e in qualsiasi momento è possibile registrare nouvi gesti in modo molto semplice e veloce.

- **Manipolazione Virtuale:** La manipolazione virtuale è una delle applicazioni più popolari del riconoscimento gesti in ambito HCI. Esistono due possibili ambiti di utilizzo per questa applicazione. Il primo consiste in un'implementazione pura dell'interazione uomo-macchina; per esempio si può utilizzare il riconoscimento per manipolare interfacce **CAD**

(computer aided design design) senza l'ausilio di mouse e tastiere. Un secondo approccio consiste nell'integrare il riconoscimento al concetto di **realtà virtuale**. In questo caso il sistema acquisisce la posizione della mano e così l'utente è in grado di manipolare un oggetto virtuale nella scena stereoscopica 3D. Nella maggior parte delle implementazioni viene utilizzato un sensore a profondità, con lo scopo di tracciare la mano dell'utente e renderizzare gli oggetti virtuali in base al punto di vista dell'utente stesso (fig 3.2).

- **Assistenza Giornaliera:** L'uso delle gesture potrebbe trovare nel tempo un'applicazione intensiva nel campo dell'assistenza giornaliera verso le categorie sociali protette. Nel caso di persone anziane, l'uso di gesti potrebbe migliorare il processo di assistenza, fornendo un aiuto nello svolgimento di alcune attività complesse come lavarsi le mani o comunicare con altre persone. Inoltre, un'interazione intelligente con dispositivi di domotica garantirebbe un confort maggiore all'anziano, facilitando l'invecchiamento presso la propria abitazione.

- **Palm Verification:** Potrebbe sembrare strano ma il riconoscimento gesti gode di impieghi anche nel dominio dei **sistemi biometrici**[4]. In questo caso viene sfruttata la verifica del palmo della mano, una tecnologia chiave per molte applicazioni di sicurezza. Esistono due tipologie di approcci al processo identificativo: **statico** e in **posa libera**. Il primo metodo si basa sull'uso di una fotocamera 2D affiancata a un tavolo di illuminazione; quando l'utente pone la mano sul tavolo, la fotocamera scatta una foto e l'algoritmo di image processing si attiva per segmentare (staticamente) la silhouette della mano in diverse regioni caratteristiche. Il secondo metodo, più evoluto, sfrutta un digitalizzatore 3D per acquisire simultaneamente immagini di intensità e gamma della mano dell'utente. In questo modo, possedendo una mappatura tridimensionale, è possibile riconoscere il palmo della mano da qualsiasi angolazione.

- **Gaming:** L'ambito gaming è forse l'ambito che più ha spinto lo sviluppo dei riconoscitori in ambito commerciale. Fra le prime aziende a proporre un sensore adibito il riconoscimento in ambito gaming troviamo Nintendo con la **Wii** e Microsoft con **Kinect**. Playstation non si è fatta attendere e dopo poco tempo ha rilasciato il proprio sensore **PS Move**. L'impor-

[4]Un sistema di riconoscimento biometrico è un particolare tipo di sistema informatico che ha la funzionalità e lo scopo di identificare una persona sulla base di una o più caratteristiche biologiche e/o comportamentali (biometria), confrontandole con i dati, precedentemente acquisiti e presenti nel database del sistema, tramite degli algoritmi e di sensori di acquisizione dei dati in input.

tanza di questi sensori stranamente non è derivata dall'ambito gaming, bensì dal loro utilizzo come piattaforme per lo sviluppo di applicazioni.

- **HRI:** Il campo di applicazione più futuristico per il riconoscimento gesti risiede sicuramente nell'ambito della **Human-Robot Interaction**, una versione avanzata della HCI. L'interfaccia basata sui gesti infatti, offre un modo molto semplice ed efficiente per interagire con il mondo dei robot. Il maggior numero delle implementazioni sfrutta algoritmi di computer vision associati al metodo della **sensor-fusion**[5]

Figura 3.2: Un esempio di manipolazione virtuale applicata all'ambito della medicina. Il chirurgo può interagire con il sistema a distanza in modo veloce e sopratutto sicuro

3.2 Sensori per il Riconoscimento di Gesture

E' interessante notare come l'evoluzione nell'ambito del riconoscimento gesti sia sempre stata strettamente correlata al progresso nell'ambito della sensoristica. Esistono essenzialmente tre tipologie di sensori che possono essere sfruttati per percepire gesti: sensori **montati su supporto**, sensori su **schermo multi-touch** e sensori basati sulla **visione**.

[5]La tecnica sensor fusion consiste nel combinare dati sensoriali o dati derivanti da fonti separate in modo tale che l'informazione risultante sia più precisa di quella ottenuta considerando le fonti separatamente.

3.2.1 Sensori montati su supporto

Definiti anche come **built-in sensors**, questa categoria di sensori la troviamo generalmente integrata all'interno di dispositivi embedded con l'acronimo di **MEMS**. La sigla MEMS sta per Micro Electro-Mechanical Systems e aggrega dentro di se un intero mondo di dispositivi di varia natura (meccanici elettrici ed elettronici), integrati in forma altamente miniaturizzata (ordine dei micro-metri) su uno stesso substrato di materiale semiconduttore. Fra gli esempi più importanti di micro-sensorisitca troviamo l'**Accelerometro e** il **Giroscopio**.

Accelerometro

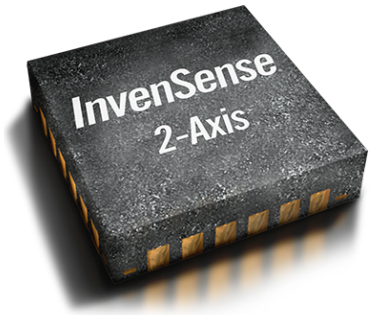
Montato su un oggetto, questo sensore è capace a misurare l'accelerazione a cui è sottoposto l'oggetto stesso (misura in realtà la forza specifica applicata in N). I Principi di funzionamento sono vari e sfruttano comunemente le caratteristiche fisiche del materiale di cui sono composti i sensori. Basandosi su questi principi di funzionamento possiamo dividere gli accelerometri in: estensimetrici, piezoresistivi, capacitivi, piezoelettrici e gravitometri (fig 3.3a).

Giroscopio

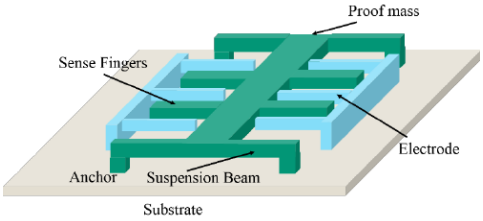
E' un dispositivo fisico rotante che mantiene il suo asse di rotazione in una direzione fissa. Questo avviene per effetto della legge di conservazione del momento angolare e permette quindi al giroscopio di essere un dispositivo invariante ai cambi di orientamento. Essenzialmente è costituito da un rotore a forma di toroide che ruota intorno al suo asse; quando il rotore è in rotazione il suo asse tende a mantenersi parallelo a se stesso e ad opporsi ad ogni tentativo di cambiare il suo orientamento (fig 3.3b).

Applicazioni nell'ambito del riconoscimento

Nell'ambito del riconoscimento, i sensori built-in hanno giocato un ruolo fondamentale sin dal loro primo impianto nei sistemi embedded. Il loro utilizzo varia dal semplice riconoscimento della posizione dell'oggetto (lettura del valore) al riconoscimento di serie temporali, per esempio gesti eseguiti con il dispositivo.
Generalmente nel caso di riconoscimento di pattern complessi, dove è richiesto un surplus di precisione da parte dello strumento, si fa uso della tecnica di **Sensor Fusion** (citata precedentemente). Per esempio fondendo i segnali forniti da un Accelerometro e da un GPS è possibile ottere misure molto precise riguardanti il moto di un oggetto.



(a) Un tipico esempio di giroscopio a 2 assi



(b) Modello di accelerometro mems di tipo capacitivo

Figura 3.3

3.2.2 Sensori Multi-Touch

In elettronica una delle rivoluzioni più incisive della storia è avvenuta sicuramente nell'ambito dello sviluppo del display. Dapprima, con l'introduzione della tecnologia lcd/plasma, i display hanno subito un rimpicciolimento drastico delle dimensioni (permettendo finalmente un loro utilizzo in ambito mobile). Successivamente il processo di rimpicciolimento si è spostato sui dispositivi stessi; la classica tastiera fisica viene eliminata e rimpiazzata da schermi touch e multi-touch. Al giorno d'oggi esistono principalmente due tipologie di schermi **multi-touch**[6]: Schermi **resistivi** e **capacitivi**.

Schermo Resistivo:

Uno schermo tattile resistivo è costituito da due pellicole trasparenti conduttive caratterizzate da una certa resistenza elettrica, sovrapposte ma separate tra loro. Quando il pennino o il dito dell'utente tocca lo schermo si crea un circuito, ai capi del quale si produce una tensione diversa a seconda del punto in cui il dito ha toccato lo schermo. Misurando questa tensione il dispositivo elettronico, su cui lo schermo è montato, è in grado di determinare la posizione del dito.

Schermo Capacitivo:

Uno schermo tattile capacitivo è un dispositivo di input/output che sfrutta la variazione di capacità dielettrica tipica dei condensatori sul vetro dello

[6]I display multitouh sfruttano gli stessi principi dei classici touch, aggiungendo però il suporto al riconoscimento di più tocchi

schermo stesso. Ai quattro angoli del pannello viene applicata una tensione che si propaga uniforme su tutta la superficie dello schermo (per via dell'ossido di metallo presente); quando il dito o un materiale conduttore tocca lo schermo, avviene una variazione di capacità superficiale che viene letta da una matrice di condensatori a film posizionati su un pannello, posto al di sotto della superficie del vetro. Questi tipi di schermi sono particolarmente usati negli smartphone e nei tablet di ultima generazione.

Applicazioni nell'ambito del riconoscimento

Nel caso di schermi touch le applicazioni nel campo di riconoscimento sono limitate. Gli usi rientrano essenzialmente nel campo dell'interazione uomo-macchina e consistono generalmente nel migliorare e semplificare il metodo di interfacciamento. Un esempio di utilizzo è da ricercarsi negli smartphones di ultima generazione, i cosiddetti full-screen smartphones. Con l'obiettivo di sfruttare massivamente ogni millimetro di schermo disponibile, gran parte delle aziende produttrici ha infatti deciso di sostituire i classici pulsanti di navigazione virtuale, in favore delle gestures (per loro natura più immediate e intuitive).

3.2.3 Sensori Visivi

Fino ad ora abbiamo incontrato due tipologie di sensori: quelli built-in, generalmente raggruppati sotto la sigla MEMS e gli schermi multitouch. Sebbene i sensori di questo tipo abbiano il pregio di essere estramente compatti e adatti a impieghi mobile, essi si dimostrano pressochè inutilizzabili in contesti in cui il contatto fisico con il dispositivo fosse limitato o addirittura impossibile. Per ovviare a questo problema ci vengono in soccorso cosiddetti sensori basati sulla visione.
E' molto complicato dare una definizione di **Sensore Visivo** data l'enorme varietà di tecnologie implementative. Generalmente qualifichiamo un sensore come visivo se agisce a distanza dall'oggetto e se, attraverso processo di sensing, viene generata qualunque modellazione del mondo (2D o 3D). Esistono essenzialmente 3 categorie fondamentali di sensori visivi, diversificate in base al tipo di acquisizione: la classe dei sensori **2D**, quella dei sensori cosiddetti **2.5D** e infine **3D** .

Sensori 2D (RGB)

Un sensore 2D o sensore di immagini è un dispositivo che converte un'immagine ottica in un segnale elettrico. Il principio base di funzionamento è diversificato in base alla classe di colore ripresa.

Per riprese in bianco e nero il procedimento è il seguente: l'immagine viene focalizzata su una griglia composta da una miriade di piccoli sensori puntiformi i quali convertono singolarmente la luminosità rilevata.
Per riprese su più colori il procedimento non cambia ma viene esteso tramite l'utilizzo di un particolare filtro, il filtro di Bayer, che garantisce un'acquisizione estesa a tutto lo spettro visibile.
Al termine di ciascuno dei procedimenti intervengono algoritmi di interpolazione che ricostruiscono l'immagine appena acquisita.

Sensori 2.5D (Depth)

Questa classe di sensori coinvolge tutti quei dispositivi che integrano durante il processo di acquisizione, il **concetto di profondità**.Il motivo della sigla 2.5D è dovuto al fatto che la nuvola di punti 3D generata da questi sensori non consente di guardare dietro gli oggetti, pertanto, gli oggetti in primo piano occludono oggetti di sfondo. La gamma di approcci all'implementazione è ampia e non mancano innovazioni in questo campo (vedi *Project soli*, una nuova tecnologia che utilizza un radar in miniatura per rilevare interazioni gestuali senza contatto). L'esempio standard di sensore 2.5D sono le **depth camera**, cioè sistemi che risolvono la distanza in base a dimensioni fisiche invarianti, come la velocità della luce o del suono, misurando il tempo di volo di un segnale tra la fotocamera e il soggetto per ogni punto dell'immagine. Fra i modelli principali di Depth camera possiamo trovare i sensori a **luce strutturata** (vedi Kinect 1.0) e i sensori **time of flight** (vedi Kinect 2.0) (fig 3.4). Maggiori dettagli sul funzionamento di queste due tecnologie verrano forniti nella sezione 4.1.2 dedicata al confronto delle versioni Kinect.

Sensori 3D

Generalmente non esistono dispositivi in grado di acquisire informazioni in 3 dimensioni, questo per il semplice fatto che i sensori hanno un campo di visione limitato e non potranno mai, singolarmente, catturare informazioni a tuttotondo.
Esistono però alcune implementazioni tecnologiche che, facendo uso di più sensori collegati assieme, permettono di ottenere una visione a 360 gradi. Fra queste la più famosa è sicuramente il **motion capture**. Il motion capture (a volte indicato come mo-cap o mocap) è il processo di registrazione del movimento di oggetti o persone. È utilizzato in applicazioni militari, di intrattenimento, sportive, mediche e per la convalida della visione artificiale. Nello sviluppo di film e videogiochi, si riferisce alla registrazione di azioni di attori umani e all'utilizzo di tali informazioni per animare modelli di personaggi digitali in animazioni computerizzate 2D o 3D.

Figura 3.4: La Time of Flight Camera è una depth camera dove gli inpulsi inviati sono di carattere luminoso. Camere di questo tipo sono molto più performanti della controparte ad ultrasuoni ma più costose.

3.3 Classificazione di Serie Temporali

Definite le motivazioni che hanno portato il riconoscimento di gesti ad essere uno dei campi più pervasivi dei nostri giorni e data una descrizione generale dei sensori più comuni; è giunto il momento di entrare nel vivo del discorso. Inizieremo la nostra esplorazione delle tipologie di riconoscimento più comuni imponendo, innanzitutto, una piccola restrizione al dominio di ricerca scelto. Dato l'obiettivo di questa tesi infatti, ci concentreremo principalmente sul concetto di **classificazione di serie temporali**, uno dei metodi più studiati e utilizzati nel campo del riconoscimento gesti con Kinect.

3.3.1 Template e Classificazione

Nell'ambito della computer science un processo di classificazione è un'attività che consiste nell' **assegnare una classe a un determinato pattern**. Definiremo con **pattern** un qualsiasi **dato, semplice o strutturato** che siamo intenzionati a classificare. Un pattern può essere rappresentato da un'immagine, un suono, numeri e cosi via. Nel caso in cui sia presente un legame tra patterns diversi a livello spaziale o temporale, definiamo la successione di questi come **sequenza**. Circoscritta la natura di un pattern a un vettore n dimensionale $\vec{x} = (x_1, x_2, ..., x_n)$ chiamato **feature vector**, siamo finalmente

pronti a definire formalmente il concetto di sequenza temporale o *Time Serie*:

$$X_k = (\vec{x_1}, \vec{x_2}, ..., \vec{x_k}) \quad X_{n,k} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,k} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,k} \end{pmatrix} \tag{3.1}$$

Una serie temporale X di lunghezza k non è altro che una **successione di feature vectors n-dimensionali nel tempo** e può essere rappresentata come una matrice di k colonne, (il numero dei vettori di feature considerati) e n righe (la dimensione di ogni vettore). Per esempio se volessimo classificare i vettori 3D ($n = 3$) generati in ogni secondo da un accelerometro con frequenza di aggiornamento a 10Hz ($k = 10$), dovremmo considerare serie temporali di questo tipo: $X_{10} = (\vec{x_1}, \vec{x_2}, ..., \vec{x_{10}})$

$$X_{3,10} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,10} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,10} \\ \vdots & \vdots & \ddots & \vdots \\ x_{3,1} & x_{3,2} & \cdots & x_{3,10} \end{pmatrix}$$

Dove $x_{i,j}$ rappresenta una singola componente nella dimensione i (altezza, larghezza, profondità) al campionamento j.
Il processo di classificazione/riconoscimento può essere applicato generalmente anche a sequenze temporali; il concetto è il medesimo definito sui pattern, l'unica differenza risiede nella tipologia degli algoritmi dedicati.

Metodi di Classificazione: Come mostrato nella figura 3.5, gli approcci utilizzati nel riconoscimento gesti di possono essere divisi approssimativamente in due macro categorie:

- **Approccio Basato su Regole:** E' l'approccio più naive; in questo caso un gesto o un'attività viene riconosciuto in base a un insieme di **regole definite manualmente**.
- **Approccio Basato su Template:** In questo approccio, la classificazione di un gesto sconosciuto viene eseguita confrontando automaticamente un template pre-acquisito tramite tecniche di **pattern recognition**. L'approccio su template può essere suddiviso in 2 sottocategorie:
  - **Matching Diretto:** Il movimento del modello viene confrontato direttamente con il movimento sconosciuto da classificare. L'algoritmo più dominante utilizzato per la corrispondenza diretta è il **dynamic time warping** (DTW)

– **Matching Basato sul Modello:** In questo approccio, viene utilizzato un **modello cinematico** o un **modello statistico**. Il template viene utilizzato per determinare i parametri del modello. Quindi, il modello adattato viene utilizzato per classificare un gesto sconosciuto. Il metodo utilizzato per addestrare il modello varia in modo significativo, da quelli semplici come l'ottenimento di angoli articolari medi, a sofisticati metodi di machine learning come gli **Hidden Markov Models** e **Reti Neurali Artificiali**.

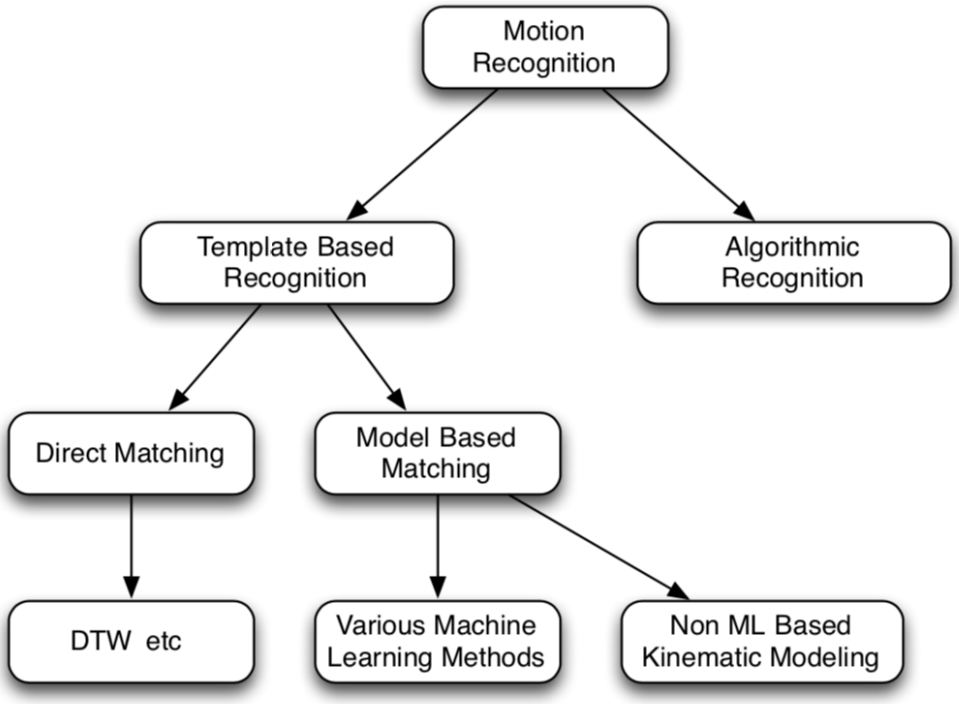


Figura 3.5: Tassonomia per gli approcci più comuni al riconoscimento

3.3.2 Approccio Basato su Regole

Il riconoscimento rule-based rappresenta una casistica a parte nel mondo dei riconoscitori. Il suo funzionamento dipende intrinsecamente da come le gesture vengono definite; in particolare deve essere possibile descrivere ogni singola gesture attraverso un **set di regole prestabilite e direttamente implementabili**. Per ogni gesture, inoltre, devono essere forniti un **congruo numero di parametri** in modo da garantire un discreto margine di adattamento. Per esempio nella codifica di uno *Swipe laterale* dovrebbe essere

possibile gestire il punto di inizio dello swipe e una percentuale di errore della traettoria eseguita.
Contrariamente a quello che si possa pensare, un approccio di questo tipo presenta alcuni benefici rispetto alla controparte **tempalte-base**; quest'ultima infatti necessita di **enormi quantitativi di dati** per garantire un buon livello di riconoscimento e spesso il **costo computazionale è molto elevato**.

Campi di Applicazione

Un approccio basato su regole è molto diffuso nel gaming e nelle applicazioni sanitarie poichè i gesti sono solitamente ben definiti, relativamente semplici e di natura ripetitiva. Ogni gesture generalmente possiede una **posa specifica** per delineare il punto di inizio e di fine. In alcuni casi i difetti di questo approccio sono utilizzati a vantaggio dell'applicazione stessa. Per esempio, nel caso di un applicazione dedicata alla supervisione di esercizi riabilitativi è importante che la gesture segua uno schema ben definito e venga quindi riconosciuta solamente quando l'esercizio è eseguito correttamente.

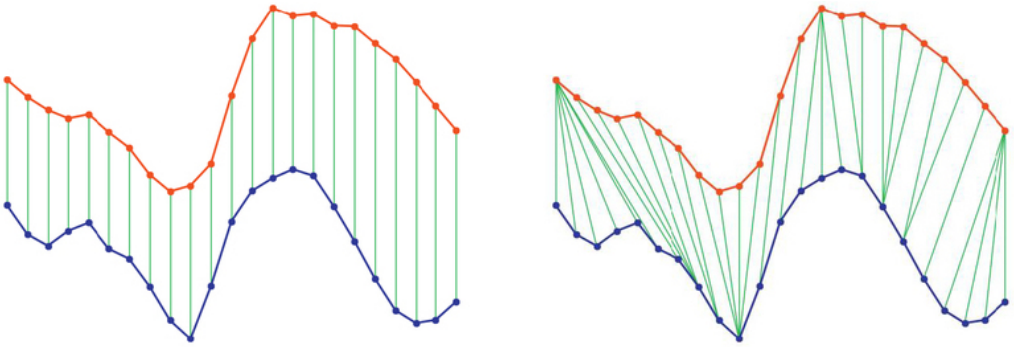
Specifica delle regole e definizione dell'algoritmo

La specifica dell'algoritmo di riconoscimento gesture non nasce improvvisamente ma deriva da uno studio attento del movimento delle parti del corpo in gioco. Solitamente per definire una specifica per il movimento robusta e **invariante alle dimensioni di spazio e tempo** (punto di inizio e fine gesture e velocità di esecuzione), è necessario utilizzare un metodo di tracciamento il più consistente possibile. Di conseguenza, la maggior parte degli studi presenti in letteratura si concentra su un piccolo insieme di regole che sono prevalentemente espresse in termini di **angoli articolari** vedi [10] e [11].
In alcuni casi l'approccio algoritmico è stato tramutato in un approccio interamente dichiarativo. Questo è il caso di [12] dove tutte le regole sono scritte in un file di testo e vengono analizzate con una grammatica **LALR-1**.

3.3.3 Dynamic Time Warping

Avanzando lungo il nostro percorso attraverso i più popolari metodi di riconoscimento, passiamo ora a studiare l'altra categoria più importante di metodi, ossia gli approcci **Template-Based** (basati su template). L'algoritmo principe di questa tipologia di metodi (per quanto riguarda il **Matching Diretto**), impiegato anche nella realizzazione del nostro framework, è il **Dynamic Time Warping** o DTW.

Il Dynamic Time Warping è un algoritmo utilizzato per calcolare la similarità (distanza) tra due serie temporali indipendentemente dalla lunghezza di ciascuna serie. Intuitivamente, le sequenze sono deformate in modo non lineare per adattarsi l'una all'altra e trovare cosi un allineamento globale ottimo 3.6. Originariamente sviluppato per il riconoscimento vocale automatico, nel corso degli anni il suo utilizzo si espanso in tantissimi settori come: la robotica, l'elaborazione di immagini, la finanza, l'elaborazione musicale ...



(a) Distanza euclidea (b) Distanza Dynami Time Warping

Figura 3.6: Due metodi utilizzati per calcolare la distanza fra due serie temporali. La distanza euclidea opera un semplice confronto 11 mentre il DTW cerca di allineare attivamente le sequenze.

Definizione Formale

Diamo ora una definizione più formale per l'algoritmo di DTW. Partendo da due sequenze $\mathbf{X}$ e $\mathbf{Y}$ temporali monodimensionali di lunghezza $n \in \mathbb{N}$ e $m \in \mathbb{N}$:

$$\begin{aligned} \mathbf{X} &= (x_1, x_2, ..., x_n)^T \\ \mathbf{Y} &= (x_1, x_2, ..., x_m)^T \end{aligned} \tag{3.2}$$

costruiamo un **warping path** (cammino di copertura) $\mathbf{W} = (w_1, w_2, ..., w_L)$ di lunghezza L, definito in questo modo

$$w_k = (x_i, y_j) \quad con\ i \in [1, ..., n],\ j \in [1, ..., m],\ k \in [1, ..., L] \tag{3.3}$$

tale per cui

$$max(n, m) \leq L < |n + m| \tag{3.4}$$

Un certo numero di vincoli sono posti sulla costruzione del warping path

- **Condizione al contorno:** Il cammino deve cominciare nel punto iniziale di entrambe le sequenze e terminare agli estremi di esse.
  Quindi: $w_0 = (1, 1)$ e $w_L = (n, m)$

- **Condizione di monotonia:** Il cammino deve esibire un comportamento monotono crescente.
  Dunque, dato $\mathbf{W}$ di lunghezza L, tale che $w_k = (i_k, j_k)$, abbiamo che:
  $i_1 \leq i_2 \leq \cdots \leq i_k \leq \cdots \leq i_L$
  $j_1 \leq j_2 \leq \cdots \leq j_k \leq \cdots \leq j_L$

- **Condizione di continuità:** Il cammino deve essere continuo.
  Dato $\mathbf{W}$, tale che $w_k = (i_k, j_k)$ deve valere questa condizione:

$$w_k - w_{k-1} \in \{(0, 1), (1, 1), (1, 0)\}$$

Questi vincoli, che inizialmente possono apparire un pò eccessivi, impongono semplicemente che il nostro cammino segua un percorso coerente con la natura delle serie temporali. Per comprendere al meglio il concetto di warping path possiamo immaginare di dover percorrere con il dito una scacchiera, partendo dal punto $(A, 1)$ *arrivando al punto* $(H, 8)$*, con l'unica condizione di non tornare indietro [13].*

Come è facile vedere, esiste un numero esponenziale di cammini che soddisfano le condizioni di cui sopra. Tuttavia, lo scopo di DTW è quello di trovare un cammino $\mathbf{W}$ tale per cui il suo costo (inteso come distanza fra le serie), definito come

$$C_w(X, Y) = \sum_{k=1}^{L} DIST(x_{k_i}, w_{k_j}) \tag{3.5}$$

sia il minimo tra tutti i cammini possibili. Indicando con $\mathbf{W}^\star$ il cammino ottimo, abbiamo che

$$DTW(X, Y) = C_{\mathbf{W}^\star}(X, Y) = min\{C_{\mathbf{W}}(X, Y)\} \tag{3.6}$$

Notiamo subito che nell'equazione (3.5) è presente una funzione di distanza; essa è fondamentale per l'algoritmo DTW e per questo motivo la tratteremo successivamente in una sezione a parte.

Calcolo della Distance Matrix

Come abbiamo appena visto, il DTW ci permette di confrontare due serie temporali che non sono esattamente le stesse ma che sono comunque molto

simili fra loro.
In particolare le basi dell'algoritmo le troviamo nella computazione di una particolare matrice n x m, $C_{n,m}$, che chiamiamo **distance matrix** (che abbiamo visto definita come matrice dei costi). I vincoli posti precedentemente ci aiutano al calcolo di tale matrice, in particolare, data:

$$C_{n,m} = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,n} \end{pmatrix} \tag{3.7}$$

Il calcolo del valore di ogni cella $C_{(i,j)}$ avviene secondo questo schema ricorsivo:

$$\begin{gathered} C_{(i,j)} = DIST(x_i, y_j) + min\{C_{(i-1,j)}, C_{(i-1,j-1)}, C_{(i,j-1)}\} \\ \text{Con} \quad 1 \leq i \leq n \quad \wedge \quad 1 \leq j \leq m \end{gathered} \tag{3.8}$$

Data la natura ricorsiva del calcolo è necessario porre alcuni casi base:

$$C_{(0,0)} = 0 \quad C_{(i,0)} = +\infty \quad C_{(0,j)} = +\infty$$

Cosi facendo possiamo calcolare la distanza DTW come:
$\mathbf{DTWDIST}(\mathbf{X}, \mathbf{Y}) = C_{(n,m)}$ e poi sfruttare la ricorsione per completare la computazione.

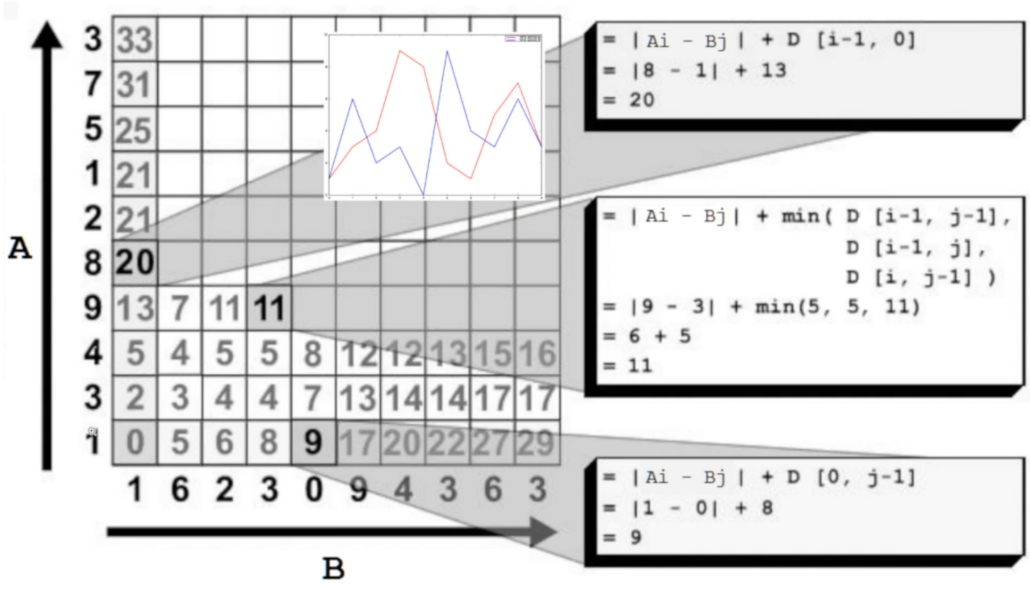


Figura 3.7: Calcolo della distanza DTW, algoritmo

Osservazioni Si possono fare alcune osservazioni sulla funzione DTW. Innanzitutto si può notare come la distanza DTW sia **ben definita** anche se possono esserci diversi cammini con un costo totale minimo. In secondo luogo, è facile vedere che la distanza DTW è **simmetrica** nel caso in cui la misura del costo locale C sia simmetrica. Tuttavia la distanza DTW è in generale non definita positiva anche se ciò vale per C.

Costo computazionale Per determinare il cammino ottimale $\mathbf{W}^\star$, fondamentale per l'algoritmo, è necessario testare ogni possibile path tra **X** e **Y**. E' evidente come tale procedura conduca ad una **complessità computazionale** che è **esponenziale** su n e m. Per questo motivo, per il calcolo della distance matrix e quindi del cammino ottimo, è stato utilizzato un algoritmo di complessità **O (nm)** basato sulla *programmazione dinamica* e impostato sui vincoli del cammino stesso. Esistono ulteriori ottimizzazioni ma queste verranno discusse in un secondo momento.

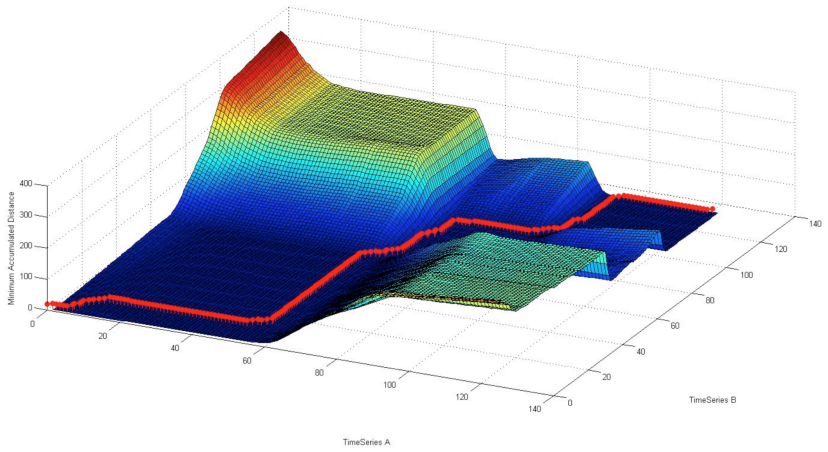


Figura 3.8: Visualizzazione 3D della matrice dei costi. In rosso possiamo vedere il cammino di copertura ottimo $\mathbf{W}^\star$. E' interessante notare come il path rimanga contenuto entro un certo limite centrale

La funzione Distanza

Nel caso del di Dynamic Time Warping abbiamo scelto, a scopo esplicativo, serie temporali monodimensionali. Nel mondo reale questa condizione risulta essere molto restrittiva, infatti come abbiamo avuto modo di vedere, è molto semplice trovarsi nella situazione di dover gestire feature vector con 3 o più dimensioni. Viene naturale quindi chiedersi se l'algoritmo appena definito possa

essere esteso a questo tipo di serie temporali. La risposta a questa domanda risiede in una particolare funzione, comune a molti algoritmi di matching[7], la **funzione distanza DIST(X, Y)**. La funzione distanza nel caso del DTW ha lo scopo di calcolare, appunto, la distanza fra i due valori assunti dalle serie temporali. Il calcolo può avvenire in modi diversi a seconda delle metriche utilizzate [4]:

Distanza Euclidea: Storicamente, la distanza euclidea, (famosa per essere stata concepita da Euclide) fornisce la migliore misura diretta fra due punti in più dimensioni. Considerati due vettori n-dimensionali $v = (v_1, v_2, \ldots, v_n)$ e $w = (W_1, w_2, \ldots, w_n)$, la distanza euclidea:

$$D_{Euclidea} = \sqrt{(v_1 - w_1)^2 + (v_2 - w_2)^2 + \cdots + (v_n - w_n)^2} \tag{3.9}$$

è la forma più semplice di misura per le distanze utilizzata in numero vastissimo di applicazioni.

Distanza di Manhattan: Conosciuta anche con il nome *City Block Distance*, questa metrica presuppone che per passare da un vettore all'altro si debba obbligatoriamente viaggiare lungo le linee della griglia evitando mosse diagonali (fig 3.9) . La distanza di Manhattan è definita in questo modo:

$$D_{Manhattan} = |v_1 - w_1| + |v_2 - w_2| + \cdots + |v_n - w_n| \tag{3.10}$$

Distanza di Chebyshev: Concepita da Pafnuty Chebyshev è anche nota come *Chessboard Distance* poiché nel gioco degli scacchi, il numero minimo di mosse necessarie da un re per passare da una casella ad un'altra è uguale alla distanza Chebyshev tra i centri dei quadrati.
Se i quadrati hanno una lunghezza laterale uno allora la distanza di Chebyshev si definisce in questo modo:

$$D_{Chebyshev} = max\{|v_1 - w_1| + |v_2 - w_2| + \cdots + |v_n - w_n|\} \tag{3.11}$$

[7]L'affinità fra due serie temporali può essere pensata come una distanza nello spazio multidimensionale, di conseguenza la misurazione di queste distanze ha un ruolo preminente in qualsiasi tipo di classificazione.

Distanza di Minkowski: La distanza di Minkowski è una metrica generale che possiede inoltre le metriche di Euclide, Manhattan e Chebyshev come casi speciali. La definizione è data dalla seguente espressione:

$$D_{Minkowski} = \left(|v_1 - w_1|^P + |v_2 - w_2|^P + \cdots + |v_n - w_n|^P\right)^{\frac{1}{P}} \tag{3.12}$$

È interessante notare come le metriche Manhattan e Chebyshev possano essere calcolate più velocemente della metrica euclidea, di conseguenza, le applicazioni basate sulle performance possono fare uso di Chebyshev quando la precisione non è di fondamentale importanza.

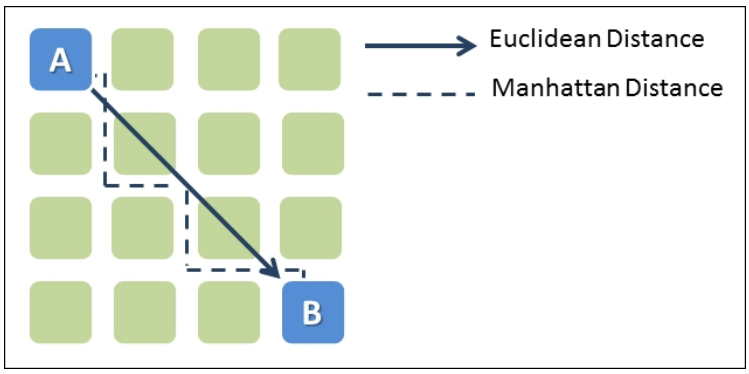


Figura 3.9: Differenza fra distanza euclidea e distanza city-block

Ottimizzazioni

Nel corso degli anni il dynamic time warping è stato oggetto di molte ricerche con lo scopo di migliorarne sempre più le prestazioni. Difatti, come sappiamo l'algoritmo nella sua versione originale ha un costo computazionale molto alto, **O (nm)** e in alcuni contesti di utilizzo, come il riconscimento realtime, risulta inefficace. Le principali tecniche di ottimizzazione verranno discusse di seguito:

DTW Pesato Per favorire la direzione diagonale, orizzontale o verticale nell'allineamento, è possibile introdurre un **vettore di peso** aggiuntivo $w = (w_d, w_h, w_v) \in \mathbb{R}^3$, ottenendo la ricorsione:

$$C(n,m) = min \begin{cases} D(n-1, m-1) + w_d * C(x_n, y_m) \\ D(n-1, m) + w_h * C(x_n, y_m) \\ D(n, m-1) + w_v * C(x_n, y_m) \end{cases} \tag{3.13}$$

Questa formula non è altro che (3.8) calcolata, in modo ricorsivo, partendo dal punto (n, m) e moltiplicando ogni volta la distanza calcolata per le componenti di w.
Nel caso ugualmente ponderato $w = (1, 1, 1)$ il DTW si riduce alla sua forma classica. Si noti che per $(w_d, w_h, w_v) = (1, 1, 1)$ si ha una preferenza della direzione di allineamento diagonale, poiché un passo diagonale (costo di una cella) corrisponde alla combinazione di uno orizzontale e uno verticale (costo di due celle). Per controbilanciare questa preferenza, spesso si sceglie $(w_d, w_h, w_v) = (2, 1, 1)$.

Approssimazioni Una strategia efficace per accelerare i calcoli DTW si basa sull'idea di eseguire l'allineamento su versioni ridotte delle sequenze $\mathbf{X}$ e $\mathbf{Y}$, riducendo così le lunghezze n e m delle due sequenze. Tale strategia è anche nota come **riduzione della dimensionalità** o astrazione dei dati.
Un approccio, in questo senso, costiste nel ridurre la velocità dei dati che permette di eseguire il **downsampling** degli stessi. Un'altra strategia è quella di approssimare le sequenze con qualche funzione a tratti lineare (o di qualsiasi altro tipo) e quindi eseguire il warping a livello di approssimazione, vedi [15, Piecewise DTW].
Un'importante limitazione di questo approccio, tuttavia, è che l'utente deve accuratamente specificare i livelli di approssimazione utilizzati nell'allineamento. Se l'utente sceglie un'impostazione troppo fine, l'incremento di velocità è trascurabile. Al contrario, se l'utente sceglie un'approssimazione troppo grossolana, per esempio, diminuendo la frequenza di campionamento delle sequenze $\mathbf{X}$ e $\mathbf{Y}$, il warping path risultante può diventare inaccurato o addirittura completamente inutile.

Vincoli Globali Un metodo comune di ottimizzazione consiste nell'imporre **condizioni globali sui path ammissibili**. Tali vincoli hanno una doppia efficacia: non solo accelerano il DTW ma prevengono anche gli **allineamenti patologici** controllando globalmente il processo di warping.
Più precisamente, sia $R \subset [1 : n]x[1 : m]$ **la regione di restrizione globale**, Un path ottimo rispetto a R è un percorso di deformazione che scorre interamente al'interno dell'insieme. Due regioni di vincoli globali ben note sono la **Sakoe-Chiba Band** [16] e il **parallelogramma Itakura**.
La banda Sakoe-Chiba corre lungo la diagonale principale e ha una larghezza fissa (orizzontale e verticale) $T \in \mathbb{N}$. Questo vincolo implica che un elemento x_i (tra gli n possibili) può essere allineato solo a uno degli m, y_m con $m \in \left[\frac{m-T}{n-T} * (i - T), \frac{m-T}{n-T} * i + T\right] \cap [1 : M]$ Il parallelogramma Itakura descrive una regione che vincola la pendenza del warping path. Più precisamente, per

un $S \in R > 1$ fissato, il parallelogramma Itakura è costituito da tutte le celle che vengono attraversate da un cammino con una pendenza tra i valori $1/S$ e S.

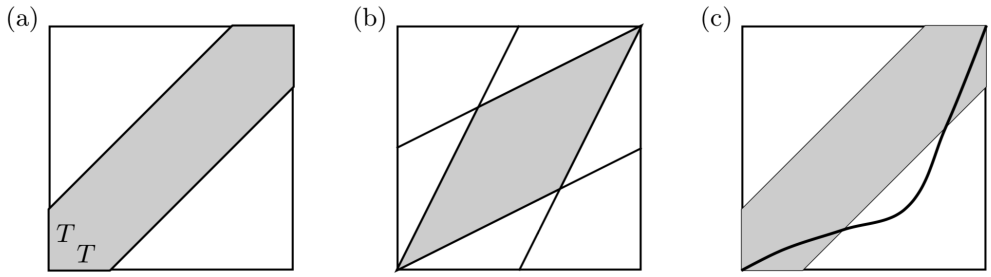


Figura 3.10: (a) Sakoe-Chiba banda di larghezza T. (b) Parallelogramma di Itakura con $S = 2$. (c) Warping path ottimo $\mathbf{W}^\star$ che non rientra nella regione di restrizione.

Classificazione con DTW

Come abbiamo visto, il Dynamic Time Warping è un algoritmo davvero interessante impiegato in un gran numero di domini applicativi. Data la sua grande versatilità spesso si tende a considerare il DTW come un classificatore. In realtà questo concetto è sbagliato; il **DTW, infatti, è un algoritmo che misura solamente la distanza fra sequenze**, non le classifica. Per effettuare un processo di classificazione di serie temporali bisogna definire un metodo con il quale classificare i template, un processo molto più complicato del calcolare una distanza.
Come vedremo nei capitoli dedicati al progetto, un possibile approccio potrebbe essere quello di eseguire l'algoritmo del DTW tra la sequenza da classificare e tutti i template del dataset, al fine di ottenere una lista di distanze in ordine crescente (a ogni distanza è associata la classe corrispondente al template originale). Il processo di classificazione si appoggerà poi all'algoritmo **K-Nearest Neighbors (K-NN)**, il quale data in input la lista di distanze fornirà in output la classe più probabile.
E' importante sottolineare che non tutte le distanze raccolte verranno passate al classificatore; bisognerà infatti porre una **soglia** che permetterà di definire se una sequenza è classificabile oppure no. Così facendo costruiremo un modello **non-gesture** molto robusto capace di escludere tutte le sequenze che non rappresentano un gesto.

Focus sull'algoritmo K-Nearest Neighbors Data una funzione distanza $DIST$ nello spazio multidimensionale il classificatore nearest neighbor (il più

vicino dei vicini), classifica un pattern x con la stessa classe dell'elemento x' ad esso più vicino nel training set TS.

$$DIST(x, x') = min\{DIST(x, x_i)\} \qquad \forall x_i \in TS \tag{3.14}$$

La regola k-Nearest Neighbors (k-NN) determina i k elementi più vicini al pattern x da classificare. Ogni pattern tra i k vicini vota la classe al quale appartiene; il pattern x viene dunque assegnato alla classe che ha ottenuto il maggior numero di voti.

3.3.4 Matching Basato su Modello

Seconda tipologia di algoritmi template-based, gli algoritmi di Matching basati su modello, rappresentano l'ultima evoluzione nel campo del riconoscimento di gesture. In questo approccio, infatti, al fine di garantire un metodo robusto di classificazione viene utilizzato un **modello astratto** il quale, dopo essere stato allenato attraverso una serie di pattern, a fronte di input sconosciuti, è in grado di eseguire il riconoscimento.

Classificazione La classificazione di serie temporali basata sull'apprendimento automatico si basa generalmente su uno o più modelli statistici sofisticati: **Hidden Markov (HMM)**, **Artifical Neural Network (ANNs)**, **Support Vector Machine (SVM)**. La maggior parte di questi modelli è costituita da un gran numero di parametri, che devono essere determinati in fase di addestramento. Tali parametri sono basati su dati di movimento pre-etichettati (inclusi sia i dati per il gesto da riconoscere, sia altri dati di movimento noti non specifici). In generale, più ampio è il set di feature usato per la classificazione e più grande sarà il dataset per il training.

Regressione Tuttavia, il riconoscimento del movimento può essere formulato in maniera diversa da un problema di classificazione adottando un approccio a **regressione**; in questo caso, il modello addestrato è indicato come **regressore**. Differentemente da un classificatore, che emette un valore discreto (una classe), l'output di un regressore è solitamente un valore continuo all'interno di un intervallo predefinito. Per utilizzare, quindi, un regressore come classificatore, è possibile scegliere una soglia in modo che quando l'output del regressore supera tale soglia, è possibile determinare la classe del template in questione. L'uso di un regressore ha il pregio di fornire non solo la classificazione, ma anche **informazioni sullo stato del gesto**, feature molto importante per un utilizzo in contesti applicativi.

Capitolo 4

Riconoscimento di Gesture con Kinect

Lanciato nel 2010 dalla Microsoft, Kinect è uno dei controller di gioco più popolari di sempre; lo dimostrano le 24 milioni di unità vendute nei primi 3 anni di commercializzazione [17].
Kinect in particolare consente agli utenti di interagire in modo naturale con un computer o una console di gioco con gesti e / o comandi vocali senza l'ausilio di fili o particolari *markers* per il tracciamento.
Fin dalla sua nascita, ha portato una rivoluzione nel campo della **NUI**[1] e del gioco a mani libere distinguendosi dai competitor, primo fra tutti Nintendo Wii, per la sua robustezza e affidabilità. Non c'è dunque da meravigliarsi che Kinect abbia continuato a crescere e sia diventato il dispositivo elettronico più venduto al mondo.
Sebbene sia considerato un controller per l'ambito videoludico (Xbox 360), l'applicabilità di Kinect va ben oltre il dominio dei giochi e si apre a un numero esponenziali di donimi applicativi.
In questo capitolo indagheremo in pronfondità il sensore kinect: in una prima sezione introdurremo le principali caratteristiche delle due versioni di Kinect (attualmente commercializzate da Microsoft), con un focus specifico all'aspetto tecnologico e sensoristico. Nella seconda sezione verrà fornito una breve riassunto sulle possibili applicazioni di questa tecnologia. Infine nella terza sezione esploreremo Kinect da un lato prettamente software, introducendo l' *SDK* fornito da Microsoft e le API di sviluppo.

[1] Sta per "Natural User Interface". Un NUI è un tipo di interfaccia utente progettata per essere il più naturale possibile per l'utente. L'obiettivo di una NUI è creare un'interazione senza interruzioni tra l'uomo e la macchina, facendo scomparire l'interfaccia stessa.

4.1 Design e Sensoristica

Kinect è dispositivo destinato prettamente a un utilizzo nell'ambito del **sensing ambientale** e del **tracciamento umano**. Esteticamente l'apparato si presenta come molto compatto, il design è elegante e i fori per i sensori gli donano un aspetto quasi futuristico. Per il momento esistono solamente due versioni del Sensore: la versione 1.0, **Kinect**, prodotta e commercializzata nel periodo 2010-2013 e la versione 2.0, **Kinect One**, prodotta per la prima volta nel 2013 e tutt'ora commercializzata asssieme all'omonima XBox. Quanto alle specifiche, Kinect si presenta come un dispositivo ultra-accessoriato e adatto a tutti i possibili contesti di utilizzo. Unico neo è la sua predisposizione a interfaccia di input statica (è presente una connessione cablata) che non consente quindi particolari utilizzi in portabilità, come succede invece per alcune interfacce di gioco di nuova generazione.

4.1.1 Sensori

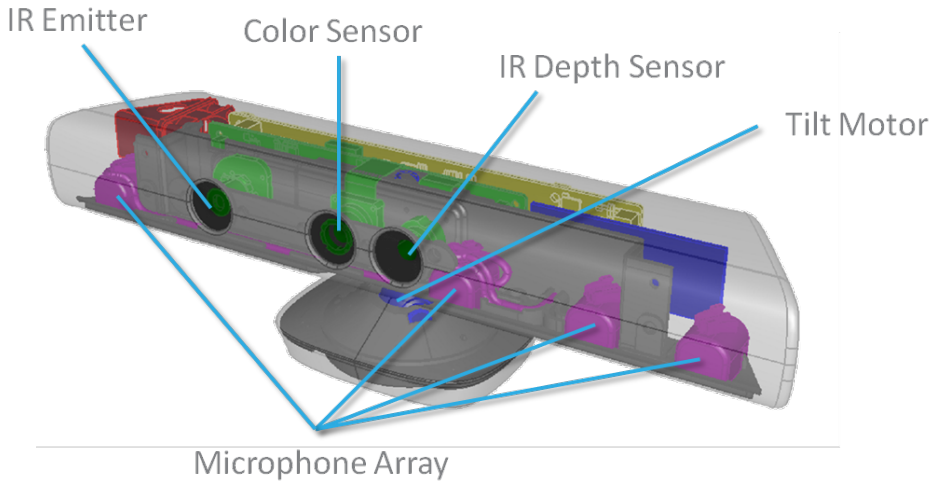


Figura 4.1: Posizione dei sensori su Kinect 1.0

Entrando in ambito sensoristica Kinect è dotato di un insieme di accessori molto vasto (fig 4.1); montati all'interno del sensore infatti troviamo:

- **Camera RGB**
- **Emettiore a infrarossi**
- **Sensore di profondità a infrarossi**

- **Array di microfoni**
- **Motore di inclinazione**
- **Led di stato**

Ognuno di questi componenti gioca un ruolo fondamentale nella fase di tracking e riconoscimento umano. Andiamo ora a vedere in dettaglio le tecnologie utilizzate, analizzando in particolare i metodi messi in campo dai progettisti per garantire a Kinect un'interazione cosi variegata.

Camera RGB

Abbiamo già visto nello scorso capitolo i principi di funzionamento di una camera di questo tipo ma non ci siamo mai addentrati in questioni più tecniche. Come sappiamo la funzione di una camera RGB consiste nel rilevare i colori rosso, blu e verde dalla sorgente, restituendo un flusso di dati rappresentato da una successione di fotogrammi di immagini fisse. Kinect è stato progettato per supportare nativamente **2 tipologie di stream RGB**: una prima con risoluzione di **640 x 480**, caratterizzata da una velocità di campionamento **a 30 fotogrammi al secondo** (FPS) e una seconda con **risoluzione 1280 x 960 a 12 FPS**. Il valore dei fotogrammi al secondo può variare in base alla risoluzione utilizzata per la cornice dell'immagine. Il **range di visione** per le telecamere Kinect è di **43 gradi in verticale** di **57 gradi in orizzontale**.

Emettitore a Infrarossi e Depth Sensor

I sensori di profondità Kinect sono costituiti da un **emettitore IR** e un **sensore di profondità IR**; entrambi lavorano insieme per far si che il tutto possa funzionare.
L'emettitore IR può sembrare una fotocamera dall'esterno, ma **è un proiettore IR** che emette costantemente luce infrarossa in un "punto pseudo-casuale" su tutto ciò che si trova di fronte. Questi punti sono normalmente invisibili per noi, ma **è possibile acquisire** le loro **informazioni** sulla profondità **utilizzando un sensore di profondità IR**.
La luce intermittente viene riflessa da diversi oggetti e il sensore di profondità IR non deve fare altro che calcolare la distanza tra il sensore e l'oggetto da cui è stato letto il punto IR. È interessante e allo stesso tempo divertente notare come questi punti non siano totalmente invisibili ma possano essere osservati da chiunque con l'utilizzo di occhiali a visione notturna (fig 4.2). Il flusso di dati di profondità supporta una risoluzione di 640 x 480 pixel, 320 x 240 pixel e 80 x 60 pixel e il raggio di visualizzazione del sensore rimane lo stesso della telecamera a colori.

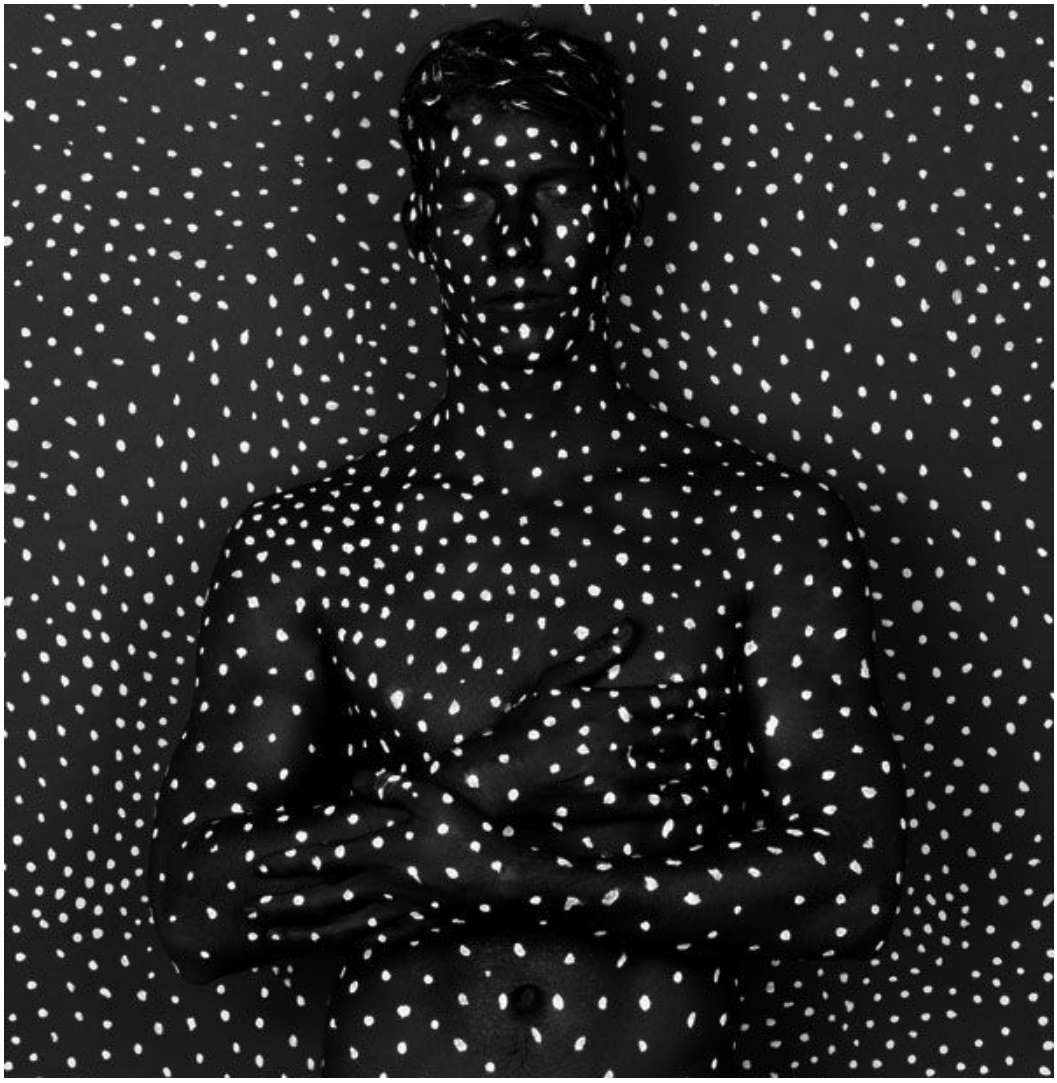


Figura 4.2: IL pattern IR visto attraverso una lente infrarossa

Array di microfoni

Il dispositivo Kinect offre un ottimo supporto per l'audio con l'aiuto di un **array di microfoni**. L'array è costituito da quattro unità di acquisizione diverse, posizionate in un ordine lineare (tre di questi sono distribuiti sul lato destro e l'altro è posizionato sul lato sinistro).
Lo scopo dell'array di microfoni non è solo quello di consentire al dispositivo Kinect di catturare il suono ma anche di localizzare la fonte dello stesso in un **panorama stereofonico**. Inoltre il loro utilizzo consente l'implementazione **algoritmi di soppressione del rumore**, che migliorano considerevolmente

il **riconoscimento della voce** in condizioni caotiche (minimizzando echi e fruscii).

Motore di inclinazione

Il basamento e il corpo del sensore sono collegati da un minuscolo motore (a tre ingranaggi) che ha lo scopo di inclinare il dispositivo ad angolazioni specifiche, mantenendo cosi un focus continuo sull'utente. **Il motore può essere inclinato** verticalmente **fino a 27 gradi**, il che significa che gli angoli di visione del sensore possono spaziare da -27 a +27 gradi.

4.1.2 Confronto fra versioni

Come abbiamo visto a inizio capitolo, Kinect possiede due versioni: Kinect e Kinect One. Kinect One è la versione più recente e porta grandi miglioramenti in tutti i settori, partendo dal design molto più curato e ricercato, fino alle specifiche nettamente migliori rispetto al modello precedente.
I camiamenti sono riassunti in questa tabella:

Feature	Kinect v1	Kinect v2
Color Camera	640 x 480 30FPS	1920 x 1080 30FPS
Depth Camera	320 x 240	512 x 424
Tecnologia Depth Cam	A triangolazione	Time of Flight
Distanza Massima	4.5m	4.5m
Distanza Minima	40cm	50cm
Campo Visivo Orizzontale	57'	70'
Campo visivo Verticale	43'	60'
Motore	SI	NO
N. Giunzioni Definite	20	26
N. di Skeleton	2	6
USB Standard	2.0	3.0
Os Supportato	Win7, Win8, Win10	Win8, Win10

Una tecnologia poco robusta

Come è facile notare, le differenze maggiori le troviamo nelle due camere montate da Kinect (RGB e Depth Camera). Se per quanto riguarda la camera RGB le uniche innovazioni si sono concentrate in un upgrade di risoluzione, per la Depth abbiamo visto un vero e proprio cambio di paradigma.
La tecnologia di rilevamento della profondità utilizzata in Kinect v1 è stata sviluppata da PrimeSense e consiste nel calcolare la profondità di ogni pixel utilizzando una semplice triangolazione. Normalmente, sono necessarie due

telecamere per facilitare il calcolo della triangolazione; in Kinect v1, invece, viene utilizzato un metodo a luce strutturata per abilitare l'uso di un singolo sensori a infrarossi (IR). Per questo motivo la fedeltà della misurazione della profondità è piuttosto bassa. Affinché il rilevamento della profondità funzioni perfettamente ci deve essere uno spazio tra due punti adiacenti abbastanza ampio da permette al sensore di distinguere profondità ma nel nostro caso i pixel sono sono troppo vicini e solo 1 ogni 20 è utilizzato per una misurazione reale della profondità (gli altri vengono interpolati).
Kinect v2 utilizzando una tecnolgia Time Of Flight non soffre di questo problema e garantisce quindi livelli di precisione di molto superiori.

4.2 Possibili Campi di Impiego

Microsoft Kinect è stato originariamente rilasciato esclusivamente per console di gioco e intrattenimento Xbox. Come sappiamo esso consente ai giocatori di giochi Xbox di controllare la console tramite gesti del corpo e comandi vocali senza utilizzare altre apparecchiature periferiche. Con l'introduzione di *Microsoft Kinect SDK* nel 2011, la tecnologia Kinect ha aperto una grande porta per lo sviluppo di altre applicazioni oltre i giochi Xbox.
La figura 4.3 mostra le principali categorie di applicazione di Kinect, che vanno dall'assistenza sanitaria, all'istruzione, alla vendita, alla formazione, al gioco, al controllo della robotica, all'interfaccia utente naturale, al riconoscimento della lingua dei segni e alla ricostruzione 3D. [17]

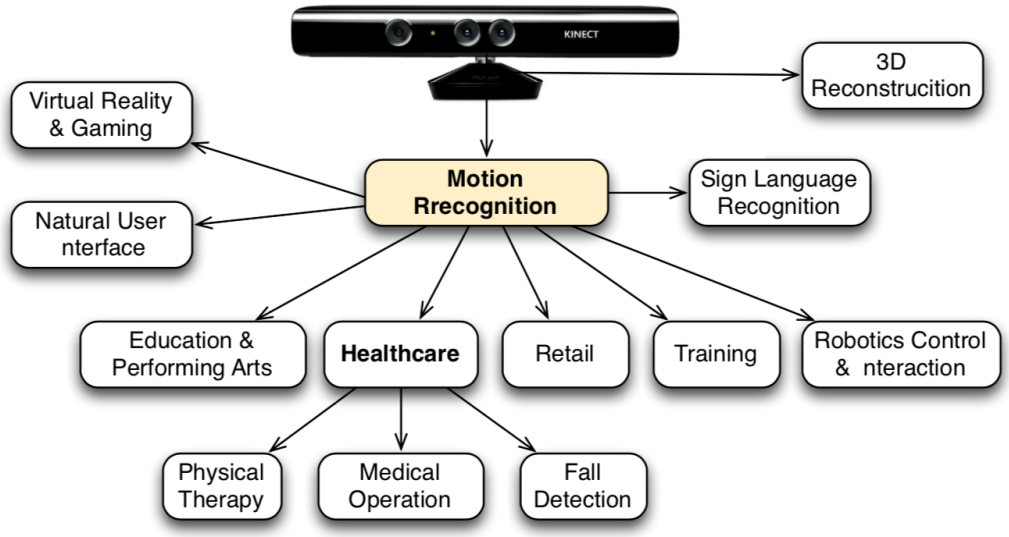


Figura 4.3: I principali campi di impiego per il sensore Kinect.

Come si può ben vedere, ad eccezione delle applicazioni di modellazione e ricostruzione 3D, praticamente tutte le altre applicazioni Kinect richiedono il riconoscimento del movimento e sfortunatamente Kinect non è dotato di librerie per il riconoscimento gesti (almeno per ora).

4.3 Kinect Development Kit

Microsoft nel corso degli anni ha sempre mantenuto una linea di business molto conservativa; precludendo la possibilità che ogni innovazione tecnologica, (sbocciata all'interno dell'azienda) venisse sfruttata in campi di ricerca esterni al colosso. Stranamente con Kinect è avvenuto esattamente l'opposto; sin da subito infatti i ricercatori di Bill Gates si sono messi all'opera con lo scopo di rilasciare un kit di sviluppo dedicato esclusivamente a Kinect, il **Kinect Development Kit**.
Una nota doverosa: *Premettendo che il progetto di tesi verrà svolto in un linguaggio differente (Java) da quello nativo per applicazioni in ambito Kinect (C# o C++), si ritiene di fondamentale importanza fornire al lettore una panoramica completa del software di sviluppo (se non altro per garantire una comprensione lineare di tutto il processo progettuale e implementativo). La libreria java verrà spiegata in seguito molto brevemente dato che rappresenta un porting per le API native.*

4.3.1 Kinect SDK

Kinect for Windows SDK è un **pacchetto software** dedicato allo sviluppo di applicazioni destinate esclusivamente a dispositivi Kinect. L'SDK include i **driver**, per garantire al sistema operativo (Windows 7,8,10) un'interazione a basso livello con il sensore, e le **API** per fornire un'interfaccia lato software ad alto livello. Nel complesso, l'SDK offre agli sviluppatori la possibilità di creare applicazioni proprietarie, utilizzando il codice gestito (C # e VB.NET), oppure applicazioni libere utilizzando il codice non gestito (C++). Inoltre possedendo un Kinect di seconda generazione, sarà possibile pubblicare i propri applicativi sul **Microsoft Marketplace**, potendo cosi monetizzare gli sforzi impiegati. Per quanto riguarda gli IDE di sviluppo, si possono utilizzare vari applicativi fra cui il famosissimo Visual Studio (2010 o versioni successive) oppure se si dispone di Unity, è possibile scaricare dei componenti aggiuntivi che garantiscono la compatibilità.

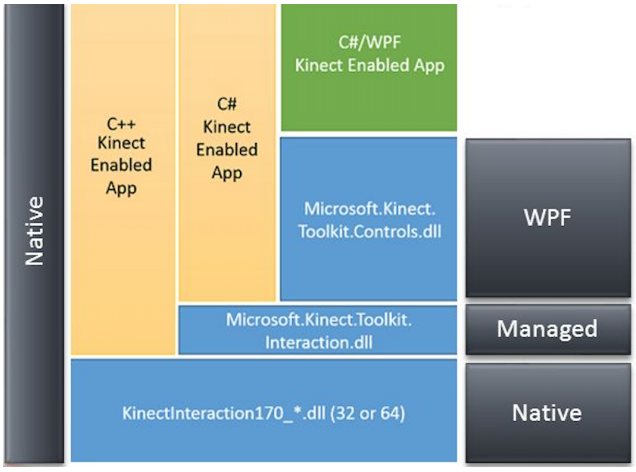


Figura 4.4: Kinect SDK una visione strutturale di insieme

Setting dell'ambiente di lavoro

Per poter sviluppare applicazioni Kinect è importante settare al meglio l'ambiente di lavoro. Il setting generico si suddivide in semplici passaggi, eseguiti i quali dovremmo giungere con un dispositivo funzionante e pronto a essere comandato. I passaggi da effettuare verranno elencati di seguito:

1. **Installazione SDK:** Riconoscere i requisiti di sistema e installare l'SDK adatto al nostro dispositivo. Nel caso ci trovassimo di fronte a un dispositivo kinect di prima generazione l'SDK avrà un'insieme di funzioni molto più limitato.
   Questa procedura installa il Runtime, i Driver, le API e i Toolkit necessari per la fase di testing; essenzialmente tutto quello di cui abbiamo bisogno.

2. **Testing audio e video:** I Toolkit installati durante la prima procedura ci permetteranno di effettuare dei semplici programmi-test su Kinect; se l'installazione è avvenuta correttamente, verranno mostrati i vari tipi di stream video. Un ulteriore test ci consentirà di testare l'aspetto audio.

3. **Preparare la configurazione di sviluppo:** Testata la corretta configurazione del dispositivo e del driver, è necessario assicurarsi che anche l'ambiente di sviluppo sia pronto. Per fare ciò dobbiamo creare un nuovo progetto in Visual Studio e agganciare il riferimento alla libreria *Microsoft.Kinect.dll*; fatto questo possiamo avviare Kinect direttamente dalle API importate.

Introduzione alle API per Kinect

Come abbiamo visto l'SDK fornisce gli strumenti e le API, sia nativi che gestiti, necessari per sviluppare applicazioni abilitate a Kinect per Microsoft Windows. Lo sviluppo di applicazioni abilitate a Kinect equivale essenzialmente allo sviluppo di altre applicazioni Windows, tranne che il Kinect SDK fornisce supporto per le funzionalità riferite al dispositivo stesso, incluse immagini a colori, immagini di profondità, input audio e dati scheletrici.
Per comprendere la funzionalità delle diverse API e conoscerne l'utilizzo, è sempre bene avere una visione chiara di come agiscono. Possiamo classificare le API nelle due seguenti categorie:

- **Controllo e accesso sensori:** Sono funzionalità riguardanti i sensori relativi a Kinect; il loro scopo è di **garantire il controllo e l'acquisizione di flussi di dati di varia tipologia**, per esempio: flusso di colore, flusso di dati a infrarossi, flusso di profondità e flusso Skeleton. Un **set di API** in questa categoria **parla direttamente con l'hardware del sensore**, mentre alcune **API sull'elaborazione** garantiscono la consistenza dei dati acquisiti dal sensore.

- **Controllo periferiche audio:** Sono le **interfacce che controllano** l'array di microfoni e il relativo flusso audio. Il loro scopo è gestire **l'interazione fra microfoni** implementando opportuni algoritmi di soppressione del suono e di stereo imaging; inoltre sono **responsabili** delle funzioni di **riconoscimento vocale**.

Nella prossima sezione andremo a fornire qualche esempio pratico sull'utilizzo delle API; definiremo prima di tutto le funzionalità base per il **controllo di Kinect**, poi la **gestione degli stream** concludendo con alcuni cenni sull'**ambito audio** e **riconoscimento vocale**.

4.3.2 API per il controllo e l'accesso ai sensori

Gran parte delle API per il controllo del sensore sono codificate all'interno della classe **KinectSensor**; dunque al fine di creare un'applicazione Kinect è necessario instanziare questa classe. Una volta creato un oggetto di tipo KinectSensor questo **costituirà l'unica runtime-pipeline completa per il sensore**, durante tutto il ciclo di vita dell'applicazione. Nel caso fossero presenti più dispositivi collegati, è presente un metodo per ottenere il device specifico e interagire con esso sempre tramite la solita interfaccia KinectSensor.

Startup e Setting del Device:

Per avviare il Kinect dobbiamo richiamare il metodo **sensor.start()**. Prima di intraprendere qualsiasi altra azione, il metodo sensor.Start() controlla innanzitutto lo stato del sensore; se il sensore è connesso, il metodo procede all'inizializzazione settando, prima i vari parametri relativi al dispositivo e poi aprendo i vari flussi di dati (colore, profondità, skeleton) con i loro valori predefiniti. Bisogna sottolineare il fatto che Kinect ponga due diversi tipi di **inizializzazione**: **senza parametri**, e **con parametri**. Nel primo caso il Kinect viene messo in esecuzione ma gli stream non sono abilitati, bisogna dunque specificare quali stream attivare. Nel secondo caso gli stream da abilitare vengono definiti prima di chiamare il metodo start.

```
if (KinectSensor.KinectSensors.Count > 0) {
   this.sensor = KinectSensor.KinectSensors[0]; //Primo presente
   this.sensor.StartSensor(); //Avvio Kinect in modalita standard
   this.sensor.ColorStream.Enable(); //Abilito gli stream
   this.sensor.DepthStream.Enable();
   this.sensor.SkeletonStream.Enable();
   this.sensor.Stop(); //Fermo il sensore
}
```

Listato 1: Inizializzazione sensore

4.3.3 Il concetto di Stream

Uno Stream di immagini non è altro che una **successione di fotogrammi** di immagini fisse. Kinect può fornire questo flusso generalmente a una velocità che è compresa tra i 12 e 30 fotogrammi al secondo (fps); valore che può variare in base al tipo di dato e alla risoluzione richiesta.

Pipelines: Come si vede nella figura 4.1, flussi diversi vengono trasferiti in singole pipelines (condutture virtuali), che è necessario abilitare insieme al tipo di dati che si desiderano dal sensore. Il **tipo di fotogrammi** dell'immagine, ovviamente, **dipende** da parametri forniti in input: come la **risoluzione del fotogramma**, il **tipo di immagine** e il **frame rate**. In base agli input inseriti, il **sensore inizializzerà il canale di streaming** per il trasferimento dei dati. Se non si specifica nulla, la libreria acquisirà il tipo di immagine e la risoluzione predefinita per quel particolare canale. L'SDK supporta due tipi di formati di flusso:

- **Color Image Stream**

- **Depth Image Stream**

Per la gestione di questi stream video, Kinect SDK fornisce una classe astratta ad alto livello chiamata **ImageStream**, con la quale è possibile effettuare operazione di processing e filtering del flusso .
Il funzionamento a basso livello del processo di streaming, invece è leggermente più complesso. I fotogrammi dell'immagine vengono memorizzati in un **buffer** prima di essere utilizzati dal'applicazione. Se c'è un **ritardo nella lettura** dei dati oppure durante processo di rendering, **il buffer si riempirà** con un nuovo frame **scartando i dati precedenti**. Questo rappresenta un punto molto critico nel caso in cui il nostro sistema dipenda fortemente dalla stabilità del flusso in entrata. Vedremo che in questo caso converrà utilizzare un approccio ad **Agenti** che permetterà di disaccoppiare nettamente il flusso di sistema da quello del sensore.

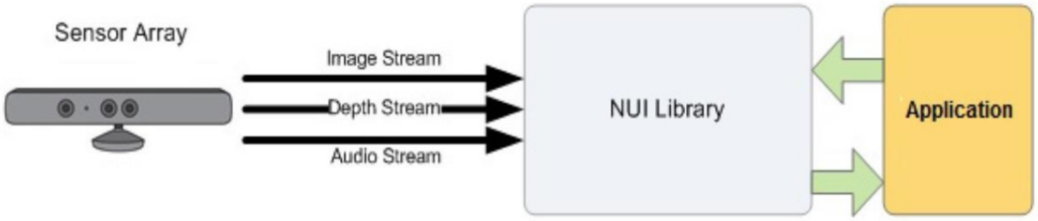


Figura 4.5: Struttura a Pipelines, Kinect v1.0

Gestione e processing del Color Image Stream:

Kinect supporta 3 tipologie di formati per la gestione dei frame a colori: **RGBa, YUV** e **Bayer** [18].

- **RGBa:** E' lo spazio colore rosso-verde-blu noto come colore RGB. Ogni pixel RGB del frame a colori è un array di dimensione quattro, disposto nel modo seguente: i primi tre valori sono per rosso, verde e blu, mentre Alpha fornisce il canale per la trasparenza.

- **YUV:** Definito in inglese con il termine **Luminance**, YUV è uno spazio colore utilizzato per la codifica di immagini o video, concepito per rispecchiare il comportamento della visione umana. Nella codifica YUV: **Y** sta per la **luminanza**, **U** è il canale del **blu** e **V** è il canale del **rosso**. E' importane chiarire che sia i dati YUV che i dati RGB rappresentano lo stesso fotogramma dell'immagine come se fossero catturati utilizzando la stessa fotocamera; l'unica differenza è nella rappresentazione dello spazio colore.

- **Bayer:** La fotocamera Kinect restituisce anche un formato di immagine a colori Bayer non elaborato, con una combinazione di colori rosso, verde e blu con un pattern di filtro colore: 50% verde, 25% rosso e 25% blu. Il pattern è definito come vettore di filtri colore Bayer o **filtro Bayer**.

I fotogrammi dell'immagine sono nel tipo **ColorImageFrame**; per recuperarli sono possibili due modi: o facendo la *subscribe* al Gestore di Eventi corrispondente o inviando esplicitamente la richiesta al sensore. Nella stragrande maggioranza dei casi, quando si desidera che il sensore notifichi la presenza di un frame, viene utilizzato il primo approccio.

Collegare l'Event Handler: La classe KinectSensor ha un **evento ColorFrameReady**, che viene richiamato ogni volta che viene inviato un nuovo frame dal sensore. Quello che dobbiamo fare consiste semplicemente nel creare un **gestore di eventi** e poi, da esso, fare la subscribe all'evento. Il processo può essere riassunto in poche linee di codice:

```
private void StartKinectCam() {
    if (KinectSensor.KinectSensors.Count > 0) {
         this.sensor = KinectSensor.KinectSensors.FirstOrDefault();
         this.StartSensor();
         this.sensor.ColorStream.Enable();
         //SUBSCRIBE
         this.sensor.ColorFrameReady += new
             EventHandler<ColorImageFrameReadyEventAr
         gs>(sensor_ColorFrameReady);
    }
   else {
         MessageBox.Show("No device is connected with system!");
         this.Close();
    }
}
//Notificato dall'evento
void sensor_ColorFrameReady(object sender,
    ColorImageFrameReadyEventArgs e) {
   ColorImageFrame imageFrame = e.OpenColorImageFrame();
}
```

Listato 2: Collegamento Event Handler a Evento

Gestione e processing del Depth Stream:

Per quanto riguarda lo Stream Depth le funzionalità proposte per l'accesso allo stream e la subscribe sono pressochè le stesse, l'unica differenza la riscon-

triamo andando ad analizzare il tipo di dati trattato.
Il sensore Kinect, infatti, restituisce i dati del frame di profondità grezza a 16 bit. I primi **tre bit** sono usati **per** rappresentare i **giocatori** identificati e i restanti **13 bit forniscono la distanza misurata** in millimetri. Discuteremo l'indice del giocatore (primi tre bit) in una parte successiva. Per ottenere la distanza registrata in uno specifico pixel (cella dell'array/immagine), è necessario quindi effettuare uno shift verso destra di 3 bit. Nel caso in cui però volessimo ottenere le distanze di tutti i pixel ci viene in aiuto questo metodo:

```
private short[] ReversingBitValueWithDistance(DepthImageFrame
    depthImageFrame, short[] pixelData) {
    short[] reverseBitPixelData = new short[depthImageFrame.
        PixelDataLength];
    int depth;
    for (int index = 0; index < pixelData.Length; index++) {
        depth = pixelData[index] >>
            DepthImageFrame.PlayerIndexBitmaskWidth;
        if (depth < 1500 || depth > 3500) {
            reverseBitPixelData[index] = (short)~pixelData[index]; ;
        } else {
            reverseBitPixelData[index] = pixelData[index];
        }
    }
    return reverseBitPixelData;
}
```

Listato 3: Creazione di un'immagine di profondità.

4.3.4 Focus sul flusso dati Skeleton

Quando parliamo di come costruire un'applicazione che interagisce con il movimento del corpo umano, prima di tutto dobbiamo catturare le informazioni degli utenti che si trovano di fronte al nostro dispositivo. Utilizzando quello che abbiamo visto precedentemente, questo compito risulta veramente complicato; un approccio potrebbe essere quello di costruire un'immagine di supporto formata solamente dagli indici dei giocatori, ma questo non permetterebbe il tracciamento del movimento (RGB + Depth). Nasce quindi la **necessità di** dover **considerare più stream** contemporaneamente al fine di ottenere un **unico flusso multifunzionale**. Fortunatamente gli sviluppatori di Kinect hanno preso in considerazione questa necessità e hanno progettato una funzionalità davvero innovativa: lo **Human Skeleton Tracking** [17].

La funzionalità completa di tracciamento dello scheletro si basa sull'**elaborazione dei dati di profondità** e su **algoritmi di visione a colori** entrambi processati attraverso tecniche di **machine learning**.
Utilizzando il tracciamento dello scheletro, il sensore Kinect può tracciare il corpo umano su vari punti di articolazione. In particolare utilizzando le librerie fornite da Microsoft, si possono monitorare fino a sei giocatori e fino a 20 giunti per ogni scheletro. Solo due utenti possono essere monitorati in dettaglio, il che significa che il sensore può restituire tutte le venti informazioni dei punti di congiunzione tracciati, mentre, per reimpostare gli utenti, fornisce solo la posizione complessiva. Questo perché il tracciamento delle informazioni comuni per tutti e sei gli utenti richiederebbe un sforzo computazionale troppo elevato.

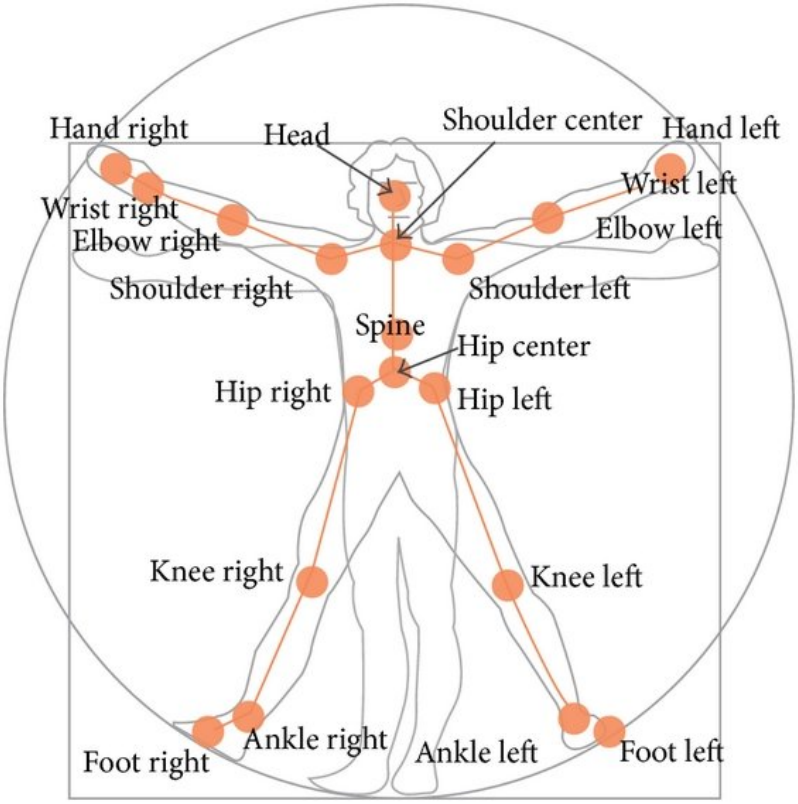


Figura 4.6: Elenco e visualizzazione delle giunzioni riconosciute da Kinect v1

Stima dello Scheletro Umano

Una versione della stima dello scheletro è incorporata all'interno dell'SDK e tramite essa si può ottenere direttamente un flusso di frame scheletrici che

può essere gestito in modo simile ai flussi già visti.
Per realizzare la stima dello scheletro umano in tempo reale, viene sviluppato un modello per rappresentare uno scheletro umano completo. Quindi, il modello viene addestrato con un numero spropositato di dati etichettati. Infine il modello addestrato è incorporato nell'SDK per il tracciamento dello scheletro in tempo reale. Quanto segue mostra i passaggi principali per la stima dello scheletro umano:

1. **Segmentazione della figura umana (Depth frames):** Analizzando i frame di profondità, il sensore identifica l'oggetto "corpo umano", che in questa fase non è altro che un insieme di dati di profondità grezzi vicini tra loro. Successivamente, attraverso un processo chiamato **segmentazione**, il sensore isola la silhouette e ricostruisce l'oggetto individuato (normalmente attraverso una semplice sottrazione del background). E' importante sottolieare che in questo punto Kinect non saprà ancora se ciò che viene inquadrato si tratta di un corpo umano o un altro oggetto.

2. **Stima della Posa Umana:** In un primo momento, il dispositivo cerca di abbinare ogni singolo pixel acquisito ai template presenti nel **dataset** installato; ogni template si compone di un codifica e di un'etichetta relativa ad una parte del corpo. In particolare questo abbinamento si basa sulla probabilità che i dati in arrivo corrispondano ai dati appresi dalla macchina.
   Il passo successivo nel riconoscimento della posa consiste nell'etichettare le parti del corpo riconosciute creando segmenti. Kinect utilizza una struttura ad albero addestrata (nota come **Decision Tree**) per abbinare i dati a un tipo specifico di corpo umano. Questo albero è noto come una **Random Forrest** [19].
   Tutti i nodi in questo albero sono dati di template diversi etichettati con nomi di parti del corpo. Alla fine, ogni singolo pixel di dati passa attraverso la foresta decisionale per adattarsi alle parti del corpo più simili. Il processo completo di corrispondenza dei dati viene eseguito più e più volte e ogni volta che ci sono dati corrispondenti il sensore inizia a segnarli creando **segmenti corporei**.

3. **Inferenza delle Giunture dello Scheletro:** Una volta identificate le diverse parti del corpo, il sensore inferisce la loro posizione confrontando le varie misure acquisite con i template corrispondenti. Con i punti di articolazione identificati il sensore è cosi in grado di seguire il corpo in tutti i suoi movimenti, fornendo all'utente un **modello tridimensionale e multiforma**, lo **Skeleton** appunto.

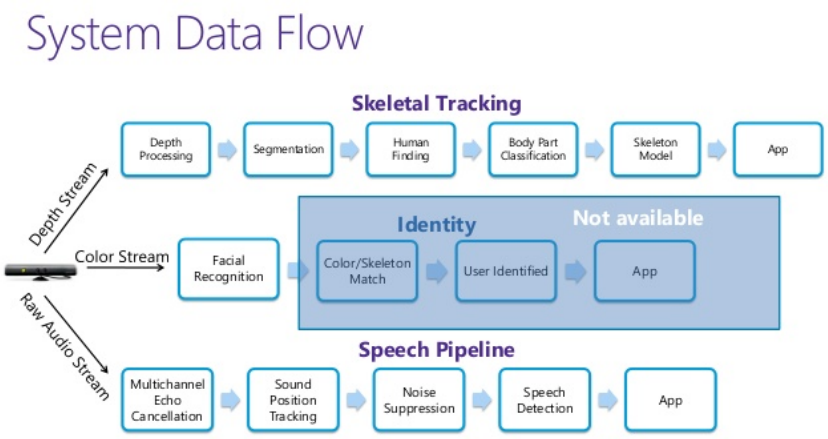


Figura 4.7: Riassunto dei processi di elaborazione che avvengono all'interno di Kinect

Kinect per Windows SDK ci fornisce un set di API che consentono un facile accesso alle giunture dello Skeleton. L'SDK supporta il monitoraggio di un massimo di 20 punti di articolazione. Ciascuna **posizione** articolare è **identificata** dal suo **nome** (testa, spalle, gomiti, polsi, braccia, colonna vertebrale, fianchi, ginocchia, caviglie e così via), e dallo **stato di tracciamento dello scheletro**: *Tracciato, Non tracciato Solo posizione.* Inoltre è possibile accedere alla **posizione di ogni giunto** compreso il suo **orientamento nello spazio**.
E' supportata anche una modalità per il tracciamento di uno scheletro da seduto che restituisce fino a 10 punti di articolazione.

API di Supporto

Per recuperare i dati dello scheletro, prima di tutto è necessario ottenere il riferimento del sensore attualmente collegato e quindi abilitare il canale dello scheletro del flusso [18].

```
KinectSensor sensor;
void MainWindow_Loaded(object sender, RoutedEventArgs e){
    this.sensor = KinectSensor.KinectSensors.Where(item => item.
    Status == KinectStatus.Connected).FirstOrDefault();

    if (!this.sensor.SkeletonStream.IsEnabled) {
        this.sensor.SkeletonFrameReady += sensor_SkeletonFrameReady;
    }
    this.sensor.Start();
```

```
}
void sensor_SkeletonFrameReady(object sender,
    SkeletonFrameReadyEventArgs e) {
    foreach (SkeletonData user in e.SkeletonFrame.Skeletons) {
        if (user.TrackingState == SkeletonTrackingState.Tracked) {
            if (user.Joints[JointType.HandRight].TrackingState
                ==JointTrackingState.Tracked) {
                Point mappedPoint =
                    this.ScalePosition(skeleton.Joints[JointType.
   HandRight].Position);
            }
        }
    }
}
private Point ScalePosition(SkeletonPoint skeletonPoint) {
    DepthImagePoint depthPoint = this.sensor.CoordinateMapper.
    MapSkeletonPointToDepthPoint(skeletonPoint, DepthImageFormat.
    Resolution640x480Fps30);
    return new Point(depthPoint.X, depthPoint.Y);
}
```

Listato 4: Creazione e gestione del flusso Skeleton.

4.3.5 API per il Riconoscimento Vocale

Come abbiamo visto precedentemente, Kinect è abilitato di default al riconoscimento audio: una caratteristica molto importante quando vogliamo costruire un'applicazione in ambito HCI.
Un modo semplice e veloce per utilizzare il riconoscimento vocale integrato consiste nel creare un'istanza della classe *SpeechRecognitionEngine*, settando tutti i parametri necessari. Creata l'istanza sarà così possibile associare un gestore di eventi all'evento corrispondente al riconoscimento, *SpeechRecognized*. In questo modo, ogni volta che l'audio è riconosciuto e internamente convertito in testo, verremo notificati con la stringa di testo riconosciuta.
Da questo punto in poi lo sviluppatore si può sbizzarrire e gestire i comandi vocali in un'infinità di modi: per esempio si possono utilizzare delle parole chiave per effettuare una particolare operazione oppure utilizzare queste parole come password di sblocco dell'applicativo.

Capitolo 5

Sviluppo del Framework

Lo scopo di questo progetto consiste nella realizzazione di un framework per il riconoscimento realtime di gestures, ***Jestures***. Il framework verrà sviluppato in linguaggio **Java** e come caso di studio verrà scelto il dispositivo **Kinect** introdotto nel capitolo precedente. Le motivazioni che hanno portato alla nascita di questa idea risiedono essenzialmente nella volontà di fornire un supporto per il riconoscimento a tutti gli sviluppatori che possiedono Kinect e hanno necessità di integrare un riconoscitore nei propri applicativi.
Per questo motivo al fine di garantire maggiore consistenza ed estendibilità verrà data grande importanza ai processi ingegneristici di sviluppo software. L'utilizzo di *DVCS* e *Software CI*, infine imporranno uno standard elevato di sviluppo e un monitoraggio costante del manufatto durante tutti gli step.
In questo capitolo illustreremo nei minimi dettagli i processi di sviluppo che hanno portato alla nascita del framework. Inizialmente, attraverso un processo di analisi, esporremo i caratteri generali e i requisiti funzionali in gioco; successivamente ci concentreremo sugli aspetti di progettazione al fine di introdurre una visione completa per l'implementazione della libreria.

5.1 Analisi

Se il processo di analisi di requisiti per un applicativo porta neccessariamente a un ragionamento in campo di funzioni lato utente, per una libreria le cose si fanno più complesse. I requisiti dovranno essere chiari, semplici e soprattutto definiti al fine di garantire un'implementazione snella e riusabile. Riferimenti alle tecnologie e ai contesti hardware dovranno essere ridotti all'osso e grande rilevanza avranno le idee volte ad assicurare la massima estendibilità. Il processo di analisi procederà nel seguente modo:

1. **Studio di Fattibilità**

2. **Analisi dei Requisiti**
    - *Requisiti Qualitativi*
    - *Studio di un modello di approccio*
    - *Requisiti Funzionali*
3. **Astrazione e Modellazione delle Entità**

5.1.1 Studio di Fattibilità

Prima di approcciarsi a un'idea di progetto è fondamentale chiedersi se quest'idea può essere implementata, in poche parole se è *fattibile*. Uno studio di fattibilità ci consente di valutare sistematicamente le caratteristiche, i costi e i benefici di un progetto; inoltre esso va a sondare le difficoltà di sviluppo in relazione alle conoscenze pregresse e alle tecnologie impiegate.

Approcci e Strategie di attuazione

Nel caso del nostro framework lo scoglio principale consiste nella ricerca di un metodo per garantire un supporto lato Java per il dispositivo Kinect. Questa necessità viola i fondamenti dell'analisi (per quanto concerne l'astrazione) posti qua sopra, ma è necessaria affinchè si possa disporre di uno strumento per effettuare il testing e lo sviluppo. Come vedremo nella sezione dedicata all'implementazione, il problema è stato risolto appoggiandosi a una libreria di binding, la quale consente un mapping 1-1 con l'SDK nativo del dispositivo. Un altro problema sorto durante lo studio di fattibilità risiede nel processo di **astrazione su più dispositivi**. Lo scopo più ambizioso del framework infatti è quello di fornire un supporto lato gesture non solo per Kinect ma anche per altre tipologie di sensori. Sfortunatamente non avendo a disposizione altri dispositivi per il testing, sorge la necessità di costruire un'**architettura modulare** grazie alla quale sarà possibile integrare col tempo ulteriori sensori.

5.1.2 Analisi dei Requisiti qualitativi

Dato che il nostro scopo è la progettazione di un framework non ci si può esimere dal rispetto degli standard qualitativi definiti nell'*Ingegneria del Software*:

- **Correttezza e affidabilità:** Sono le caratteristiche fondamentali di un software, senza di esse non ha senso intraprendere un progetto. Relativamente a questo framework l'*affidabilità* sarà ovviamente legata all'accuratezza del riconoscimento che dovrà essere idealmente del 100%.

Un sistema affidabile è anche un sistema da cui un utente può dipendere e per raggiungere questo scopo è necessario curare non solo l'aspetto implementativo ma anche quello di utilizzo.

- **Riusabilità, facilità d'uso e robustezza:** La bontà di un framework nella maggior parte dei casi risiede nella sua *usabilità* e *riusabilità* anche a fronte di performance non proprio ottimali. E' interessante però notare come framework chiari e ben organizzati spesso siano anche quelli più performanti, evidenziando quindi il fatto che la *riusabilità* e la *buona progettazione* siano un'ottima accoppiata per garantire *robustezza e performance*.

- **Modularità ed estendibilità:** Nel nostro caso uno dei fattori fondamentali per ottenere *estendibilità* risiederà nel definire un *core framework* robusto affiancato a *moduli* dedicati all'interfacciamento su vari dispositivi (Leap Motion, Kinect, ecc). Questo disaccoppiamento oltre ad semplificare l'utilizzo della libreria, permetterà la creazione di nuovi moduli esterni garantendo cosi piena *compatibilità* con nuovi sensori.

- **Performance:** Nell'ambito di progetti dinamici come questo le *performance* sono un aspetto qualitativo preliminare e non contrattabile. Un tuning delle performance spesso è necessario per garantire un funzionamento robusto e affidabile: allo stesso tempo però è un processo molto costoso in termini di tempo e conoscenze e deve essere impiegato solo se l'affidabilità del sistema viene messa in discussione. Idealmente un buon tuning delle performance si pone l'obbiettivo di migliorare il sistema senza snaturare o degradare la qualità del modello sottostante.

- **User Experience:** Nella prima parte della tesi abbiamo parlato approfonditamente dell'importanza della user experience nel campo delle tecnologie destinate alla Human-Computer Interaction. Nel caso del nostro framework, al fine di migliorare l'esperienza utente, è di fondamentale importanza gestire al meglio i processi di interazione. Per esempio bisognerà fornire all'utente un metodo intuitivo per registrare il proprio dataset di gesti e un ulteriore feedback nel momento in cui tali gesti dovranno essere riconosciuti.

5.1.3 Studio di un modello di approccio

Completato lo studio di fattibilità e analizzati i requisiti qualitatitivi richiesti per lo sviluppo di una libreria, è giunto il momento di entrare nel vivo del discorso e indagare fino in fondo le **specifiche del problema**.

Al fine di garantire specifiche il più coerenti e corrette possibili, l'approccio al progetto partirà da uno studio ristretto all'ambito del riconoscimento con lo scopo di **generalizzare e astrarre le entità**, fondamentali per i processi successivi di analisi e modellazione.

Il modello fonologico:

Nel capitolo 2 abbiamo studiato in profondità il concetto di gesture, in particolare nella sezione 2.1.2 abbiamo definito due modelli linguistici principali per la modellazione di gesture, fondamentali per un approccio corretto al riconoscimento:

- Phonological Model: Secondo questo modello una gesture è individuata in modo univoco da 3 caratteristiche complementari che sono: **movimento, posizione** e **forma**.
- Movement Hold Model: Secondo questo modello una gesture è rappresentata come una sequenza lineare di **fasi di movimento** alternate a **fasi di attesa**. Sia le fasi di movimento che le fasi di attesa sono definite da unità semantiche.

Il modello che si predispone meglio per un approccio a riconoscimento è sicuramente il secondo infatti è molto naturale per un elaboratore ottenere informazioni discretizzate da informazioni nel continuo.
E' interessante infine osservare come entrambi i modelli linguistici dimostrino la dipendenza delle gestures da aspetti di tipo temporale e spaziale.

Spazio Il concetto di **Spazio** nel riconoscimento è di fondamentale importanza. Nel caso delle Gesture possiamo individuare due tipi di spazio:

- **Spazio Assoluto:** La gesture avviene in uno spazio di visione determinato e le **coordinate sono relative solo al sensore**. Per esempio Kinect fornisce di default coordinate spaziali tridimensionali assolute dipendenti dal proprio campo visivo. In uno spazio assoluto, affinchè una gesture venga riconosciuta bisognerà eseguire i movimenti nello stesso identico modo e nella stessa identica posizione con la quale è stata registrata.
- **Spazio Relativo:** La gesture avviene su un dominio di spazio ristretto e relativo all'utente. Le coordinate sono calcolate in base alle metriche del sensore ma **il sistema di riferimento viene relativizzato all'utente**. E' il caso di un accelerometro all'interno di un dispositivo wearable; le coordinate fornite sono ristrette alla tecnologia del sensore ma dipendono intrinsecamente dal movimento dell'utente che indossa il dispositivo.

Nel caso del riconoscimento di gesti, appurato che il focus sia rivolto ai soli movimenti delle mani, non si richiede un tracciamento generico dell'utente e quindi nel nostro caso è più consono un modello a spazio relativo. Dato che Kinect non fornisce vettori relativi all'utente sono state sviluppate alcune tecniche con lo scopo di relativizzare la posizione delle gesture.

Tempo Infine anche il **tempo** gioca un ruolo di primaria importanza nel rilevamento di gesti. Grazie alla **discretizzazione temporale** infatti possiamo campionare e quindi suddividere un gesto nelle sue componenti statiche fondamentali rendendo, cosi, più facile la loro analisi.

Single Joint e Spazi Vettoriali Come sappiamo esistono infinità di gestures, da quelle più semplici che richiedono movimenti semplici, a quelle più complesse dove i movimenti sono molto più intricati e possono addirittura riguardare diverse parti del corpo. Nel caso delle Gesture **Single Joint** la posizione della joint nel tempo può essere rappresentata come un **vettore in uno spazio bidimensionale o anche tridimensionale**, questo le rende delle candidate molto appetibili per un sistema robusto di riconoscimento.

5.1.4 Analisi dei Requisiti Funzionali

Definito un approccio standard al *riconoscimento gesture* e identificati i confini del dominio del problema, possiamo finalmente studiare un metodo al fine di *adattare gli aspetti teorici al nostro caso di studio.*
Come abbiamo già specificato a inizio capitolo, uno degli aspetti fondamentali del progetto risiederà nella sua estendibilità a più sensori, da cui la scelta di un'architettura **modulare**. Dato che non è possibile sviluppare e testare il sistema su ogni tipologia di sensore esistente si è deciso di prendere in esame un solo dispositivo. La scelta come abbiamo detto è ricaduta sul famosissimo *Kinect* della Microsoft, sensore nato in ambito gaming ma portato poi anche in altri ambiti quali la computer grafics, virtual reality e ovviamente gesture recognition. E' stato scelto proprio questo sensore per tre motivi molto semplici:

1. Gode di una diffusione elevata e di un costo generalmente abbordabile per un utente medio.

2. E' supportato oltre che da Microsoft anche da un'ampia community indipendente che rilascia costantemente aggiornamenti.

3. E' equipaggiato con una sensoristica da far gola a qualsiasi smanettone.

Kinect oltre a godere di queste qualità, come abbiamo già visto, possiede la fenomenale capacità di **mappare e tracciare i movimenti** del corpo umano **su uno spazio tridimensionale**, da qui l'idea di sviluppare insieme al framework anche il modulo dedicato a Kinect.

Studio dei segnali forniti da Kinect

Kinect, grazie all'SDK fornito da Microsoft, ci permette di sfruttare varie tipologie di flussi di dati. Come definito nella sezione 4.3.3 la scelta può ricadere su 3 tipologie:

- **Depth Stream:** Flusso di immagini di profondità.
- **Color Stream:** Flusso di immagini a colori.
- **Skeleton Stream:** Flusso di Skeleton. Ogni skeleton è formato da parti del corpo ognuna delle quali fornisce un flusso di vettori tridimensionali.

Inoltre è possibile sfruttare l'**accelerometro** montato sul dispositivo al fine di ottenere un ulteriore stream di dati.
In letteratura ognuno di questi stream è stato impiegato con scopi di riconoscimento, tuttavia gran parte delle tecniche proposte, per essere implementate, richiedono conoscenze (computer vision, machine learning) troppo avanzate e ciò implicherebbe sforzi enormi. Per questo motivo si è scelto di intraprendere la strada del **riconoscimento basato su serie temporali**, sfruttando i **feature vector** forniti dal flusso Skeleton.
Fortunatamente, come vedremo, l'SDK ci fornisce anche la possibilità di filtrare il flusso per alcune parti del corpo. Questa rappresenta una feature molto importante, infatti bastera sviluppare il riconoscimento su una giunzione generica e questo funzionerà su tutte le giunzioni; ovviamente la giunzione predefinita sarà una delle due mani.

Il problema delle coordinate assolute Definito il metodo di acquisizione dei segnali dobbiamo far fronte a un altro problema, ossia l'implementazione di un metodo che permetta di rappresentare i dati dello scheletro nel sistema di riferimento dell'utente anziché in quello del sensore. Questo è necessario dal momento che il Kinect SDK ci riporta solamente **vettori in coordinate assolute** dello spazio. Durante la fase di analisi sono state studiate e testate varie tecniche di relativizzazione. Lo scopo di queste tecniche consiste principalmente nello spostare il sistema di riferimento, da quello assoluto, calcolato sul campo visivo di Kinect a quello relativo, centrato sull'utente.

- **Spazio Vettoriale Relativo:** Lo spazio vettoriale assoluto viene ricalcolato e sfasato in base alla posizione dell'utente. Se $\vec{v} = (v_1, v_2, v_3)$ è un vettore tridimensionale assoluto, il suo calcolo come vettore relativo $\vec{v'}$ avverrà nel seguente modo: $\vec{v'} = (v_1 - x, v_2 - y, v_3 - z)$ dove x, y, z rappresentano le posizioni assolute dell'origine del sistema di riferimento relativo. Come origine del sistema si dovrà scegliere una giuntura del corpo che sia equidistante da tutti gli arti, per esempio il bacino.
  Il problema fondamentale di questo approccio risiede essenzialmente nella precisione che utilizza Kinect per calcolare lo Skeleton e quindi la posizione delle giunture. Durante i primi test effettuati infatti, la posizione del gesto, relativa al nuovo sistema di riferimento, risulta spesso imprecisa, sopratutto durante un movimento generico di tutto il copro. Calcolando il nostro sistema di riferimento sulla base di 2 vettori distinti abbiamo introdotto anche 2 punti d'errore. **Se Kinect misura in modo sbagliato solo una delle 2 giunzioni (bacino, mano), l'errore si ripercuoterà su tutto il sistema** portando a un tracking errato del movimento.

- **Starting Vector:** Questo è il nome che si è voluto dare alla tecnica utilizzata; l'approccio è molto semplice. Si decide un tempo di campionamento invariabile per esempio di 30 FPS; di conseguenza una gesture sarà formata da 30 vettori tridimensionali. Al fine di relativizzare il movimento si potrebbe prendere, fra questi 30, un **vettore-origine** ossia un vettore che fissa il sistema di riferimento su se stesso per tutta la durata della gesture. Si è scelto per questioni temporali il primo vettore.
  In altri termini definita la gesture (feature vector) $G = (v_1, v_2, \ldots, v_{30})$, fissiamo come vettore-origine $v_1 \Rightarrow (0, 0, 0)$. Di conseguenza tutti gli altri vettori saranno calcolati in base a v_1 tramite una semplice differenza, quindi $G' = (v_1 - v_1, v_2 - v_1, \ldots, v_{30} - v_1)$ sarà la nostra gesture relativizzata.
  Questo approccio è molto buono ma non ottimo poichè **impone una dipendenza molto forte su un solo vettore**. Imponendo per esempio un campionamento di 30 FPS il punto di inizio gesture dovrà necessariamente coicidere con il primo vettore. Se questo vettore viene letto in modo errato l'intera gesture, vanificando così la possibilità di un riconoscimento sovrapposto.

- **Derivata:** L'ultimo approccio è presente in molti paper riguardanti il riconoscimento di gesture con Kinect. L'idea è molto semplice, in ogni momento, ossia per ogni fotogramma, **viene calcolata la derivata del vettore tridimensionale direttore della gesture** (per esempio la mano destra). Essenzialmente stiamo calcolando i **vettori velocità** delle

gesture. Per farlo si sottrae semplicemente il vettore precentemente rilevato con quello attuale; in alcuni casi si può differenziare ogni 2 o più vettori con lo scopo di diminuire la frequenza di campionamento.
In altri termini se $\vec{v_i} = (x_i, y_i, z_i)$ è un vettore tridimensionale assoluto al tempo i, il suo calcolo come vettore derivata $\vec{v'}$ avverrà nel seguente modo: $\vec{v'_i} = (x_i - x_{i-1}, y_i - y_{i-1}, z_i - z_{i-1})$.
Sono stati svolti numerosi test e tutti hanno portato a considerare questo approccio come il miglior metodo di relativizzazione gesture. Ovviamente sono presenti alcuni problemi ma questi dipendono essenzialmente dal concetto stesso di derivata; per esempio sarà necessario eseguire le gesture a velocità molto simili tra loro. Vedremo però che utilizzando il DTW anche questo problema potrà essere risolto.

Studio delle entità a livello funzionale

Chiariti i metodi e le tecniche di gestione dei segnali provenienti da Kinect, possiamo finalmente concentrarci sull'analisi e lo studio di un'architettura per la nostra libreria. Punto fondamentale di questo studio, è l'identificazione delle entità fondamentali.
Al fine di poter identificare al meglio le entità in gioco è bene ricordare le caratterstiche e le specifiche richieste dal nostro framework:

- **Estendibilità** Per quanto riguarda l'estendibilità sensoristica sarà necessario implementare un'**architettura modulare** nella quale sarà molto semplice agganciare sensori di diversa natura.

- **Flessibilità** Bisognerà utilizzare un **criterio modulare e flessibile** anche all'aspetto di **codifica dei vettori**. Per esempio alcuni sensori potrebbero fornire prestazioni migliori utilizzando codifiche diverse da quella derivativa.

- **Usabilità** Per incrementare l'usabilità sarà necessario implementare dei criteri che garantisco un **interfacciamento semplificato** ai processi di recording e riconoscimento.

Al fine di trarre beneficio da questi fattori, è stato scelto di suddividere l'architettura del framework in due parti separate: il **Gesture Recognizer** e il **Gesture Recorder**. Il primo, sviluppato con un approccio modulare, costituirà il cuore del riconoscimento; mentre il secondo fornirà un supporto esterno, sottoforma di **tool**, per quanto riguarda l'**acquisizione** delle gesture.

Gesture Recognizer

Focalizzandoci sulla parte dedicata al riconoscitore possiamo individuare alcune entità fondamentali.

Il Core Framework Il core framework rappresenta il centro nevralgico di tutta l'elaborazione dei segnali e il riconoscimento. Esso dovrà necessariamente possedere queste caratteristiche:

1. **Agganciamento a moduli sensore:** Dovrà essere possibile agganciarsi a moduli relativi ai sensori. Al fine di semplificare l'operazione, bisognerà definire delle specifiche standard per un ogni modulo.

2. **Codifica dei dati:** Bisognerà implementare un metodo di codifica robusto e concorde con le richieste del riconoscimento. Tale metodo verrà modularizzato in un componente a parte con lo scopo di garantire estendibilità nel momento in cui la codifica dovesse cambiare.

3. **Gestione del framerate:** Per garantire un supporto maggiore ai sensori, occorrerà definire un metodo di gestione del framerate durante il riconoscimento. Per esempio nel caso in cui si volessero utilizzare due sensori con frequenze diverse, il framework dovrà adattarsi a tale richiesta.

4. **Classificazione delle sequenze:** Feature fondamentale per il funzionamento del framework. Sarà d'obbligo in questo caso fornire un riconoscimento preciso con tassi di errore molto bassi.

5. **Gestione degli utenti:** Gli aspetti di gestione saranno necessari per garantire una buona usabilità del framework da parte degli utenti finali. Per fare ciò bisognerà concentrasi su tre punti fondamentali:

   - *Creazione e eliminazionse di profili utente*
   - *Gestione del dataset personalizzata*
   - *Impostazioni di riconoscimento personalizzate*

6. **Interfaccia grafica di supporto:** Occorrerà infine includere un'interfaccia graficia di supporto alla registrazione, al riconoscimento di gesture e alla gestione degli utenti.

Lato Utente Lato utente il framework dovrà necessariamente utilizzare un **approccio ad eventi**. Gli eventi possono essere di varie tipologie:

- **Gesture Event:** Nel momento in cui il recognizer riconoscere una gesture viene generata una notifica testuale o visiva contente il nome della gesture riconosciuta.

- **Tracking Event:** Un tracking-event non è altro che un evento che traccia un segnale, *semplice* o *codificato* proveniente dal sensore.

  - *Absolute Tracking Event:* Viene tracciata la posizione assoluta del giunto.
  - *Derivative Tracking Event:* Viene tracciata la velocità del giunto.
  - *Starting-Vector Tracking Event:* Viene tracciato un vettore codificato secondo il metodo *Starting-vector*.
  - *Accelerometer Tracking Event:* Viene tracciato un vettore 3D generato dall'accelerometro.

Moduli Sensore I **moduli sensore saranno entità esterne** che dovranno rispettare certe condizioni specifiche per permettere l'agganciamento al *Core Framework*. Le condizioni riguarderanno vari aspetti tra cui:

- I tipi di dati scambiati tra i due moduli.

- La frequenza di invio dei dati.

- Le caratteristiche fisiche, meccaniche nonchè le metodologie di utilizzo del sensore.

Gesture Recorder

Gesture recorder si affianca, come anticipato, al Recognizer come tool di supporto. Il suo scopo è quello di **garantire all'utente un metodo facile, veloce e soprattutto usabile per registrare le proprie gesture**. L'architettura del Recorder è molto semplice, si compone essenzialmente di:

- **Interfaccia grafica:** Fornisce un supporto visivo durante tutto il processo di acquisizione, partendo dalla visualizzazione della gesture al suo salvataggio nel dataset.

- **Gestione lato utente:** Il recorder ha anche lo scopo di **creare dei profili utente**; in questo modo è possibile associare a ogni utilizzatore il proprio dataset con associate le proprie informazioni.

Casi d'uso Il framework nelle sue caratteristiche funzionali è estremamente semplice e conciso, il suo scopo infatti è quello di riconoscere gesture a fronte di segnali provenienti dai sensori. E' bene comunque aggiungere alcune funzionalità di supporto utili all'utente in modo da evitare problemi di rigidità funzionale. Per fare ciò si è deciso di aggiungere un accesso a medio-basso livello sul sistema in modo da permettere all'utente di acquisire informazioni sui dati preprocessati. Una funzionalità utile in questo senso è data dalla possibilità di accedere alla posizione di ogni parte del corpo in qualsiasi momento. (fig 5.1)

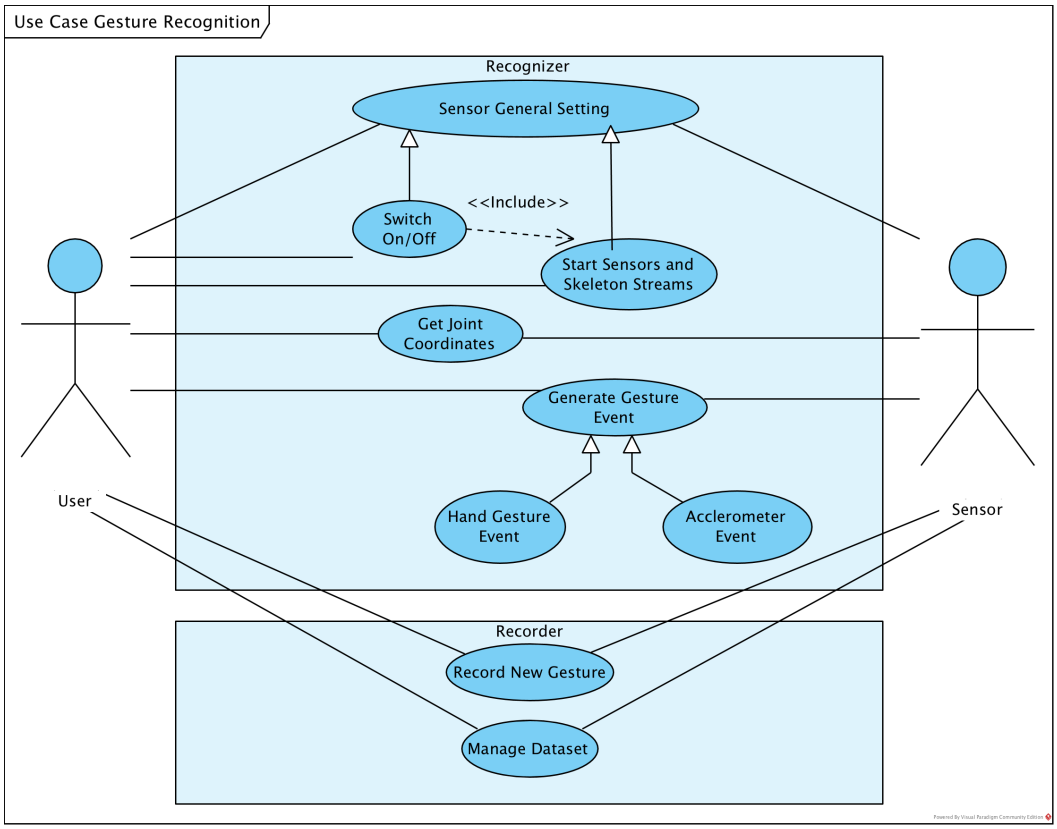


Figura 5.1: UMl dei casi d'uso: rappresenta una rappresentazione visiva dei requisiti funzionali analizzati precedentemente.

5.1.5 Astrazione e Modellazione delle Entità

La modellazione di analisi può e deve essere effettuata in due modi: il primo modo, *lato utente*, permette di definire una serie di entità utili per l'utente finale della libreria, mentre il secondo, *lato progettista*, permette di definire tutte le entità di backend necessarie per il funzionamento della libreria stessa.

Sarà poi necessario definire un mapping che permetta di tradurre i vari livelli di astrazione e complessità in modo da rendere il passaggio da lato utente a lato progettista meno netto.

Astrazione utente ed Entità da modellare

L'utente deve poter affidarsi a una serie di entità che permettano di fornire tutte le funzionalità necessarie all'utilizzo della libreria. Esse devono necessariamente rispecchiare le entità nate durante l'analisi dei requisiti e non devono essere né troppo specifiche né troppo generali.

- **Event:** Rappresenta un *evento* generato dal riconoscitore
  - **Gesture Event**
  - **Tracking Event**
- **Sensor:** E' la rappresentazione del *sensore* ad alto livello nonchè quella visibile dal *Core Framework*. Ogni *modulo sensore* deve conformarsi alle caratteristiche di *Sensor* per poter essere agganciato.
- **Event Generator:** Rappresenta un generatore di eventi. Questa entità lavora in sincronia con *recognizer* e *Tracker* e permette di elaborare e restituire gestures sottoforma di notifica *GestureEvent* o *TrackingEvent*.
- **Sensor Module:** E' il modulo specifico che rappresenta ogni sensore, esso permette di accedere a tutte le configurazioni specifiche garantendo così un controllo ideale ad alto livello.

Astrazione progettista ed Entità da modellare

Il progettista ha il grande compito di realizzare delle entità abbastanza astratte da essere completamente mappabili sulle entità utente, e abbastanza concrete da poter lavorare sui dati provenienti dal sensore.

- **Tracker:** Entità nata durante l'analisi dei requisiti, il *Tracker* ha lo scopo di fornire funzionalità per il tracciamento di un segnale, *semplice* o *codificato* proveniente dal sensore (vedi Tracking Event).
- **Serializer:** Entità del *Core Framework* dedicata alla *gestione del dataset* e degli *utenti*.
- **Codifier:** Modulo fondamentale del *Core Framework*, esso ha il compito di *acquisire, analizzare* e *discretizzare* il segnale continuo generato dal Sensore.

- **Recognizer** Rappresenta il vero *Core Framework* in tutte le sue parti. Il suo scopo è quello di coordinare armoniosamente *Codifier, Serializer* in modo da generare degli eventi di tipo *Gesture*.

Architettura per il Recognizer

L'ultima parte del processo di analisi è volta ad identificare, a grandi linee, un struttura del framework. Come possiamo vedere nella figura 5.2, l'architettura di massima è molto semplice e rispecchia fedelmente lo studio fatto fino ad ora.

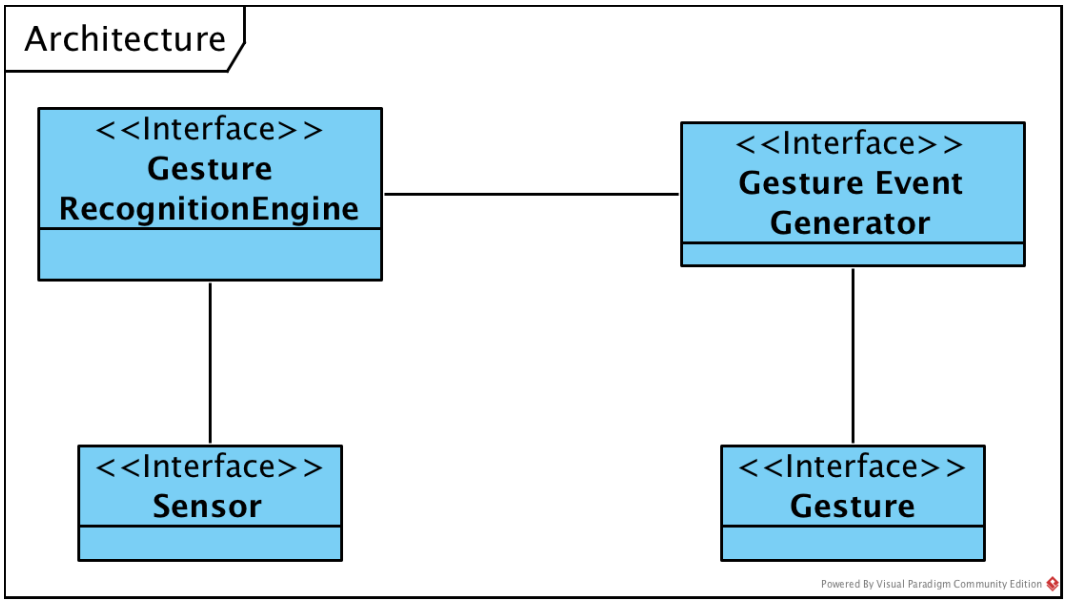


Figura 5.2: Architettura. Uml dell'architettura del framework

5.2 Progettazione

In questa sezione andremo a definire nel dettaglio il processo di mappatura da classi di analisi a classi di progettazione, in particolare ci concentreremo sui problemi riscontrati e sulle soluzioni adottate per risolverli. La fase di progettazione verrà divisa in due parti: la prima, dedicata al **Framework** e la seconda, dedicata al **tool di recording**.
Per ogni singola parte, lo studio dell'intera attività di progettazione seguirà uno schema ben preciso: la descrizione dell'aspetto *statico* del dominio avverrà attraverso **diagrammi UML** mentre gli aspetti *dinamici* e *funzionali* verranno descritti attraverso **specifiche testuali**.

5.2.1 Design Dettagliato: Recognizer

Al fine di garantire una descrizione completa del processo di progettazione, seguiremo la suddivisione proposta alla sezione 5.1.4 dell'analisi.

Modulo Sensore:

Il modulo che gestisce l'interfacciamento con il sensore è uno dei moduli più delicati dal punto di vista progettuale. Questo modulo ha la grande responsabilità di adattare l'approccio, definito dalle varie librerie a basso livello, all'architettura del framework (fig 5.4).
Al fine di rendere compatibili due componenti così diversi si è deciso di utilizzare il **Pattern Adapter**. E' stata definita un'interfaccia di adattamento (Sensore) e una classe *Adapter* che permette la compatibilità tra il *Core Framework* e il *componente* della libreria a basso livello.
Nel caso di Kinect, la libreria nativa (*J4K*) fornisce un flusso continuo di dati in uscita; affinchè questi dati possano essere inviati al **Core Framework** è necessario che quest'ultimo osservi il sensore. Per fare ciò ci si è avvalsi dell'utilizzo del **Pattern Observer** il quale attraverso due interfacce di osservazione (*Kinect Observer, Sensor Observer*) consente di *notificare* l'osservatore.
Grazie a questi accorgimenti è possibile scollegare un sensore e collegarne un altro senza problemi garantendo quindi un'ottima *estendibilità*.

Patterns I pattern utilizzati sono:

- **Pattern Adapter:** Il sensore viene adattato al *Core Framework*. Questa scelta garantisce al sensore un adattamento completo al sistema e nasconde gran parte delle funzioni di basso livello presenti nella libreria nativa.

- **Pattern Observer:** Il sensore notifica il *Core Framework* attraverso una catena di osservatori.

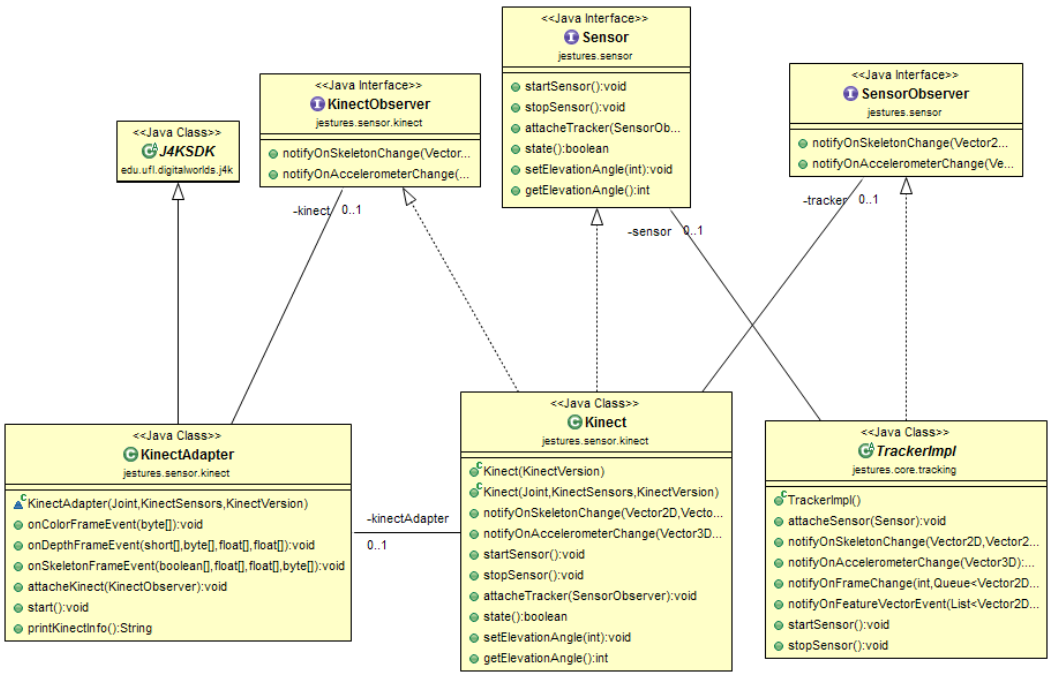


Figura 5.3: Pattern Observer e Adapter applicati alla comunicazione fra moduli.

Dati Scambiati Dato che dobbiamo confrontarci con sensori molto diversi tra loro, è naturale che anche i tipi di dato trasmessi differiscano. Onde evitare problemi di compatibilità si è deciso di standardizzare la tipologia di dati inviata, ovvero i *vector 2D*. In un primo momento si era pensato di creare un tipo di dato generico, lo *Skeleton* che codificasse l'insieme delle giunture del corpo, *Joints*. Sfortunatamente, dopo aver svolto alcuni test, si è visto che la costruzione di un oggetto di questo tipo era estremamente pesante e nella maggior parte dei casi inutile dato che generalmente, durante il riconoscimento, il focus è riservato a pochissime giunture.

Core Framework

Codifier Il codifier o codificatore è il primo componente di medio basso livello che gestisce ed elabora i *vector2D* provenienti dal sensore. Il suo ruolo è essenzialmente quello di **relativizzare** e **discretizzare** il flusso continuo di dati in serie di vettori bi-dimensionali (*Feature Vector*) di lunghezza variabile

(settabile dall'utente).
Le operazioni di relativizzazione dipendono dalle modalità di riconoscimento che deve essere effettuato; è possibile operare sui flusso in modo *derivativo* con lo scopo di ottenere vettori relativi legati alla velocità di esecuzione, oppure, come abbiamo visto precedentemente, è possibile utilizzare altre tipologie di codifica come quella basata su *spazio vettoriale* o *starting vector*. Poichè gli algoritmi di relativizzazione non sono limitati solamente a questi tre metodi, si è deciso di **modularizzare** il codificatore al fine di garantire una completa **estendibilità**.
Implementare questo tipo di modularizzazione non è semplice. Infatti, poichè il Codifier si innesta al centro nevralgico dell'elaborazione dei dati, bisogna assicurarsi che il flusso degli stessi non venga interrotto. Inoltre, i dati in arrivo dal sensore devono essere elaborati *in modo asincrono* così da disaccoppiare completamente i due moduli. Per modellare un tale comportamento è stato scelto nuovamente il *Pattern Observer*, implementato tramite l'interfaccia *Tracker Observer*.
Per quanto riguarda gli aspetti di discretizzazione, il *Codifier* ha lo scopo di regolare e normalizzare il framerate dei vari sensori in modo da renderlo compatibile con la lunghezza delle gesture.

Patterns I pattern utilizzati sono:

- **Pattern Singleton:** Esiste una sola entità del tipo *Codifier*.
- **Pattern Observer:** I dati provenienti dal *Sensore* vengono notificati al *Tracker*, il quale a sua volta delega l'elaborazione degli stessi al *codifier*. Quest'ultimo, notifica in modo asincrono il *Tracker* attraverso l'interfaccia, *Tracker Observer*.

Serializer Come il codifier anche il *serializer* appartiene al *Core Framework*. Questa classe ha l'obbiettivo di agire da *boundary class* e quindi effettuare il collegamento tra il framework e il *File System*. Per prima cosa il serializer effettua operazioni di raggruppamento sui dati (*Feature Vectors*), quindi procede alla serializzazione dei *Dataset* precedentemente costruiti. Una volta che i dati sono stati scritti su disco è necessario poi poterli recuperare. Il serializer ovviamente effettua anche questa operazione permettendo di caricare in memoria o un solo *Feature Vector* o tutto il *Dataset*.
Un'altra feature molto importante fornita da questa classe, risiede nella gestione dei profili utente. Il serializer, in questo caso, lavora a stretto contatto con il *tool di recording* il quale, attraverso interfaccia grafica, fornisce metodi per la creazione e l'eliminazione di utenti e la personalizzazione di gesture.

Patterns I pattern utilizzati sono:

- **Pattern Singleton:** Esiste una sola entità del tipo *Serializer*.

Tracker Nato come classe di supporto agli eventi di tracking a basso livello (*Tracking Event*), il *Tracker* astrae la logica di collegamento e gestione flussi fornendo un'*interfaccia di backend* a tutto il sistema. Un esempio è dato dalla gestione dell'interfacciamento con la GUI, che avviene solamente a questo livello e in modo indipendente da tutti gli altri moduli. Il *Recognizer* sfrutterà questa caratteristica estendendo da questa classe al fine di disaccoppiare il processo di riconoscimento da quello di tracking.

Recognizer Infine siamo giunti alla classe più importante del *Core Framework* il *Recognizer*. Il recognizer racchiude dentro di sé tutti i moduli precedentemente descritti nel core framework e li utilizza per giungere al riconoscimento di una Gesture (fig 5.4). In particolare, (estendendo la classe *Tracker*) il recognizer si avvale del *codifier* per discretizzare il flusso continuo di dati e ottenere un *feature vector*. Inoltre si serve anche del *Serializer* per precaricare in memoria l'intero *Dataset* di template. Ottenuto da un lato il *Feature Vector* e dall'altra un *Dataset* il recognizer procede dunque al confronto del feature vector con ognuno dei template presenti nel dataset. Il confronto avviene appoggiandosi ad algoritmi di Machine Learning e Classificatori che possiamo pensare racchiusi in una classe *Classifier*. Il classificatore agisce da black box e ci restituisce il Feature Vector più simile a quello che stiamo osservando. Ottenuto il feature vector viene letta la classe di gesture rappresentante e viene generato un *Gesture Event*.

Patterns I pattern utilizzati sono:

- **Pattern Singleton:** Esiste una sola entità del tipo *Recognizer*.
- **Pattern Observer:** Tutto il Core Framework fa ampio uso del pattern observer, con l'obiettivo di creare un flusso unico di dati.
- **Pattern Template:** La classe Recognizer estendendo da Tracker nasconde alcuni aspetti implementativi di quest'ultima e ne ridefinisce altri.

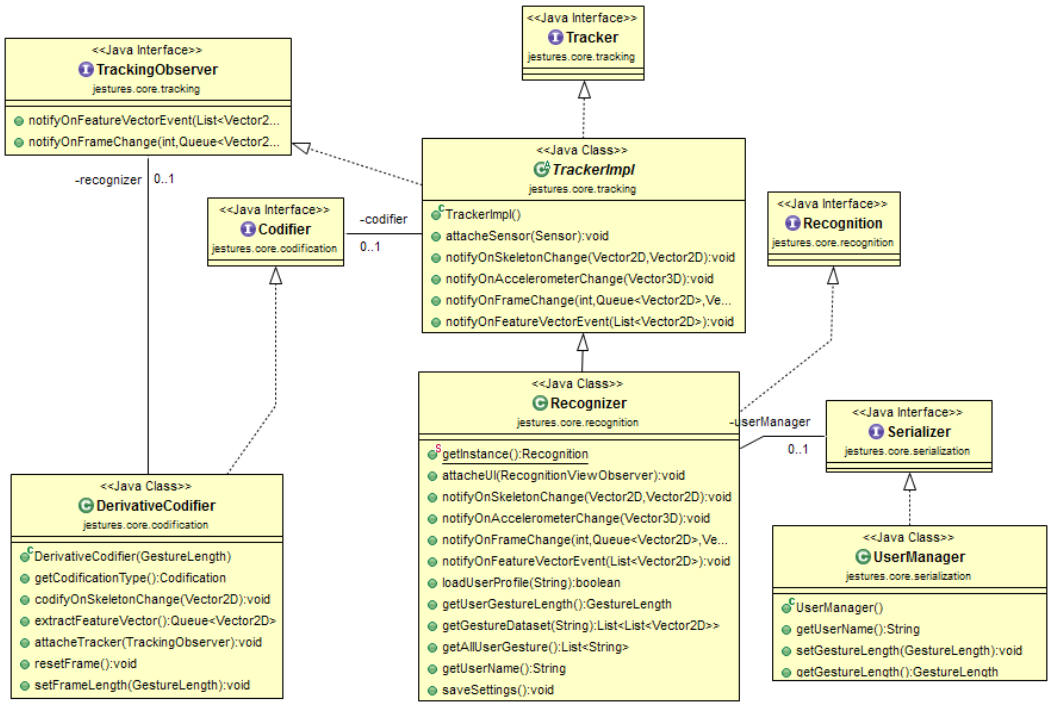


Figura 5.4: Il Recognizer si avvale di *Codifier*, *Serializer* e *Classifier* per riuscire a classificare una Gesture.

Generazione di Eventi

Esistono essenzialmente due approcci alla generazione di eventi: l'approccio a *polling* e l'*approccio a eventi*. L'approccio a polling consiste nel controllare continuamente lo stato di un sistema fino a quando non si ottiene la configurazione attesa. Nell'approccio a eventi invece, l'osservatore non deve verificare continuamente lo stato di un sistema: è il sistema stesso che genera un evento e lo notifica all'osservatore. Come possiamo vedere, nel nostro caso è conveniente implementare un approccio a eventi. Un aiuto molto grande al fine di costruire un sistema ad eventi ci viene fornito dalle API Swing in Java. Nell'SDK, infatti, è comune l'utilizzo **Listeners**, ossia *interfacce funzionali* che fungono da ascoltatori per gli eventi di sistema.
Agganciando (tramite **Observer**), all'interno della nostra libreria un'interfaccia funzionale di questo tipo, forniamo al framework l'abilità di notificare all'esterno eventi che avvengono all'interno di esso. Una volta che l'evento viene notificato, è possibile gestirlo in maniera assolutamente indipendente dal contesto di libreria; questo è possibile solamente attraverso il **Pattern Strategy** (fig 5.5).

Patterns I pattern utilizzati sono:

- **Pattern Observer:** Attraverso il pattern observer il *GestureListener* rimane in attesa di ricevere un evento. Nel momento in cui l'evento è ricevuto questo viene immediatamente notificato all'utente.

- **Pattern Strategy:** Notificato l'evento, l'utente può decidere di implementare la strategia nel modo che preferisce; solitamente per brevità si utilizza una lambda expression.

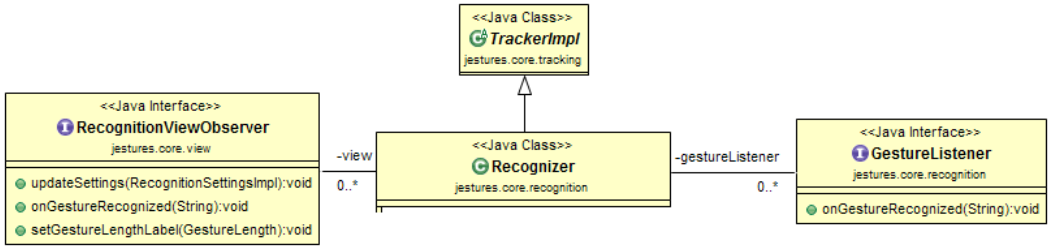


Figura 5.5: Un'implementazione elegante per la generazione eventi.

5.2.2 Design Dettagliato: Recorder

Per quanto riguarda la progettazione del tool di recording, è stato seguito uno modello di sviluppo standard basato sul pattern *MVC* (Model View Controller) (fig 5.6).

Gestione del *Model* con MVC Nel nostro caso, data la semplicità dell'applicativo si è deciso di ignorare completamente il concetto di Model e le funzionalità necessarie al tool sono state integrate nel modulo Serializer del Framework. Questa scelta ha dei vantaggi e degli svantaggi.
Guardando i lati negativi, una decisione di questo tipo potrebbe portare in futuro a *problematiche di estendibilità* a entrambi i moduli. Nel caso in cui infatti, si decidesse di aggiungere funzionalità al tool queste dovranno essere implementate all'interno della libreria aumentando l'entropia della stessa.
Concentrandoci invece sui lati positivi, questa decisione permette di *ridurre* considerevolmente il quantitativo di *codice duplicato*, assicurando inoltre una forte dipendenza fra i sistemi. Per esempio, nel caso in cui si decidesse di modificare il processo di salvataggio dei dati, il processo di lettura dipendendo da esso, verrebbe aggiornato di conseguenza, fornendo così più robustezza.

Interazione View → Controller Il controller nel nostro tool agisce fondamentalmente da intermediario tra l'interfaccia e il modulo Serializer. Nel momento in cui l'utente effettua un'azione sull'interfaccia, l'evento viene propagato al controller il quale esegue un metodo definito all'interno del serializer.

Interazione Controller → View Nel verso opposto l'interazione è più complicata, infatti l'interfaccia deve essere aggiornata in modo *asincrono* dal controller. Dato che un approccio a polling in questo contesto non è auspicabile, dovremo rivolgerci nuovamente al **Pattern Observer**. Il funzionamento è il solito: il controller genera un evento e questo viene propagato all'interfaccia grafica, la quale aggiorna il componente correlato.

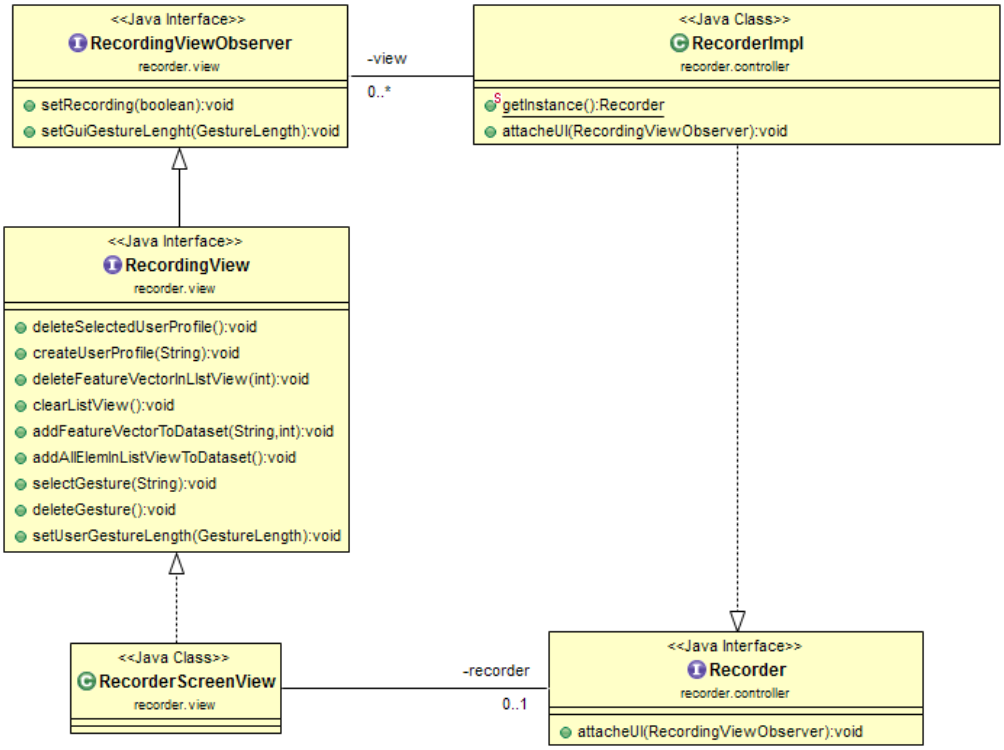


Figura 5.6: Pattern MVC semplificato.

5.3 Implementazione

Questa sezione sarà dedicata alla descrizione dettagliata dell'implementazione della libreria. La descrizione avverrà seguendo i passi principali del riconoscimento; questo ci consentirà di esprimere congiuntamente sia gli aspetta

implementativi (di codice) che i conseguenti aspetti funzionali.
La sezione seguirà questa suddivisione:

1. **Acquisizione dei segnali:** Verrà fornita una descrizione del processo di acquisizione dei segnali provenienti da Kinect. Dopo una breve introduzione sul processo mapping tramite libreria *J4K* verranno discusse le classi in gioco e le metodologie di elaborazione dei segnali.

2. **Estrazione dei feature vector:** In questa parte studieremo approfonditamente il processo di estrapolazione delle feature con un focus particolare sugli algoritmi e le classi implementate.

3. **Gesture recording e costruzione del dataset:** Lo studio di questa sezione verterà principalmente sul processo di recording delle gesture, in particolare forniremo una specifica dettagliata sulla struttura e gestione del dataset creato.

4. **Riconoscimento di gesture:** Per fornirne una descrizione esaustiva del processo di riconoscimento, ci concentreremo in un primo momento sullo studio delle classi in gioco spostandoci poi sul campo degli algoritmi implementati.

5.3.1 Acquisizione dei segnali e Estrazione di Feature

Come già sottolineato durante lo studio di fattibilità (sez 5.1.1), uno dei problemi più grandi sorti durante la progettazione di questo framework, risiede essenzialmente nella definizione di una mappatura **Kinect SDK → Java**. L'unica libreria capace di fornire una mappatura di questo tipo è J4K[1] [20].

Libreria J4K

La libreria J4K è una famosa libreria Java open source che implementa un **binding Java** per l'SDK Kinect di Microsoft. Comunica con una libreria nativa di Windows, che gestisce i flussi di profondità, colore, infrarossi e scheletro di Kinect utilizzando la **JNI** (Java Native Interface).

Compatibilità La libreria J4K è compatibile con tutti i dispositivi kinect (Kinect per Windows, Kinect per XBOX, Kinect o Kinect 2) e consente di controllare più sensori di qualsiasi tipo da una singola applicazione, a condizione

[1] Allo stato dell'arte, sono presenti varie API in linguaggio Java interfacciate con Kinect; purtroppo però, la maggior parte di esse non consente una gestione globale e semplificata del dispositivo.

che le capacità del sistema lo consentano. Ad esempio, è possibile controllare due sensori Kinect v1 o uno Kinect v1 e uno Kinect v2 collegati tramite USB 3.0 allo stesso computer. Inoltre, la libreria J4K contiene diverse comode classi Java che convertono i frame di profondità, i frame di scheletro e i frame di colore ricevuti da un sensore Kinect in oggetti Java di facile utilizzo.

Estensioni Facoltativamente, è possibile utilizzare la libreria Java **JOGL** (JogAmp) per visualizzare, tramite **OpenGL**, i dati Kinect come superfici 3D [21]. L'uso della libreria JOGL è *facoltativo* e non è necessario se non si utilizzano i metodi di disegno forniti nelle classi Java J4K.
Questa (fig 5.7) è una schermata della finestra di dialogo Kinect Viewer, che è inclusa nella libreria Java di J4K e offre un modo semplice per visualizzare i dati correnti e una semplice interfaccia grafica per la modifica dei parametri del sensore.

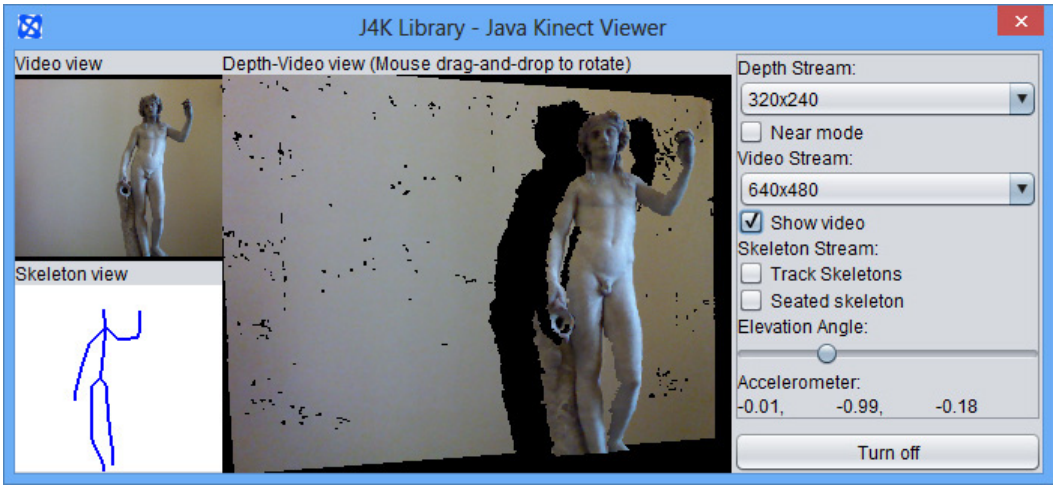


Figura 5.7: Il pannello di controllo per Kinect fornito dalla libreria J4K

Come scrivere un programma Kinect-Java con J4K Tralasciando per un attimo le funzionalità mappate da J4K, concentriamoci sullo studio della struttura di un programma Kinect.
Il codice qua sotto mostra un esempio canonico di programma Kinect-Java.

```
import edu.ufl.digitalworlds.j4k.J4KSDK;
import edu.ufl.digitalworlds.j4k.DepthMap;
import edu.ufl.digitalworlds.j4k.Skeleton;
import edu.ufl.digitalworlds.j4k.VideoFrame;

public class Kinect extends J4KSDK{
```

```
    VideoFrame videoTexture;

    public Kinect() {
        super();
    }

    @Override
    public void onDepthFrameEvent(short[] depth_frame, byte[]
        player_index, float[] XYZ, float[] UV) {
        DepthMap map=new
            DepthMap(getDepthWidth(),getDepthHeight(),XYZ);
    }
    @Override
    public void onSkeletonFrameEvent(boolean[] skeleton_tracked,
        float[] joint_position, float[] joint_orientation, byte[]
        joint_status) {
        Skeleton skeletons[] = new
            Skeleton[getMaxNumberOfSkeletons()];
         for(int i = 0; i < getMaxNumberOfSkeletons(); i++) {
            skeletons[i] = Skeleton.getSkeleton(i, skeleton_tracked,
                joint_position, joint_orientation, joint_status, this);
      }
    }
    @Override
    public void onColorFrameEvent(byte[] data) {
        videoTexture.update(getColorWidth(), getColorHeight(), data);
    }
}
```

Listato 5: La struttura di un semplice programma Kinect-Java

Come vediamo l'implementazione è molto semplice. La parte più importante del codice in tutti gli esempi forniti è la definizione di una classe Java che gestisce i flussi live ricevuti da un sensore Kinect. È possibile definire facilmente tale classe anche in meno di 10 righe di codice java. La definizione di una classe Kinect-Java avviene in 2 semplici passaggi:

- Prima di tutto, la classe dovrebbe estendere la classe **J4KSDK** dalla libreria J4K, come mostrato sopra. Facoltativamente, puoi definire il tuo costruttore e i parametri personalizzati per la tua classe, come mostrato nell'esempio seguente.

- Infine, bisogna implementare l'interfaccia dei 3 metodi **FrameEvent**. Questi metodi sono fondamentali poichè verranno automaticamente chiamati ogni volta che viene ricevuto un nuovo frame di profondità, video

o scheletro dal sensore Kinect. È possibile personalizzare il contenuto di questi 3 metodi in base alle esigenze della propria applicazione.

Funzionalità offerte: J4K oltre a definire 3 metodi dedicati alla gestione degli stream, garantisce anche un buona gestione lato sensore. La classe **J4KSDK** in particolare effettua un binding per il controllo totale di Kinect, definendo una quanitità impressionante di funzionalità. Tramite questa classe possiamo controllare l'angolo di elevazione di Kinect, impostare la modalità *"Seated"* e addirittura acquisire un flusso di dati provenienti dall'accelerometro.

Integrazione di J4K lato Framework

L'integrazione di J4K con il nostro framework avviene in modo molto semplice ed intuitivo. Data la natura modulare dei Sensori è possibile disaccoppiare totalmente le funzionalità offerte dalla libreria da quelle del framework. In particolare al fine di nascondere aspetti implementativi a basso livello è stato fatto uso del **Pattern Adapter**.
La classe che implementerà J4KSDK sarà dunque un Kinect adapter, il quale agirà da filtro ed elaboratore di segnali.

```
public void onSkeletonFrameEvent(final boolean[] skeletonTracked,
    final float[]
   jointPosition, final float[] jointOrientation, final byte[]
      jointStatus) {
    Skeleton skeleton;
    for (int i = 0; i < this.getMaxNumberOfSkeletons(); i++) {
        if (skeletonTracked[i] && this.first) {
            this.first = false;
            skeleton = Skeleton.getSkeleton(i, skeletonTracked,
                jointPosition,
               jointOrientation, jointStatus, this);
            Vector2D joint1;
            Vector2D joint2;
            joint1 = new Vector2D(
                    (int) (skeleton.get3DJoint(Skeleton.HAND_RIGHT)[0]
                     * KinectAdapter.MULTIPLIER),
                    (int) (skeleton.get3DJoint(Skeleton.HAND_RIGHT)[1]
                     * KinectAdapter.MULTIPLIER));
            joint2 = new Vector2D((int)
                (skeleton.get3DJoint(Skeleton.HAND_LEFT)[0]
                     * KinectAdapter.MULTIPLIER),
                    (int) (skeleton.get3DJoint(Skeleton.HAND_LEFT)[1]
```

```
                    * KinectAdapter.MULTIPLIER));
            this.kinect.notifyOnSkeletonChange(joint1, joint2);
        }
    }
    this.first = true;
}
 @Override
public void attacheKinect(final KinectObserver kinect) {
    this.kinect = kinect;
}
```

Listato 6: La struttura di un semplice programma Kinect-Java

Come vediamo nel listato 6, l'elaborazione avviene in modo abbastanza semplice:

1. *Identificazione e Estrazione Skeleton:* Viene costruito un oggetto di tipo Skeleton e inizializzato con i valori dell'utente tracciato. Per fare ciò si ciclano tutti gli skeleton presenti e si considera solamente il primo tracciato.

2. *Creazione dei vettori-giuntura:* Ottenuto lo Skeleton riguardante l'utente, si inizializzano i due vettori 2D riguardanti le due giunture richieste, in questo caso *Mano Sinistra* e *Mano Destra*.

3. *Notifica dei vettori:* I vettori vengono notificati attraverso una catena di Observer alla classe *Tracker* passando per la classe *Kinect*.

Conversione 3D → 2D L'algoritmo appena definito effettua anche una riduzione di dimensionalità sui vettori processati. Nel nostro caso la riduzione è avvenuta semplicemente ignorando la terza dimensione. Questa operazione potrebbe in alcuni casi generare problemi, infatti, non effettuando una normalizzazione rispetto alla profondità è possibile che il riconoscitore riscontri difficoltà nel classificare gesture avvenute a una distanza diversa da quella di registrazione. In altri termini, se registrassimo la gesture "Cerchio" a una distanza di 50 centimetri e poi la andassimo a riconoscere a una distanza di 2 metri, la **prospettiva indurrebbe un cambio di scala** nei valori ottenuti falsando cosi tutto il processo.

5.3.2 Estrazione di Serie Temporali (*Time Serie*)

Abbiamo terminato la scorsa sezione descrivendo il processo di elaborazione di dati e trasmissione al core framework (tramite catena di Observer). In

questa sezione ci concentreremo proprio sul *Core Framework*, in particolare ci focalizzeremo sull'algoritmo adibito all'estrazione e alla costruzione di **serie temporali**.
Ricordiamo velocemente la definizione di serie temporale: Circoscritta la natura di un pattern a un vettore n dimensionale, definiremo come serie temporale la seguente matrice: $\vec{x} = (\vec{x_1}, \vec{x_2}, ..., \vec{x_n})$. Solitamente una serie temporale viene considerata come un vettore le cui componenti possono essere mono o multi dimensionali.
All'interno del Core-Framework la costruzione di serie temporali è completamente adibita a un solo modulo, il **Codifier**. Poichè le tipologie di codifica sono diverse, ci concentreremo solamente su l'unica tipologia implementata nella libreria, la codifica derivativa (**Derivative Codifier**) 6. Come vediamo nel diagramma di attività 5.8, l'algoritmo di codifica derivativa è abbastanza semplice e può essere descritto in pochi passaggi.

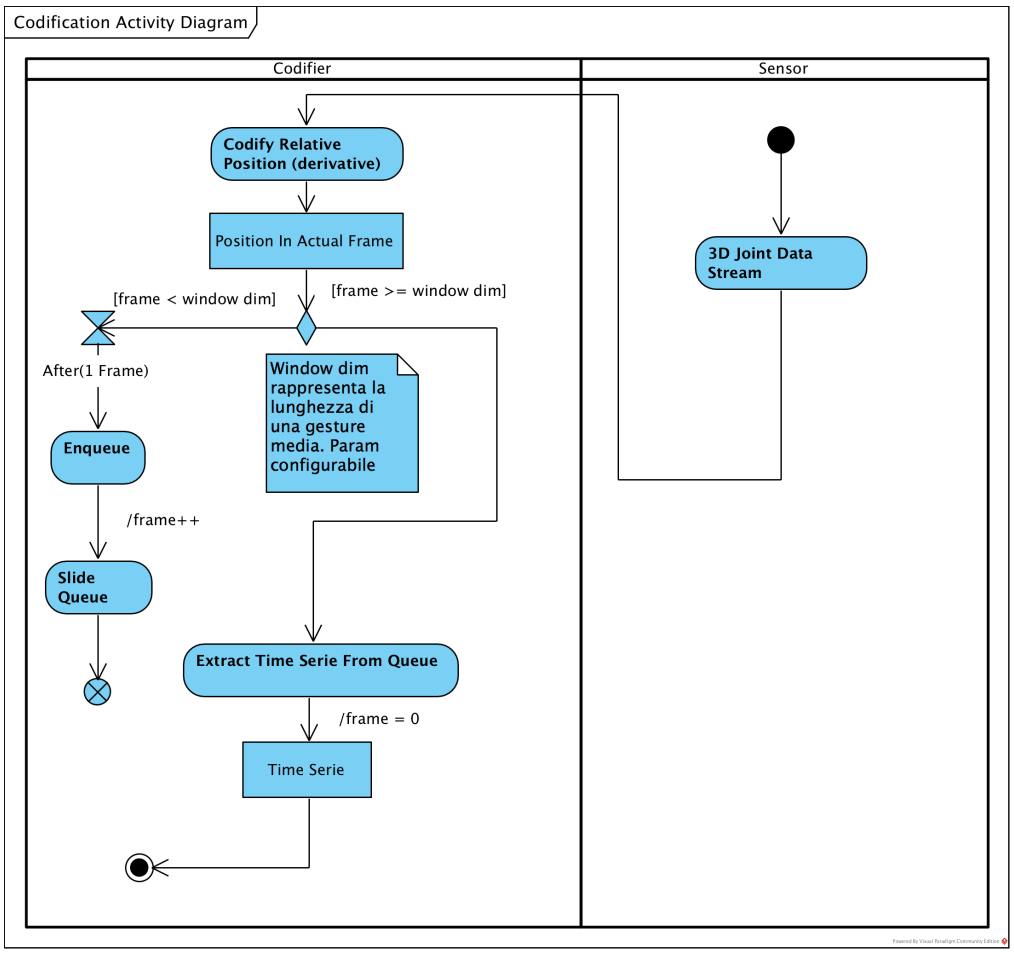


Figura 5.8: Diagramma UML di attività che descrive l'algoritmo di codifica.

1. *Instanziazione del Codificatore e Inizializzazione Parametri:* Vengono inizializzati i parametri dell'algoritmo dedicati al conteggio dei frame e al calcolo della derivata. Inoltre viene creata una *Sliding Queue* (coda scorrevole) con lunghezza k (lunghezza di una gesture), che conterrà le derivate calcolate per ogni vettore.
   Eseguito per ogni frame:

   (a) *Calcolo della derivata:* Al frame i si calcola la derivata per l'i-esimo vettore e la si inserisce in cima alla coda scorrevole, la quale avanzerà di una posizione.

   (b) *Estrazione:* Quando anche il k-esimo vettore verrà codificato e inserito nella coda, quest'ultima verrà estratta e passata superiormente al *Tracker* come **Serie Temporale**.

   (c) *Reset dei parametri:* Una volta che l'estrazione è completa, l'indice i, dedicato al conteggio dei frame, viene resettato e si riparte.

L'implementazione Java si discosta poco dalle specifiche definite tramite UML; le uniche differenze le troviamo nella chiamata al metodo *notifyOnFrameChange* utilizzato per aggiornare il tracker e quindi l'interfaccia grafica interattivamente.

```
public class DerivativeCodifier implements Codifier {
    private Queue<Vector2D> featureVector;
    private Vector2D oldVector;
    private Vector2D derivative;
    private int frame;
    private GestureLength gestureLength;
    private TrackingObserver tracker;

    public DerivativeCodifier(final GestureLength frames) {
        this.featureVector =
            EvictingQueue.create(frames.getFrameNumber());
        this.oldVector = new Vector2D(0, 0);
        this.frame = 0;
        this.gestureLength = frames;
    }
    public void codifyOnSkeletonChange(final Vector2D newVector) {
        this.derivative = newVector.subtract(this.oldVector);
        this.featureVector.offer(this.derivative);
        this.oldVector = newVector;
        if (this.frame == 0) {
            this.startingVector = newVector;
        }
```

```
        if (this.frame > this.gestureLength.getFrameNumber() - 1) {
            this.tracker.notifyOnFeatureVectorEvent(new
                ArrayList<Vector2D>(this.featureVector));
            this.resetFrame();
        } else {
            this.tracker.notifyOnFrameChange(this.frame,
                this.featureVector, this.derivative,
                    this.startingVector.subtract(newVector));
            this.frame++;
        }
    }
    public void attacheTracker(final TrackingObserver recognizer) {
        this.recognizer = recognizer;
    }
}
```

Listato 7: La classe DerivativeCodifier, focus sull'algoritmo di codifica

5.3.3 Pattern Recognition

Nelle sezioni precedenti abbiamo visto come i vettori generati da Kinect vengano elaborati e aggregati al fine di costruire delle sequenze temporali. Una **time serie** non **è** altro che **una gesture nella sua forma più grezza**; il **sistema** infatti, a questo livello, **non è in grado di discernere i legami tra vettori di una serie temporale** e quindi risulta incapace di riconoscerla. Il nostro scopo, a questo punto del percorso, consiste nel fornire al sistema una *serie di esempi etichettati* in modo da portarlo piano piano a comprenderne i pattern sottostanti. L'idea che sta dietro a questo approccio risiede in una particolare sottoarea del *machine learning*, il **riconoscimento di pattern**.
Generalmente intendiamo come Pattern Recognition, *la costruzione di algoritmi che possano apprendere da un insieme di dati e fare delle predizioni su questi, costruendo in modo induttivo un modello basato su campioni.*
Nel nostro caso, l'apprendimento e il riconoscimento dei pattern non sarà basato su un modello ma avverrà tramite **matching diretto**, ossia la comparazione avrà luogo fra la time-serie (da classificare) e la collezione di esempi (**dataset**), al fine di trovare l'affinità migliore.

5.3.4 Acquisizione del Dataset

Come abbiamo appena accennato, per implementare un buon sistema di riconoscimento è necessario definire un metodo di acquisizione e di etichettatura per i pattern d'esempio. Relativamente al nostro framework si è deciso di

implementare un tool a parte dedicato a questo scopo, il **Recorder**. Grazie al recorder, il processo di acquisizione risulta estremamente semplificato ed è possibile registrare un intero dataset di gesture nel giro di pochi minuti. Gli step al fine di memorizzare un nuovo gesto, tramite il tool, sono molto semplici e possono essere schematizzati in questo modo:

- **Scelta dell'utente:** L'utente intenzionato a registrare un gesto deve innanzitutto creare uno profilo personale. La creazione di un utente genera un spazio dedicato solamente all'utilizzatore in questione; in particolare viene *generato un dataset* e vengono *inizializzati i parametri di riconoscimento*. Nel caso in cui il profilo sia già presente all'interno dell'applicativo, basterà selezionarlo (fig 5.9a).

- **Scelta della gesture da registrare:** Scelto il profilo utente e di conseguenza caricato il dataset corrispondente, bisognerà a questo punto registrare una gesture. Anche in questo caso sono possibili due strade: se la gesture non è definita sarà possibile crearne una personalizzata mentre, al contrario, se è già presente, basterà selezionarla (fig 5.9a).

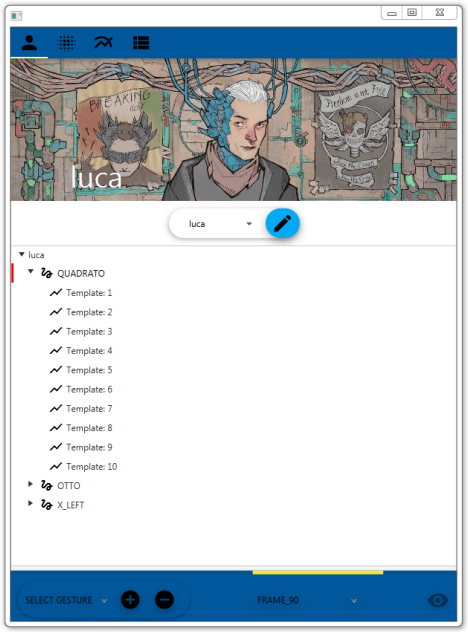
- **Setting del framerate:** E' importante sottolineare che, se necessario, è fornito un supporto a framerate diversi da quelli nativi; in questo caso non appena verrà registrata una gesture, il framerate verrà codificato sul profilo utente e non sarà più possibile modificare la frequenza di aggiornamento (fig 5.9a).

- **Accensione del sensore e visualizzazione del tracking:** Solamente dopo aver selezionato l'utente e un gesture sarà possibile abilitare il sensore e iniziare il tracciamento. L'interfaccia grafica è stata pensata per fornire un **tracking costante e multimodale** del gesto; in particolare sono presenti due visualizzazioni, una di tipo **starting-vector** fornita tramite *canvas* (fig 5.9b) e l'altra, di tipo **derivativo** graficata tramite *line-chart* (fig 5.9c).

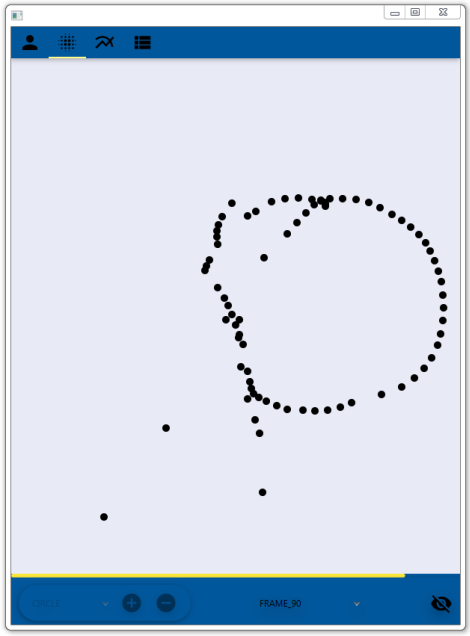
- **Abilitare il recording:** Dato che non è possibile, per ovvi motivi, abilitare il recording di una gesture tramite input da tastiera, è necessario fornire un supporto per un **innesco a distanza**. Per fare ciò è stato messo a punto un approccio ingegnoso che permette di **abilitare la registrazione semplicemente elevando il braccio sinistro** (o destro se si è scelto sul sensore il profilo di recognition con braccio sinistro).
  Nel momento in cui il braccio sinistro è alzato **una progress bar indicherà la percentuale di gesture acquisita**.

- **Selezione delle gesture:** Una volta che una gesture viene registrata, il sistema impila una sua rappresentazione (ottenuta tramite screenshot del

canvas) in una listview. Nel momento in cui si riterrà di aver registrato un congruo numero di gesture, si potrà selezionare quelle meglio riuscite per poi successivamente serializzarle nel dataset (fig 5.9d).

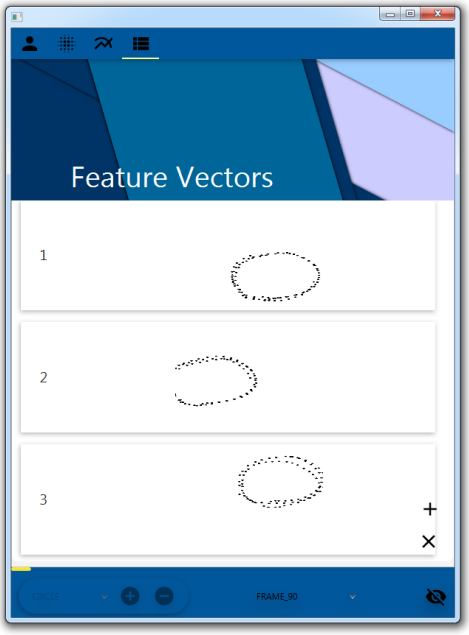


(a)



(b)

(c)



(d)

Il processo di gestione delle gesture è sicuramente l'aspetto più complicato per quanto riguarda il Recorder. A livello implementativo il Recorder rappresenta un'estensione della classe Tracker, di conseguenza tutta la logica di tracking e gestione delle sequenze è presente anche in questo contesto senza il minimo sforzo. Come ben intuibile non è possibile esporre dettagliatamente tutte le funzionalità implementate, di conseguenza ci limiteremo a fornire una descrizione tecnica limitatamente al processo di acquisizione e recording.

Focus sul processo di Recording

Il processo di recording, come abbiamo visto sopra, consiste nell'abilitare la registrazione semplicemente elevando il braccio sinistro. Come vediamo nel codice 8 al fine di garantire un comportamento ottimale, è stato introdotto un valore di soglia *THRESHOLD* utile per definire in modo netto quando la mano si trova al di sopra o al di sotto della spalla. Studiando nel dettaglio il metodo *Sec Joint Trigger* possiamo notare come l'innesco del riconoscimento avvenga attraverso l'uso di un semplicissimo flag di stato *isRecording*.
Nel complesso l'implementazione di questa funzionalità è risultata estremamente agevole; tramite il Recorder, infatti, è possibile appoggiarsi direttamente al livello di Tracking, ottenendo così un accesso semplificato al flusso Skeleton.

```
public final class RecorderImpl extends TrackerImpl implements
    Recorder {
    private static final int THRESHOLD = 50;
    public void attacheUI(final RecordingViewObserver view) {
        this.view.add(view);
        this.view.forEach(t -> t.refreshUsers());
    }
    public void notifyOnSkeletonChange(final Vector2D primaryJoint,
        final Vector2D secondaryJoint) {
        super.notifyOnSkeletonChange(primaryJoint, secondaryJoint);
        this.secJointTrigger(primaryJoint, secondaryJoint);
    }
    private void secJointTrigger(final Vector2D primaryJoint, final
        Vector2D secondaryJoint) {
        if (secondaryJoint.getY() > primaryJoint.getY() +
            RecorderImpl.THRESHOLD && !this.isRecording) {
            this.isRecording = true;
            this.view.forEach(t -> t.setRecording(true));
            this.resetCodificationFrame();
        } else if (primaryJoint.getY() - RecorderImpl.THRESHOLD >
            secondaryJoint.getY() && this.isRecording) {
            this.isRecording = false;
```

```
            this.view.forEach(t -> t.setRecording(false));
        }
    }
}
```

Listato 8: La classe DerivativeCodifier, focus sull'algoritmo di codifica

Focus sul processo di Salvataggio

Per quanto riguarda l'implementazione del processo di salvataggio gesture, abbiamo utilizzato intensamente i servizi di Serializzazione forniti dalla libreria. In particolare, come vediamo nel codice 9, abbiamo istanziato una lista di appoggio per le gesture salvate *listOfRecordedFeatureVector*, in modo da garantire una gestione online delle stesse prima della loro serializzazione.

```
public final class RecorderImpl extends TrackerImpl implements
    Recorder {
    private final List<List<Vector2D>> listOfRecordedFeatureVector;
    private RecorderImpl() {
        this.listOfRecordedFeatureVector = new ArrayList<>();
        this.userManager = new UserManager();
    }
    public void notifyOnFeatureVectorEvent(final List<Vector2D>
        featureVector) {
        if (this.isRecording) {
            this.view.forEach(t -> t.notifyOnFeatureVectorEvent());
            this.listOfRecordedFeatureVector
            .add(Collections.unmodifiableList(featureVector));
        }
    }
    public void addRecordedFeatureVector(final String gesture, final
        int index) throws IOException {
        this.userManager.addFeatureVectorAndSerialize(gesture,
            this.listOfRecordedFeatureVector.get(index));
    }
    public void addAllRecordedFeatureVectors(final String gesture)
        throws IOException {
        this.userManager.addAllFeatureVectorsAndSerialize(gesture,
            this.listOfRecordedFeatureVector);
    }
    public void deleteRecorderGestureDataset(final String
        gestureName) throws IOException {
        this.userManager.deleteGestureDataset(gestureName);
```

```
    }
    public void deleteRecordedGestureFeatureVector(final String
        gestureName, final int index) {
        this.userManager.deleteGestureFeatureVector(gestureName,
            index);
    }
}
```

Listato 9: La classe DerivativeCodifier, focus sull'algoritmo di codifica

I servizi di salvataggio proposti sono vari e operano essenzialmente nello stesso modo:

1. *Caching delle gesture:* Nel momento in cui l'utente avvia il sensore e alza il braccio sinistro, le gesture vengono temporaneamente acquisite nella lista di caching (*listOfRecordedFeatureVector*)

2. *Rimozione gesture indesiderate:* In qualsiasi istante, l'utente può visualizzare la lista di gesture salvate e, se necessario, eliminare quelle che ritiene imperfette. Il processo di eliminazione avviene cliccando con tasto destro la gesture desiderata; il Recorder in questo caso procede ad eseguire il metodo *deleteRecorderGestureDataset*, il quale elimina la gesture corrispondente.

3. *Salvataggio gesture:* Quando l'utente riterrà di aver registrato un congruo numero di gesture valide potrà salvarle. I metodi di salvataggio sono due e fanno riferimento ai metodi *add...* presenti nel codice. Cliccando una singola volta su una gesture verrà chiamato il metodo addRecordedFeatureVector che serializzerà la singola gesture. Clickando invece sul bottone $\oplus$ si avrà una completa serializzazione di tutti i template relativi alla gesture selezionata (metodo *addAllRecordedFeatureVectors*).

5.3.5 Riconoscimento di Gesture

Una doverosa premessa

Sebbene il riconoscimento di gesture costituisca il topic fondante dell'elaborato, esso si compone di una serie di **tecniche estremamente eterogenee**, molte delle quali esulano dagli ambiti di classificazione. I metodi messi in atto, in questo senso, hanno il solo scopo di **abilitare il riconoscimento**; il quale ricopre una piccolissima parte del lavoro svolto.
Ciò che abbiamo espresso sopra, rappresenta un monito per il lettore affinchè comprenda l'importanza dei **processi preparatori e abilitativi a una tecnologia** e non si focalizzi solamente sui risultati finali.

Il Recognizer

Il *Recognizer* costituisce l'ultimo step al riconoscimento; esso mette a frutto tutti gli sforzi fatti fino ad ora e li concentra al fine di raggiungere un solo obiettivo, l'identificazione di gesture. Dal punto di vista strutturale il Recognizer ricorda molto il *Recorder*; entrambi infatti rappresentano un'estensione della classe *Tracker* e si interfacciano con essa al fine di gestire le sequenze temporali e il dataset. L'unica differenza fra le due classi sta nel metodo di gestione del dataset: la prima, infatti, agisce in scrittura su di esso, mentre la seconda, in lettura.
Vediamo ora nel dettaglio come è stato sviluppato il processo di riconoscimento:

Definizione dei parametri di riconoscimento Come vediamo nel codice 10 al fine di implementare il recognizer è necessario definire una serie parametri utili al riconoscimento.

```
public final class Recognizer extends TrackerImpl implements
    Recognition {
    private final Serializer userManager;
    private final Set<RecognitionViewObserver> view;
    private final Set<GestureListener> gestureListener;
    private Map<Integer, List<Vector2D[]>> userDataset;
    private final Map<Integer, String> intToStringGestureMapping;
    private long lastGestureTime;
    private final DynamicTimeWarping<Vector2D> dtw;
    private RecognitionSettingsImpl recognitionSettings;
    private boolean gestureRecognized;
    private Recognizer() {
        this.view = new HashSet<>();
        this.gestureListener = new HashSet<>();
        this.intToStringGestureMapping = new HashMap<>();
        this.userManager = new UserManager();
        this.userDataset = null;
        this.lastGestureTime = 0;
        this.recognitionSettings = new RecognitionSettingsImpl
                                    (UpdateRate.FPS_10, 0, 0, 0, 0,
                                       0);
        this.dtw = new DynamicTimeWarping<Vector2D>((a, b) ->
            a.distance(b), this.recognitionSettings.getDtwRadius());
        RecognitionScreenView.startFxThread();
    }
    public void notifyOnSkeletonChange(final Vector2D primaryJoint,
                                    final Vector2D secondaryJoint) {
```

```
        super.notifyOnSkeletonChange(primaryJoint, secondaryJoint);
    }
}
```

Listato 10: Recognizer: inizializzazione delle variabili

- *Serializer:* E' il modulo dedicato al recupero e all'elaborazione del dataset al fine di renderlo compatibile con gli algoritmi di riconoscimento. Il dataset elaborato viene caricato in memoria come mappa di tipo <*Integer,List<String>>*, al fine di recuperare il nome delle gesture è stata affiancata al dataset una mappa di tipo <*Integer, String*> (*intToStringGestureMapping*) che garantisce una mappatura da codice numerico a nome di gesture.

- *DTW:* Rappresenta l'algoritmo utilizzato per la comparazione delle sequenze temporali.

- *Recognition Settings:* sono i parametri di riconoscimento; fra questi possiamo trovare i valori di soglia per il DTW, il numero di *nearest neighbor* di KNN e tanti altri.

Nel codice inoltre, è mostrato il metodo di tracciamento ed elaborazione di serie temporali ereditato dalla classe Tracker *NotifyOnSkeletonChange*.

Un Algoritmo per il Riconoscimento Instanziato il riconoscitore e agganciato il metodo di tracciamento, non manca che definire l'algoritmo di riconoscimento.
Nella sezione dedicata al Dynamic Time Warping abbiamo mostrato come è possibile implementare il riconoscitore tramite un approccio a *Matching Diretto*. Quello che andremo a fare ora consisterà nel definire una specifica completa e implementabile del processo; a tale scopo ci avvarremo dell'utilizzo di diagrammi UML.
Come illustrato nel diagramma di attività 5.10, i passi del riconoscimento sono i seguenti:

- *Codifica:* Come sappiamo, le operazioni di tracking e codifica sono attuate dal Tracker e di conseguenza sono presenti nel Recognizer.

- *Calcolo della Distanza:* Ottenute le serie temporali bisognerà effettuare un *matching* diretto con ogni template del dataset al fine di costruire una **lista di corrispondenze** di tipo (*Nome Gesture* $\rightarrow$ *Distanza*).

- *Classificazione:* Ottenuta la lista di distanze occorrerà, infine, definire un metodo per ottenere il template più simile alla gesture (time serie) da classificare. La scelta nel nostro caso è ricaduta sull'utilizzo dell'algoritmo KNN, poichè fornisce prestazioni molto elevate.

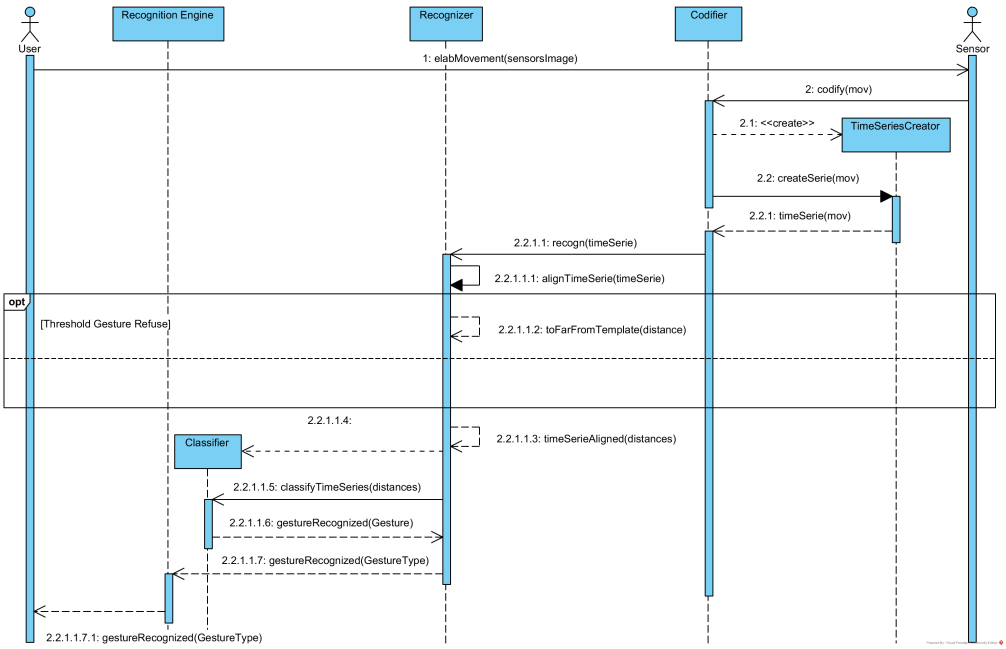


Figura 5.10: Diagramma UML di sequenza che descrive l'algoritmo di riconoscimento.

DTW

L'algoritmo di Dynamic Time Warping utilizzato è quello implementato all'interno di una nota libreria di machine learning scritta in java, **Smile** [22]. La versione proposta da Smile introduce una serie di ottimizzazioni volte a migliorare le prestazioni di DTW. Degna di nota è l'implementazione della già citata *Sakoe and Chiba Band*, che previene deformazioni del warping path eccessive e migliora i costi computazionali.

Modello *Gesture Avoidance* e introduzione della Soglia Poichè DTW rimane un algoritmo molto costoso, si è deciso di mettere in atto una serie di strategie per limitare il più possibile il suo utilizzo. A tale scopo è stato sviluppato un modello definito "*Gesture Avoidance*" atto a filtrare tutte le sequenze che non possono essere accettate come gesture, da qui il suo nome.
A livello di codice, l'algoritmo risulta implementato all'interno del metodo

notifyOnFrameChange, il quale permette la ricezione dei dati provenienti dal Codifier.

```
public void notifyOnFrameChange(final int frame,
     final Queue<Vector2D> featureVector, final Vector2D
        derivative, final Vector2D distanceVector) {
  if ((frame + 1) % this.settings.getUpdateRate().getFrameNumber()
     == 0) {
     final Vector2D[] arrayFeatureVector = new
        Vector2D[featureVector.size()];
     featureVector.toArray(arrayFeatureVector);
     final long currentSec = System.currentTimeMillis();
     if (this.gestureRecognized
           && currentSec - this.lastGestureTime >
              this.settings.getMinTimeSeparation()) {
        this.lastGestureTime = currentSec;
        this.recognize(arrayFeatureVector);
     } else if (!this.gestureRecognized) {
        this.lastGestureTime = currentSec;
        this.recognize(arrayFeatureVector);
     }
  }
   private void recognize(final Vector2D... featureVector) {
     final List<Pair<Double, Integer>> distances = new
        ArrayList<>();
     for (final Integer gestureKey : this.userDataset.keySet()) {
        for (final Vector2D[] gestureTemplate :
           this.userDataset.get(gestureKey)) {
           final double dtwDist = this.dtw.d(gestureTemplate,
              featureVector);
           if (dtwDist < this.settings.getMaxDTWThreashold()
                 && dtwDist > this.settings.
                                       getMinDtwThreashold()) {
              distances.add(new Pair<Double, Integer>(dtwDist,
                 gestureKey));
           }
        }
     }//......SEQUE KNN
```

Listato 11: Gesture Avoidance e DTW

Come si può osservare nel codice 11 l'algoritmo si appoggia principalmente su una serie di parametri definiti dall'utente. Tali parametri, presenti all'interno della classe *RecognitionSettings* hanno lo scopo principale di garantire una messa a punto ottimale del filtraggio delle gesture.
L'algoritmo di Gesture Avoidance è stato progettato in modo stratificato consentendo cosi un filtraggio delle gesture a pià livelli:

- *Update Rate:* L'update rate rappresenta la *velocità di campionamento* utilizzata dal riconoscitore. A velocità maggiori la reattività del sistema migliora a discapito però di **carico computazionale maggiore** (più confronti con l'intero dataset) e di una maggiore probabilità di **gesture-bouncing** (riconoscimento multiplo della stessa sequenza).

- *Time Separation:* Il problema del bouncing di gesture può essere gestito introducendo un **periodo di attesa** fra il riconoscimento di una gesture e la successiva,*lastGestureTime*. In particolare, per ogni frame viene calcolato l'intervallo di tempo che intercorre dall'ultimo evento di riconoscimento; se questo intervallo è maggiore di un parametro preimpostato (*MinTimeSeparation*) allora è possibile procedere al riconoscimento.

- *Calcolo distanza con DTW:* Arrivati a questo punto è finalmente possibile intraprendere il riconoscimento. A livello di codice questo passaggio è marcato con la chiamata al metodo privato *Recognize*.
  A questo punto avviene il cosiddetto *Direct Matching* in cui ogni singolo template del dataset viene confrontato con la sequenza da classificare calcolando una distanza.

- *Soglia DTW:* A questo punto entra in gioco la nostra soglia; il suo scopo è quello di filtrare ulteriormente le sequenze classificabili, rifiutando tutte quelle che hanno una distanza troppo elevata.

KNN

Ottenute tutte le sequenze e memorizzate le loro distanze in una lista dedicata, non rimane altro da fare che sviluppare una strategia per scoprire la classe del nostro template.
Per trovare la classe corrispondente alla sequenza in oggetto, un'idea consiste nello scegliere quella associata al template con la distanza minima. Tuttavia, questo approccio è incline a dare risposte errate, dato che la classe con distanza minima potrebbe essere diversa da quella codificata dalla sequenza. Un approccio più consistente risiede nell'utilizzo dell'algoritmo k-Nearest Neighbors (kNN) per eseguire la classificazione. Fissato un numero $k \in \mathbb{N}$, si selezionano i k template aventi distanza minima, e tra questi, si individua la classe avente

la frequenza massima (cioè quella presente più volte tra i k risultati). Questo è l'approccio utilizzato nel nostro framework (vedi 12).

```
private void recognize(final Vector2D... featureVector) {
    final List<Pair<Double, Integer>> distances = new ArrayList<>();
    \\DTW
    if (distances.size() > this.settings.getMatchNumber()) {
        distances.sort((a, b) -> a.getKey().compareTo(b.getKey()));
        final double[] kNearestNeighbor = new
            double[this.settings.getMatchNumber()];
        for (int i = 0; i < this.settings.getMatchNumber(); i++) {
            kNearestNeighbor[i] = distances.get(i).getValue() + 0.0;
        }
        final String gesture =
            this.intToStringGestureMapping.get((int)
            StatUtils.mode(kNearestNeighbor)[0]);
        this.gestureListener.forEach(t ->
            t.onGestureRecognized(gesture));
        this.view.forEach(t -> t.onGestureRecognized(gesture));
        this.gestureRecognized = true;
    } else {
        this.gestureRecognized = false;
    }
}
```

Listato 12: Algoritmo K-Nearest Neighbor

L'implementazione dell'algoritmo non ha presentato grosse difficoltà; esso consiste in un semplicissimo ciclo *for* all'interno del quale avviene l'estrazione, dalla lista ordinata delle distanze, dei primi k template più vicini. Raggruppati i k template in un array si può ricavare l'indice dela classe più frequente utilizzando la funzione di *Moda Campionaria* sull'array. Per ottenere il nome della classe basta effettuare un semplice matching inverso tramite *intToStringGestureMapping*.

5.4 Risultati Sperimentali e Sviluppi futuri

Prima di realizzare esperimenti su un sistema di riconoscimento è necessario effettuare il cosiddetto **Tuning dei Parametri**. Per ottenere una messa a punto perfetta, infatti bisognerà testare tutte le possibili configurazioni, studiando per ognuna di esse come cambia il tasso di riconoscimento. Per quanto riguarda la nostra libreria, l'aumento del tempo di separazione (*Time Separation*) delle gesture associato a un tempo di campionamento più breve hanno

permesso di limitare il gesture-bouncing migliorando cosi la precisione e la reattività del sistema (fig 5.11).

Figura 5.11: Schermata del framework dedicata al tuning dei parametri

5.4.1 Risultati Sperimentali

Affichè i risultati siano i più reali possibile si è deciso di svolgere tre tipologie di test sperimentali. I test hanno una doppia finalità: dal un lato permettono di conoscere le performance del sistema, dall'altro forniscono de-

gli input molto importanti riguardo a concetti come usabilità, flessibilità e soddisfazione utente.

Test Standard

Rappresenta un utilizzo tipico della libreria; l'utente registra un proprio dataset di gesture e effettua il tuning autonomamente. Il test mira a verificare il tasso di riconoscimento del sistema su 10 tentativi per ogni tipologia di gesture e non è possibile ripetere il riconoscimento.
Indichiamo con una **x** l'effettivo riconoscimento e con una **o** il mancato riconoscimento. Nel caso in cui sia presente un **falso positivo** lo evidenzieremo indicando l'indice della gesture riconosciuta.

Test standard: lunghezza gesture → 90 frame (3 secondi)

PROVE	1	2	3	4	5	6	7	8	9	10
(1) QUADRATO	x	x	x	o	x	x	x	x	x	x
(2) OTTO	x	x	x	x	x	x	x	x	x	x
(3) CERCHIO	x	x	x	x	o	x	x	x	x	x
(4) MEZZALUNA	x	x	x	x	x	x	x	x	x	x
(5) TRIANGOLO	x	x	o	o	x	x	o	x	x	x

Il test mostra come, con un singolo utente, le performance siano più che accettabili. Il rate di riconoscimento estremamente elevato dipende strettamente dalla lunghezza della gesture impostata. Con gesture molto lunghe infatti, la probabilità di falsi positivi scende enormemente a svantaggio però della reattività del sistema.

Test standard: lunghezza gesture → 30 frame (1 secondo)

PROVE	1	2	3	4	5	6	7	8	9	10
(1) QUADRATO	x	x	x	o	x	x	x	x	x	x
(2) OTTO	x	o	x	x	x	x	x	x	o	x
(3) CERCHIO	x	x	**4**	x	**4**	x	**4**	x	x	x
(4) MEZZALUNA	x	**3**	x	**3**	x	o	x	x	o	x
(5) TRIANGOLO	x	o	x	x	x	x	o	x	x	x

In questo caso il test è stato svolto su gesture di lunghezza standard (30 frame). Come vediamo sono presenti alcuni falsi positivi; il motivo della loro presenza dipende essenzialmente da due fattori. Il primo fattore risiede nel basso numero di feature che compongono una gesture (solo 30); basta infatti che alcuni vettori non vengano acquisiti correttamente e la distanza calcolata dal DTW risulterà completamente errata. Il secondo fattore consiste nella

similarità dell'esecuzione per alcune gesture. Per esempio in questo test la totalità di falsi positivi si sono verificati durante il riconoscimento del cerchio "Cerchio", gesture molto simile alla "Mezzaluna".

Test su Dataset Univoco

In questo test è stato distribuito a due utenti diversi un unico dataset preregistrato. A ogni utente è stato prima illustrato il movimento da effettuare e poi è stato lasciato libero di provare. Il test di riconoscimento consta di 10 tentativi per ogni tipologia di gesture e non è possibile ripetere il riconoscimento.

Utente 1

PROVE	1	2	3	4	5	6	7	8	9	10
(1) QUADRATO	x	x	x	o	x	x	o	x	o	x
(2) OTTO	x	x	o	x	o	x	o	x	o	x
(3) CERCHIO	x	x	o	x	x	x	o	x	o	x
(4) X	x	x	x	o	x	o	x	x	o	x
(5) Z	x	x	x	o	o	o	x	x	x	x

Utente 2

PROVE	1	2	3	4	5	6	7	8	9	10
(1) QUADRATO	x	o	o	x	x	x	o	x	o	x
(2) OTTO	x	o	x	x	x	x	o	o	o	o
(3) CERCHIO	x	x	x	x	o	x	x	x	o	o
(4) X	x	o	o	o	x	x	x	o	x	x
(5) Z	x	x	o	o	x	x	x	x	o	x

Come previsto, i tassi di riconoscimento di questo test sono molto bassi. Il problema in questo caso non dipende dal riconoscitore, bensì risiede nell'utente stesso il quale, non avendo avuto l'opportunità di registrare il proprio dataset, non è abituato al movimento richiesto.

5.4.2 Sviluppi Futuri

Il framework è stato sviluppato al fine di garantire una completa estendibilità futura. In particolare come abbiamo visto sarà possibile, in un futuro prossimo, progettare e sviluppare nuovi moduli sensore dedicati. Per esempio si potrebbe utilizzare il sensore Leap Motion al fine di estendere il riconoscimento alle singole dita della mano.
Per quanto riguarda il Riconoscitore, esso è stato progettato con l'obiettivo

di essere modulare ed espandibile. Alcuni sviluppi futuri prevederanno un miglioramento del tasso di riconoscimento grazie all'integrazione di sofisticati algoritmi di machine learning come le Reti Neurali o gli Hidden Markow Models.
Infine un ulteriore obiettivo futuro consisterà nel miglioramento del tool di recording, per esempio fornendo un supporto migliorato alla gestione del dataset di gesture.

Conclusioni

La mia valutazione sul percorso che si va concludendo non può essere che positiva. Inizialmente l'approccio con il progeto di tesi è stato comprensibilmente complicato; essere proiettati all'interno di argomenti che si studiano durante la magistrale non è affatto facile e all'inizio può portare a grandi frustrazioni. Queste frustrazioni ben presto però si sono tramutate in un vero e proprio spirito di avventura e frenesia nel raggiungere l'obiettivo.
Questa immensa positività (nata improvvisamente), legata alla mia grande voglia di sapere, hanno contribuito nel tempo ad incrementare la mia autonomia durante tutto lo sviluppo del framework. In particolare ho fatto tesoro di tutte le nozioni apprese durante i tre anni di studio e ho saputo applicarle nel momento di bisogno. Per esempio grazie al corso di Ingegneria del Software ho potuto implementare personalmente un modello prescrittivo. La documentazione sviluppata nel corso di tutto il progetto ha così costruito una routine di sviluppo estremamente solida e consistente.
Per quanto riguarda i requisiti funzionali definiti all'inizio del percorso, possono felicemente affermare che sono stati completamente soddisfatti. In alcuni casi i risultati dei test compiuti sono stati, infatti, al di sopra delle aspettative, producendo dati praticamente identici a quelli presenti in letteratura. Di questo però non c'è da stupirsi; il DTW associato all'algoritmo KNN rappresenta, infatti, una delle implemetazioni più performanti nel campo della classificazione di serie temporali.
Per concludere, posso affermare che mi ritengo interamente soddisfatto del cammino intrapreso; esco da questi mesi di duro lavoro sicuramente arricchito, sia sul piano culturale, sia nella consapevolezza dei miei mezzi e dei miei punti di forza.

Ringraziamenti

Un ringraziamento particolare va alla mia famiglia, che mi ha sempre supportato nel percorso che ha portato al lavoro di tesi.
Ringrazio la Prof. Annalisa Franco e il Prof Alessandro Ricci per avermi dato l'opportunità di mettermi in gioco, sviluppando questo progetto.
Ci tengo, inoltre, a ringraziare la mia ragazza perchè se io sono arrivato fino a questo punto un pò è anche grazie a lei. Un ultimo ringraziamento va ai miei amici che mi hanno sempre incoraggiato durante tutta questa avventura.

Bibliografia

[1] Peter Smart *Design for the Future of Human-Computer Interaction*, Fantasy Interactive, 2017.

[2] Myers Brad *A Brief History of Human-Computer Interaction Technology*, 1998.

[3] Ping Zhang, Dennis Galletta, *Value Sensitive Design and Information Systems*, Anno.

[4] Ram Pratap Sharma, Gyanendra K. Verma, *Human Computer Interaction using Hand Gesture*, IMCIP, 2015.

[5] Andrea Corradini, *Real-Time Gesture Recognition by Means of Hybrid Recognizers*, Springer-Verlag, 2001.

[6] William C. Stokoe, *Sign Language Structure: An Outline of the Visual Communication Systems of the American Deaf*, Oxford University Press , 2005.

[7] Scott K Liddell, Robert E Johnson, *American sign language: The phonological base. Sign Language Studies*, Gallaudet University Press, 1989.

[8] Feng Jiang, Shengping Zhang, Shen Wu, Yang Gao, Debin Zhao *Multi-layered Gesture Recognition with Kinect*, Isabelle Guyon, Vassilis Athitsos, and Sergio Escalera, 2015.

[9] Hong Cheng, Lu. Yang, Zicheng Liu, *A Survey on 3D Hand Gesture Recognition*, 2015.

[10] L. Cheng, Q. Sun, H. Su, Y. Cong, and S. Zhao, *Design and implementation of human-robot interactive demonstration system based on kinect*, In Control and De- cision Conference (CCDC),2012.

[11] R. A. Clark, Y.-H. Pua, A. L. Bryant, and M. A. Hunt, *Validity of the microsoft kinect for providing lateral trunk lean feedback during gait retraining*, 2013.

[12] T. Hachaj, M. R. Ogiela, *Rule-based approach to recognizing human body poses and gestures in real time*, Multimedia Systems, 2014.

[13] Nicholas Gillian, R. Benjamin Knapp, Sile O'Modhrain, *Recognition Of Multivariate Temporal Musical Gestures Using N-Dimensional Dynamic Time Warping.*

[14] Meinard Muller, *Information Retrieval for Music and Motion*, Springer, 2007.

[15] E. Keogh and M. Pazzani, *Iterative deepening dynamic time warping for time series*, Conf. on Data Mining, 2002.

[16] H. Sakoe, S. Chiba, *Dynamic programming algorithm optimization for spoken word recognition*, Transactions on Acoustics, Speech and Signal Processing, 1978.

[17] Roanna Lun, Wenbing Zhao, *A Survey of Applications and Human Motion Recognition with Microsoft Kinect*, Cleveland State University, 2015.

[18] Prashan Premaratne, *Human Computer Interaction Using Hand Gestures*, Series Editor, 2014.

[19] Abhijit Jana, *Kinect for Windows SDK Programming Guide*, Pckt publishing, 2012.

[20] Angelos Barmpoutis, *J4K*, *Sito Internet*

[21] Angelos Barmpoutis, *Tensor Body: Real-Time Reconstruction of the Human Body and Avatar Synthesis From RGB-D*, IEEE, 2013.

[22] Haifengl, *Smile - Statistical Machine Intelligence and Learning Engine*, *Sito Internet*