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 IN AMBIENTE ANDROID PER LA REALIZZAZIONE DI INTERFACCE UTENTE SEE-THROUGH E HANDS-FREE

Elaborato in
PROGRAMMAZIONE DI SISTEMI EMBEDDED

Relatore
Prof. ALESSANDRO RICCI

Co-relatore
Ing. ANGELO CROATTI

Presentata da
FEDERICO GIANNONI

Prima Sessione di Laurea
Anno Accademico 2015 – 2016

PAROLE CHIAVE

Hands Free Interaction

Augmented Virtuality

Wearable Computing

User Interface

Android

"You miss 100% of the shots you don't take"

Indice

Introduzione

"The most profound techonlogies are those that disappear[...] They weave themselves into the fabric of everyday life until they are undistinguishable from it [...]" [5], così scriveva Mark Weiser nel 1991, illustrando la sua idea di una completa pervasività computazionale, nella quale il mondo digitale permea quello fisico in maniera del tutto trasparente, mettendo a disposizione degli utenti le funzionalità di cui esso dispone, in modo che esse siano utilizzabili in maniera diretta e spontanea.

Benché ci si trovi ancora lontani dalla realtà concepita da Weiser, ci si sta muovendo sempre di più verso di essa. Tale affermazione trova riscontro nella continua introduzione di nuove forme di tecnologia, sempre più simili ad oggetti appartenenti al mondo reale (*smart objects*), che sta portando ad un progressivo assottigliamento del confine tra realtà e virtualità.
Questa tesi si colloca nell'ambito di un settore specifico di queste nuove tecnologie: quello del *wearable computing*. Un dispositivo wearable, è un dispositivo indossabile, che punta a rendere più naturale ed immediata l'interazione con il mondo digitale e ad aumentare le capacità di chi lo indossa.
Nonostante i numerosi vantaggi che l'utilizzo di questi dispositivi possa portare, specialmente in ambito lavorativo, la loro diffusione è ancora limitata, in particolar modo dalla mancanza di applicazioni designate su misura per questi device.

Un'applicazione progettata appositamente per uno wearable, dovrebbe fare leva su quelli che sono i suoi punti di forza, adattarsi alle sue specifiche e al suo funzionamento. Questo però, non è semplice. Per incoraggiare lo sviluppo di sistemi di questo tipo, ci si è proposti di realizzare un framework che renda possibile la creazione di interfacce utente, pensate appositamente per adattarsi alle caratteristiche dei dispositivi wearable. Queste interfacce, sono pensate per essere il meno intrusive e il più dirette possibile, qualità che possono fare la differenza, specialmente per quanto riguarda alcuni campi lavorativi.

Capitolo 1

Human Computer Interaction e Wearable

In questo primo Capitolo introduttivo, contestualizzerò il lavoro svolto illustrando brevemente il campo della *Human Computer Interaction (HCI)* ed i suoi concetti chiave, esponendo anche quella che è la situazione attuale nel campo dell'interazione con le nuove tecnologie. La trattazione del Capitolo, proseguirà poi spostandosi proprio verso queste ultime, o meglio, verso il settore delle *Wearable Technologies* e dei dispositivi *Hands Free*.

1.1 Il campo della HCI

La Human Computer Interaction, è il campo che si occupa dello studio della pratica dell'usabilità in ambito informatico. In altre parole, è un ramo dell'ingegneria rivolto allo studio e alla creazione di interfacce hardware o software per dispositivi elettronici, che risultino essere non solo ***"easy to learn, easy to use"***, ma anche **comode**, **accattivanti** (tanto da invogliare l'utente a volerle utilizzare) ed **efficaci** nello svolgere i compiti per cui sono state progettate.

1.1.1 Il concetto di Human Centered Design

Dalla definizione proposta, risulta facile notare come il campo della HCI studi l'interazione tra uomo e macchina concentrandosi su quello che è il punto di vista dell'utente, per il quale appunto, le interfacce sono progettate. Questo tipo di approccio, che mette in primo piano i bisogni e le necessità dell'utilizzatore, viene chiamato *Human Centered* ed ha come obiettivo la realizzazione di *Human Centered Systems*, termine spesso usato in maniera non consona

per indicare un qualunque sistema che possa risultare in qualche modo "utile" all'uomo.

In realtà, progettare un sistema **centrato** sull'uomo, significa realizzare un sistema **calibrato** sulla base di quelle che sono le sue limitazioni e le sue capacità, in modo da **adattarsi** ad esse ed eventualmente **completarle** o **aumentarle**. Per riuscire a produrre tali risultati quindi, il settore della HCI deve preoccuparsi anche di studiare le particolarità dell'uomo, i suoi processi cognitivi e i suoi pattern d'interazione, fondendo così tematiche provenienti dal campo della psicologia cognitiva, ad altre relative invece a quello della *Human Factors Engineering* (campo dell'ingegneria che si occupa di migliorare ed ottimizzare le performance di sistemi che prevedono interazioni con l'uomo, proprio considerando e studiando quelle che sono le capacità e le limitazioni dell'uomo all'interno del sistema).

1.1.2 HCI e le nuove tecnologie

Nel corso degli anni, a causa della continua introduzione di nuove forme di tecnologia, l'interazione uomo–macchina è profondamente cambiata ed il settore della HCI ha dovuto adattarsi di conseguenza, facendo fronte a sempre nuove sfide. Al giorno d'oggi, grazie soprattutto alla diffusione di *cellular networks*, *personal area networks*, del *Cloud* e del *mobile computing*, abbiamo a che fare con dispositivi di ogni forma e funzionalità, che pervadono la nostra realtà quotidiana e sono in grado di comunicare tra loro per offrirci qualsivoglia tipo di servizio.

Ci si sta, così, avvicinando sempre di più alla visione che Mark Weiser espresse nel 1991, di un mondo in cui fisico e virtuale non sono più l'uno il contrario dell'altro, bensì un tutt'uno; ovvero un mondo in cui la tecnologia **permea la realtà** in maniera "invisibile" agli occhi dell'uomo, che *"può così farne uso senza fatica, concentrandosi esclusivamente su quelli che sono i suoi obiettivi"* [5]. I rapidi avanzamenti nel campo della *realtà aumentata* e la diffusione dei cosiddetti *smart objects*, sono due segni a testimonianza di questo movimento verso un nuovo paradigma di pervasività computazionale e sono anche topic rilevanti nell'ambito di questa tesi. Lasciando per ora in sospeso il concetto di *augmented reality*, che verrà ripreso ed approfondito nel Capitolo successivo, concentriamoci invece sugli smart objects.

Uno smart object, è un oggetto appartenente alla realtà quotidiana le cui capacità sono state **potenziate** tramite interfacciamento con il mondo digitale. Questi oggetti, sono in grado di offrire funzionalità aggiuntive oltre a quelle che già mettevano a disposizione; possono *fare di più rispetto a quello per cui erano stati originariamente ideati.* La Figura 1.1 riporta un esempio di smart object, più in particolare di uno weather forecasting umbrella, ovvero

un ombrello in grado di comunicare all'utente la probabilità di precipitazioni nella zona tramite l'intensità di illuminazione del suo manico.

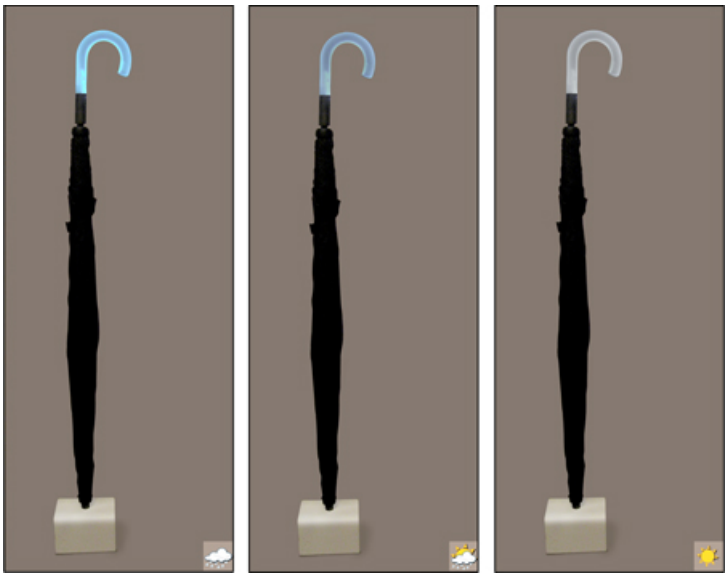


Figura 1.1: *Weather forecasting umbrella, il cui manico si illumina a seconda della probabilità di precipitazioni nella zona in cui si trova.*

1.2 Il settore Wearable

Tra le numerose categorie di smart object che sono emerse e che stanno tutt'ora emergendo nel mercato e in ambito lavorativo, quella più rilevante nell'ambito di questa tesi, è la categoria delle tecnologie wearable. Benché nel corso degli anni siano state offerte più definizioni che si discostano leggermente l'una dall'altra, possiamo vedere un sistema wearable come un **dispositivo elettronico indossabile** e quindi a stretto contatto con l'utilizzatore.

1.2.1 Requisiti funzionali di un dispositivo wearable

Come qualsiasi altra forma di tecnologia, anche gli wearable sono pensati per sostenere ed aiutare chi li utilizza. Più in particolare, questi dispositivi vengono progettati per rispondere alle esigenze dei cosiddetti *on-the-go users*, ossia utenti in movimento, impegnati nello svolgimento di altri task e quindi con capacità di interazione, concentrazione e mobilità, ridotte rispetto al normale. Un dispositivo wearable deve **compensare** quelli che sono **i deficit di un utilizzatore on-the-go** ed aiutarlo nell'esecuzione delle sue attività.

Per poter raggiungere un tale obiettivo, esso deve innanzitutto mettere a disposizione un sistema di **interfacciamento diretto** e rapido, per far fronte al fatto che l'utente, già impegnato, non potrà dedicare lunghi periodi di tempo all'interazione col dispositivo. Allo stesso tempo però, tale interfaccia **non dovrà risultare intrusiva** o monopolizzare l'attenzione dell'utilizzatore, distogliendola da ciò che egli stava facendo.

Un altro requisito importante per un dispositivo wearable, è quello di essere in grado di monitorare l'ambiente, ovvero essere **context-aware**. Il termine *context-awareness* viene utilizzato in ambito informatico per descrivere la capacità di un sistema di percepire (tramite l'impiego di opportuni sensori) e comprendere la realtà in cui esso è immerso, così da modificare il suo comportamento in base ad essa. Un dispositivo wearable può fare uso di una tale funzionalità per incrementare le capacità sensoriali dell'utente (tramite feedback audio o visivi), che risultano essere in gran parte concentrate sullo svolgimento di altri compiti; oppure per capire quando poter interagire con esso senza risultare intrusivo.

Per finire, uno wearable deve essere **sempre in funzione** e risultare **completamente osservabile e controllabile** dal punto di vista dell'utente, in modo che esso possa eventualmente correggerne o modificarne il comportamento in base alle sue esigenze, quando il dispositivo non è in grado di farlo da solo.

1.2.2 Requisiti tecnici di un dispositivo wearable

Aldilà di quelli che sono i requisiti funzionali, un dispositivo wearable deve anche soddisfare, come ogni altro dispositivo appartenente all'ambito del mobile computing, determinati requisiti tecnici. Primo fra tutti, il dover trovare un giusto **compromesso tra dimensioni, capacità di calcolo, connettività, consumo e dissipazione di calore**.

Idealmente, dal momento che gli wearable sono dispositivi indossabili, dovrebbero avere **dimensioni ridotte**, così da migliorare la loro comodità di utilizzo e ridurre la loro intrusività. Tuttavia, progettare sistemi di dissipazione adeguati per device piccoli, risulta essere alquanto complicato e ciò va tenuto fortemente in considerazione, dal momento che gli wearable, essendo a contatto con l'epidermide, **non dovrebbero mai superare i 40 gradi Celsius**, sempre per motivi relativi al comfort, ma anche alla sicurezza dell'utente.

Tutto ciò va così a pesare su quelle che sono le capacità di calcolo e di autonomia di questi dispositivi. Dal momento che le loro dimensioni ridotte comportano grandi difficoltà nella gestione del calore, l'unico modo per evitare che le temperature di questi device vadano oltre la soglia sopra menzionata è quello di **utilizzare hardware a basso consumo** e quindi, generalmente

poco efficiente. Questo vale anche per il sistema di alimentazione che però deve essere in grado di **far fronte al requisito funzionale di costanza** che uno wearable deve avere e che abbiamo già menzionato.

Il discorso dissipazione e autonomia vanno ad influire anche su quelle che sono le capacità di connettività di questo tipo di dispositivi, che devono essere in grado di comunicare con altri. Come sappiamo, l'utilizzo della rete risulta essere altamente dispendioso in termini di alimentazione e porta anche ad un notevole riscaldamento dei device e va quindi, tenuto in considerazione.
Teniamo inoltre a mente, che più piccole risultano essere le dimensioni del dispositivo, maggiori saranno le difficoltà nell'interazione con esso, a causa delle ridotte dimensioni del display e delle periferiche di input.

Possiamo vedere, arrivati a questo punto, quanto possa essere complicato progettare un sistema wearable. Va inoltre considerato il fatto, che per questo genere di dispositivi, una forte importanza va attribuita anche al **design**. Dal momento che essi devono essere indossati, non devono solo soddisfare requisiti di comodità, ma **devono anche avere un certo appeal estetico** per invogliare l'utente a volerli utilizzare.

1.2.3 Il concetto di Human Augmentation

Benché pensati per far fronte alle esigenze dell'on-the-go user, i sistemi wearable vengono anche impiegati come *Human Enhancing Technologies* (HET). La *Human Enhancement* o *Human Augmentation*, è un campo emergente nel settore della medicina e della bioingegneria, che ha come obiettivo il **potenziamento delle capacità dell'uomo** oltre quelli che sono i suoi limiti naturali, sfruttando gli avanzamenti nei diversi campi dell'ingegneria, delle nanotecnologie, della cibernetica e della farmacologia.

Quando si parla di "aumento delle capacità", ci si riferisce, non solo a quelle fisiche, ma anche a quelli sensoriali, emotive e mentali. Le possibili applicazioni nel campo della human enhancement sono pressoché illimitate e si basano tutte sul raggiungimento della cosiddetta *man–computer symbiosis*, ossia un perfetto mix tra uomo e macchina, o, se vogliamo, un "essere umano aumentato". Thad Starner, uno dei maggiori esponenti nel campo dello wearable computing, definì proprio il concetto di una man–computer symbiosis come *"un traguardo ottenibile tramite l'impiego di un dispositivo wearable ipoteticamente ideale"* [7]. Benché ancora distanti dal raggiungimento della visione di Starner, negli ultimi anni sono stati fatti grandi passi verso di essa, grazie anche a sviluppi nel settore delle nanotecnologie, che hanno dato inizio al movimento dei dispositivi wearable da fuori, a dentro il corpo umano. Un esempio di progetto per tecnologie wearable sottopelle è illustrato in Figura 1.2.

Figura 1.2: *Dai creatori di Fitbit, uno dei primi concetti di tecnologia wearable sotto pelle. Underskin è un dispositivo pensato come tatuaggio digitale; impiantato nel palmo dell'utilizzatore ed alimentato esclusivamente dall'energia elettro-chimica del suo corpo, è pensato per essere in grado di interagire con altri dispositivi tramite il tocco, spedendo segnali NFC e rendendo possibile, ad esempio, sbloccare le proprie serrature semplicemente toccandole, scambiare dati con un altro utilizzatore tramite una semplice stretta di mano, o fare in modo che la propria carta di credito funzioni solo quando a contatto con la propria mano.*

1.2.4 Wearable nel mercato

Per quanto riguarda l'influenza dei sistemi wearable nel mercato, benché questa risulti essere in continua crescita, è ancora limitata dall'esistenza di barriere che inibiscono il loro utilizzo da parte dei consumatori. Alla base di queste barriere vi è un problema di percezione, dovuto al fatto che la maggior parte dei potenziali utenti, non riesce a vedere quali possano essere i benefici nell'utilizzo di questi dispositivi. Altri **problemi che limitano la diffusione degli wearable**, includono:

- La **necessità di dover spesso essere connessi ad uno smartphone** o ad un tablet per fornire determinate funzionalità;
- Il loro **prezzo elevato**;
- La **carenza di funzionalità e di appeal estetico**;
- Le **limitate dimensioni del display**;
- Uno **scarso tempo di autonomia**;

- La **mancanza di "killer app"**;
- La **manca di un interfacciamento efficiente**;
- L'**esistenza di problemi legati alla privacy**[1].

Nonostante questi fattori limitanti, il mercato degli wearable risulta comunque essere in rapida crescita, grazie anche all'ingresso nel settore di grandi compagnie quali Apple e Google. I dati dei rapporti del Business Insider Intelligence [10] riportano un aumento di oltre cinquanta milioni di dispositivi venduti negli ultimi cinque anni, ovvero un incremento di circa dieci milioni ogni anno. Sempre basandoci sulle stesse fonti, possiamo osservare che il settore continuerà a crescere in maniera esponenziale nei prossimi tre anni (come illustrato in Figura 1.4).

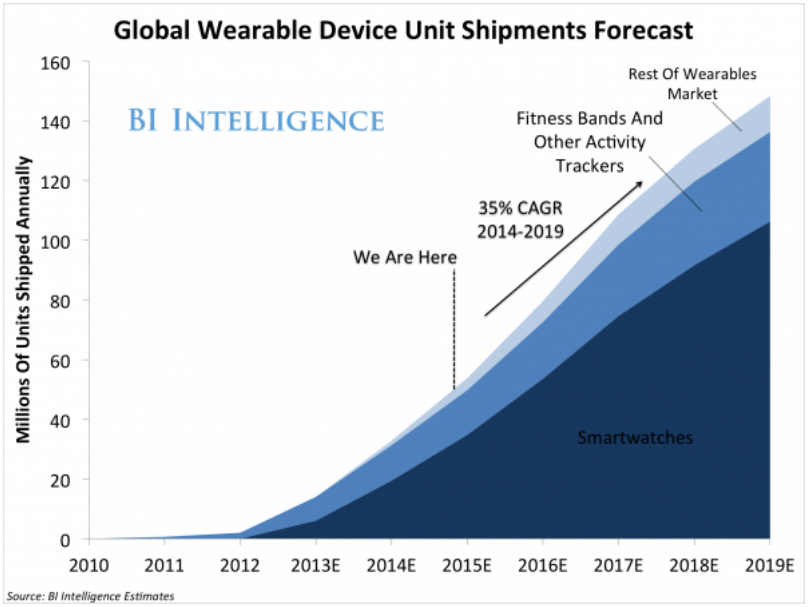


Figura 1.3: *Report e proiezioni del mercato degli wearable dal 2010 al 2019. I dati sono relativi all'anno 2014, nel quale sono stati venduti oltre 33 milioni di device. Il grafo riporta una crescita del mercato con un tasso annuale composto pari al 35%, secondo la quale, nel corso dell'anno 2019 verranno venduti quasi 150 milioni di dispositivi.*

Osservando il grafico possiamo notare come la fetta più grande di dispositivi wearable in ambito commerciale sia costituita dagli smartwatch, che continueranno, anche negli anni successivi, a dominare il settore. Tra i vari produttori

[1] `http://www.abajournal.com/magazine/article/google_glass_is_already_causing_legal_experts_to_see_problems`

di smartwatch inoltre, le statistiche suggeriscono che saranno proprio Apple e Google a sovrastare la concorrenza. A seguire troviamo i fitband, dispositivi utilizzati in ambito sportivo, seguiti da *smart eyewear* e poi da tutte le altre tipologie di device.

Figura 1.4: *L'immagine mostra un confronto tra due smartwatch: un Samsung Gear S (a sinistra), dotato di sistema operativo Android e un Applewatch (a destra), basato invece su iOS.*

1.2.5 Wearable nel mondo del lavoro

Dal momento che sono pensati per sostenere un utente mentre questo sta svolgendo altri compiti, gli wearable sembrerebbero essere perfetti per l'utilizzo in campo lavorativo. Tuttavia, l'espansione di questi dispositivi nel mondo del business, è ancora molto limitata, principalmente a causa degli stessi fattori che inibiscono il loro utilizzo in campo commerciale.
Questi vincoli vanno a pesare ancora di più quando si tratta di utilizzo in ambito lavorativo, dove il funzionamento corretto e costante del dispositivo, diventa assolutamente vitale per fare in modo che si possa dipendere su esso.

In ogni caso, il loro impiego ha anche qui cominciato a prendere lentamente piede. In particolare, per quanto riguarda il mondo del lavoro, uno tra i settori nel quale l'uso delle tecnologie wearable sta aumentando in maniera sensibile, è quello dell'healthcare, dove il loro utilizzo come HET (Human Enhancing Technologies) sta rendendo possibili cose sempre più stupefacenti. Principalmente, l'uso degli wearable in ambito medico, ha come scopo la trattazione di deficit, non più temporanei dell'utente on-the-go, bensì permanenti, dovuti a malattie, malformazioni o incidenti.
In Figura 1.5 possiamo vedere un'immagine esemplificativa dell'utilizzo di queste tecnologie per migliorare la qualità di vita di persone con handicap signifi-

cativi[2]. Più in particolare, l'immagine mostra l'impiego di due dispositivi Myo per far muovere un braccio prostetico tramite impulsi nervosi.

Figura 1.5: *I dispositivi Myo qui illustrati sono dispositivi in grado di captare i segnali elettrici che percorrono il sistema nervoso e muscolare del braccio. Le registrazioni prodotte da entrambi vengono combinate per far muovere la protesi come se questa fosse un braccio vero. Il movimento risulta essere ancora poco fluido ma incredibilmente preciso.*

Possiamo vedere come in questo campo, i sistemi wearable diventino qualcosa di più di semplici dispositivi, ma siano piuttosto delle vere e proprie estensioni del corpo dell'utilizzatore, che ne fa uso senza dovervisi concentrare. Il settore healthcare inoltre, beneficerà enormemente anche del movimento sottopelle di questi dispositivi, di cui abbiamo già parlato nella Sezione 1.2.3. Ricordiamo tuttavia, che il campo medico è il campo nel quale è richiesta massima affidabilità e che quindi, saranno ancora necessari svariati progressi prima di poter assistere ad una diffusione pervasiva di wearable nel settore.

Oltre che in campo sanitario, l'impiego di questi sistemi sta lentamente prendendo piede anche in altri ambiti, quale quello industriale o dei trasporti (esempi in Figura 1.6) e con opportuni miglioramenti e abbassamenti dei prezzi nel corso dei prossimi anni, non vi è dubbio che i dispositivi wearable si affermeranno come uno standard nel mondo lavorativo.

[2] `https://www.youtube.com/watch?v=LSuzMxQDmzg`

(a) DAQRI Smart Helmet.

(b) Smartcap.

Figura 1.6: *In Figura 1.8a è rappresentato uno smart helmet. Questo dispositivo, pensato per l'utilizzo in ambito industriale, ha la capacità di connettere il suo wearer all'ambiente lavorativo e alle relative infrastrutture digitali, fornendogli informazioni sul contesto in cui è immerso ed aiutandolo dunque nello svolgimento delle sue mansioni. In Figura 1.8b invece, viene mostrato uno smart cap, dispositivo pensato per monitorare lo stato di attenzione del suo utilizzatore e tipicamente utilizzato nel mondo dei trasporti. Esso è in grado di prevedere, grazie ad opportuni sensori, quando il pilota, che sta indossando il cappello, sta per avere un colpo di sonno, riducendo così incidenti dovuti alla fatica.*

1.3 Hands Free Interaction

Al fine di garantire un'interazione rapida, semplice e diretta, rispettando quelli che sono alcuni dei suoi requisiti funzionali, un sistema wearable aspira spesso ad essere *hands free*, ovvero utilizzabile senza richiedere l'utilizzo delle mani da parte dell'utente. Anche il semplice maneggiare un touchscreen infatti, potrebbe richiedere un livello di concentrazione e mobilità che in molti contesti on-the-go non è possibile avere. Proprio per questo, i sistemi wearable sono costretti a discostarsi da quelli che sono i paradigmi tradizionali di interazione, muovendosi verso l'esplorazione di nuove interfacce per input e output.

Tra queste, per quello che riguarda il **mobile input**, il riconoscimento vocale risulta essere una soluzione sempre più invitante, grazie al continuo miglioramento degli algoritmi, delle reti cellulari e all'impiego del Cloud per poter delegare l'elaborazione del suono a server anziché al proprio dispositivo. Anche l'interfacciamento vocale tuttavia ha dei lati negativi, primo fra tutti il fatto che non risulta essere utilizzabile in luoghi rumorosi e quindi, ad esempio, in una buona parte dei settori lavorativi (industriale, edile, ecc...).

1.3.1 Gesture Recognizers

Un'alternativa all'utilizzo della voce, è l'impiego di gesture. Esistono dispositivi in grado di intercettare e decodificare i segnali elettrici propagati attraverso il nostro sistema nervoso e muscolare ed associarli all'esecuzione di un determinato gesto. Un esempio è offerto dal dispositivo Myo[3], dei Thalmic Labs, che è in grado di riconoscere cinque gesture diverse effettuate utilizzando la mano. Benché questi dispositivi, dal momento che richiedono l'utilizzo delle mani, non siano effettivamente hands free, sono comunque utilizzabili in maniera facile ed immediata anche in contesti dove mobilità e concentrazione sono ridotti al minimo.

Figura 1.7: *Il Myo è un dispositivo wearable da indossare sul braccio. Grazie ad otto sensori EMG posizionati sulle placche esterne, è in grado di captare e registrare le propagazioni dei segnali elettrici che percorrono le terminazioni nervose e la muscolatura del braccio, ricombinandoli poi per individuare quando viene effettuata una determinata gesture. Il dispositivo monta anche un sistema IMU a 9 assi (accelerometro, giroscopio e magnetometro tutti a 3 assi), rendendo così possibile anche tracciare il movimento del braccio.*
Il Myo non dispone di una scheda di rete, ma può connettersi a qualunque altro dispositivo che disponga di Bluetooth Low Energy.

1.3.2 Head Mounted Displays

Per quello che riguarda le interfacce di output, l'opzione più utilizzata in campo wearable, che garantisce un'interazione immediata e hands free, è costituita dagli *Head Mounted Display* (HMD).
Questa categoria di dispositivi, viene indossata direttamente sulla testa e pre-

[3] `https://www.myo.com/`

senta un display posizionato davanti ad uno (*monocular HMD*) o entrambi (*binocular HMD*) gli occhi.

L'idea di head mounted display risale al 1960 e fu concepita da un cinematografo di nome Morton Heilig. Le prime sperimentazioni con questi dispositivi presero il via circa un decennio dopo in campo militare, per poi sbarcare in quello commerciale attorno agli anni '80 - '90.
Nonostante la tecnologia sia stata a nostra disposizione ormai da decenni, è solo recentemente che questi dispositivi hanno cominciato a diffondersi drasticamente. Questo è dovuto in parte alla loro applicazione nei vari campi della Mixed Reality, che analizzeremo nel Capitolo 2.

Al giorno d'oggi esistono diverse categorie di HMD, le cui particolarità sono dettate dal loro contesto applicativo. Alcune di queste sono:

- **Smart glasses**, dispositivi *see-through* molto simili a degli occhiali. Sono caratterizzati da display e campo visivo tipicamente molto piccoli (di solito intorno ai 20 gradi per quanto riguarda il campo visivo, e risoluzioni attorno a 500x500 pixel per occhio per quello che riguarda il display). Tutto ciò significa che le immagini digitali riprodotte da questi dispositivi, appaiono come racchiuse all'interno di una piccola finestrella sul mondo reale e sono più o meno trasparenti, rendendo possibile all'utente di vedere anche ciò che sta oltre ad esse, in modo da non risultare intrusivi, facendo perdere contatto con la realtà fisica.
  Alcuni smart glass inoltre, offrono la possibilità di montare lenti da vista.

(a) Google Glass. (b) Epson Moverio BT200.

Figura 1.8: *Confronto tra Google Glass ed Epson Moverio BT200. Entrambi i dispositivi sono forniti di sistema operativo Android (4.4 per i Google Glass e 4.0.3 per i Moverio), sistema IMU a 9 assi (comprensivo di accelerometro, giroscopio e magnetometro a 3 assi) e fotocamera. Per quando riguarda i display, si parla di una risoluzione di 640x360 pixel con campo visivo monoculare pari a 14 gradi per i Google Glass, contro i 960x540 pixel con campo visivo binoculare (480x540 per ogni occhio) e campo visivo pari a 23 gradi per i Moverio.*

- **Virtual Reality gaming headsets**, dispositivi ancora pesanti ed ingombranti, ma con campi visivi molto ampi. Sono tipicamente binoculari ed includono un impianto audio stereo 3D. Essendo impiegati nel campo della realtà virtuale, non sono see-through e forniscono un esempio di un ulteriore campo applicativo per le tecnologie wearable.

Figura 1.9: *Oculus Rift, noto headset per virtual reality gaming.*

- **Headsets per la realtà aumentata**, dispositivi see-through più grandi rispetto a degli smart glasses e quindi con una maggiore capacità computazionale. Tipicamente si trovano nella forma di caschi e sono utilizzati in alcuni campi, ancora ristretti, del mondo del lavoro, come il campo medico, di salvataggio, logistico e in vari settori dell'ingegneria. Sono utilizzati, come suggerisce il nome, per fare realtà aumentata. I campi visivi di questi dispositivi, benché più ampi rispetto a quelli degli smart glass, sono comunque ancora molto limitati, principalmente dal fatto che sono see-through.
  Un esempio di dispositivo di questo tipo è già stato illustrato precedentemente, nella Figura 1.8a.

Capitolo 2

Il campo della Mixed Reality

Nel Capitolo precedente, abbiamo visto cosa sono le tecnologie wearable e alcuni dei loro campi applicativi. Ci concentreremo ora nell'analizzare più approfonditamente uno di questi, ovvero il campo della Mixed Reality.
Quando si parla di wearable nel contesto di Mixed Reality, ci si riferisce, in particolare, ad una categoria specifica di dispositivi, che abbiamo già illustrato, ovvero quella degli HMD. Tenendo in mente ciò, possiamo addentrarci nella trattazione del Capitolo.

2.1 Reality Virtuality Continuum

Il concetto di *Reality Virtuality Continuum* (RVC) venne espresso da Milgram nel 1994 [11], con lo scopo di mettere in relazione:

- la **realtà fisica**, costituita dal mondo in cui viviamo e
- la **realtà virtuale**.

Il termine *realtà virtuale* (VR), viene utilizzato per esprimere un ambiente completamente sintetico, artificiale. Le proprietà di un mondo virtuale, possono essere simili a quelle del mondo reale, così come possono anche discostarsi completamente da esse. Una realtà virtuale, ad esempio, può non attenersi a quelle che sono le leggi della fisica o dello scorrimento del tempo, le quali possono apparire, nel contesto virtuale, completamente assenti o più o meno alterate. Benché quindi, reale e virtuale costituiscano semanticamente due opposti, secondo Milgram, sarebbe meglio vedere queste due entità come gli estremi di un *continuum*, denominato Reality Virtuality Continuum.

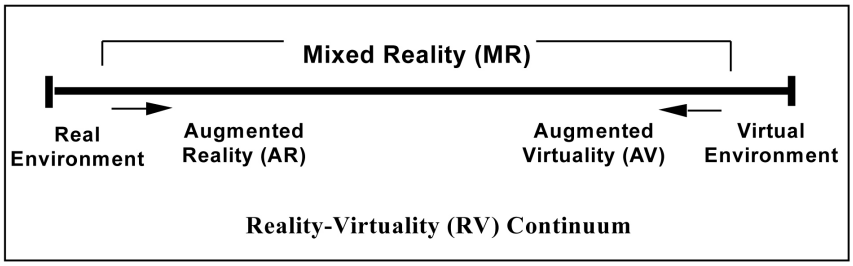


Figura 2.1: *Schema esemplificativo del RV Continuum espresso da Milgram.*

Come possiamo vedere in Figura 2.1, localizzato al centro del continuo, si trova il concetto **Mixed Reality** (MR), suddivisa ulteriormente in **Augmented Reality** (AR) e **Augmented Virtuality** (AV).

2.1.1 Augmented Reality

Il termine Augmented Reality fa riferimento all'inserimento di elementi computazionali nella realtà fisica. In altre parole, possiamo considerare realtà aumentata, una realtà fisica che è stata "potenziata" tramite l'inserimento di immagini e/o informazioni virtuali al suo interno. Diversamente dalla realtà virtuale, in un ambiente di realtà aumentata, il mondo a cui si fa riferimento è quello reale. All'interno di questo però, sono presenti entità computazionali che coesistono ed eventualmente interagiscono con quelle fisiche. Queste entità possono essere veri e propri oggetti virtuali, oppure informazioni riferite ad oggetti fisici (si parla in questo caso di *oggetti aumetnati*).
E' definibile realtà aumentata, anche una realtà nella quale, invece di venire aggiunti oggetti virtuali, vengono rimossi alcuni oggetti fisici (concetto di *diminished reality* o realtà diminuita).

L'inserimento di oggetti virtuali in un contesto fisico viene ottenuto principalmente tramite due approcci:

- secondo **approccio video**, ovvero catturando tramite una fotocamera una vista del mondo reale ed elaborando poi l'immagine inserendo dentro di essa gli oggetti virtuali esattamente nel punto in cui essi devono trovarsi. All'utente è presentato esclusivamente il risultato finale dell'elaborazione. Questo approccio risulta essere poco utilizzato, principalmente per quello che è il problema della latenza dovuta all'elaborazione dell'immagine;

- secondo **approccio ottico**, ovvero elaborando in real time la vista del mondo reale e giustapponendo su di essa gli oggetti virtuali utilizzando opportuni metodi ottici.

Entrambi gli approcci possono poi suddividersi ulteriormente in:

- **approccio see-through**, ottenuto tramite l'impiego di un dispositivo see-through, ovvero con display trasparente, quale ad esempio un paio di smart glasses. In questo modo è possibile vedere veramente il mondo fisico e disegnare esclusivamente gli oggetti virtuali in esso posizionati;
- **approccio "window-on-the-world"**, in cui invece il mondo fisico viene mostrato attraverso delle registrazioni, assieme agli oggetti virtuali in esso disposti.

Dal momento che il substrato di riferimento nel campo della realtà aumentata è la realtà fisica, sono gli oggetti virtuali a doversi conformare ad essa e alle sue proprietà.
E' ovvio che per riuscire a conseguire un tale obiettivo, è necessario l'impiego di dispositivi che siano context-aware, in modo da riuscire a modificare il comportamento degli oggetti virtuali sulla base di quella che è la situazione nel mondo fisico.

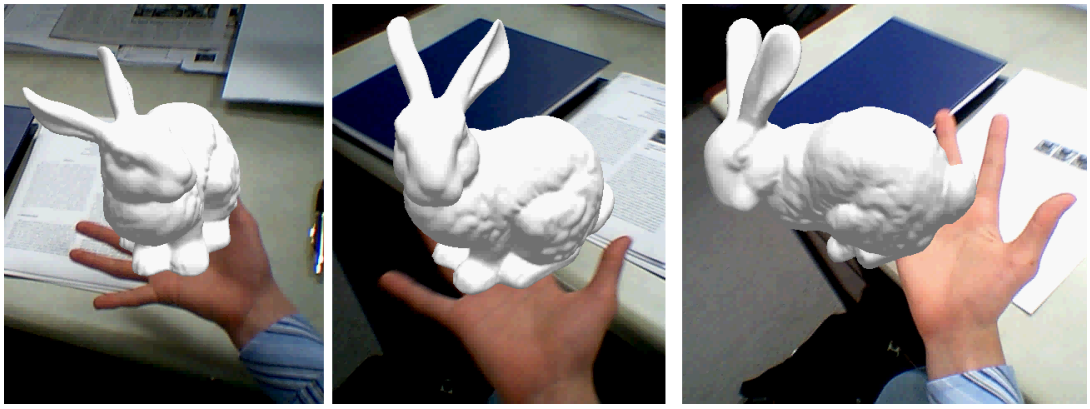


Figura 2.2: *L'immagine mostra un esempio di oggetto virtuale nel mondo reale. La figurina del coniglio viene renderizzata diversamente a seconda del posizionamento della mano, come se essa si trovasse fisicamente su di essa.*

Gli oggetti virtuali devono quindi essere soggetti, ad esempio, ad effetti quali profondità (apparendo tanto più piccoli e meno nitidi quanto più sono lontani dall'osservatore) ed occlusione.
Il problema dell'occlusione è uno tra i più complessi nell'ambito del rendering di oggetti virtuali. Prendiamo ad esempio un caso in cui un osservatore stia osservando un oggetto A. Tra l'oggetto A e l'osservatore, è situato però un oggetto B. Nel mondo reale, l'oggetto B occlude l'oggetto A, coprendolo parzialmente o totalmente. Per dare il massimo effetto di immersione possibile,

questo deve accadere anche per quanto riguarda gli oggetti virtuali. Benché risulti relativamente semplice fare in modo che oggetti virtuali si occludano tra di loro o occludano oggetti fisici, risulta invece essere estremamente complicato fare in modo che oggetti reali occludano oggetti virtuali.

Un altro problema che incide molto sul fattore immersione, è legato al fatto che la rappresentazione del mondo reale sul quale disporre gli oggetti virtuali, viene tracciata utilizzando la fotocamera del dispositivo. Questa fotocamera, spesso, per via della sua posizione, ha un campo visivo diverso da quello dell'utente, ovvero inquadra la realtà da un altro punto di vista. E' necessario quindi, fare in modo, tramite opportune trasformazioni geometriche, che i due campi visivi coincidano, prima di procedere con l'inserimento delle entità computazionali.

2.1.2 Augmented Virtuality

Il termine *Augmented Virtuality* fa riferimento invece ad un contesto in cui l'ambiente predominante è puramente virtuale, ma è in grado di percepire oggetti o avvenimenti appartenenti al mondo fisico.
In una virtualità aumentata quindi, è il mondo virtuale ad essere potenziato da quello reale, contrariamente a ciò che accade in una realtà aumentata.
L'influenza del mondo fisico su una virtualità, può essere ottenuta in più modi:

- **Inserendo oggetti reali nel contesto virtuale** di riferimento;
- **Utilizzando registrazioni di sensori collegati al mondo reale per modificare il contesto virtuale** di riferimento;

Anche per quanto riguarda l'augmented virtuality, gli oggetti che vanno ad aumentare il contesto di riferimento (in questo caso il mondo virtuale), dovranno attenersi a quelle che sono le sue proprietà. Nell'immagine 2.3 possiamo vedere un esempio di virtualità aumentata.

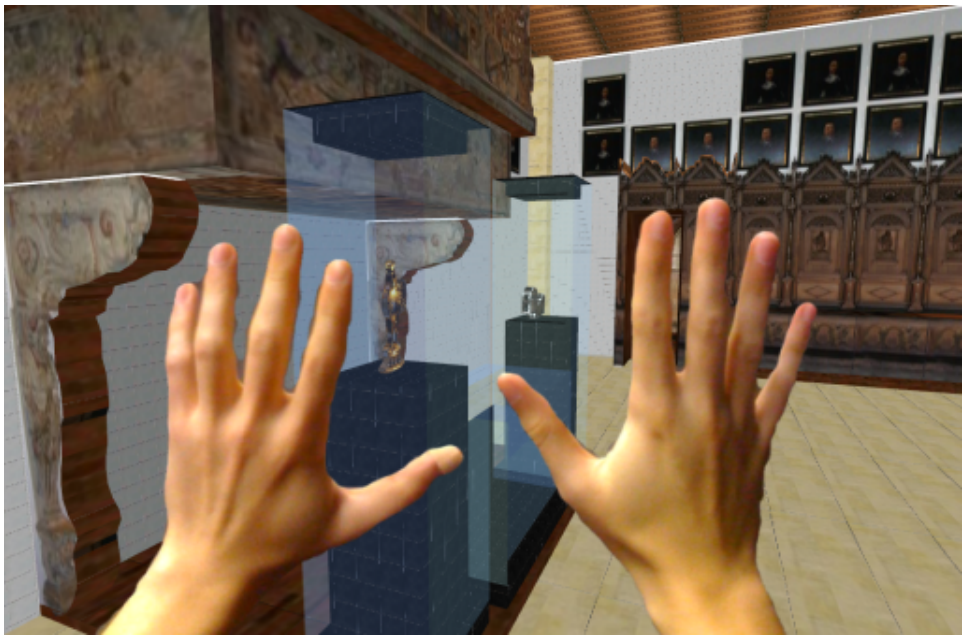


Figura 2.3: *Un esempio di augmented virtuality. L'immagine rappresenta un mondo virtuale con il quale l'utente interagisce tramite le sue mani. Le mani rappresentano, in questo caso, l'apporto proveniente dal mondo fisico che aumenta la virtualità.*

2.2 Proprietà di un Mixed Reality System

Il concetto di RV Continuum espresso da Milgram, suddivide la mixed reality in augmented reality ed augmented virtuality. Quando parliamo di sistemi di mixed reality però, una distinzione di questo tipo risulta essere troppo banale e semplicistica, perché non tiene conto di quelle che sono tutte le possibili sfumature all'interno di questi due campi. Un sistema di realtà aumentata infatti, non presenta necessariamente le stesse caratteristiche di un altro sistema di realtà aumentata. Lo stesso discorso può essere fatto per ciò che riguarda i sistemi di virtualità aumentata.

Per poter suddividere in maniera più nitida un sistema di mixed reality da un altro, Milgram introdusse una tassonomia tridimensionale, basata su quelle che sono le proprietà principali di tali sistemi.

2.2.1 Tassonomia tridimensionale di Milgram

La tassonomia proposta da Milgram si configurava inizialmente come un iperspazio a più dimensioni.

Tali dimensioni costituivano le proprietà principali di un sistema di AR/AV, che andava quindi, sulla base di esse, a collocarsi in uno specifico punto di tale iperspazio, differenziandosi così dagli altri. Per semplificare le cose, l'iperspazio può essere visto anche come uno spazio tridimensionale, se, tra le varie proprietà che ne costituiscono le dimensioni, ne vengono considerate solo tre.
Le tre proprietà/dimensioni più importanti secondo Milgram, sono: l'**Extent of World Knowledge** (EWK), la **Reproduction Fidelity** (RF) e l'**Extent of Presence Metaphor** (EPM).
Sulla base di queste tre quindi, è possibile tracciare uno spazio tridimensionale all'interno del quale un qualunque sistema di AR/AV può essere distintamente collocato.
Per poter capire però come posizionare tale sistema lungo i tre assi sulla base delle sue caratteristiche, andiamo ora ad analizzare queste ultime.

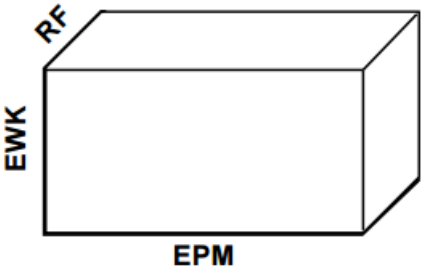


Figura 2.4: *Rappresentazione dello spazio tridimensionale all'interno del quale è possibile collocare un qualunque sistema di mixed reality.*

2.2.2 Extent of World Knowledge

La dimensione dell'EWK classifica i sistemi di mixed reality sulla base di quanto questi sanno sulla realtà di interesse che riproducono. L'estremo sinistro di questa dimensione, rappresenta il caso in cui il sistema non sappia nulla su tale realtà, che viene quindi definita come *Unmodelled*, ossia non modellata computazionalmente. L'estremo destro, al contrario, rappresenta invece il caso in cui il sistema conosca e manipoli completamente la realtà d'interesse. Questo è il caso in cui la realtà di interesse è puramente virtuale. Un sistema situato su questo estremo conosce posizione, forma e caratteristiche di ogni oggetto appartenente al mondo rappresentato e sa anche come questi interagiscono tra loro e come l'utente può manipolarli.

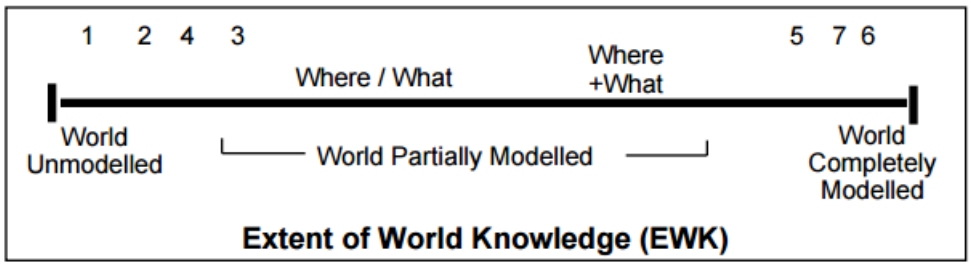


Figura 2.5: *Rappresentazione grafica della dimensione dell'Extent of World Knowledge.*

2.2.3 Reproduction Fidelity

La dimensione della RF incide particolarmente sulla sensazione di immersione e presenza dell'utente all'interno della mixed reality rappresentata dal sistema. Il termine Reproduction Fidelity si riferisce alla qualità delle immagini riprodotte dal sistema, sia che queste rappresentino oggetti del mondo reale, sia che rappresentino oggetti puramente virtuali. Anche in questo caso, come nel precedente, ci si muove da sinistra verso destra in maniera crescente. Un sistema posizionato all'estrema destra di questo asse quindi, sarà un sistema con una notevole qualità di rappresentazione delle immagini.

La RF, in generale, dipende da diversi fattori quali l'hardware del dispositivo utilizzato, le tecniche di rendering impiegate, ecc e per questo, sarebbe più chiaro suddividerla in più dimensioni. Tuttavia, per mantenere il discorso più semplice e sintetico, viene qui rappresentata come un tutt'uno.

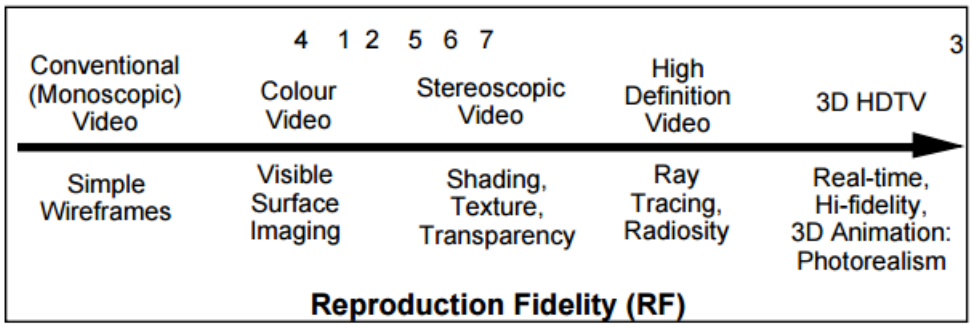


Figura 2.6: *Rappresentazione grafica della dimensione della Reproduction Fidelity.*

2.2.4 Extent of Presence Metaphor

La dimensione dell'EPM, classifica un sistema sulla base di quanto un utente si senta immerso nella realtà riprodotta. All'estremo sinistro di questo asse, si trovano i sistemi che fanno uso di un approccio window-on-the-world per rappresentare la loro mixed reality. In sistemi del genere ovviamente, l'utente non si sente affatto parte del mondo rappresentato, ma ha piuttosto la sensazione di osservarlo da fuori ("da una finestra"). All'estremo destro invece, si trovano quei sistemi che aderiscono alla metafora del *real-time imaging*, ovvero sistemi in cui l'utente si sente completamente immerso nella realtà riprodotta.

Più in generale, l'Extent of Presence Metaphor serve a distinguere quelli che sono sistemi esocentrici (in cui l'utente non viene posto al centro), dai sistemi egocentrici (in cui l'utente viene posto al centro). La dimensione dell'EPM e quindi il suo asse, non dovrebbe, in un certo senso, essere ortogonale a quella della RF, in quanto entrambe posizionano nel loro estremo destro, sistemi nei quali l'utente ha una sensazione di immersione completa.

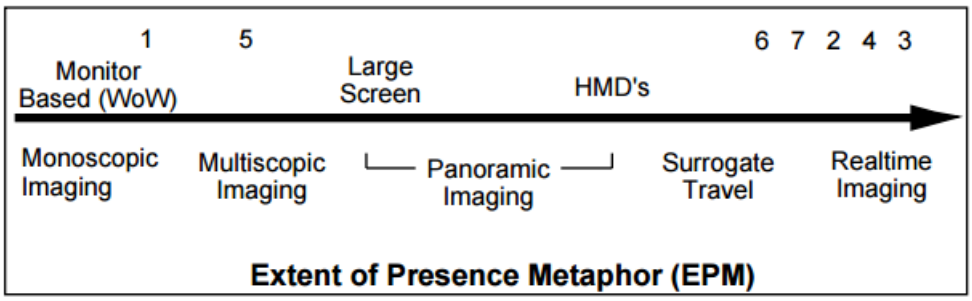


Figura 2.7: *Rappresentazione grafica della dimensione dell'Extent of Presence Metaphor.*

2.2.5 Metodi di interazione con le entità virtuali

Oltre a basarsi sulla tassonomia di Milgram, è possibile differenziare un sistema di mixed reality da un altro, anche analizzando quelle che sono le tecniche che esso impiega per permettere all'utente di interagire con gli oggetti virtuali appartenenti alla realtà rappresentata.
Tra le vari interfacce di interazione, le principali sono:

- **Interfacce tangibili**, ovvero costituite da oggetti del mondo fisico, mediante i quali si è in grado di interagire con oggetti puramente virtuali;
- **Interfacce ibride**, che combinano un insieme di interfacce diverse, ma complementari, in modo da rendere possibile l'interazione con gli oggetti

virtuali in più modi. Queste forniscono una piattaforma per interazioni con dispositivi di vario tipo, dei quali, magari, non si conoscono le specifiche anticipatamente;

- **Interfacce multimodali**, che utilizzano invece mezzi quali il riconoscimento vocale, lo sguardo, o i movimenti/gesti delle mani come interfaccia di interazione con gli oggetti virtuali. Queste interfacce, emerse recentemente, sono quelle che si stanno diffondendo più rapidamente.

2.3 Esempi di Mixed Reality System

Per concludere, vediamo ora alcuni esempi di mixed reality system, in modo da rendere più chiaro il concetto di cosa essi siano. Verranno illustrati tre sistemi in maniera sintetica, facendo riferimento a ciò che è stato esposto nel corso della trattazione del Capitolo.

2.3.1 Magic Paddle

Il primo esempio è costituito da Magic Paddle [15], un sistema di realtà aumentata che fa uso di interfacce di interazione tangibili.
Il progetto Magic Paddle, permette all'utente di arredare delle stanze in miniatura, disponendo in esse dei mobili virtuali facendo uso di una piccola paletta per manipolare (spostare e ruotare) questi ultimi. Il rendering degli oggetti virtuali, è fatto da un apposito HMD. Oltre a questo e alla paletta per manipolare gli oggetti, il sistema utilizza un particolare libro che funge da "catalogo" e da workspace. Una volta aperto, esso presenta una pagina in cui sono visibili illustrazioni di mobili ed una pagina bianca, che rappresenta invece la stanza da arredare. Indossando l'HMD, sopra la pagina in cui sono illustrate le figure dell'arredamento, è possibile vedere dei modelli tridimensionali virtuali di tali figure. L'utente può quindi, utilizzando la paletta, spostare, copiare e disporre questi modelli virtuali sulla pagina bianca, creandosi una piccola stanzina aumentata.

In Figura 2.8 vediamo un'immagine che mostra il sistema in funzione.

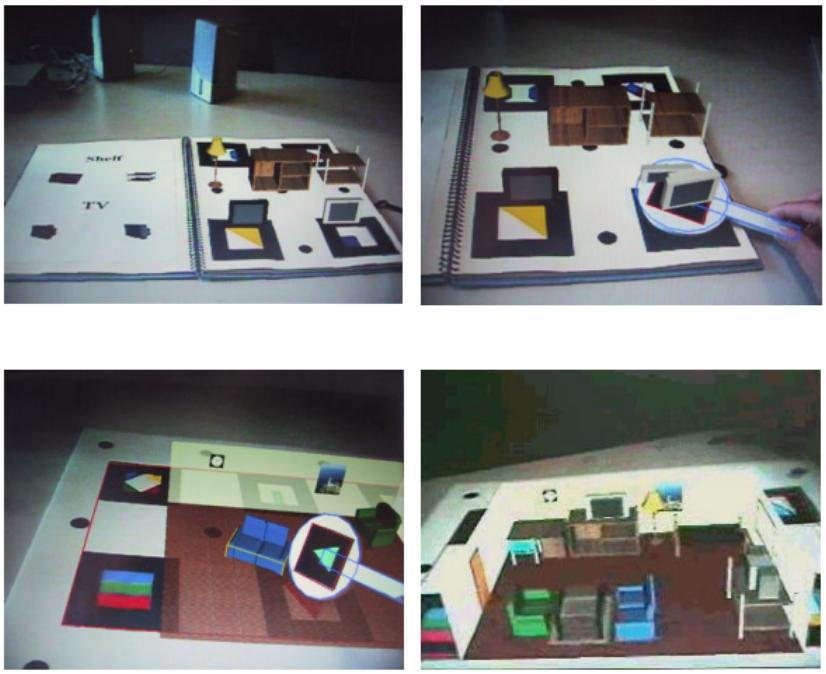


Figura 2.8: *L'immagine mostra il libro catalogo/workspace utilizzato in Magic Paddle e l'utilizzo della paletta per sistemare i modelli dei mobili virtuali.*

2.3.2 Second Surface

Second Surface [16] è un ulteriore esempio di sistema basato su realtà aumentata, che segue però un approccio del tipo window-on-the-world. In Second Surface, si fa uso di un tablet per disegnare su una canvas virtuale, linkata ad un oggetto appartenente al mondo reale. A partire da un'immagine che inquadra un oggetto reale, Second Surface è in grado di associare ad essa una canvas virtuale, sulla quale l'utente può disegnare, sempre utilizzando il tablet. I contenuti prodotti dall'utente, vengono poi mantenuti all'interno di quella canvas, che a sua volta viene salvata in un database e linkata a quello che è l'oggetto su cui essa è stata posizionata.

I database che mantengono le informazioni relative agli oggetti su cui le canvas vengono posizionate, sono chiamati *Dictionary*. Gli utenti possono condividere fra di loro uno stesso Dictionary e lavorare quindi sulle stesse canvas, modificando o semplicemente osservando i contenuti creati da altri.
Un esempio è offerto dall'immagine 2.9.

Figura 2.9: *La figura mostra due utenti, uno atto a disegnare su una canvas virtuale, mentre l'altro lo osserva. Il secondo utente, è in grado di vedere la canvas virtuale sulla quale il primo sta lavorando perché fa uso dello stesso Dictionary del primo. Come possiamo vedere inoltre, anche i contenuti della canvas (in questo caso dei fiori disegnati dal primo utente) sono visibili ad entrambi.*

2.3.3 Sistemi basati su Myo Armband e Smart Glass

All'Augmented World Expo (AWE) 2015, Stefan Alexander, dei Thalmic Labs, ha esposto l'idea dell'utilizzo di un'interfaccia multimodale (in questo caso il Myo) accoppiata ad uno smart glass, per applicazioni di augmented virtuality ed augmented reality hands-free e see-through. Dal momento che questi concetti ricadono proprio nello stesso campo in cui si colloca il lavoro svolto per questa tesi, consiglio di prendere visione del video della presentazione[1], per riuscire a comprendere meglio le idee che verranno esposte nel Capitolo successivo.

[1] `https://www.youtube.com/watch?v=9jT78WMxf5c`

Capitolo 3

Caso di studio: Framework per nuove interfacce utente

Avendo terminato l'esposizione dei concetti introduttivi, è ora possibile passare ad un'esposizione, seppur ancora sommaria, di quello che è stato il progetto svolto nell'ambito di questa tesi.
In questo Capitolo, verranno esposte a grandi linee le funzionalità del framework progettato e la sua utilità in campo pratico, facendo riferimento a ciò che è stato oggetto di trattazione dei capitoli precedenti.

3.1 Applicazioni basate su sistemi wearable hands-free

Abbiamo già visto, nel Capitolo 1, come i dispositivi wearable siano pensati appositamente per far fronte alle esigenze dell'utente on-the-go, fornendogli un interfacciamento al mondo digitale senza richiedere un impegno oneroso da parte sua per poterli utilizzare.
Abbiamo anche visto come questi dispositivi aspirino, il più delle volte, ad essere hands-free, ovvero utilizzabili senza impegnare le mani del loro utilizzatore. Nonostante questo però, il fatto che non esistano ancora killer-app progettate su misura per questi dispositivi, ne limita notevolmente l'utilizzo sia in campo lavorativo, che nell'ambito del mercato. La carenza di applicazioni per dispositivi wearable hands free, è dovuta anche, al fatto che non esistono ancora librerie o framework a supporto del loro sviluppo.
Ma cosa si intende per "app ideate su misura per tecnologie wearable hands-free"?

Proviamo a pensare di dover progettare un'applicazione per uno o più dispositivi di questo tipo. Viste le loro caratteristiche, sarebbe una buona idea

progettare il tutto in modo che questo faccia leva su quelli che sono i punti di forza di tali tecnologie. In altre parole, se queste tecnologie sono state pensate per essere facili da usare, dirette e non intrusive, allora sarebbe bene fare in modo che anche le applicazioni rispettino questi criteri, sfruttando al meglio le potenzialità dei dispositivi sulle quali dovranno eseguire.

3.2 L'idea di base del framework sviluppato

Per semplificare lo sviluppo di applicazioni basate appositamente su dispositivi wearable hands-free, abbiamo deciso di progettare ed implementare un framework che fornisca un supporto per la realizzazione di interfacce utente, pensate appositamente per questo tipo di device. In altre parole, quello che si vuole mettere a disposizione, è una piattaforma mediante la quale sia possibile costruire, nel modo più semplice possibile, interfacce utente innovative.

Queste interfacce, sono pensate per distogliere il meno possibile l'attenzione dell'utilizzatore da ciò che egli sta facendo, in modo da adattarsi perfettamente alle caratteristiche dei dispositivi wearable. Esse devono, inoltre, permettere un'interazione hands free tra utente e contenuti. Applicazioni disposte di interfacce utente simili, possono sfruttare in maniera migliore i punti di forza di un dispositivo wearable, fornendo agli utenti un aggancio a contenuti digitali senza limitare in alcun modo le loro capacità. Questo può fare la differenza in contesti lavorativi quali quello medico o quello industriale, nei quali spesso, si ha bisogno di mantenere libero il proprio campo visivo e le proprie mani per poter svolgere determinate attività.

3.2.1 Un'interfaccia utente innovativa

Quando si parla di interfacce utente nell'ambito hands-free, ci si deve per forza allontanare dai paradigmi standard, quali interfacce basate su WIMP (Windows, Icons, Menus, Pointer), o su Touch and Swipe Gestures. Entrambe infatti, non sono applicabili in un contesto hands-free e richiedono una concentrazione visiva che non sempre è possibile avere quando si è in contesti on-the-go.

La nuova tipologia di interfaccia utente, è pensata per andare oltre i dispositivi, fondendosi, in un certo senso, con quello che è il mondo reale che circonda l'utilizzatore. In parole povere, possiamo descrivere questa nuova interfaccia, come uno spazio virtuale bidimensionale, posizionato relativamente al suo utilizzatore. Tale spazio non è limitato dalle dimensioni del display sul quale viene riprodotto (che sarà head mounted, visto che si parla di wearable) e può essere utilizzato come una "lavagna", sulla quale disegnare qualsiasi cosa si voglia. Dal momento che le dimensioni di tale lavagna possono eccedere

quelle del display, questa dovrà essere navigabile in un qualche modo. Per rimanere in linea con il concetto hands-free, che rappresenta il punto focale di queste interfacce, tale navigazione dovrà essere resa possibile tramite il solo movimento della testa. Né la lavagna, né i suoi contenuti inoltre, dovranno mai occludere completamente il campo visivo dell'utente, in modo che esso non perda mai contatto con la realtà. Per finire, la nuova interfaccia è pensata per disporre anche di un sistema di puntamento. Il puntatore, sempre visibile sulla lavagna, dovrà essere manovrabile tramite il movimento del braccio. Utilizzando specifiche gesture inoltre, sarà possibile interagire con gli elementi virtuali posti sulla lavagna, nel caso ce ne sia bisogno.
Qui sotto, è offerta un'immagine esemplificativa che aiuti a visualizzare ciò che è stato appena esposto.

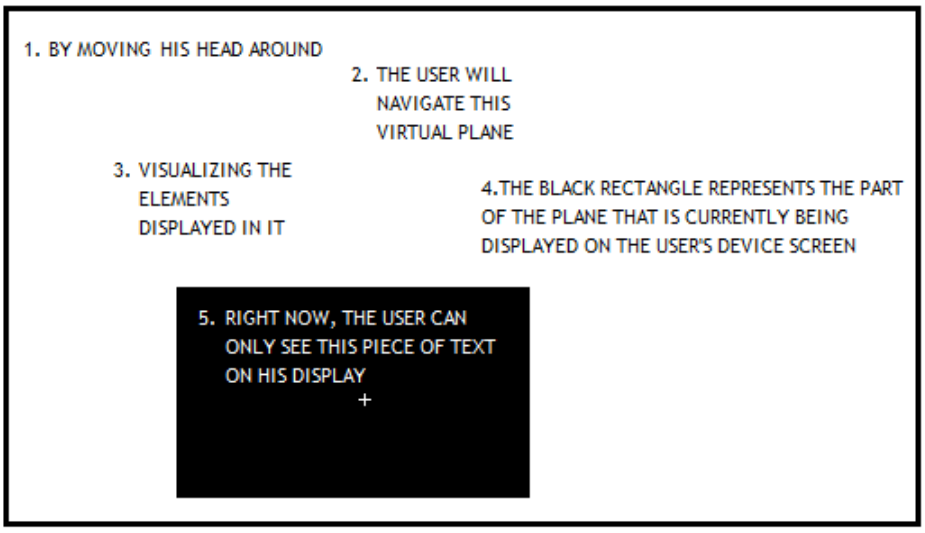


Figura 3.1: *Il rettangolo bianco rappresenta il piano virtuale sul quale è possibile disporre gli elementi che comporranno la propria interfaccia grafica (in questo esempio, in esso sono disposte delle semplici scritte numerate). Benché virtuale, tale piano è posizionato di fronte all'utilizzatore ed è quindi collegato alla realtà fisica. Il rettangolo nero invece, rappresenta il campo visivo dell'utente, ovvero la parte di interfaccia che viene, in un determinato momento, riprodotta sullo schermo del dispositivo utilizzato (HMD). Tale rettangolo si sposta all'interno del piano in maniera concorde a come l'utente sta muovendo la testa. Ad esempio, se l'utente volesse visualizzare il testo 4, gli basterebbe muovere la testa leggermente verso destra e verso l'alto. Quella specie di mirino bianco all'interno del rettangolo nero rappresenta il puntatore, che verrà spostato invece tramite il movimento del braccio dell'utente.*

3.3 Tecnologie di riferimento

Il progetto si colloca a metà tra il campo dell'augmented virtuality (in quanto la realtà manipolata dal sistema è puramente virtuale, ma viene influenzata dagli input registrati dei sensori inerziali appartenenti al mondo fisico, che saranno utilizzati per far scorrere lavagna e puntatore) e l'augmented reality (perché essendo see-through, non occlude la visione del mondo reale) ed è pensato per offrire un supporto nello sviluppo di applicazioni che facciano uso di:

- Un HMD see-through, sul quale verrà visualizzata la lavagna;
- Un'interfaccia multimodale, più nello specifico un Gesture Recognizer, da utilizzare come fonte di input per il cursore e come mezzo per interagire con gli elementi virtuali disposti nella lavagna.

Per quanto riguarda lo sviluppo, è stato scelto di utilizzare come piattaforma Android, vista la sua diffusione nel campo del mobile computing (e quindi anche in quello delle tecnologie wearable) e le funzionalità messe a disposizione dal suo SDK per manipolare i sensori inerziali, necessari per far muovere opportunamente la lavagna sulla quale vengono disposti i contenuti dell'interfaccia.

Capitolo 4

Cenni su sensori inerziali e quaternioni

Prima di procedere con la descrizione tecnica del lavoro svolto, dedichiamo un breve Capitolo alla trattazione dei sensori inerziali e del loro funzionamento. Nelle pagine successive, vedremo quelle che sono le caratteristiche di un IMU, la descrizione degli angoli di roll, pitch e yaw e i loro punti deboli. Verrà poi esposto, brevemente, il concetto di quaternione, mettendone in luce i pregi e fornendo qualche esempio di utilizzo.

4.1 Sensori inerziali e filtri

Un IMU (Inertial Measurement Unit), è un **dispositivo elettronico in grado di misurare la velocità angolare, l'accelerazione e il campo magnetico di un corpo sui suoi tre assi**. Gli IMU possono essere utilizzati, come nel nostro caso, per monitorare il movimento di un dispositivo, in modo da sapere verso dove e di quanto questo si è mosso o si sta muovendo. La maggior parte dei dispositivi mobili oggigiorno, sono dotati di IMU a 9 assi, ovvero che dispongono di 9 gradi di libertà (alcuni ne hanno solo 6, in quanto non sempre il magnetometro è presente) nella percezione del moto. La posizione degli assi sui quali si basano le misurazioni dei sensori, varia da dispositivo a dispositivo. Per quanto riguarda device basati su Android però (sia smart phone, che smart glass, che smart watch), di solito la loro disposizione è la seguente:

- L'asse Z ha direzione uscente dallo schermo del dispositivo;
- L'asse X è ortogonale all'asse Z ed attraversa lo schermo del dispositivo da sinistra verso destra;

- L'asse Y è ortogonale agli altri due e attraversa lo schermo del dispositivo dal basso verso l'alto.

Un esempio della disposizione degli assi di un IMU di un Android device è offerta in Figura 4.1.

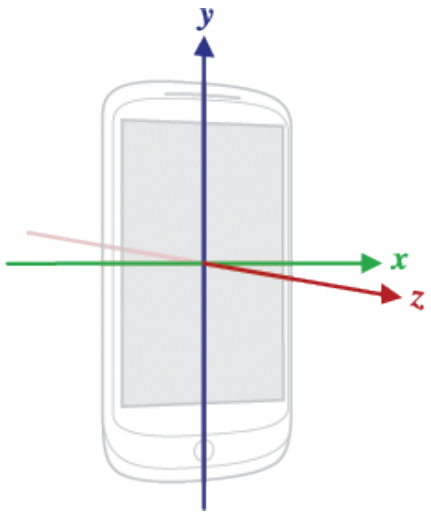


Figura 4.1: *Disposizione degli assi di un dispositivo Android.*

4.1.1 Accelerometro e magnetometro

Un accelerometro è un sensore inerziale in grado di **misurare l'accelerazione di un corpo** (o più precisamente la forza inerziale alla quale esso è sottoposto) sui suoi tre assi. Basandosi sul vettore dell'accelerazione registrato, è possibile risalire all'angolo di inclinazione del corpo rispetto a ciascun asse, ricostruendo quindi quella che è la posizione attuale del corpo. La registrazione dell'accelerazione alla quale il corpo è sottoposto in un determinato momento, è espressa sotto forma di vettore, le cui componenti rappresentano la componente della forza inerziale rispetto a ciascun asse di riferimento. Una misurazione di un accelerometro, è quindi esprimibile come: `a = (ax, ay, az)`, dove `ax`, `ay` e `az` sono le componenti del vettore accelerazione `a`, in riferimento agli assi x, y e z rispettivamente. La lunghezza di tale vettore, è calcolabile tramite il teorema di Pitagora, il che significa che essa è sempre maggiore di ciascuna delle sue componenti. Da questo inoltre, possiamo ricavare che ciascuna componente è ottenibile moltiplicando la lunghezza di `a` per il coseno che il vettore `a` forma con l'asse al quale la componente è riferito, ovvero:

- `ax = a * cos(alfa)`, dove alfa è l'angolo che `a` forma con l'asse x;

- `ay = a * cos(beta)`, dove beta è l'angolo che `a` forma con l'asse y;
- `az = a * cos(gamma)`, dove gamma è l'angolo che `a` forma con l'asse z.

Avendo `ax`, `ay` e `az` all'interno del vettore che rappresenta l'accelerazione registrata dal sensore, possiamo ricavare, tramite formula inversa, i valori degli angoli alfa, beta e gamma, ottenendo così la posizione, relativa ai tre assi, del corpo.

Il funzionamento di un magnetometro è simile a quello di un accelerometro, solamente che esso, invece di misurare la forza inerziale alla quale un corpo è sottoposto, **misura il campo magnetico**, sempre in relazione ad un sistema di riferimento a tre assi. Basandosi su tali misurazioni, anch'esso è in grado di determinare la posizione di un corpo ricavando i valori degli angoli di inclinazione rispetto a ciascun asse.

4.1.2 Giroscopio

Il giroscopio è un sensore in grado di **calcolare la velocità angolare** di un corpo rispetto ai tre assi di riferimento. Sulla base di queste misurazioni, è possibile tenere traccia del suo spostamento, integrando nel tempo la velocità angolare misurata. Questo può essere fatto, più semplicemente, moltiplicando la velocità angolare misurata nel periodo t0, per l'intervallo di tempo t1 - t0, dove t1 rappresenta l'istante in cui la velocità angolare viene ricalcolata dal sensore. Continuando in questo modo, è possibile ottenere lo spostamento relativo ad ogni intervallo di tempo che intercorre tra una misurazione e l'altra, che è tipicamente molto breve per quanto riguarda il giroscopio. Questo metodo risulta quindi efficace per tracciare lo spostamento di un dispositivo che dispone di tale sensore. Questo spostamento, può essere visto come l'orientamento relativo del dispositivo rispetto a quella che era la sua posizione iniziale. Per calcolare la posizione iniziale tuttavia, il giroscopio da solo non è sufficiente.

4.1.3 Filtri complementari

La vera efficacia di un IMU risiede nel poter combinare le registrazioni di tutti e tre i sensori, fondendole ed eliminando quelli che sono i punti deboli di ciascuno di essi. Come abbiamo visto, per poter ricavare l'orientamento (posizione) assoluta di un dispositivo, occorre basarsi sulle registrazioni di magnetometro ed accelerometro. Questi sensori tuttavia, non sono precisi per quanto riguarda il calcolo della variazione di tale posizione, prima di tutto perché le loro misurazioni vengono registrate con intervalli di tempo relativamente lunghi, inoltre perché, per ottenere lo spostamento a partire da un'accelerazione,

occorre integrare tale valore due volte nel tempo. Questa doppia integrazione farebbe esplodere errori anche minimi nella misurazione, ai quali tra l'altro, sia magnetometro che accelerometro sono particolarmente vulnerabili (specialmente per quanto riguarda le interferenze dovute al rumore).

Il giroscopio al contrario, fornisce misurazioni ad una frequenza molto elevata ed è molto più preciso e resistente alle interferenze (seppure anch'esso vulnerabile al rumore), rendendolo adatto a calcolare la variazione di posizione, ma non la posizione assoluta (cosa che non è proprio in grado di fare).
Per calcolare la variazione di posizione tuttavia, anche il giroscopio necessita di integrare nel tempo le sue misurazioni. Dal momento che un giroscopio misura la velocità angolare, per ottenere lo spostamento relativo, è sufficiente integrare nel tempo una sola volta i valori registrati. Anche la singola integrazione però, fa incrementare sempre di più errori minimi nella misurazione, portando al cosiddetto *gyro-drift*.

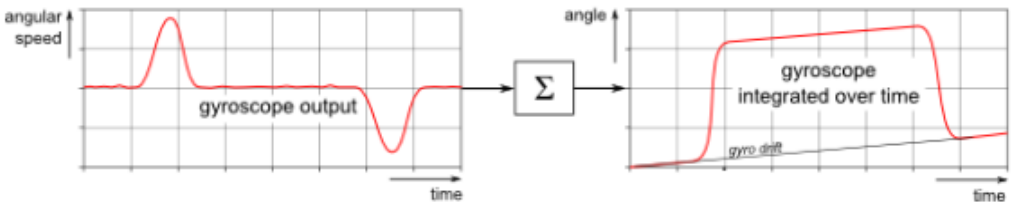


Figura 4.2: *Il grafo mostra la variazione delle registrazioni di un giroscopio quando un dispositivo viene ruotato di 90 gradi in una direzione e, dopo un breve periodo, viene riportato alla posizione iniziale. Dai grafici è possibile notare come gli errori in ciascuna misurazione influiscano su quella successiva quando si va ad integrare. L'accumulo di questi errori porta al fenomeno del drift qui illustrato.*

Questi problemi possono essere eliminati utilizzando magnetometro ed accelerometro per calcolare la posizione iniziale del dispositivo e basandosi poi sulle registrazioni del giroscopio per calcolare come tale posizione varia nel tempo. Per evitare il fenomeno del drift poi, non appena comincia ad accumularsi l'errore nelle misurazioni del giroscopio, si ricalcola l'orientamento assoluto utilizzando nuovamente accelerometro e magnetometro. La nuova posizione calcolata, viene utilizzata come nuovo punto di partenza in relazione al quale viene calcolato lo spostamento, sempre basandosi sulle registrazioni del giroscopio.

Questa soluzione è efficace, ma non tiene conto di quelli che sono gli errori nel calcolo della posizione assoluta di magnetometro ed accelerometro. Come abbiamo detto, entrambi questi sensori sono vulnerabili ad interferenze e, benché per calcolare la posizione assoluta, al contrario della variazione di po-

sizione, non sia necessaria alcuna integrazione, tali errori possono comunque risultare significativi pur non essendo incrementali. Per eliminare questo tipo di errori, si può far uso dei cosiddetti filtri. Esistono diversi tipi di filtri, alcuni più semplici (filtri complementari), altri più complessi (filtri di Kalman). Noi qui, in relazione a ciò che è stato fatto nel progetto di tesi, esporremo quella che è l'idea per un semplice filtro complementare.

Un **filtro complementare si basa sull'idea di utilizzare il giroscopio per calcolare variazioni dell'orientamento che avvengono in brevi intervalli di tempo, mentre le registrazioni di accelerometro e magnetometro vengono utilizzate come informazioni di supporto lungo periodi temporali più ampi**. Questo equivale ad applicare un low-pass filter alle registrazioni di magnetometro ed accelerometro (facendo passare solo segnali sotto una determinata frequenza) ed applicare invece un high-pass filter (facendo passare solo segnali sopra una determinata frequenza) all'integrazione delle misurazioni effettuate dal giroscopio.

Fare low-pass filtering delle registrazioni di accelerometro e magnetometro, è equivalente a fare una media in base al tempo di tali registrazioni. In altre parole, ogni volta che viene effettuata una misurazione dai sensori, questa viene moltiplicata per un peso ed aggiunta all'orientamento assoluto:

```
AccMagOrientation = (1 - factor) * AccMagOrientation + factor *
                                   newAccMagValue
```

Dove AccMagOrientation è l'orientamento calcolato sulla base delle registrazioni di accelerometro e magnetometro e factor è una costante di peso scelta arbitrariamente (tanto più alta sarà, maggiore sarà il peso dato alle registrazioni più recenti).

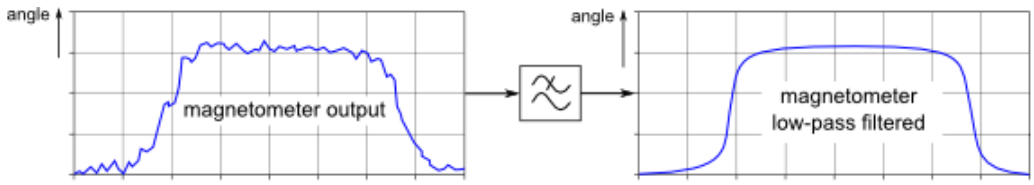


Figura 4.3: *Il grafico mostra come cambiano le registrazioni di magnetometro e accelerometro quando un dispositivo viene ruotato di 90 gradi verso una direzione e, dopo un breve periodo, riportato nella posizione iniziale. Sulla destra è possibile vedere il risultato a seguito dell'applicazione del low-pass filter illustrato.*

L'high-pass filtering delle misurazioni integrate del giroscopio invece, viene fatto sostituendo la componente filtrata dal low-pass filter di accelerometro e magnetometro, con i valori registrati dal giroscopio, il che equivale a fare:

```
fusedOrientation = (1 - factor) * newGyroValue + factor *
                   newAccMagValue
```

Il risultato complessivo, viene mostrato in Figura 4.4.

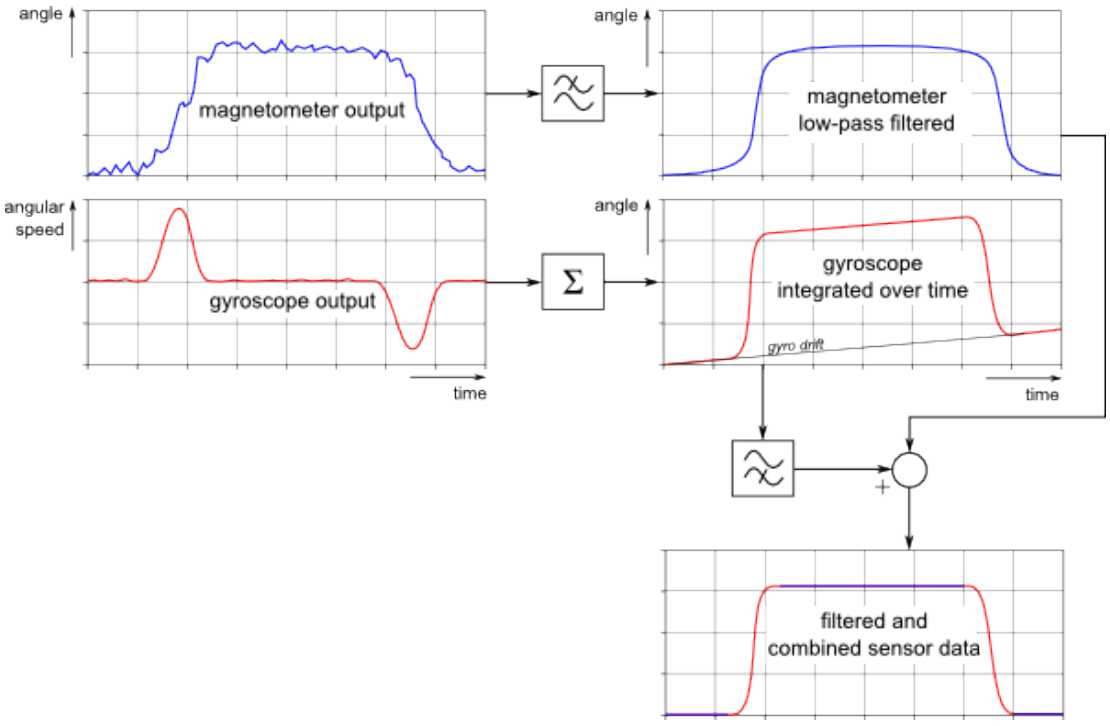
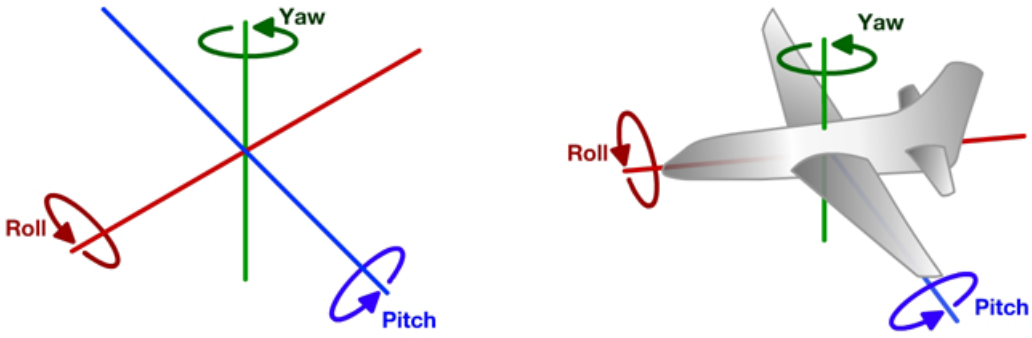


Figura 4.4: Risultato dell'applicazione del filtro complementare descritto.

4.2 Angoli di roll, pitch e yaw

La posizione, o orientamento, di un dispositivo dotato di sensori inerziali, può essere espressa, come già detto, sulla base del valore degli angoli che il dispositivo forma con i tre assi di riferimento.
Questi angoli, prendono il nome di **roll**, **pitch** e **yaw**, o più comunemente, angoli di Eulero.

- L'angolo di **pitch** è l'angolo di rotazione attorno all'asse x;
- L'angolo di **roll** è l'angolo di rotazione attorno all'asse y;
- L'angolo di **yaw** (detto anche *azimuth*) è l'angolo di rotazione attorno all'asse z.



4.2.1 Problemi legati all'utilizzo

L'utilizzo degli angoli di Eulero, per quanto semplice possa essere, non risulta affatto efficace, in quanto è soggetto a diversi problemi, quale ad esempio quello del **gymbal lock**.
Il problema del gymbal lock si verifica quando due assi si allineano e si perde un grado di libertà nella percezione del moto.

4.3 I quaternioni

I quaternioni sono entità matematiche che **rappresentano una rotazione o un orientamento**. In termini pratici, essi sono dei **vettori a quattro dimensioni** della forma `[w, x, y, z]` o anche `[x, y, z, w]`. Al contrario

degli angoli di Eulero, i quaternioni **non soffrono del problema del gymbal lock e risultano essere privi di qualsiasi ambiguità**. Tralasciando la matematica sulla quale questi si basano, visto che comunque, sono disponibili tantissime librerie diverse per manipolarli, vediamo ora cosa è possibile fare con questi quaternioni.

4.3.1 Esempi di utilizzo

Abbiamo detto innanzitutto, che essi rappresentano una rotazione o un orientamento. Affinché un quaternione rappresenti una rotazione, deve essere normalizzato (ovvero $x^2 + y^2 + z^2 + w^2 = 1$). E' possibile, applicare una rotazione espressa sotto forma di quaternione ad un orientamento, anch'esso espresso sotto forma di quaternione, moltiplicando il quaternione che rappresenta l'orientamento per quello che rappresenta invece la rotazione da applicare. Il risultato sarà un nuovo quaternione che rappresenta il nuovo orientamento. Occorre tenere presente che la moltiplicazione tra quaternioni non è commutativa.

Utilizzando i quaternioni, è possibile anche ottenere la rotazione necessaria per passare da un orientamento ad un altro. Avendo i due orientamenti espressi sotto forma di quaternioni, è possibile calcolare la "differenza" tra i due, moltiplicando il quaternione che rappresenta l'orientamento finale per l'inversa del quaternione che rappresenta l'orientamento iniziale. L'inversa di quaternione è calcolabile dividendo la sua coniugata per la sua magnitudine. La magnitudine di un quaternione si calcola come per un qualsiasi altro vettore, ovvero elevando al quadrato le sue componenti e sommandole tra di loro. La coniugata invece si ottiene negando tutti gli elementi del quaternione. A partire da un quaternione è possibile anche ricavare quelli che sono gli angoli di roll, pitch e yaw effettuando delle operazioni sulle sue componenti.

Quelle viste finora, sono solo alcune delle operazioni eseguibili su/con i quaternioni. Tralasciando quelle più complicate e tediose (quale ad esempio la spherical linear interpolation, o *slerp*), limitiamoci a tenere queste a mente, in modo da riuscire a seguire le trasformazioni effettuate nel codice del framework che verranno illustrate nei capitoli successivi.

Capitolo 5

Analisi dei requisiti e definizione delle API

Questo Capitolo segna l'inizio della fase di descrizione del progetto svolto[1]. Cominceremo questa seconda parte della tesi rivedendo, da un punto di vista più formale, quelli che sono i requisiti che il sistema deve soddisfare; dopodiché daremo una definizione, ancora piuttosto astratta, di quelle che sono le entità principali che compongono il framework e delle API che esso deve mettere a disposizione.

5.1 Diagramma dei casi d'uso

Come già menzionato, il framework sviluppato ha come scopo quello di mettere a disposizione una piattaforma in grado di facilitare la realizzazione di interfacce utente innovative see-through e hands-free, per applicazioni Android basate su smart glass e gesture recognizers.

Le funzionalità che il sistema deve mettere a disposizione, sono indicate nel diagramma dei casi d'uso proposto in Figura 5.1.

[1] repo: `https://www.bitbucket.org/djanno/wearableui`

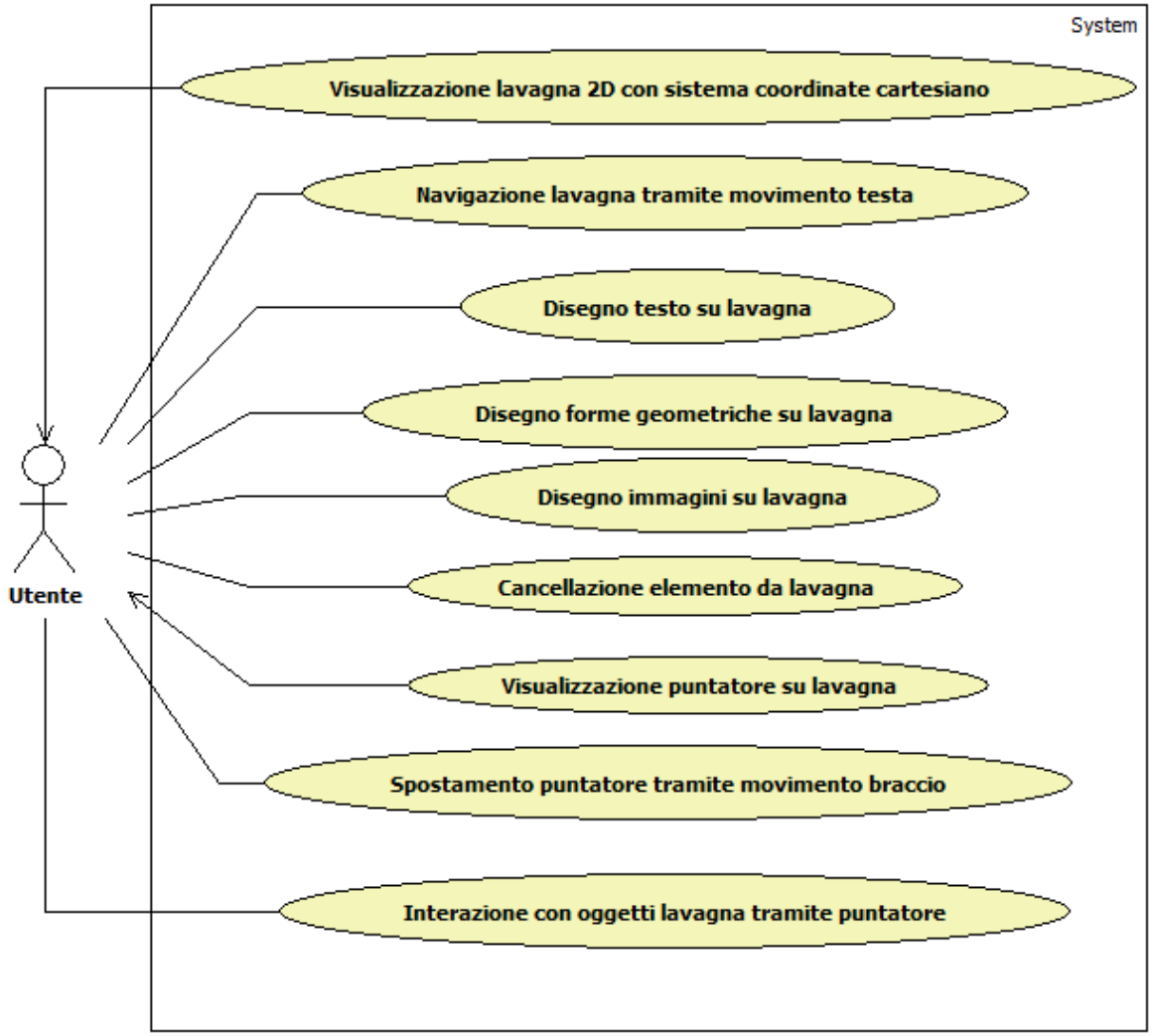


Figura 5.1: *Diagramma dei casi d'uso.*

5.2 Definizione delle entità principali

Prima di procedere con la descrizione della parte progettuale e di mettere in luce quella che è l'architettura di base del sistema, definiamo la terminologia che verrà utilizzata nel corso dei capitoli successivi, eliminando così la possibilità di fraintendimenti e ambiguità.

All'interno del framework esistono quattro entità principali, ovvero:

- La **Viewport**, che rappresenta il concetto della lavagna virtuale 2D, navigabile tramite il movimento della testa e con sistema di coordinate centrato;
- Il **Gaze**, che rappresenta l'orientamento dello sguardo dell'utente. Sarà proprio sulla base di questo, che la Viewport verrà fatta scorrere;
- Il **Cursor**, che rappresenta il cursore all'interno di una Viewport, mediante il quale si può interagire con altri oggetti in essa disposti e che deve essere spostabile tramite il movimento del braccio;
- Il **Finger**, che rappresenta invece l'orientamento del braccio sul quale l'utente sta indossando il gesture recognizer e dal quale dipende quindi il movimento del Cursor.

5.3 Definizione delle API

Le API messe a disposizione dal framework, permettono all'utilizzatore di disegnare sulla Viewport. Queste sono:

- **drawText**, che permette di disegnare una stringa di testo all'interno della Viewport, definendone colore, posizione e dimensioni;
- **drawRectangle**, che permette di disegnare un rettangolo all'interno della Viewport, definendone colore, posizione e dimensioni;
- **drawCircle**, che permette di disegnare un cerchio all'interno della Viewport, definendone colore, posizione e dimensioni;
- **drawPoint**, che permette di disegnare un punto all'interno della Viewport, definendone colore e posizione;
- **drawLine**, che permette di disegnare una linea retta all'interno della Viewport, definendone colore, punto di partenza e di arrivo;
- **drawImage**, che permette di disegnare un'immagine all'interno della Viewport, definendone posizione e dimensioni;
- **lock** e **unlock**, che permettono di bloccare e sbloccare rispettivamente lo scorrimento della Viewport tramite il movimento della testa.

Inoltre, qualsiasi cosa venga disegnata sulla Viewport tramite queste API, può registrare un listener che innescherà un'opportuna callback ogni qualvolta il sistema registri il verificarsi di un Touch Event all'interno dei suoi confini. In altre parole il framework mette a disposizione non solo funzionalità per disegnare elementi all'interno della Viewport, ma anche per interagire con essi. Per finire, tutto ciò che viene disegnato sulla Viewport, deve essere modificabile via programmatica. Ad esempio deve essere possibile modificare il contenuto di un testo illustrato sulla Viewport, o il colore, o le dimensioni o la posizione. Lo stesso vale per ogni altro elemento disegnabile tramite le API sopra illustrate.

Capitolo 6

Progettazione dell'architettura

In questo Capitolo verrà illustrata l'architettura di base del framework. Nel corso della trattazione, sarà possibile notare come molte delle scelte progettuali effettuate, siano state condizionate dall'utilizzo della piattaforma Android.

6.1 Flussi di controllo

Iniziamo mettendo in mostra la suddivisione del carico computazionale sui vari flussi di controllo. Il framework da realizzare, deve mettere a disposizione diverse funzionalità, alcune delle quali richiedono una quantità di risorse di calcolo non indifferente.
Per questo motivo, è stato necessario spartire il tutto su più thread.

6.1.1 Main Thread

Quando si parla di Android, il Main Thread è l'unico thread in grado di aggiornare la View. Ciò significa che l'aggiornamento della Viewport e dei suoi contenuti, potrà essere fatto esclusivamente su questo thread. Tale aggiornamento inoltre, che consiste nell'invalidare la Viewport e ridisegnarla, dovrà avvenire con una frequenza relativamente alta.

Tipicamente, una GUI classica, basata ad esempio su menù e finestre, viene aggiornata con una frequenza che si aggira attorno ai 15 *frames per second* (fps). Benché la Viewport sia anch'essa una GUI, per riuscire a soddisfare efficientemente i suoi requisiti, ha bisogno di un refresh rate più elevato. Ricordiamo infatti, che la Viewport deve essere navigabile tramite il movimento della testa. Per fare in modo che tale navigazione avvenga in maniera fluida, è necessario che il rateo con il quale essa viene ridisegnata, sia alto abbastanza da non venire percepito dall'occhio umano. Questo significa che la nostra

Viewport deve essere ridisegnata all'incirca tra le 27 e le 30 volte al secondo. Ciò tiene già il Main Thread sufficientemente impegnato, impedendogli di dedicarsi anche al calcolo del Gaze (tramite la gestione delle registrazioni provenienti dai sensori inerziali degli smart glass), che deve quindi essere delegato ad un altro thread.

6.1.2 Thread addetto alla gestione dei sensori

Il thread delegato alla gestione dei sensori e quindi al calcolo del Gaze, risulta essere tanto impegnato quanto il Main Thread, per quanto concerne la frequenza delle computazioni che esso deve eseguire. Abbiamo detto infatti, che la Viewport deve avere un refresh rate che si aggira attorno ai 30 fps. Tuttavia, per poter essere ridisegnata, essa ha bisogno di conoscere il valore aggiornato del Gaze. La parte di Viewport che deve essere resa visibile infatti, dipende proprio da dove l'utente sta guardando. Se il Gaze venisse calcolato più lentamente rispetto alla frequenza di aggiornamento della Viewport, l'effetto di scorrimento ottenuto risulterebbe poco fluido, a causa della latenza nell'aggiornamento del Gaze. Questo significa che anche l'aggiornamento del Gaze, deve avvenire con una frequenza che si aggira attorno ai 30 refresh al secondo, come per il ridisegnamento della Viewport effettuato dal Main Thread. Inoltre, tale aggiornamento deve essere effettuato subito prima che la Viewport venga ridisegnata.

Dal momento che ci troviamo su Android inoltre, è possibile inquadrare il ricalcolo della posizione degli smart glass e quindi del Gaze, come un Service. Questo Service però, dovrà essere fermato ogni qualvolta un'applicazione che fa uso del framework passi in background, così da rilasciare i sensori, nel caso qualche altra applicazione debba utilizzarli.

6.1.3 Thread addetto alla gestione della connessione

Rimane ora da gestire il discorso Finger. La posizione del braccio dell'utente, rappresentata dal Finger, è calcolata da un gesture recognizer, indossato proprio su tale braccio. L'unico modo che il sistema ha per mantenere traccia di tale orientamento, è di farselo comunicare dal gesture recognizer. Per poter fare ciò, è necessario stabilire una connessione tra smart glass e gesture recognizer.

Vista la limitata distanza e la limitata quantità di informazioni che devono essere trasferite, il protocollo Bluetooth risulta essere il più adatto per instaurare la connessione. Sarà quindi necessario un ulteriore Service per la gestione di tale connessione ed un protocollo applicativo per far comunicare smart glass e gesture recognizer. Il gesture recognizer infatti, può essere usato per inviare

non solo aggiornamenti sulla sua posizione, ma anche comandi codificati da specifiche gesture, per eseguire un click con il Cursor, o magari bloccare lo scorrimento della Viewport. Anche questo Service quindi, viste le funzionalità che deve mettere a disposizione, ha bisogno di un suo thread sul quale eseguire.

6.2 Comunicazione intra-processo

Spartire le computazioni su più thread, porta per forza di cose a complicazioni quali la gestione di corse critiche. Vista la difficoltà e le limitazioni nell'utilizzo di token di sincronizzazione in Java per evitare questo genere di problemi, è stato scelto di seguire un approccio diverso, organizzando l'architettura del sistema in modo che nessuna corsa critica possa verificarsi.

In questo momento, avendo descritto quelli che sono i principali flussi di controllo all'interno del sistema, la situazione è la seguente:

- Abbiamo il Main Thread, che è l'unico in grado di ridisegnare la Viewport e i suoi contenuti. Per poter fare ciò però, ha bisogno di accedere in lettura a Gaze (per ridisegnare correttamente la Viewport) e Finger (per ridisegnare correttamente il Cursor);

- Il thread che si occupa della gestione dei sensori inerziali, accede in lettura al Gaze;

- Il thread che si occupa della gestione della connesione, accede in lettura al Finger, per aggiornarlo concordemente con ciò che gli viene comunicato dal gesture recognizer.

Per poter evitare corse critiche, è sufficiente eliminare gli accessi in scrittura a Gaze e Finger dei due Service. Non si può intervenire sul Main Thread, perché esso è l'unico che può aggiornare la View, ma nel nostro caso, per farlo, ha bisogno anche di accedere a Gaze e Finger.

6.2.1 IntraProcessMessageHandler

L'architettura è allora stata organizzata in modo che i due thread che si occupano di mantenere aggiornato Gaze e Finger, non settino autonomamente il loro valore, ma si limitino a comunicarlo al Main Thread. In questo modo, l'unico thread ad accedere a Finger, Gaze, Cursor e Viewport, è il Main Thread e questo rimuove ogni possibilità di corsa critica. Questa soluzione inoltre, non innalza di molto il carico di lavoro del Main Thread, al quale si richiede, in più, solo di settare alcuni campi ai valori che gli vengono comunicati.

Per poter far comunicare più flussi di controllo, Android mette a disposizione oggetti chiamati Handler (*android.os.Handler*). Quando viene creato un Handler, questo viene associato ad una coda di messaggi. Ciascun thread in Android (HandlerThread), ha infatti, integrata al suo interno, una coda di messaggi chiamata Looper. Un Handler può poi essere configurato come gestore di tale coda, facendo l'override del suo metodo `handleMessage`. Ogni chiamata effettuata dall'Handler all'interno di `handleMessage`, viene eseguita sul thread al quale appartiene il Looper che l'Handler sta gestendo (i.e. se un Handler è collegato al Looper del Main Thread, tutte le chiamate che esso effettuerà per gestire i messaggi ricevuti, verranno eseguite sul Main Thread). E' sufficiente quindi, assegnare un Handler al Looper del Main Thread (MainLooper) e fare in modo che gli altri thread inviino su tale Looper i loro messaggi. Quando questi verranno gestiti dall'Handler, il codice per la loro gestione verrà eseguito necessariamente sul Main Thread.

Anche il protocollo per la comunicazione intra processo, è messo a disposizione dalla piattaforma. I messaggi inviati sui Looper, sono modellati da degli oggetti chiamati Message (*android.os.Message*). Il contenuto di ciascun messaggio è definito da un flag intero. Ciascun messaggio può poi contenere dei valori (o degli oggetti) extra al loro interno, recuperabili tramite una specifica chiave (codificata da una stringa). Quello che basta fare quindi, è definire una specie di "catalogo", che associa, ad un intero, un significato ed eventualmente specifica le chiavi per recuperare i valori (o oggetti) extra contenuti nei messaggi.

All'interno del nostro sistema, l'Handler che si occupa di gestire i messaggi intra processo, che gli altri thread inviano verso il Main Thread, è chiamato IntraProcessMessageHandler.

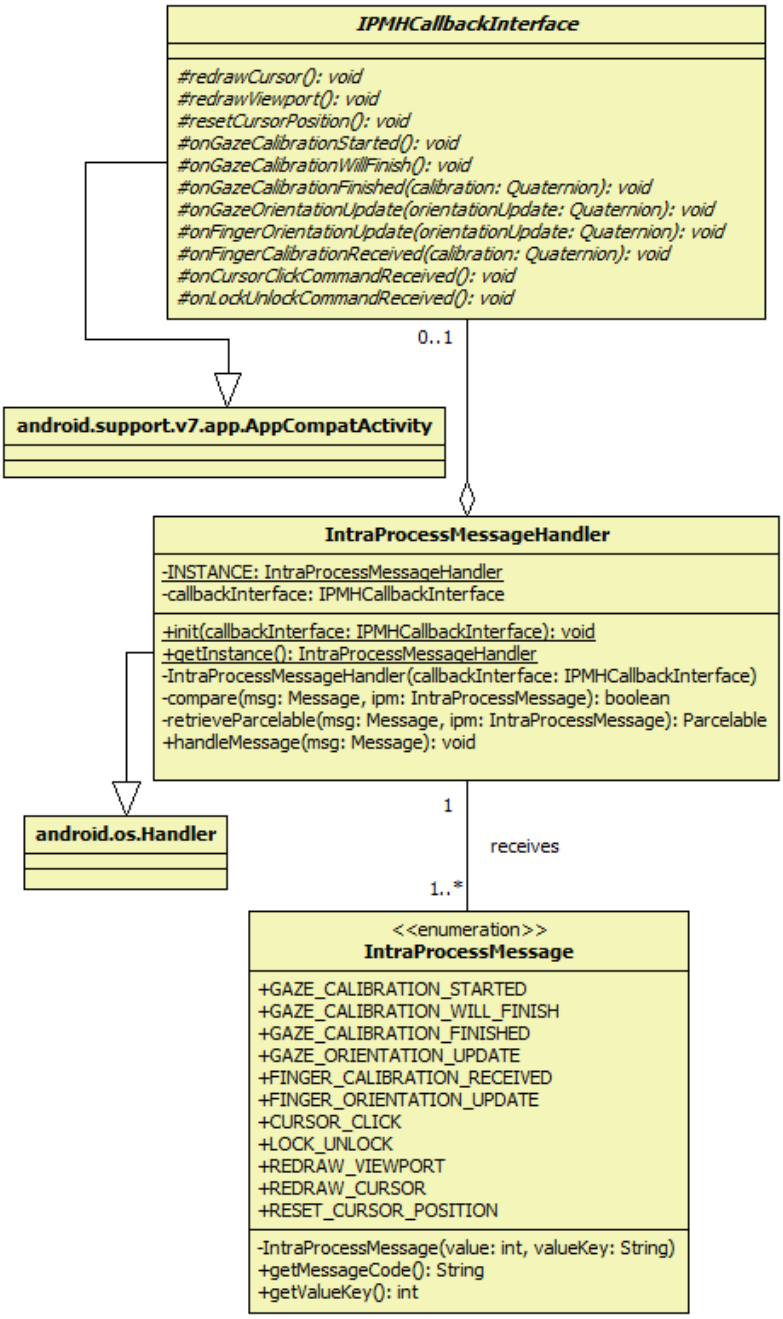


Figura 6.1: *Diagramma delle classi raffigurante l'IntraProcessMessageHandler.*

Il diagramma delle classi rappresentato qui sopra, in Figura 6.1, mostra la struttura dell'IntraProcessMessageHandler. Notiamo innanzitutto l'utilizzo del Singleton pattern, sensato dal momento che deve esistere una sola istanza

di questa classe. Tale pattern inoltre, permette ai Service per la gestione dei sensori e della connessione, di recuperare facilmente l'istanza dell'Handler, alla quale devono inviare i messaggi per l'aggiornamento di Gaze e Finger rispettivamente.

Quando l'IntraProcessMessageHandler riceve un messaggio, dopo averlo decodificato, richiama la callback opportuna dell'IPMHCallbackInterface. L'Handler quindi, ha bisogno di un riferimento ad un oggetto che implementi tale interfaccia (settabile tramite il metodo `init`). Notiamo inoltre, come l'IPMHCallbackInterface non sia propriamente un interfaccia, bensì una classe astratta. Questa scelta è legata al fatto che i metodi di un'interfaccia in Java, devono essere esclusivamente `public`, mentre nel nostro caso, trattandosi di callback, sarebbe bene che queste fossero `protected`. Inoltre, modellando l'IPMHCallbackInterface come classe astratta e facendole estendere AppCompatActivity, ha permesso anche di esprimere il vincolo per cui chi implementa le callback necessarie all'IntraProcessMessageHandler per la gestione dei messaggi intra processo, debba essere per forza un'Activity.
Qui sotto, in Figura 6.2, viene mostrato un diagramma di sequenza che illustra lo scambio di messaggi tra i vari flussi di controllo.

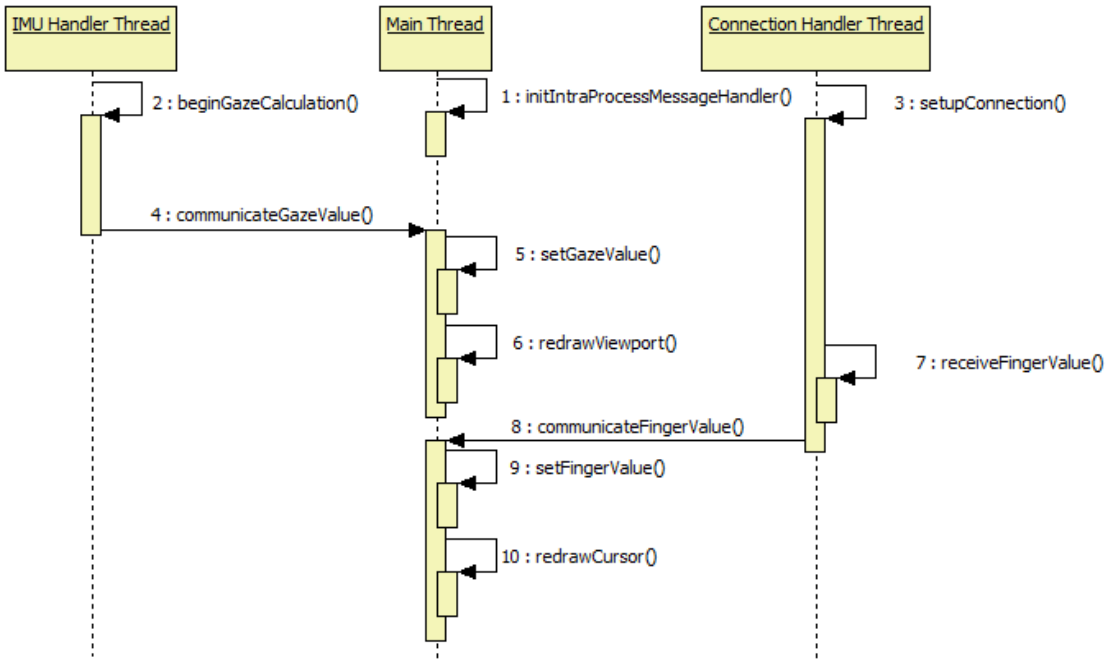


Figura 6.2: *Diagramma di sequenza che mostra l'interazione tra i flussi di controllo.*

6.3 Definizione del nucleo dell'architettura

Definiamo ora quello che è il core dell'architettura del sistema, ovvero il controller principale.
Come accade tipicamente quando si sviluppa in Android, è stato scelto di affidare il ruolo di controller ad un'Activity. Tale Activity, sarà la stessa Activity che implementerà IPMHCallbackInterface e le callback necessarie per l'handling dei messaggi da parte dell'IntraProcessMessageHandler.

6.3.1 ViewportActivity

L'Activity che funge da nucleo dell'architettura, è chiamata ViewportActivity. Oltre ad implementare IPMHCallbackInterface e le callback in essa definite, essa presenta anche la particolarità di accettare solo una Viewport come content View.

Qua sotto, in Figura 6.3, viene mostrato un diagramma delle classi che raffigura quello che è il nucleo dell'architettura del sistema, inclusa, ovviamente la ViewportActivity. Notiamo, innanzitutto, che essa è una classe astratta e che quindi, un'applicazione che fa uso del framework, dovrà innanzitutto dichiarare un'Activity che estenda tale classe.

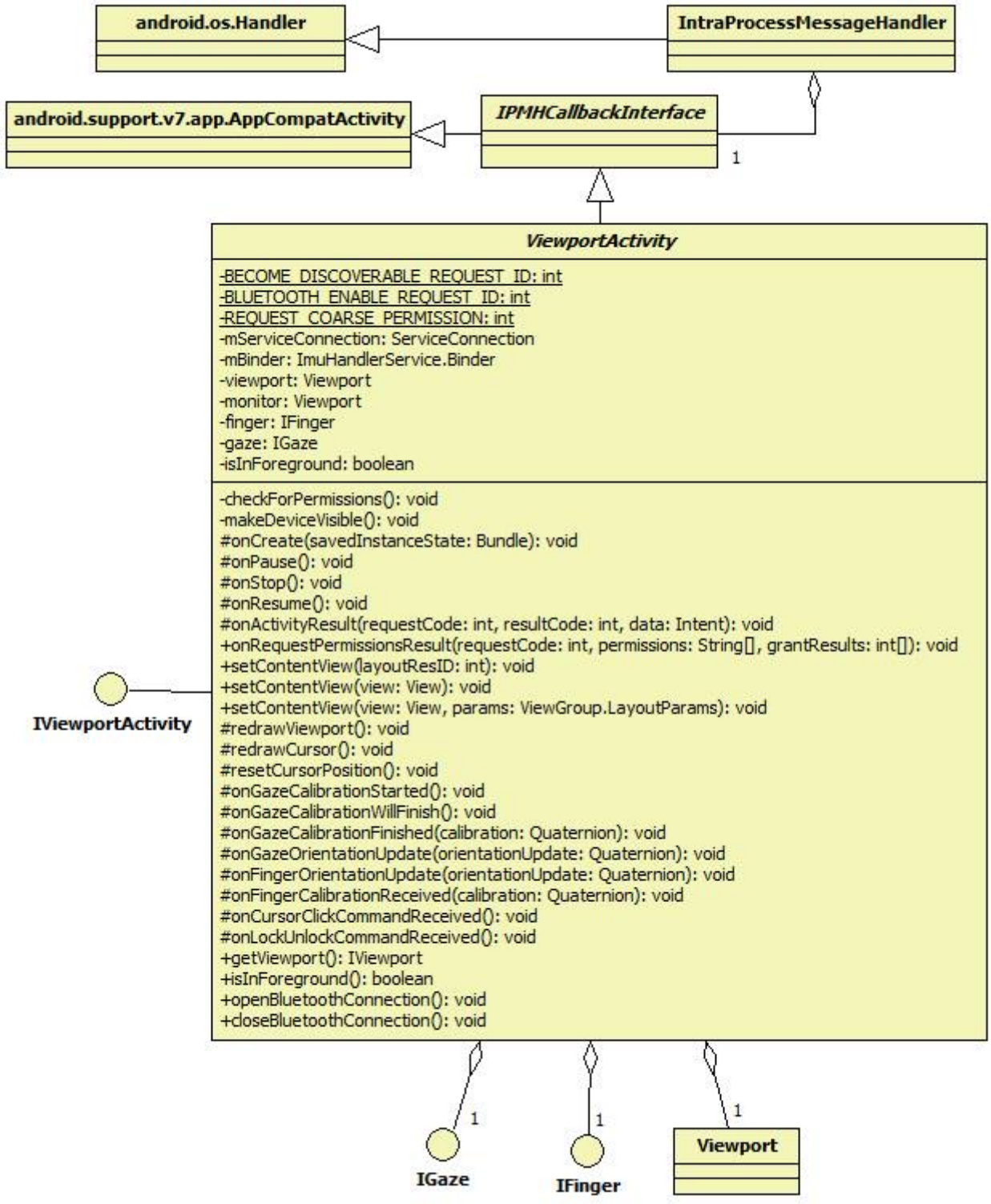


Figura 6.3: *Diagramma delle classi che illustra il core architetturale.*

6.3.2 Esempio di funzionamento del sistema

Concludiamo la trattazione del Capitolo con un esempio di funzionamento del sistema, che mette in mostra l'interazione tra le varie parti dell'architettura finora descritte per ridisegnare la Viewport.

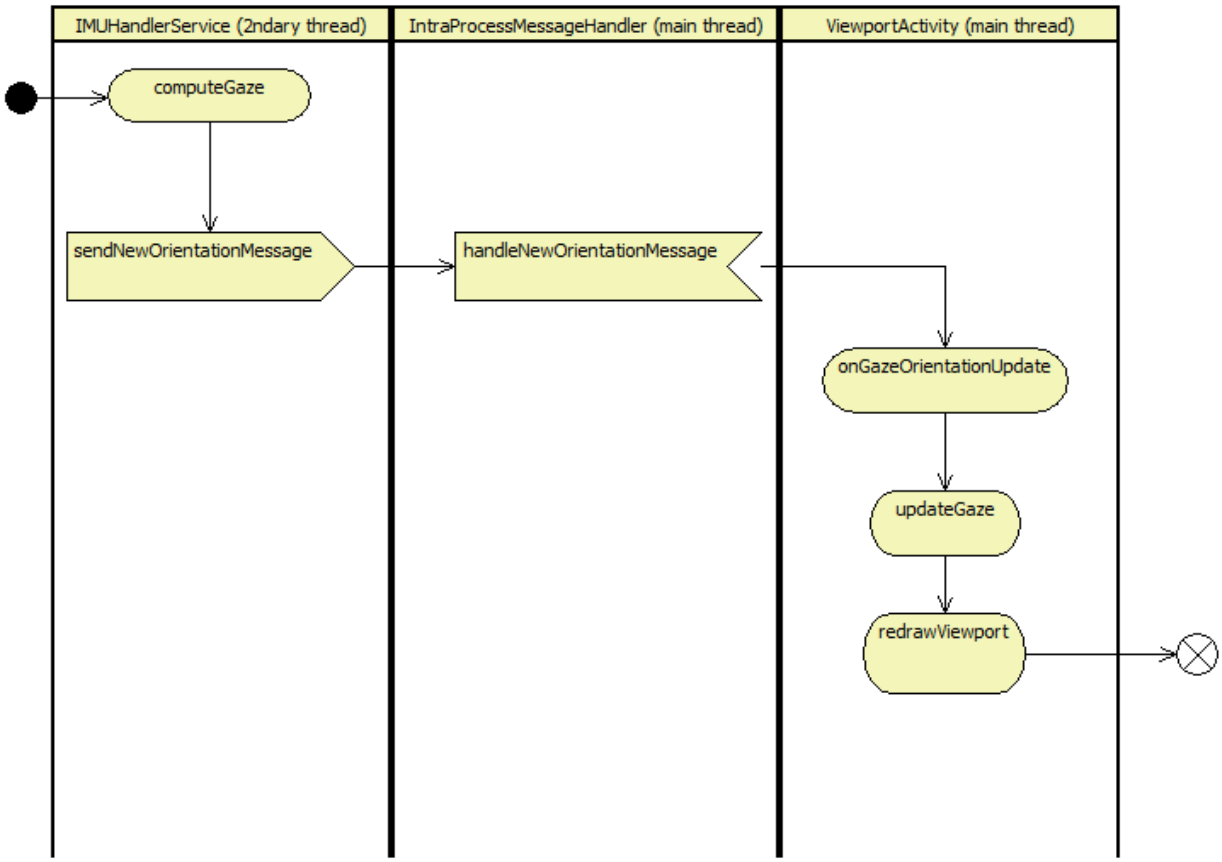


Figura 6.4: *Diagramma di attività che illustra come venga ridisegnata la Viewport, seppure ancora in maniera astratta.*

Capitolo 7

Tracking del Gaze

Questo Capitolo, verrà dedicato all'analisi dettagliata della parte di sistema impiegata nel tracking del Gaze. Verrà descritto come il sistema mantiene traccia dell'orientamento dello sguardo dello wearer e come questo venga impiegato per far scorrere la Viewport.

7.1 Come far scorrere la Viewport

Il sistema, come sappiamo, deve permettere la navigazione della Viewport tramite il movimento della testa. Per poter tracciare tale movimento, è necessario memorizzare la posizione iniziale e calcolare poi la differenza tra questa e quelle successivamente misurate. Il sistema fa infatti uso di una fase di calibrazione, durante la quale viene calcolata la posizione della testa dell'utente, manipolando opportunamente le registrazioni dei sensori inerziali a bordo degli smart glass sui quali il framework sta eseguendo.

Una volta calcolata questa posizione iniziale, la Viewport viene disposta in modo da apparire centrata rispetto a tale posizione. Lo scorrimento viene fatto sulla base della differenza tra gli orientamenti successivamente calcolati e l'orientamento iniziale ottenuto al termine della fase di calibrazione. Questa differenza viene espressa dagli angoli di pitch e yaw relativi alla rotazione da eseguire per passare dall'orientamento calcolato al termine della fase di calibrazione, all'orientamento attuale.
Vengono utilizzati questi due angoli, per via di come sono mappati gli assi del sistema di riferimento utilizzato dai sensori per il calcolo della posizione, sul quale ci concentreremo verso la fine del Capitolo. La Viewport verrà fatta scorrere orizzontalmente e verticalmente sulla base dei due angoli riportati. Ciò significa che questi angoli dovranno essere ricalcolati ogni volta che un nuovo orientamento degli smart glass (e quindi della testa dell'utente) viene calcolato.

7.2 Servizi per la gestione dei sensori

Passiamo ora ad illustrare i Service, utilizzati all'interno del sistema, per calcolare l'orientamento degli smart glass e dunque della testa dell'utente (Gaze). Per poter rendere più facilmente estendibile e modificabile il framework, è stato scelto di definire un Service astratto che si occupi di offrire funzionalità di base di setup e di deallocazione delle risorse necessarie per l'utilizzo dei sensori su Android. Un Service che voglia utilizzare i sensori, può poi estendere tale servizio e preoccuparsi esclusivamente di definire quali sensori esso voglia utilizzare e come intenda farlo, senza doversi preoccupare del codice per richiedere l'accesso a tali sensori e rilasciarli nei momenti opportuni. Tale Service astratto inoltre, si occupa anche di spostare le computazioni per la gestione delle registrazioni di ciascun sensore, su dei thread a parte (rispettando l'organizzazione dei flussi di controllo vista nel Capitolo precedente).

7.2.1 ImuHandlerService

Su Android, quando un'applicazione passa in background, è bene che questa rilasci le risorse del dispositivo che sta impegnando, in modo che altre applicazioni possano utilizzarle. Nel nostro caso, tali risorse sono i sensori inerziali. Ogni Service che voglia utilizzare i sensori inerziali, dovrà preoccuparsi di registrarsi come listener per tali sensori ed annullare tale registrazione ogni qualvolta l'applicazione passi in background. Nel caso in cui si voglia definire un nuovo Service per la gestione dei sensori inerziali quindi, che implementi magari un algoritmo diverso per il calcolo del Gaze, sarebbe necessario riscrivere il codice per la richiesta e il rilascio dei sensori.
Sarebbe necessario inoltre, organizzare il tutto in modo che tale algoritmo esegua su un flusso di controllo a parte, per garantire un corretto funzionamento del sistema. Per evitare tutto ciò e rendere il framework più facilmente estendibile, è stato inserito un Service astratto chiamato ImuHandlerService, che si occupa di offrire queste funzionalità a tutte le sue sottoclassi.

Nel caso in cui si voglia cambiare il modo in cui il Gaze viene calcolato, basterà definire un nuovo Service, fargli estendere ImuHandlerService e preoccuparsi unicamente di definire quali sensori utilizzare e come farlo. Più nel dettaglio, la sottoclasse dovrebbe aggiungere i sensori che vuole utilizzare alla `sensorList` di ImuHandlerService nell'`onCreate` e chiamare, subito dopo, il metodo `registerSensors` (sempre di ImuHandlerService). Questo metodo si occupa di registrare la classe come listener per i sensori specificati ed alloca anche un thread per ciascuno di essi, in modo che le computazioni relative alla gestione delle registrazioni di ogni sensore, siano eseguite su un flusso di controllo dedicato. La sottoclasse dovrà poi definire come intende

manipolare le registrazioni fornitegli dai sensori, facendo l'override del metodo `onSensorChanged` dell'interfaccia SensorEventListener (appartenente all'SDK Android) che ImuHandlerService implementa. Il codice per il rilascio dei sensori e la distruzione dei thread ad essi dedicati, è già definito nell'`onDestroy` di ImuHandlerService. Sarà sufficiente distruggere il Service che lo estende ogni qualvolta l'applicazione passa in background, per eseguire il rilascio e il cleanup delle risorse.
Qui sotto, in Figura 7.1, viene illustrato il diagramma UML di ImuHandlerService e di un Service che lo estende, chiamato SensorFusionService.

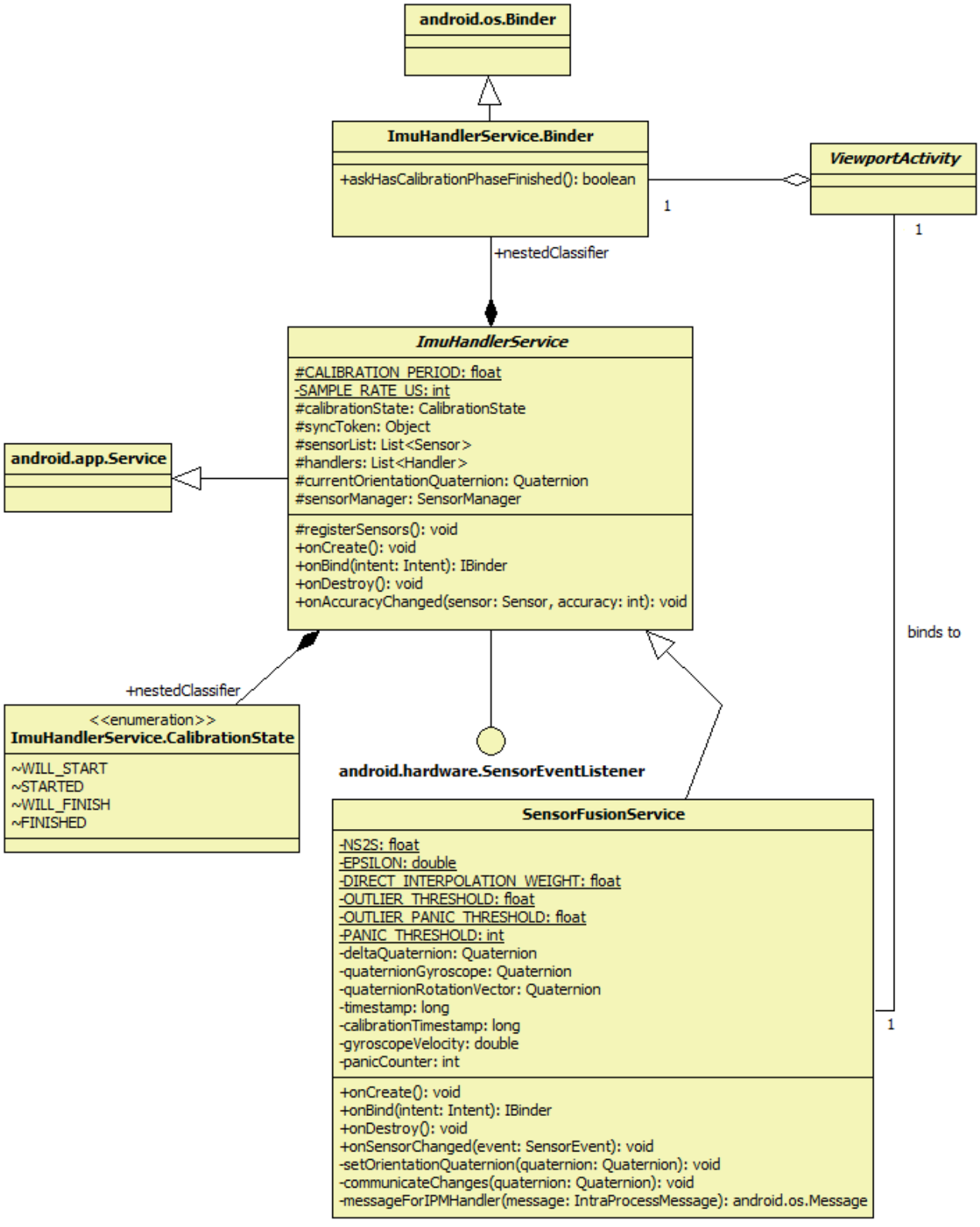


Figura 7.1: *Diagramma delle classi raffigurante ImuHandlerService e una sua sottoclasse.*

Notiamo come ImuHandlerService metta a disposizione di eventuali sottoclassi, anche tutto il necessario per implementare la fase di calibrazione.

7.2.2 SensorFusionService

SensorFusionService è il Service che implementa l'algoritmo di sensor fusion utilizzato attualmente all'interno del framework per il calcolo del Gaze. Tale algoritmo (preso da un'applicazione open source [20] e poi riadattato alle esigenze del nostro caso) si basa sull'utilizzo di un filtro complementare che unisce le misurazioni di rotation vector sensor e giroscopio. Il rotation vector, è un sensore particolare messo a disposizione dall'SDK Android, che esprime, sotto forma di quaternione, la posizione del dispositivo, calcolata sulla base delle registrazioni di magnetometro e accelerometro.

L'algoritmo registra, su un thread, la posizione del dispositivo ottenuta tramite rotation vector e su un altro, la posizione del dispositivo ottenuta integrando nel tempo le registrazioni del giroscopio. Per quanto riguarda il lato giroscopio, le sue misurazioni cominciano a venire campionate solo dopo che un orientamento iniziale è stato stabilito dal rotation vector sensor, in quanto, come già detto nel Capitolo 4, il giroscopio può solo calcolare la variazione nella posizione ed ha quindi bisogno di un punto di partenza, offerto in questo caso, proprio dal rotation vector. Il ricalcolo della posizione tramite i sensori viene fatto ogni 33 millisecondi circa, in modo da ottenere una frequenza di aggiornamento della posizione del dispositivo (e quindi del Gaze) che si aggira attorno ai 30 aggiornamenti al secondo (soglia minima richiesta per il ridisegnamento della Viewport). L'algoritmo inoltre, fonde le misurazioni dell'orientamento ottenute sulla base delle registrazioni dei due sensori ed espresse sotto forma di quaternioni, tramite *spherical linear interpolation* (slerp). La fusione viene fatta in modo che le registrazioni del rotation vector, influiscano solo di poco su quelle del giroscopio (il peso utilizzato per l'interpolazione, è un'euristica, come la maggior parte delle costanti utilizzate nell'algoritmo). Tale fusione, viene però fatta solo nel caso in cui le due misurazioni dell'orientamento (sia quella ottenuta tramite rotation vector, che quella ottenuta tramite giroscopio) siano all'incirca uguali. Nel caso in cui le due si discostino invece, è probabile che almeno uno dei due sensori, abbia effettuato una misurazione non corretta. Per distinguere quali tra i due ha "sbagliato", l'algoritmo si basa sulla seguente supposizione:

- Se si verificano errori sporadici, questi vengono attribuiti al rotation vector sensor. Il motivo è dovuto al fatto che tale sensore si basa su magnetometro e accelerometro, sensori più vulnerabili ad interferenze rispetto al giroscopio.

Quando errori del genere si verificano, la fusione delle misurazioni della posizione, non viene effettuata, ma viene considerata solo quella ottenuta tramite il giroscopio;

- Se si verificano invece molti errori consecutivi, è improbabile che questi siano dovuti a continue interferenze. Il problema non è quindi legato al rotation vector, bensì al giroscopio e, più precisamente, al fatto che esso stia driftando, portandosi dietro un errore accumulato nel corso delle varie integrazioni.
  Quando ciò accade, l'orientamento ottenuto tramite le registrazioni del giroscopio, viene reinizializzato al valore calcolato invece dal rotation vector, così da eliminare la discrepanza nelle misurazioni dei due sensori. Questo fenomeno prende il nome di *panic reset*.

Vediamo ora come tale Service, comunica i risultati al Main Thread, in modo che esso riesca ad aggiornare il Gaze concordemente.
I primi 15 secondi di esecuzione dell'algoritmo, vengono dedicati alla fase di calibrazione. La durata di questa fase è stata scelta a seguito di diversi test, dai quali è emerso che tale algoritmo ha bisogno, tipicamente, di eseguire per circa 10 secondi, prima di fornire una misurazione affidabile della posizione del dispositivo sul quale si trovano i sensori. Questa fase di calibrazione inoltre, viene ripetuta ad ogni panic reset. Ciò è necessario per garantire il corretto funzionamento del sistema. Al termine della fase di calibrazione, il Service informa il Main Thread inviando sul suo Looper un messaggio di calibrazione terminata, contenente il quaternione che rappresenta la posizione iniziale calcolata. Sarà poi l'IntraProcessMessageHandler, una volta ricevuto tale messaggio, ad innescare la callback `onGazeCalibrationFinished` della ViewportActivity, la quale setterà l'orientamento ricevuto come orientamento iniziale del Gaze.

Una volta terminata la fase di calibrazione, ogni volta che il Service calcola una nuova posizione, eseguendo o meno la fusione tra le misurazioni basate sui due sensori, questa viene comunicata, sempre tramite messaggio intra processo, al Main Thread e ricevuta quindi dall'IntraProcessMessageHandler, che innescherà, questa volta, la callback `onGazeCalibrationUpdate` della ViewportActivity.
All'interno di tale callback, verrà comunicato il nuovo orientamento al Gaze. Sarà poi lui ad occuparsi di trovare la differenza tra questo e quello iniziale, in modo da ottenere gli angoli necessari allo scorrimento della Viewport.

7.3 Gaze

Illustriamo ora, con un diagramma delle classi, come il Gaze venga modellato all'interno del sistema.

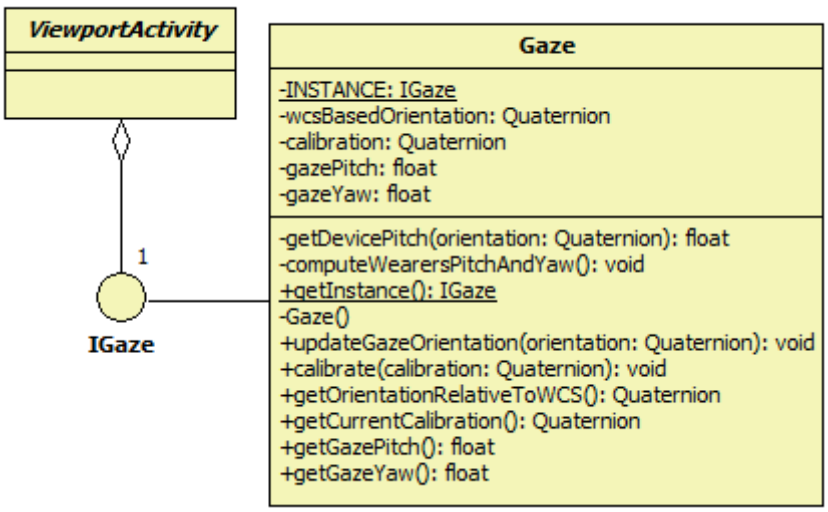


Figura 7.2: *Diagramma delle classi del Gaze.*

L'utilizzo del Singleton pattern, è dovuto al fatto che debba esistere una sola istanza del Gaze.

7.3.1 Aggiornamento del Gaze

I metodi `calibrate` e `updateGazeOrientation` vengono utilizzati, rispettivamente, per settare la posizione iniziale degli smart glass e quelle successivamente calcolate. Sulla base di queste due, come già detto, verranno calcolati gli angoli utilizzati per far scorrere la Viewport. Tali angoli vengono calcolati dal Gaze stesso. Ogni chiamata ad `updateGazeOrientation` infatti, include una chiamata a `computeWearersPitchAndYaw`, che esegue, su un AsyncTask (in modo da lasciare libero il Main Thread, dal quale `updateGazeOrientation` viene chiamato), la differenza tra i due quaternioni che esprimono la posizione iniziale e quella corrente, fornita ad `updateGazeOrientation` (tale differenza viene fatta moltiplicando il quaternione che rappresenta l'orientamento corrente per l'inverso di quello che rappresenta invece l'orientamento iniziale). Una volta ottenuta tale differenza, viene calcolato, sulla base di questa, il valore degli angoli necessari per far scorrere concordemente la Viewport. Fatto ciò, prima di terminare, l'AsyncTask invia un messaggio all'IntraProcessMessageHandler. Tale messaggio, informa che il sistema ha ora a disposizione

tutto l'occorrente per ridisegnare la Viewport. L'IntraProcessMessageHandler quindi, innescherà la callback `redrawViewport` della ViewportActivity.

7.3.2 Remapping del sistema di coordinate

Abbiamo già menzionato, che gli angoli utilizzati per far scorrere la Viewport, sono di quelli di pitch e yaw. L'utilizzo di questi due angoli, dipende dal modo in cui sono disposti gli assi del sistema di riferimento utilizzato dai sensori per il calcolo della posizione. L'algoritmo utilizzato infatti, utilizza come punto di partenza, una posizione registrata tramite il rotation vector sensor. Questo sensore, utilizza come sistema di riferimento il *World Coordinate System* (WCS).
Gli assi del WCS, sono disposti come segue:

- L'asse x è tangenziale alla superficie terrestre e punta, indicativamente, ad Est;
- L'asse y è ortogonale all'asse x e punta verso il Nord magnetico;
- L'asse z è ortogonale agli altri due e punta verso il cielo.

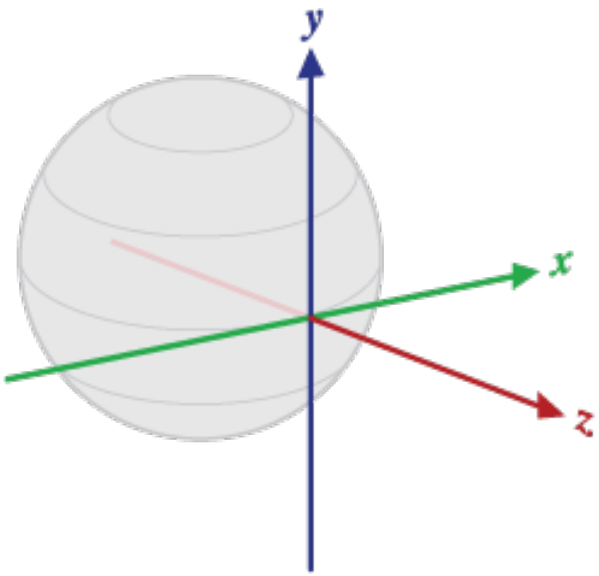


Figura 7.3: *Rappresentazione del World Coordinate System, detto anche ENU based (East - North - Up).*

Basandosi su questo sistema di coordinate, è possibile associare all'angolo di yaw (angolo attorno all'asse z) la magnitudine del movimento orizzontale della

testa, sulla base del quale, verrà fatta scorrere orizzontalmente la Viewport. Più in particolare, la magnitudine dello spostamento orizzontale della testa, sarà identificata dal valore assoluto dell'angolo di yaw.
Sappiamo inoltre che gli angoli di rotazione attorno ad un asse hanno segno positivo quando rappresentano una rotazione antioraria.
Nel caso dello yaw quindi, si avrà un valore positivo quando questo rappresenta una rotazione verso sinistra, mentre si avrà un valore negativo quando questo rappresenta una rotazione verso destra (prendendo sempre, come punto di riferimento, la posizione calcolata al termine della fase di calibrazione). Per poter identificare anche il movimento verticale però, è necessario rimappare gli altri due assi.

E' stato scelto quindi, di rimappare l'asse x del WCS, all'asse -x del dispositivo Android. Il sistema di riferimento degli assi Android, è già stato illustrato nel Capitolo 4. L'asse -x, in ogni caso, è inquadrabile come un asse tangente allo schermo del dispositivo (quindi dei nostri smart glass) e diretto verso sinistra.
Utilizzando questo asse come riferimento, è possibile inquadrare l'angolo di pitch (attorno all'asse x) come l'angolo di rotazione della testa verso l'alto (se il pitch è positivo) o verso il basso (se il pitch è negativo). Per effettuare tale rimappaggio, occorre utilizzare i metodi messi a disposizione del SensorManager di Android. Qui sotto, in Figura, viene mostrato lo snippet di codice che effettua tale operazione.

```
private float getDevicePitch(final Quaternion orientation) {
        final Matrixf4x4 rm = new Matrixf4x4();
        // Get the rotation matrix from the given quaternion
        SensorManager.getRotationMatrixFromVector(rm.getMatrix(),
            orientation.ToArray());
        final Matrixf4x4 rmRemapped = new Matrixf4x4();
        // Remap the axes to the following, so that our pitch will be
            relative to the -x axis of the device
        // (tangential to the screen and pointing left) - therefore a
            pitch > 0 will indicate a rotation towards
        // the sky, a pitch < 0 will indicate a rotation towards the
            ground.
        // The y axis is remapped to the z axis so that the roll will
            be the angle around the z axis of the device
        // (coming out of the screen towards the user) - this is not
            used for now.
        SensorManager.remapCoordinateSystem(rm.getMatrix(),
            SensorManager.AXIS_MINUS_X, SensorManager.AXIS_Z,
                rmRemapped.getMatrix());
```

```
        float[] angles = new float[3];
        // Recover the Euler angles (yaw in 0, pitch in 1, roll in 2)
           that express the remapped orientation
        SensorManager.getOrientation(rmRemapped.getMatrix(), angles);
        // Return only the pitch angle
        return angles[1];
    }
```

Listato 7.1: *Come viene ottenuto l'angolo di pitch rimappato.*

Il pitch relativo all'asse -x del dispositivo, dovrà essere ricalcolato sia per la posizione iniziale, sia per quelle successivamente misurate, in modo da poterne fare la differenza ed ottenere l'angolo, sulla base del quale, la Viewport verrà fatta scorrere verticalmente (tale angolo rappresenta la rotazione verso l'alto, o verso il basso, della testa, rispetto alla posizione ottenuta al termine della fase di calibrazione).

7.4 Diagramma di attività

In Figura 7.4, è illustrato un diagramma di attività che riassume il funzionamento della sotto parte di sistema descritta in questo Capitolo.

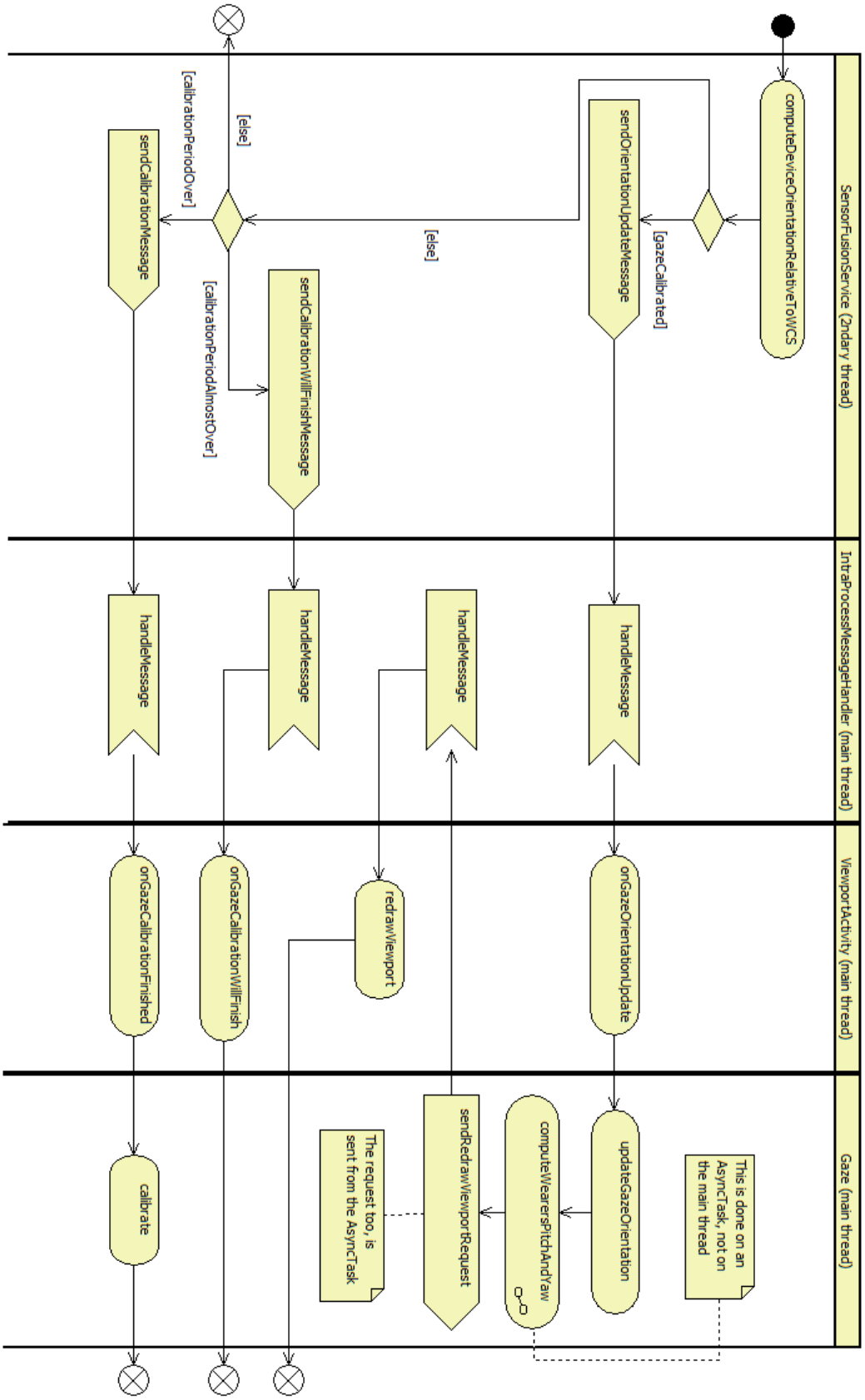


Figura 7.4: *Diagramma di attività che mostra come il Gaze venga aggiornato.*

7.5 Recuperare ed utilizzare il Gaze

Nel caso in cui un'applicazione abbia bisogno di recuperare i valori di orientamento mantenuti all'interno della classe Gaze, essa può farlo recuperando inizialmente l'istanza del Gaze tramite il metodo `getGaze` della ViewportActivity ed utilizzando poi i metodi appropriati, come mostrato nello snippet qui sotto.

```
public void gazeUse(final ViewportActivity context) {
   // Recovering the instance
   final IGaze instance = context.getGaze();

   // This is how to recover the orientation relative to the WCS
   // for a personal purpose
   final Quaternion wcsOrientation =
       instance.getOrientationRelativeToWCS();
   this.useWCSOrientation(wcsOrientation);

   // This is how to recover the orientation relative to the WCS
   // that represents the starting position (the one recorded at
   // the end of the calibration phase) of the Gaze and use it for
   // a personal purpose
   final Quaternion wcsCalibration =
       instance.getCurrentCalibration();
   this.useWCSCalibration(wcsCalibration);

   // This is how to recover the pitch and yaw angles of the Gaze
   // relative to the starting position. These are also the angles
   // used to scroll the Viewport
   final float pitch = instance.getGazePitch();
   final float yaw = instance.getGazeYaw();
   this.useGazePitchAndYaw(pitch, yaw);

}
```

Listato 7.2: *Esempio di recupero ed utilizzo del Gaze.*

Capitolo 8

Tracking del Finger

In questo Capitolo, verrà analizzata la parte di sistema che si occupa di aggiornare il Finger. Verrà mostrato come il sistema recupera e mantiene traccia della posizione del gesture recognizer e come fa uso di tale posizione per spostare, concordemente, il Cursor.

8.1 Come far muovere il Cursor

Similmente a ciò che è stato detto per Gaze e Viewport, anche per spostare il Cursor all'interno della Viewport, è necessario mantenere traccia dello spostamento del Finger dalla sua posizione iniziale. Anche qui quindi, c'è bisogno di una fase di calibrazione. Come vedremo però, per quanto riguarda il Finger, le cose sono leggermente più semplici, in quanto il calcolo della posizione del gesture recognizer viene delegato direttamente a tale dispositivo e non deve essere quindi il sistema ad occuparsene.

Facendo la differenza tra orientamento corrente ed orientamento iniziale (ottenuto in fase di calibrazione), otteniamo la rotazione necessaria per raggiungere la posizione attuale. Da tale rotazione, è possibile ricavare gli angoli di pitch e yaw sulla base dei quali è possibile mappare il movimento del Cursor, similmente a come viene fatto per Gaze e Viewport.

8.2 Servizio per la gestione della connessione

Mentre il Gaze veniva calcolato direttamente dal sistema, sulla base delle registrazioni dei sensori inerziali degli smart glass, per calcolare il Finger, è necessario affidarsi alle registrazioni dei sensori a bordo del gesture recognizer. L'unico modo per recuperare tali registrazioni, è quello di farsele comunicare

direttamente dal gesture recognizer. Per raggiungere tale obiettivo, il sistema mette a disposizione un Service di connessione.

Questo Service, configura il sistema come un Bluetooth server in grado di accettare, al massimo, solo una connessione (quella proveniente appunto dal gesture recognizer). Tale servizio inoltre, è programmato in modo da rendere il sistema sempre e comunque raggiungibile dal gesture recognizer. Nel caso infatti, in cui la connessione con tale dispositivo cada, il Service se ne accorge e riapre immediatamente la welcome socket, in modo che il sistema sia nuovamente raggiungibile. Questo servizio, oltre a gestire la connessione, deve essere anche in grado di riconoscere e reagire opportunamente ai messaggi ricevuti dal gesture recognizer. Dal momento che tale dispositivo viene usato non solo per muovere il Cursor, ma anche per l'invio di specifici comandi (codificati da gesture), quali il click o il blocco dello scorrimento della Viewport, per riuscire a comunicare, sistema e gesture recognizer hanno bisogno di un opportuno protocollo applicativo.

Va inoltre detto, che la connessione tra gesture recognizer e sistema, viene immaginata non come una connessione diretta, ma come una connessione mediata da uno smart phone. Questa scelta è dovuta sia a motivi legati all'hardware, sia all'obiettivo di poter offrire orizzonti applicativi più ampi, coinvolgendo nel sistema anche lo smart phone e, di conseguenza, tutte le funzionalità che può mettere a disposizione.

8.2.1 Protocollo applicativo

Visto che entrambi i lati della connessione sono dispositivi Android based (smart phone lato client e smart glass, sui quali è in esecuzione il framework, lato server), è stato scelto di utilizzare come protocollo applicativo, l'invio di oggetti serializzabili. Ciascun messaggio all'interno del protocollo, è modellato da un oggetto. I messaggi utilizzati per trasportare i comandi inviati dal gesture recognizer al sistema (quali ad esempio il click), vengono codificati da un oggetto Message, che contiene un flag per distinguerne il contenuto, ovvero quale comando tale messaggio sta trasportando.

Aldilà dei messaggi semplici, esistono due tipi di messaggi più specifici, i CalibrationMessage e gli OrientationChangedMessage, che vengono invece utilizzati, rispettivamente, per settare la posizione iniziale o per comunicare la nuova posizione del gesture recognizer. Per poter fare ciò, questi messaggi, oltre ad un flag che ne distingue il contenuto, hanno bisogno di contenere anche l'orientamento che vuole essere comunicato (codificato da un quaternione).
In Figura 8.1, viene mostrato un diagramma delle classi che illustra la struttura del protocollo applicativo. Il `content` di ciascun messaggio, che ne distingue

il contenuto, altro non è che il valore di un'enumerazione che definisce tutti i possibili messaggi.

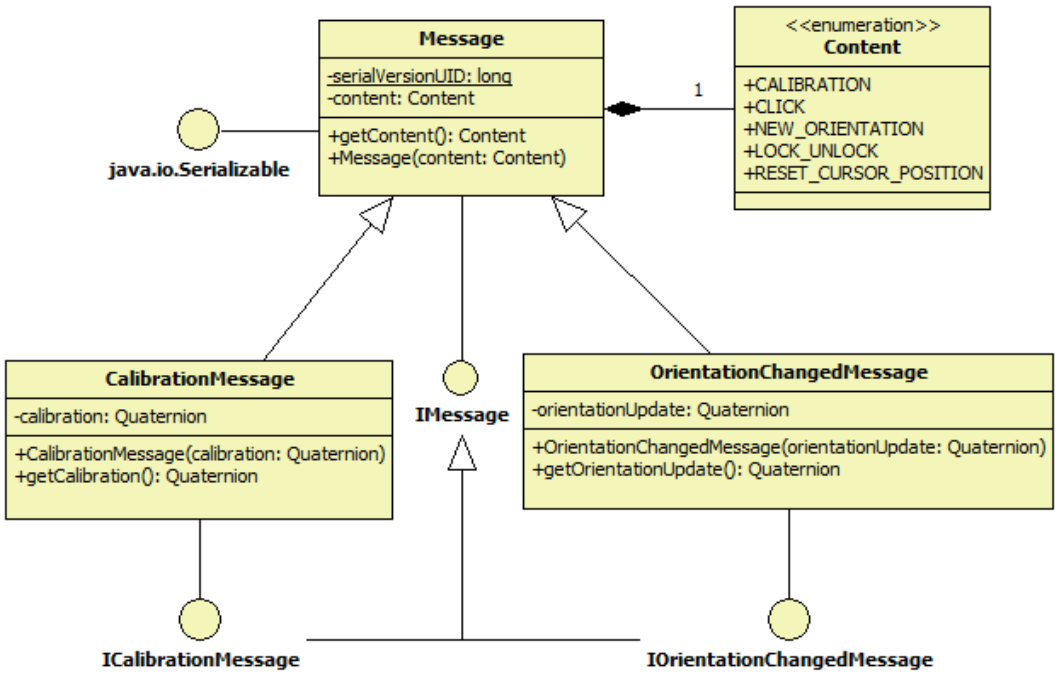


Figura 8.1: *Diagramma delle classi che illustra il protocollo applicativo.*

Il significato di ciascun contenuto è il seguente:

- Il contenuto *CALIBRATION* identifica un CalibrationMessage;
- Il contenuto *CLICK* identifica un messaggio che richiede il dispatch di un TouchEvent nella posizione in cui si trova il Cursor;
- Il contenuto *NEW ORIENTATION* identifica un OrientationChanged-Message;
- Il contenuto *LOCK UNLOCK* identifica un messaggio che richiede di bloccare, o sbloccare (a seconda della situazione in cui ci si trova) la Viewport;
- Il contenuto *RESET CURSOR POSITION* identifica un messaggio che richiede di riportare il Cursor al centro della parte di Viewport visibile all'utente (ovvero al centro del FieldOfView).

Il protocollo, per ora, non prevede alcun messaggio di risposta da parte del server, che si limita ad ascoltare.

8.2.2 MessageParserService

Qui sotto, in Figura 8.2, viene mostrato un diagramma delle classi che illustra le proprietà del MessageParserService.

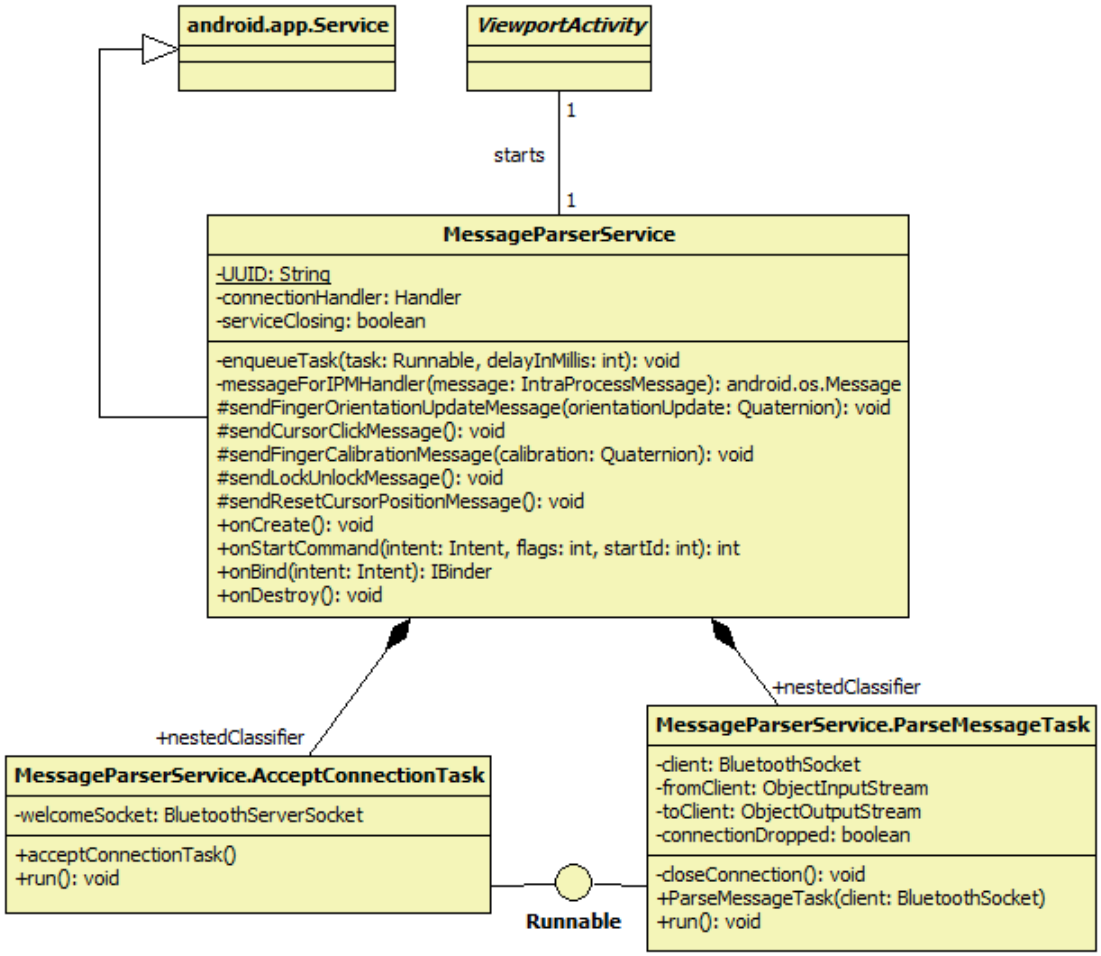


Figura 8.2: *Diagramma della classe MessageParserService.*

Questo Service, come abbiamo già detto, implementa le funzionalità di gestione della connessione e di parsing dei messaggi. Più nel dettaglio, il Service, dopo aver ricevuto un messaggio proveniente dal gesture recognizer, ne costruisce un altro del tipo *android.os.Message*, che abbia lo stesso contenuto

di quello ricevuto. Il nuovo messaggio, può essere poi inviato all'IntraProcessMessageHandler, che si occuperà di innescare la callback opportuna della ViewportActivity:

- `onFingerCalibrationReceived`, nel caso in cui il messaggio ricevuto dall'Handler sia stato costruito a partire da un CalibrationMessage ricevuto dal MessageParserService;
- `onFingerOrientationUpdate`, nel caso in cui il messaggio ricevuto dall'Handler sia stato costruito a partire da un OrientationChangedMessage ricevuto dal MessageParserService;
- `onCursorClickCommandReceived`, nel caso in cui il messaggio ricevuto dall'Handler sia stato costruito a partire da un Message con contenuto *CURSOR CLICK*;
- `onLockUnlockCommandReceived`, nel caso in cui il messaggio ricevuto dall'Handler sia stato costruito a partire da un Message con contenuto *LOCK UNLOCK*;
- `resetCursorPosition`, nel caso in cui il messaggio ricevuto dall'Handler sia stato costruito a partire da un Message con contenuto *RESET CURSOR POSITION*.

Sulla base di queste callback, verrà aggiornato il Finger, similmente a come viene fatto per il Gaze. Diversamente da quanto accadeva per il Gaze però, la calibrazione del Finger, è pensata per essere immediata ed effettuabile qualora si voglia. La calibrazione del Finger infatti, è effettuata tramite una gesture, con la quale viene comunicato al sistema di settare l'orientamento del gesture recognizer, nel momento in cui tale gesture è stata effettuata, come posizione iniziale.
In questo modo è possibile cambiare il sistema di riferimento per il movimento del Cursor a proprio piacimento.

8.3 Finger

La struttura della classe che modella il concetto di Finger, è illustrata in Figura 8.3.

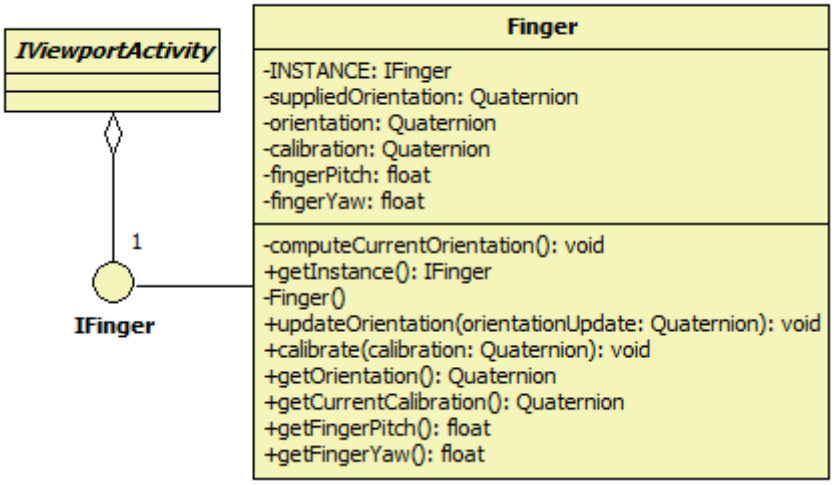


Figura 8.3: *Diagramma della classe Finger.*

Notiamo come questa sia molto simile a quella del Gaze, incluso l'utilizzo del Singleton pattern.

8.3.1 Aggiornamento del Finger

L'aggiornamento del Finger, avviene in maniera quasi del tutto identica all'aggiornamento del Gaze.
I metodi `calibrate` e `updateFingerOrientation`, utilizzati rispettivamente per settare la posizione iniziale (`calibrate`) e la nuova posizione ricevuta (`updateFingerOrientation`), vengono richiamati all'interno delle callback `onFingerCalibrationReceived` e `onFingerOrientationUpdate`.

All'interno del metodo per aggiornare la posizione del Finger inoltre, è inclusa una chiamata ad un altro metodo, chiamato `computeFingersPitchAndYaw`, che, analogamente al `computeWearersPitchAndYaw` del Gaze, esegue, su un AsyncTask, le trasformazioni e i calcoli per ottenere gli angoli di pitch e yaw necessari a far muovere il Cursor. Prima di terminare, anche questo AsyncTask comunica all'IntraProcessMessageHandler che il sistema ha a disposizione tutto il necessario per aggiornare il Cursor. L'Handler, a sua volta, innescherà la callback `redrawCursor` della ViewportActivity.

Diversamente da quanto accade per Gaze e Finger, la frequenza di aggiornamento del Cursor dipende dalla velocità della comunicazione tra sistema e gesture recognizer. Tanto più velocemente il sistema riceverà gli aggiornamenti della posizione del gesture recognizer, tanto più fluido risulterà essere l'aggiornamento del Finger e quindi lo spostamento del Cursor sulla Viewport. Nel

caso in cui questo fosse troppo veloce, sarebbe bene limitarlo facendo input conditioning lato client (quindi non a livello di framework).

8.4 Diagramma di attività

Per finire, in Figura, è illustrato un diagramma di attività che esemplifica il funzionamento della parte di sistema descritta nel corso della trattazione del Capitolo.

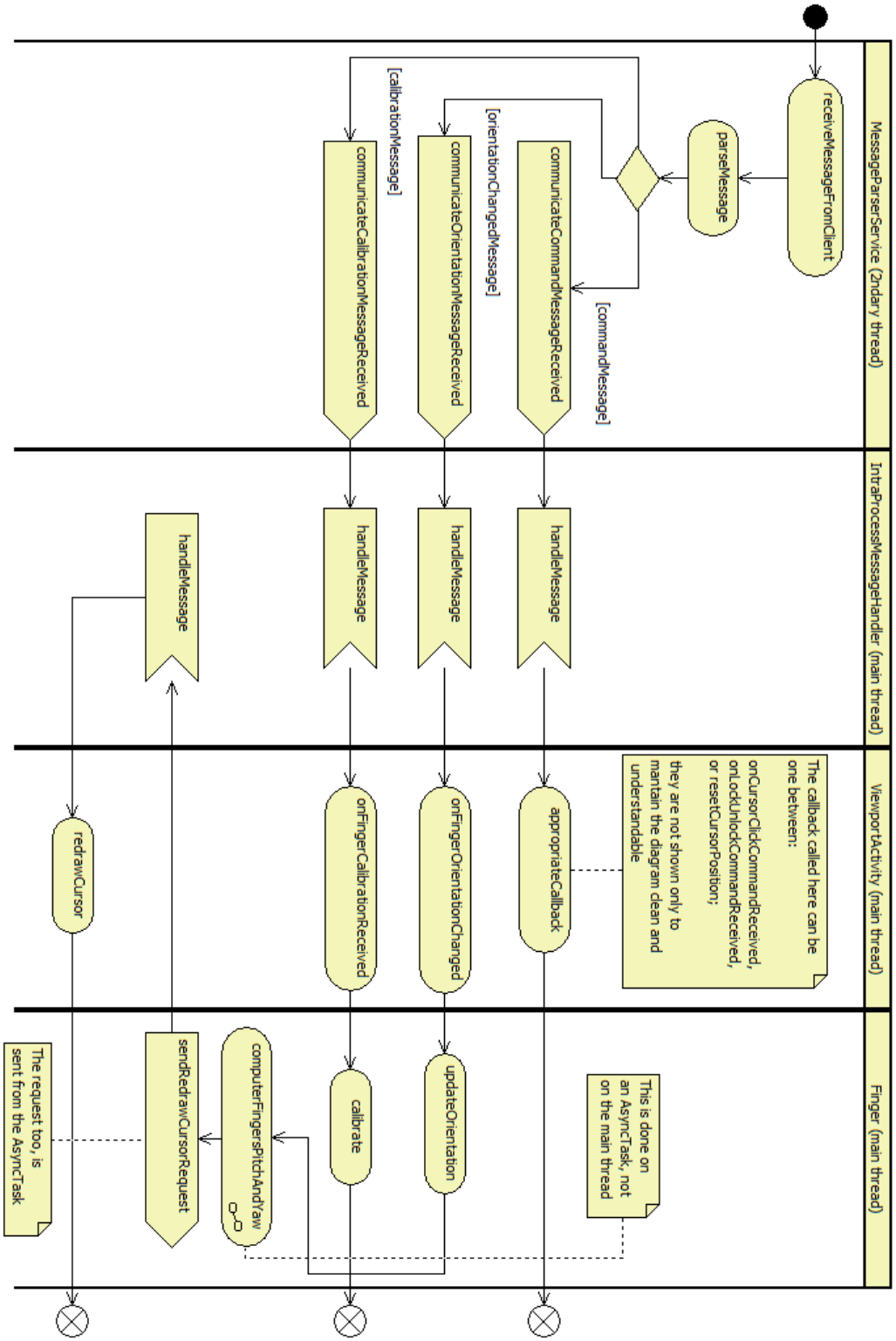


Figura 8.4: *Diagramma di attività che mostra come il Finger viene aggiornato e come vengono registrati i comandi ricevuti dal gesture recognizer.*

8.5 Recuperare ed utilizzare il Finger

Analogamente a ciò che è stato detto per il Gaze, anche il Finger può essere recuperato nello stesso modo per poter essere utilizzato da un'applicazione basata sul framework.

```
public void fingerUse(final ViewportActivity context) {
   // Recovering the instance
   final IFinger instance = context.getFinger();

   // This is how to recover the last recorded orientation of the
   // gesture recognizer. Such orientation is expressed according
   // to the coordinate system of the gesture recognizer
   final Quaternion fingerOrientation = instance.getOrientation();
   this.useFingerOrientation(fingerOrientation);

   // This is how to recover the starting orientation of the gesture
   // recognizer. This orientation represents the starting point of
   // the Finger and it's set when the system receives a Calibration
   // Message
   final Quaternion fingerCalibration =
       instance.getCurrentCalibration();
   this.useFingerCalibration(fingerCalibration);

   // This is how to recover the pitch and yaw angles of the Finger
   // relative to the starting position. These are also the angles
   // used to scroll the Cursor
   final float pitch = instance.getFingerPitch();
   final float yaw = instance.getFingerYaw();
   this.useFingerPitchAndYaw(pitch, yaw);

}
```

Listato 8.1: *Esempio di recupero ed utilizzo del Finger.*

Capitolo 9

Viewport

In questo Capitolo verrà descritta la parte rimanente del sistema, ovvero quella che comprende la Viewport ed i suoi contenuti. Verrà mostrato come questa viene fatta scorrere e come vengono modellate le componenti in essa rappresentate.

9.1 Proprietà della Viewport

La Viewport è una custom View, di dimensioni maggiori o uguali (ma tipicamente strettamente maggiori) dello schermo del dispositivo. Il background di tale View, è di colore nero, per poter ottenere un miglior effetto di trasparenza. Come sappiamo, la Viewport deve poter essere navigata tramite il movimento della testa. Lo scorrimento, viene ottenuto modificando i valori del `leftMargin` e del `topMargin` di questa componente, sulla base di un mappaggio che prende come input i valori degli angoli di pitch e yaw calcolati dal Gaze in `computeWearersPitchAndYaw` (la formula matematica utilizzata per il mappaggio verrà mostrata nella Sezione successiva).

Un'altra particolarità di questa componente, è data dal fatto che l'utente deve poter interagire con essa, come se il suo sistema di coordinate fosse analogo a quello di un piano cartesiano. Android tuttavia, utilizza per le View un sistema di coordinate in cui lo (0, 0) è situato nell'estremo in alto a sinistra. Benché da fuori l'utente debba poter interagire con la Viewport basandosi sul nuovo sistema di coordinate, al suo interno la Viewport deve però utilizzare quello di Android per poter disegnare i suoi elementi, in quanto anche le canvas assegnate alle View, sulle quali vengono disegnati i loro contenuti, utilizzano questo sistema di coordinate. La conversione da un sistema di coordinate all'altro però, dovrà essere resa invisibile da fuori, in modo che chi utilizza il framework non debba preoccuparsene. Il mappaggio per passare da sistema di

coordinate Android a sistema di coordinate cartesiano, è piuttosto semplice e si ottiene nel modo seguente:

$$vcsX = \frac{-viewportWidth}{2} + acsX$$
$$vcsY = \frac{viewportHeight}{2} - acsY$$

Dove vcs sta per Viewport Coordinate System e rappresenta quindi la coordinata, x o y, espressa nel sistema di coordinate cartesiano, mentre acs sta per Android Coordinate System e rappresenta quindi la coordinata, x o y, espressa nel sistema di coordinate Android.

Passiamo ora ad illustrare come la Viewport è stata modellata, mostrandone il diagramma delle classi.

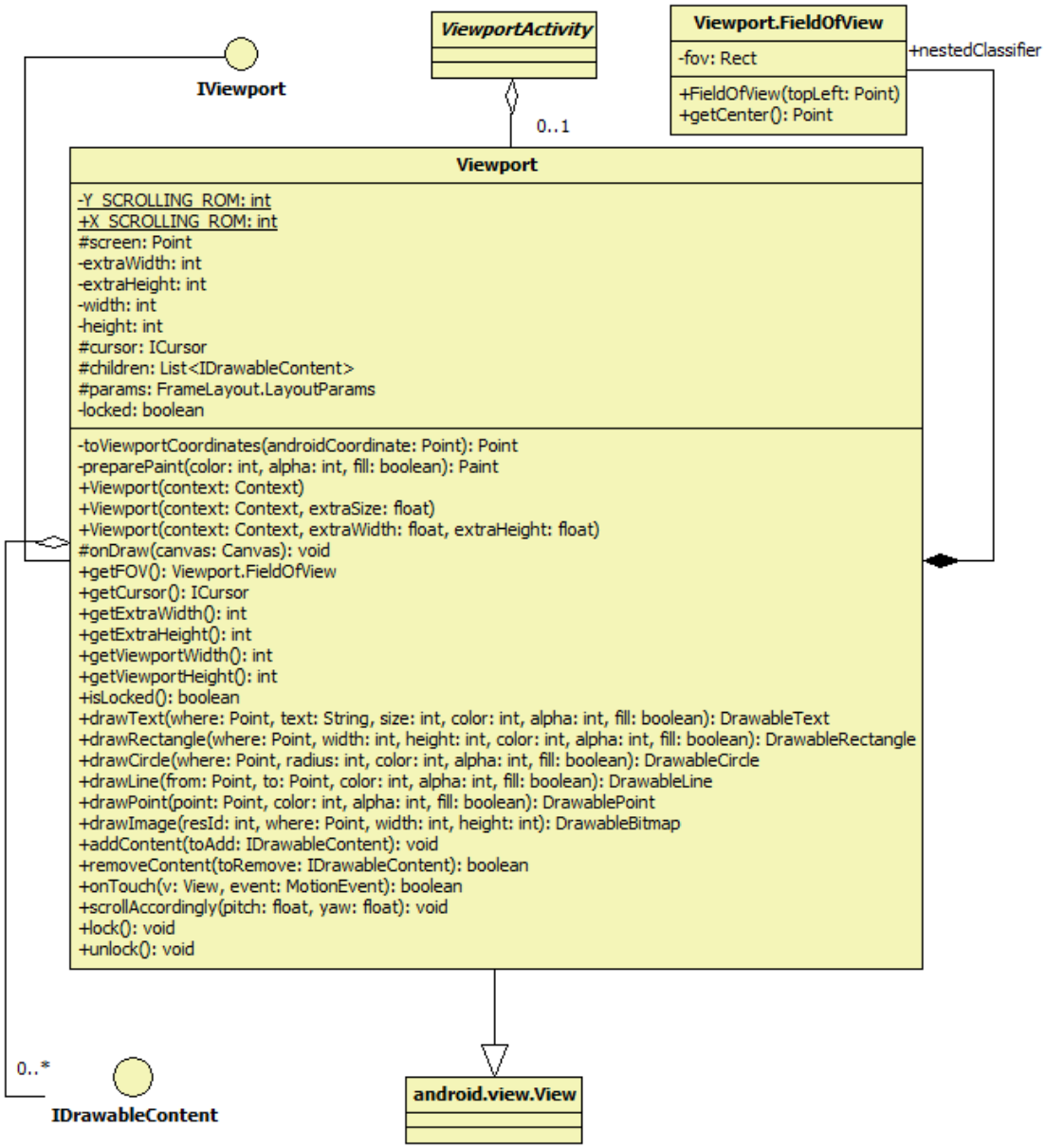


Figura 9.1: *Diagramma della classe Viewport.*

Notiamo la presenza di un'inner class chiamata FieldOfView. Tale classe modella la porzione di Viewport che è visibile, in un determinato momento, all'utente. Tale porzione è ottenibile a partire dal suo estremo in alto a sinistra, ricavabile basandosi sui valori dei `marginLeft` e `marginTop` della Viewport. Per ottenere poi l'estremo opposto (in basso a destra) è sufficiente sommare al

vertice in alto a sinistra la larghezza e l'altezza dello schermo del dispositivo utilizzato (calcolabile a run time). Il FieldOfView ci serve per mappare il movimento del Cursor, facendo in modo che esso non esca mai dal campo visivo dell'utente. Il come ciò viene fatto verrà spiegato verso la fine del Capitolo.

La Viewport, come possiamo notare, memorizza l'insieme dei suoi contenuti (`children`) in una lista. Il come questi figli vengano disegnati al suo interno, verrà spiegato nell'ultima Sezione del Capitolo.

Prima di passare alla descrizione del mappaggio impiegato per far scorrere la Viewport, vale la pena menzionare l'utilizzo, anche qui, di un AsyncTask. In questo caso, l'AsyncTask viene impiegato per individuare se un determinato TouchEvent coinvolge o meno uno dei figli della Viewport (ovvero viene utilizzato per rilevare l'interazione con i vari elementi contenuti nella Viewport). Ogni qualvolta si verifica un TouchEvent, nella callback `onTouchEvent`, viene ciclata, su tale AsyncTask, a partire dal fondo, la lista (o meglio una copia, in modo da evitare concurrent modification exceptions) dei figli della Viewport, per verificare se tale evento è accaduto all'interno dei confini di uno di questi. Non appena viene individuato un elemento coinvolto nel TouchEvent, se questo ha registrato un EventListener, tale EventListener "consuma" il TouchEvent, che non viene propagato oltre nella lista di figli. Nel caso in cui, invece, l'elemento non abbia registrato un EventListener, vengono controllati gli altri figli della Viewport. Ciò sta a significare, che se un TouchEvent si verifica all'interno dei confini di due figli della Viewport, solo l'EventListener associato a quello che è stato aggiunto più tardi dei due (o, se vogliamo, quello con la coordinata z più grande) registrerà l'evento.

9.2 Scorrimento della Viewport

Passiamo ora al descrivere come gli angoli di pitch e yaw vengano utilizzati per fare scorrere la Viewport concordemente al movimento della testa. Come abbiamo detto, il pitch è l'angolo che rappresenta lo spostamento verticale del Gaze, mentre lo yaw rappresenta quello orizzontale. Lo scorrimento, viene ottenuto, come già detto, manipolando i margini della Viewport.

Sappiamo, innanzitutto, che quando ci si trova nella posizione iniziale, ovvero quando `wearersPitch` e `wearersYaw` valgono 0, la parte di Viewport che viene visualizzata è quella centrale. Per poter visualizzare il centro della Viewport, i margini devono avere, rispettivamente, i valori:

$$\texttt{marginLeft} = \frac{-extraWidth}{2}$$

$$\texttt{marginTop} = \frac{-extraHeight}{2}$$

Dove extraWidth rappresenta di quanto la Viewport è più lunga, in pixel, rispetto allo schermo del dispositivo ed extraHeight rappresenta di quanto la Viewport è più alta, in pixel, rispetto allo schermo del dispositivo.

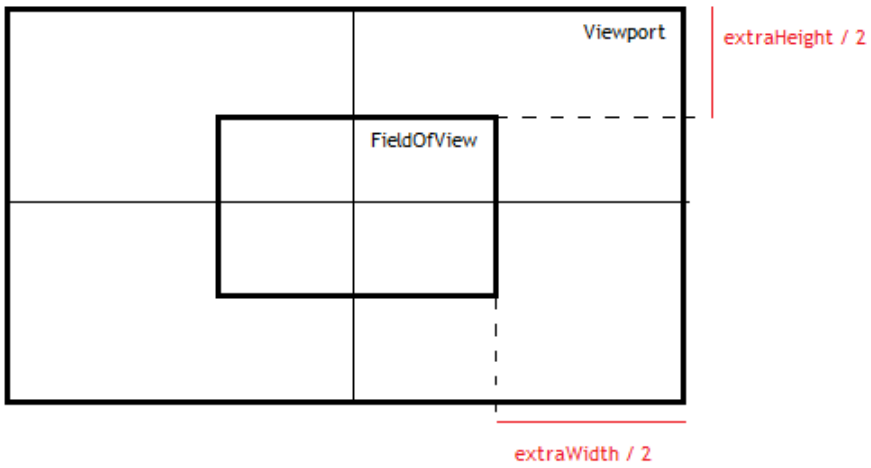


Figura 9.2: *Rappresentazione grafica di Viewport e FieldOfView.*

Quello che ci serve fare, ora, è definire una costante di *gain*. Con gain si intende di quanti pixel far scorrere un determinato margine in relazione all'ampiezza dell'angolo di pitch o yaw, che esprime di quanto il Gaze si discosta dalla sua posizione iniziale.

Non volendo utilizzare una costante assoluta, la quale potrebbe portare a problemi quali rendere impossibile l'intera navigazione della Viewport nel caso in cui questa fosse troppo grande, è stato scelto di impiegarne una relativa.

La magnitudine dello scorrimento viene calcolata sulla base delle dimensioni della Viewport. Per calcolare tale valore, ci si è basati sul seguente ragionamento. Supponiamo di voler rendere l'intera Viewport navigabile in orizzontale con una rotazione della testa di 120 gradi e in verticale con una rotazione della testa di 90 gradi. Queste costanti definiscono il *range of motion* (rom) necessario per navigare l'intera Viewport. Chiamiamo tali costanti $X\text{-}ROM = 120$ (range of motion per scorrere orizzontalmente l'intera Viewport) e $Y\text{-}ROM = 90$ (range of motion per scorrere verticalmente l'intera Viewport). La scelta dei valori è stata data a seguito di alcuni test, per vedere di quanti gradi fosse effettivamente possibile ruotare la testa prima che tale movimento diventasse scomodo o innaturale.

Avendo definito gli estremi per ciò che riguarda gli angoli, definiamo ora gli estremi per ciò che riguarda i margini. Rifacendoci sempre alla Figura

9.2, è possibile notare come, per quanto riguarda lo scorrimento orizzontale, il FieldOfView si trovi nell'estrema sinistra quando il valore di `marginLeft` è uguale a 0, mentre si trovi nell'estrema destra quando tale valore è uguale a $-extraWidth$. Questo significa che, per quanto riguarda lo scorrimento orizzontale, per poter inquadrare l'estrema sinistra della Viewport (`marginLeft` = 0), sarà necessario ruotare la testa di *X-ROM / 2*, mentre per poter inquadrare l'estrema destra (`marginLeft` = $-extraWidth$) sarà necessario ruotare la testa di *-X-ROM / 2* (ricordiamo che l'angolo di yaw è positivo, nel nostro caso, per le rotazioni verso sinistra).

Il range per ciò che riguarda l'angolo di yaw, è quindi pari a *X-ROM*, mentre il range per ciò che riguarda il `marginLeft` è pari a $-extraWidth$. Basandoci su questi dati, la proporzione per calcolare lo spostamento orizzontale rispetto al centro (ovvero rispetto alla posizione iniziale) è la seguente:

$$wearersYaw : X - ROM = marginLeft : extraWidth$$

dalla quale è possibile calcolare il valore del `marginLeft`:

$$marginLeft = \frac{wearersYaw * extraWidth}{X - ROM}$$

visto che l'angolo `wearersYaw` rappresenta l'angolo di yaw in relazione alla posizione iniziale, al margine ottenuto andrà sommato il margine relativo a tale posizione, ovvero a quando il FieldOfView è situato al centro della Viewport e quindi:

$$marginLeft = \frac{-extraWidth}{2} + \frac{wearersYaw * extraWidth}{X - ROM}$$

In maniera del tutto analoga, è possibile calcolare il valore del `marginTop`:

$$wearersPitch : Y - ROM = marginTop : extraHeight$$

$$marginTop = \frac{wearersPitch * extraHeight}{Y - ROM}$$

$$marginTop = \frac{-extraHeight}{2} + \frac{wearersPitch * extraHeight}{Y - ROM}$$

Per fare scorrere la Viewport quindi, la ViewportActivity chiama, nella callback `redrawViewport`, il metodo `scrollAccordingly`, al quale passa i valori di `wearersPitch` e `wearersYaw` del Gaze.
A questo punto, la Viewport, dopo aver effettuato i mappaggi sopra descritti (a meno che non sia bloccata), viene invalidata e quindi ridisegnata.
Lo scorrimento della Viewport, viene applicato anche al Cursor, per fare in modo che esso rimanga all'interno del FieldOfView.

9.3 DrawableContent

Passiamo ora ad illustrare come vengano modellati i contenuti della Viewport. Questi prendono il nome di DrawableContent. I DrawableContent sono inquadrabili come degli oggetti che conoscono la loro posizione all'interno della Viewport e sanno come disegnarsi all'interno di essa.

Il ridisegnamento dei figli della Viewport, viene innescato nella callback `onDraw` della Viewport stessa, richiamata ogni qualvolta questa viene invalidata. A questa callback viene passata la Canvas sulla quale essa deve ridisegnare sé e tutte le sue componenti. La Viewport a questo punto, passa la Canvas a tutti i suoi figli tramite il metodo `drawOnCanvas`. I figli (DrawableContent), data tale Canvas e conoscendo le loro coordinate all'interno della Viewport, sono in grado di disegnarsi su di essa, richiamando il metodo astratto `draw`. Per potersi disegnare, essi devono prima convertire le loro coordinate, espresse nel sistema di coordinate della Viewport ((0,0) al centro), nel sistema di coordinate Android, utilizzato dalle Canvas ((0,0) in alto a sinistra).

Questi oggetti sono anche in grado di fornire le coordinate della loro *hitbox*, ovvero dei loro confini. Anche le coordinate delle hitbox però, sono espresse secondo il sistema di coordinate della Viewport, la quale dovrà quindi convertirle in quello Android per rilevare se un determinato TouchEvent (che utilizza appunto il sistema di coordinate Android) si è verificato o meno all'interno di queste. Anche per il calcolo delle hitbox ci si basa sull'utilizzo del template method. Il punto in basso a sinistra della hitbox infatti, è definito dalla coordinata di posizionamento del DrawableContent all'interno della Viewport. Quello in alto a destra viene calcolato a seconda dell'elemento rappresentato dal DrawableContent. Ciò viene fatto nel metodo astratto `computeUpperBound`.

Qui sotto, in Figura 9.3, viene mostrato il diagramma delle classi dei vari DrawableContent.

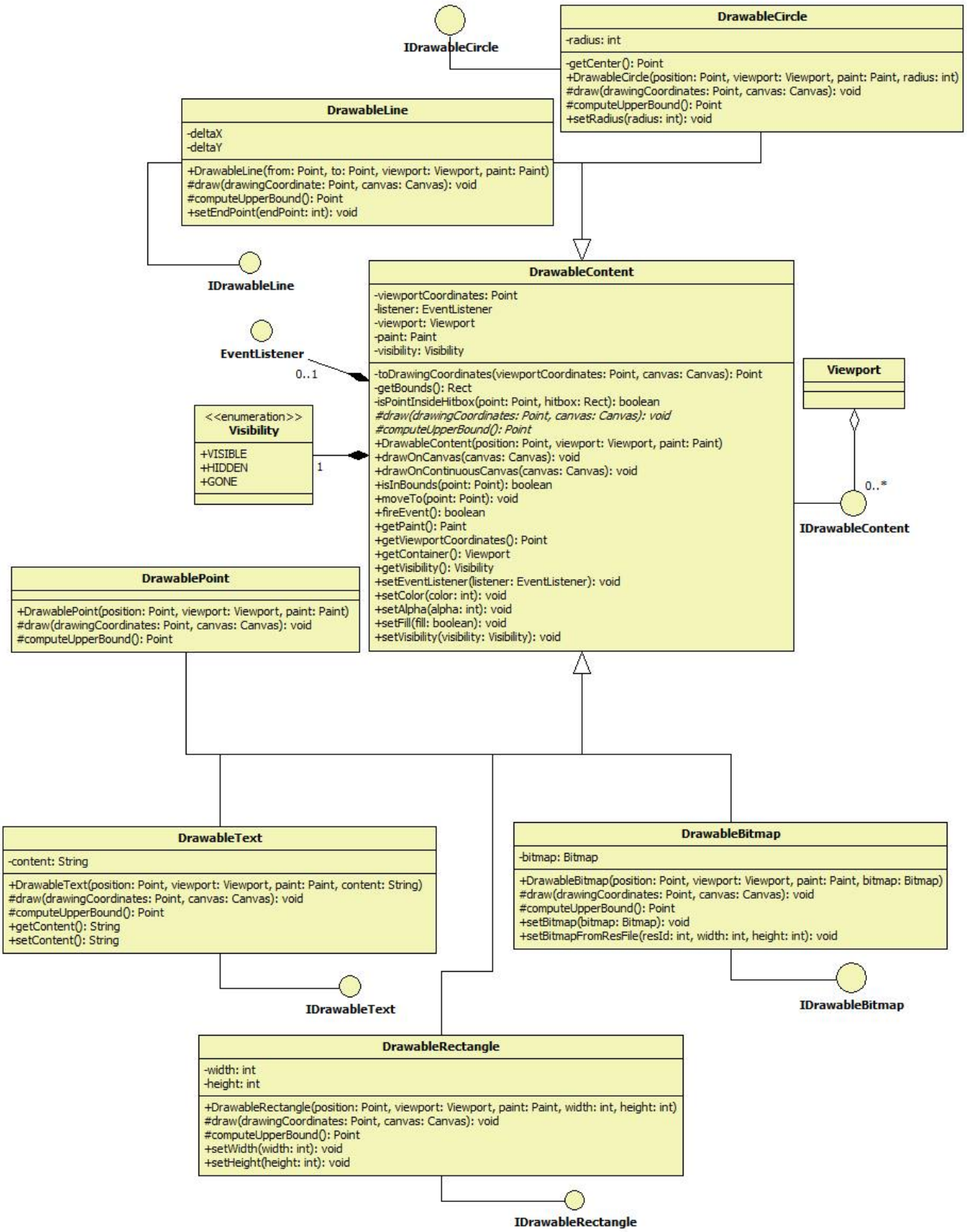


Figura 9.3: *Diagramma delle classi che illustra come i DrawableContent sono stati modellati.*

E' importante notare che, dal momento che questi oggetti non rappresentano delle vere e proprie View, è possibile intervenire su di essi (modificandone contenuti, colore, dimensioni o posizione), anche al di fuori del Main Thread. Visto che la Viewport viene invalidata 30 volte al secondo circa e quando ciò accade anche tutti i DrawableContent vengono ridisegnati, qualsiasi modifica venga apportata a questi contenuti verrà riportata visivamente in maniera quasi immediata.

9.3.1 Cursor

Anche il Cursor viene modellato come un DrawableContent, ma non come un figlio della Viewport. La Viewport infatti, mantiene il riferimento al suo Cursor al di fuori della lista dei suoi figli.
Nel diagramma delle classi illustrato qui sotto, in Figura 9.4, possiamo notare il diagramma delle classi che rappresenta il Cursor.

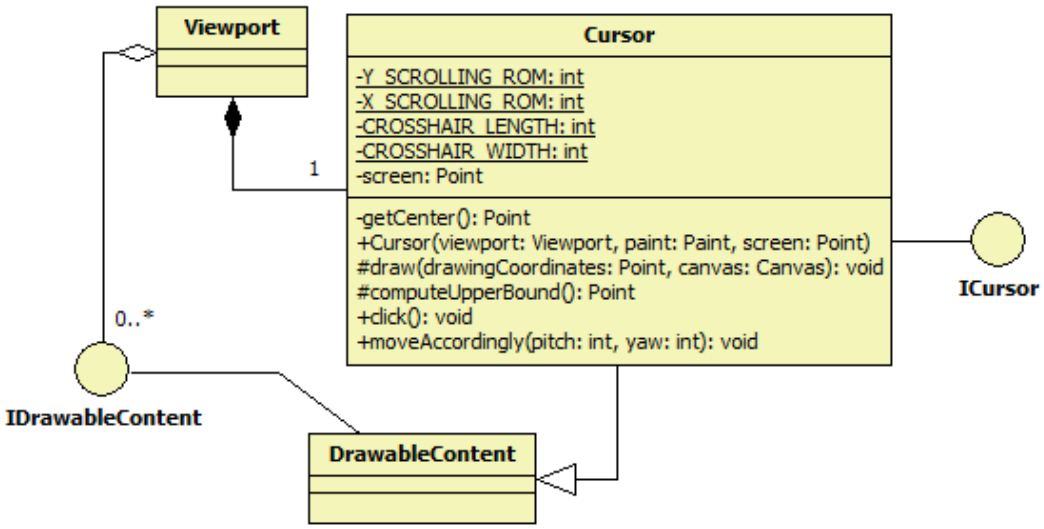


Figura 9.4: *Diagramma della classe Cursor.*

Notiamo come anche il Cursor disponga di un metodo `moveAccordingly`, simile a quello della Viewport, per farlo muovere. Anche a tale metodo, chiamato nella callback `redrawCursor` della ViewportActivity, prende come input gli angoli `fingersPitch` e `fingersYaw` del Finger e li utilizza come base per far muovere il Cursor.

La tecnica di mappaggio utilizzata, è del tutto analoga a quella già esposta per la Viewport. L'unica differenza è data dal fatto che, per evitare che il

Cursor possa essere spostato al di fuori del FieldOfView, se gli angoli di pitch e yaw registrati vanno al di sopra di una determinata soglia (che porterebbe a muovere il Cursor al di fuori della parte di Viewport visibile), questo non viene fatto muovere.

9.4 Recuperare ed utilizzare la Viewport

La Viewport è recuperabile nello stesso modo in cui sono recuperabili anche Finger e Gaze, ovvero tramite la ViewportActivity. Per poter recuperare la Viewport associata ad una ViewportActivity tuttavia, questa deve prima essere stata settata come sua content View. Una volta recuperata l'istanza della Viewport associata all'Activity, è possibile disegnare o cancellare elementi su di essa tramite i metodi che mette a disposizione.

```
public void viewportUse(final ViewportActivity context) {
   // Recovering the Viewport
   final Viewport viewport = context.getViewport();

   // The user can now draw on the Viewport by using its
   // methods
   final DrawableText test = viewport.drawText(new Point(0, 0),
       "TEST", 15, Color.WHITE, 120, true);

   // Or erase what was drawn on it
   viewport.removeContent(test);

}
```

Listato 9.1: *Esempio di recupero ed utilizzo della Viewport.*

Capitolo 10

Estensioni apportate al framework

In questo Capitolo verranno mostrate le estensioni apportate al framework, che non facevano originariamente parte delle specifiche richieste.

10.1 ContinuousViewport

E' stata aggiunta al framework, una Viewport che avvolge completamente l'utente ed offre l'astrazione di apparire come continua. L'illusione di continuità, è stata ottenuta impiegando una Viewport che sia in realtà, tre volte più grande di quanto richiesto. E' possibile inquadrare tale Viewport, come tre Viewport normali disposte una a fianco all'altra. Solo su quella centrale però, è possibile disporre contenuti e solo questa verrà navigata.

Per quanto riguarda gli elementi (DrawableContent) in essa rappresentabili, essi sono gli stessi utilizzati da una Viewport normale, tuttavia il loro comportamento è stato leggermente alterato per quanto riguarda il ridisegnamento e il calcolo del loro bordi. Se un DrawableContent si trova infatti su una ContinuousViewport, esso viene disegnato tre volte, una volta in ogni Viewport, sulla stessa coordinata e i suoi confini (la sua hitbox) vengono calcolati diversamente, in determinati casi.
Le immagini mostrate qui sotto, offrono una rappresentazione grafica più chiara e comprensibile.

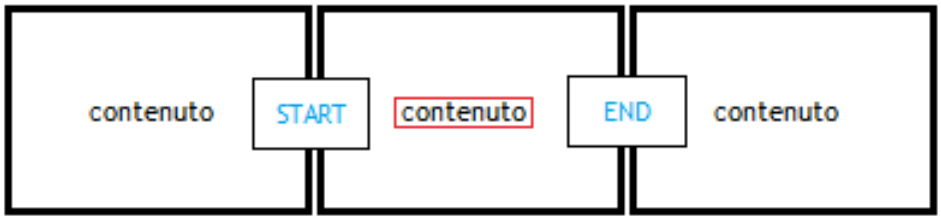


Figura 10.1: *L'immagine mostra come un contenuto venga disegnato tre volte all'interno della ContinuousViewport, rappresentata qui come tre Viewport disposte una a fianco all'altra. Il rettangolo rosso attorno al contenuto centrale, ci mostra l'hitbox di tale contenuto. I rettangoli start ed end, sono invece le posizioni, rispettivamente, di estrema sinistra ed estrema destra del FieldOfView, all'interno di una ContinuousViewport.*

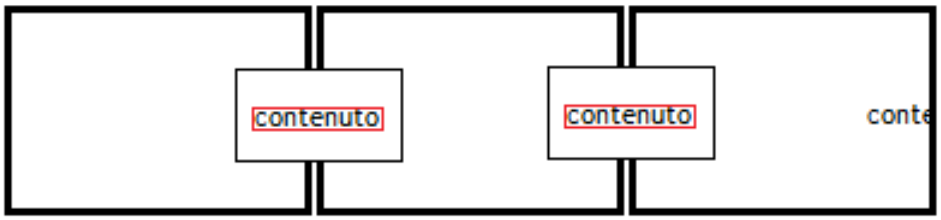


Figura 10.2: *In questa immagine, viene fatto vedere invece come la hitbox di un contenuto venga calcolata diversamente, se esso si trova in una situazione simile a quella qui illustrata. In questo modo, gli eventi che interessano tale elemento, saranno sempre intercettabili.*

Qua sotto invece, in Figura 10.3, è illustrato il diagramma delle classi. Come si può notare, la ContinuousViewport è per gran parte identica ad una Viewport normale.

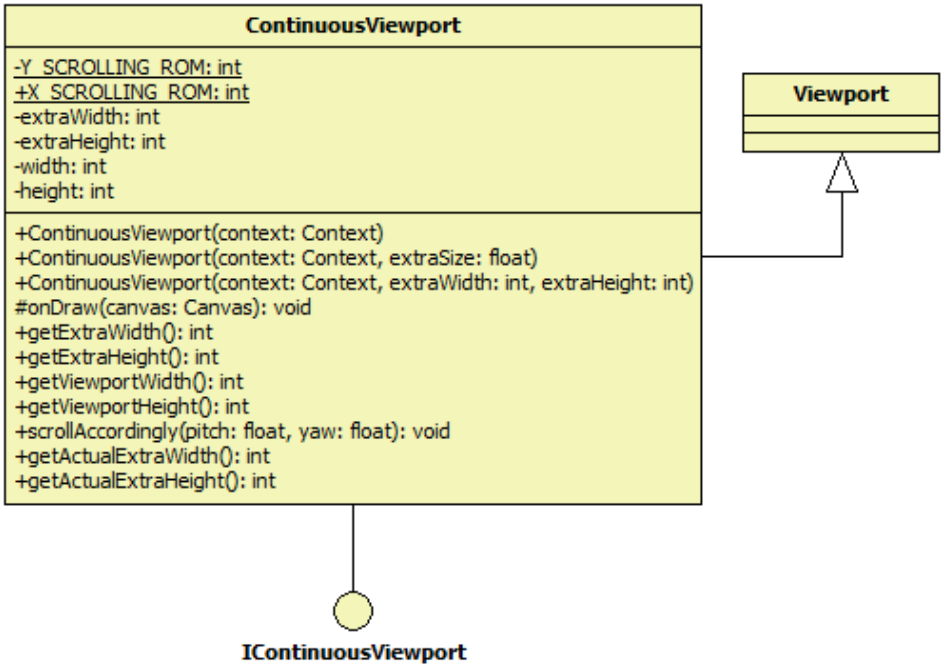


Figura 10.3: *Diagramma della classe ContinuousViewport.*

La distinzione principale sta nel come essa viene navigata e nel come ridisegna le sue componenti. Innanzitutto, per quanto riguarda il ridisegnamento di un DrawableContent, esso avviene solo se si trova all'interno della Viewport centrale, per evitare errori di rendering che farebbero comparire nella Viewport elementi che invece non ne fanno parte.

Per quanto riguarda la navigazione, la logica seguita è sempre la stessa, l'unica differenza è il fatto che cambiano gli estremi per ciò che riguarda lo scorrimento orizzontale (quello verticale è del tutto analogo ad una Viewport normale). Innanzitutto, cambia il nostro range of motion. X-ROM sarà, in questo caso, pari a 360, per dare l'illusione che tale Viewport avvolga l'utente. Per quanto riguarda il `marginLeft` inoltre, notiamo, facendo riferimento ai disegni di prima, che questo ha valore:

$$\frac{-extraWidth}{2} - width$$

quando il FieldOfView è al centro della Viewport. Con width si intende la lunghezza di una sola delle tre Viewport, la vera lunghezza dell'intera ContinuousViewport è pari a 3 * width, o più semplicemente, actualWidth.

Similmente, anche extraWidth rappresenta di quanto una sola delle tre Viewport è più larga rispetto allo schermo del dispositivo. La vera extraWidth della ContinuousViewport è invece pari a 2 + 3 * extraWidth, o più semplicemente, actualExtraWidth. L'intero range di scorrimento inoltre, è aumentato e non è più pari a $-extraWidth$, bensì a $-width$. Sulla base di questo, possiamo ricostruire la nuova formula per il calcolo del `marginLeft`, in modo del tutto analogo a quanto già fatto per una comune Viewport:

$$marginLeft = \frac{-extraWidth}{2} - width + \frac{wearersYaw * width}{X - ROM}$$

10.2 BridgeConnectionService

E' stato inserito a livello di framework, anche un Service astratto che offre metodi per collegarsi ed inviare messaggi, dallo smart phone connesso al gesture recognizer, al sistema in esecuzione sugli smart glass. Grazie a questo Service, viene semplificato ancora di più il lavoro dell'utente che voglia utilizzare il framework, il quale non dovrà occuparsi del setup della connessione Bluetooth e del corretto utilizzo del protocollo applicativo. Ciò che egli dovrà fare invece, consiste semplicemente nell'estendere questo servizio, chiamato BridgeConnectionService e chiamarne i metodi appropriati per comunicare la posizione del gesture recognizer e i comandi, codificati da gesture, al sistema. Della connessione tra smart phone e gesture recognizer tuttavia, dovrà occuparsene l'utente.
Qui sotto, viene presentato il diagramma delle classi di tale Service:

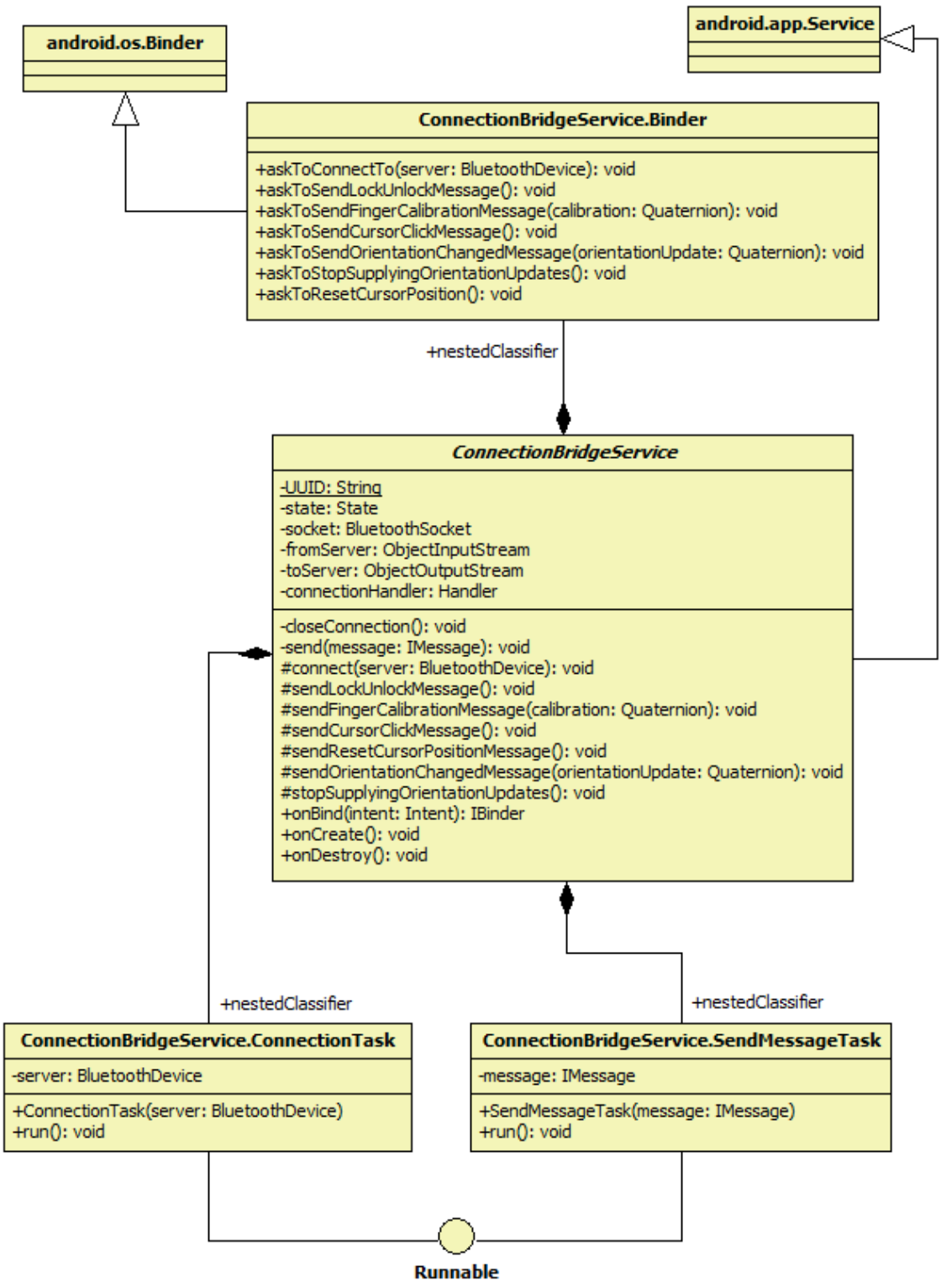


Figura 10.4: *Diagramma della classe ConnectionBridgeService.*

Un'altra particolarità di questo Service, è data dal fatto che esso può fornire, in un determinato momento, o aggiornamenti della posizione, o comandi (quali click, ecc...). Quando il Service è nel suo stato *SUPPLYING ORIENTATION UPDATES*, tutti i tentativi di invio di comandi vengono filtrati. Analogamente, quando esso si trova nello stato *SUPPLYING COMMANDS*, tutti i tentativi di invio di aggiornamento della posizione verranno ignorati.

Il passaggio dallo stato *SUPPLYING COMMANDS* allo stato *SUPPLYING ORIENTATION UPDATES* avviene a seguito dell'invio di un comando di calibrazione (che è necessario per poter fornire aggiornamenti della posizione). Il passaggio inverso, è effettuato chiamando il metodo `stopSupplyingOrientationUpdates`, che può essere eventualmente associato ad una determinata gesture.

La scelta di distinguere queste due fasi, è legata al fatto che spesso, quando si muove il braccio, può capitare che il gesture recognizer individui delle gesture che in realtà non sono state effettuate, provocando l'invio involontario di comandi ad esse associati.

Capitolo 11

Testing ed esempio di utilizzo

11.1 Dispositivi utilizzati e risultati ottenuti

Benché il framework debba essere utilizzabile con qualunque accoppiata smart glass - gesture recognizer, gli unici dispositivi a disposizione nella fase di sviluppo e di testing sono stati gli Epson Moverio BT 200 per quanto riguarda gli smart glass e il Myo per quanto riguarda il gesture recognizer.

Per quello che riguarda il rendering dei contenuti, i risultati ottenuti sono, tutto sommato, soddisfacenti, anche se il sistema comincia a faticare nel momento in cui vengono disposti, all'interno della Viewport, un numero molto elevato di contenuti. Questo è dovuto al fatto che ad ogni aggiornamento, il Main Thread è costretto a ciclare tutta la lista di figli della Viewport e comunicare a ciascuno di essi di ridisegnarsi. Se tale attività impegna troppo tempo, questo porta ad un progressivo decadimento della reattività del sistema, in quanto i messaggi che vengono ricevuti dall'IntraProcessMessageHandler cominciano ad accumularsi, non potendo essere gestiti immediatamente, dal momento che il Main Thread è occupato nel ciclare i contenuti della Viewport.

Questo porta anche ad un ulteriore considerazione, sempre legata al Main Thread. Visto il carico che esso deve già sopportare, sarebbe meglio che le applicazioni basate su tale framework, utilizzino il meno possibile tale thread. Ciò non dovrebbe essere un problema visto che, come abbiamo già illustrato, i DrawableContent sono modificabili anche da altri thread, non essendo propriamente delle View.

Sono stati effettuati anche testing con animazioni. Il framework ha risposto, anche qui, in maniera soddisfacente, a patto che tali animazioni non vengano delegate al main thread.

Per quanto riguarda il refresh rate della Viewport, il sistema è riuscito a resistere anche ad una frequenza che si aggira attorno ai 50 fps. Le condizioni di testing però, prevedevano una Viewport con pochi elementi. Proprio per que-

sto, è stato scelto di limitare tale rateo a 30 fps, ossia il minimo indispensabile, in modo che la Viewport possa venire riempita il più possibile.

11.2 Considerazioni sul filtro impiegato

Benché il filtro complementare utilizzato abbia portato dei risultati che rispettano le specifiche, questo presenta comunque dei difetti. Primo fra tutti, la quantità di tempo richiesta dalla sua fase di calibrazione, che al momento ammonta a ben 15 secondi e nel fatto che tale fase, debba essere ripetuta ad ogni panic reset (evento completamente sporadico e quindi non prevedibile).
Una volta trascorsa tale fase tuttavia, l'algoritmo offre dei risultati perlopiù precisi. Interferenze dovute al suono, o ad altre vibrazioni, vengono filtrate quasi completamente, anche se può capitare che la Viewport venga fatta scorrere sulla base di queste interferenze, che vengono erroneamente considerate dei movimenti. Questi spostamenti non voluti però, sono molto leggeri e non compromettono il corretto funzionamento del sistema, inoltre, nel corso del tempo, tendono a bilanciarsi a vicenda.

In ogni caso, come già descritto nel Capitolo 7, introdurre un nuovo algoritmo per il tracking del Gaze all'interno del framework non risulterebbe essere complicato.

11.3 Problemi nella comunicazione Bluetooth

Per quanto riguarda la comunicazione via Bluetooth, per motivi legati all'hardware, non è stato possibile inviare più di cinque aggiornamenti al secondo della posizione del gesture recognizer.
Questo significa che la frequenza di aggiornamento del Finger e, di conseguenza, il movimento del Cursor, sono risultati essere veramente poco fluidi. Tutto ciò, è però dovuto esclusivamente alla versione datata di Bluetooth impiegata dai Moverio BT200. Nel caso si voglia mettere in esecuzione il framework su tali smart glass, è bene fare input conditioning in modo da limitare la frequenza di invio degli orientamenti a tale dispositivo, altrimenti si finirebbe con il saturare i buffer di invio e ricezione. Se la velocità di invio supera quella di ricezione infatti, ciò porta ad una decadenza della reattività del sistema, che si troverà a processare orientamenti del dispositivo relativi ad istanti di tempo già passati.

Ci si aspetta tuttavia, che con l'introduzione di nuovi dispositivi dotati di versioni più recenti di Bluetooth, questo problema scomparirà da solo.

11.4 Esempio di utilizzo del framework

Terminiamo la trattazione del capitolo e della tesi, mostrando un rapido esempio di utilizzo del framework. Innanzitutto, è necessario istanziare il controller principale, ovvero la ViewportActivity. Dal momento che essa è astratta però, l'unico modo che si ha per farlo, è creare una nuova classe che la estenda.

```
package com.example.federico.demo;

import com.example.federico.wearableui.controller.ViewportActivity;

// Create a new ViewportActivity that will serve as the Main
// Activity of the application
public class DemoActivity extends ViewportActivity {

   @Override
   onCreate(final Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
   }

}
```

Listato 11.1: *Estensione di ViewportActivity.*

Fatto ciò, sappiamo che tale Activity accetta come content View solo una Viewport, quindi, inizializziamo una nuova Viewport. I parametri passati (`extraWidth`, `extraHeight`) al costruttore della Viewport, corrispondono alle dimensioni extra espresse in percentuale (i.e. specificare `extraWidth` = 0.5, significa richiedere l'istanziamento di una Viewport con una larghezza del 50% superiore a quella del display del dispositivo utilizzato). E' necessario sbloccare e rendere la Viewport navigabile tramite il metodo `unlock`, visto che alla creazione essa risulta essere bloccata.

A questo punto, è necessario richiamare, preferibilmente nell'`onCreate`, il metodo `openBluetoothConnection` della ViewportActivity, per configurare il dispositivo sul quale l'applicazione eseguirà come Bluetooth server e rendere possibile il collegamento con il gesture recognizer. Questo step non è necessario, nel caso in cui l'interfaccia voglia essere utilizzata solo come mezzo per illustrare le varie informazioni e non abbia quindi bisogno dell'impiego del Cursor.

```
package com.example.federico.demo;

import com.example.federico.wearableui.controller.ViewportActivity;
```

```
public class DemoActivity extends ViewportActivity {

   @Override
   onCreate(final Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      // Create a new Viewport and set it as the content View
      // of the Activity
      final Viewport contentView = new ContinuousViewport(this, 10,
          10);
      this.setContentView(contentView);

      // Unlock the Viewport, since by default it's locked and,
      // therefore, unable to be scrolled
      contentView.unlock();

      // Make it possible for the gesture recognizer to connect
      // in order to provide commands such as clicks and lock\unlock
      // requests and Finger orientation updates to the system
      this.openBluetoothConnection();
   }

}
```

Listato 11.2: *Inizializzazione di una ContinuousViewport del 1000% più larga e più alta dello schermo del dispositivo e setup per la connessione Bluetooth.*

Una volta settata la Viewport come content View, è possibile disegnare su essa tramite gli opportuni metodi di *draw* messi a disposizione da essa. Ricordiamo che la coordinata specificata come posizione, è la coordinata dell'estremo in basso a sinistra di ciò che si vuole disegnare.

```
package com.example.federico.demo;

import com.example.federico.wearableui.controller.ViewportActivity;

public class DemoActivity extends ViewportActivity {

   @Override
   onCreate(final Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      /* ... */
```

```
        // Draw a "WELCOME" String in position (0, 0) with dimension
        // 25, color white, 120 alpha value and fill style
        final DrawableText welcomeMessage = contentView.drawText(new
            Point(0, 0), "WELCOME", 25, Color.WHITE, 120, true);
        // Draw a circle in position (300, 400) with a radius of 100
        // pixel, color yellow, 255 alpha value and fill style
        final DrawableCircle sun = contentView.drawCircle(new
            Point(300, 400), 100, Color.YELLOW, 255, true);
    }

}
```

Listato 11.3: *Esempio di disegno di una stringa di testo e di un cerchio.*

Per finire, qui sotto viene mostrato un esempio di modifica delle proprietà degli elementi disegnati nello snippet precedente, effettuato da un opportuno EventListener ad essi associato. Ricordiamo che, dal momento che i DrawableContent non estendono View, possono essere modificati da qualunque thread.

```
package com.example.federico.demo;

import com.example.federico.wearableui.controller.ViewportActivity;

public class DemoActivity extends ViewportActivity {

    @Override
    onCreate(final Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        /* ... */

        // Setting an EventListener for the welcomeMessage content
        // that will change its content from "WELCOME" to "GO AWAY"
        // and its color from white to red when a TouchEvent will
        // be registered inside the bounds of the element
        welcomeMessage.setEventListener(new EventListener() {

            @Override
            public void onEventFired(final DrawableContent content) {
                content.setContent("GO AWAY");
                content.setColor(Color.RED);
            }
```

```
      });

      // Same thing for the sun content, but its color will be
      // changed to blue instead
      sun.setEventListener(new EventListener() {

         @Override
         public void onEventFired(final DrawableContent content) {
            content.setColor(Color.BLUE);
         }

      });
   }

}
```

Listato 11.4: *Esempio di come associare ad un DrawableContent un EventListener che ne modifichi il contenuto ad ogni "click".*

E' possibile, all'interno della nostra Activity, effettuare l'override delle callback ereditate da IPMHCallbackInterface, nel caso si vogliano aggiungere ad esse delle funzionalità extra.
Affinché il sistema funzioni correttamente però, è necessario che la super venga sempre invocata.

Per quanto riguarda invece il lato smartphone, sul quale implementare la parte che permette la connessione tra gesture recognizer e smart glass, è possibile utilizzare, come già detto, le funzionalità messe a disposizione dal BridgeConnectionService. Qui sotto, è offerto uno snippet esemplificativo, che mostra come utilizzare le funzionalità di tale Service (o meglio, di uno che lo estende, visto che BridgeConnectionService è astratto) per utilizzare un Myo come input source.

```
// AbstractDeviceListener belongs to the Myo SDK and it's
// used to register the events captured by the armband
class MyoEventListener extends AbstractDeviceListener {

    private static final int ORIENTATION_PROVISION_RATE_MS = 200;

    private final IBinder mBinder;
    private long timestampLastSent;
    private Quaternion lastOrientationProvided;

    public MyoEventListener(final BridgeConnectionService.Binder
        binder) {
```

```
        this.mBinder = binder;
        this.timestampLastSent = -1;
        this.lastOrientationProvided = null;
    }

    @Override
    public void onPose(final Myo myo, final long timestamp, final
        Pose pose) {
        super.onPose(myo, timestamp, pose);
        if (this.mBinder == null || this.lastOrientationProvided ==
            null) {
            return;
        }

        Log.i("MyoListener", "Pose registered: " + pose.name());

      // Sending the appropriate command codified by the
      // gesture by calling the corresponding method of
      // the binder of the ConnectionService. For examp-
      // le, a WAVE_OUT gesture is codified as a click
      // command, therefore each time such gesture is
      // registered, the program will ask the Service
      // to send the click command, through the binder
        switch (pose){
            case WAVE_IN:
                this.mBinder.askToStopSupplyingOrientationUpdates();
                break;
            case WAVE_OUT:
                this.mBinder.askToSendCursorClickMessage();
                break;
            case DOUBLE_TAP:
                this.mBinder.askToSendLockUnlockMessage();
                break;
            case FIST:
                this.mBinder.askToSendFingerCalibrationMessage(this.
            lastOrientationProvided);
                break;
            case FINGERS_SPREAD:
                this.mBinder.askToResetCursorPosition();
                break;
            default:
                break;
        }
    }
```

```
    @Override
    public void onOrientationData(final Myo myo, final long
        timestamp, final com.thalmic.myo.Quaternion rotation) {
      // Check if the time elapsed since the last orientation sent
      // is greater than the specified interval. This is done to
      // limit orientation updates to 5 updates per second, in case
      // the smart glass can't handle a high transfer rate
      if (this.mBinder == null || timestamp - this.timestampLastSent
          < ORIENTATION_PROVISION_RATE_MS) {
         return;
      }

        final Quaternion q = new Quaternion();
       // Creating a Quaternion object used by the framework
       // from the Quaternion object used by the Myo
        q.w((float) rotation.w());
        q.x((float) rotation.x());
        q.y((float) rotation.y());
        q.z((float) rotation.z());

        this.mBinder.askToSendOrientationChangedMessage(q);
        this.timestampLastSent = timestamp;
        this.lastOrientationProvided = q;
    }

    @Override
    public void onDisconnect(final Myo myo, final long timestamp) {
        Toast.makeText(getApplicationContext(), "Disconnected from "
            + myo.getName(), Toast.LENGTH_SHORT).show();
    }

}
```

Listato 11.5: *MyoDeviceListener che comunica al framework tutto ciò che viene registrato dal Myo.*

E' ovvio che il BridgeConnectionService non può occuparsi anche della parte di discovery dei dispositivi. Quello che può fare però e che fa, è mettere a disposizione una funzionalità, sempre richiamabile tramite il suo Binder, per connettersi ad uno specifico BluetoothDevice (`askToConnectTo`).

Conclusioni

Il progetto svolto per questa tesi, mi ha portato a dovermi misurare con argomenti e tematiche che non avevo mai affrontato prima, quali la manipolazione delle registrazioni di sensori inerziali, l'utilizzo di sensor fusion filters e l'impiego di quaternioni per tracciare la posizione di un dispositivo; ma anche, più in generale, il semplice fatto di dover progettare un framework e non un'applicazione, rappresenta per me una cosa del tutto nuova.
Similmente, ho anche avuto modo di approfondire conoscenze delle quali già ero in possesso, quali lo sviluppo su piattaforma Android e la gestione di connessioni e comunicazioni via Bluetooth.
Per poter raggiungere il risultato conseguito, del quale mi ritengo complessivamente soddisfatto, sono state necessarie numerose ore di studio degli argomenti sopra citati.

Ovviamente, le difficoltà non sono mancate, sia per quanto riguarda i campi che risultavano a me nuovi, sia per quanto riguarda invece quelli su cui avevo già esperienza. Nonostante ciò, quasi tutte sono state superate, alcune un po' meglio di altre. Con questo ovviamente, non si vuole affermare che il framework realizzato sia, nel suo stato attuale, impeccabile, perché non lo è. Tuttavia esso soddisfa i requisiti che erano stati posti.

Aldilà di quelli che sono i risultati tecnici, ho gradito molto lavorare sul progetto presentato e ritengo che questa esperienza abbia arricchito il mio bagaglio culturale e le mie capacità, sia di modellazione, che di implementazione di un sistema.

Sviluppi Futuri

In futuro, sarebbe interessante inserire, all'interno del framework, funzionalità per condividere il proprio spazio visivo con gli altri.
Con questo si intende, non semplicemente condividere i contenuti della Viewport, ma anche la posizione nella quale essa e quindi i suoi contenuti, si trovano. Questo è, teoricamente, ottenibile condividendo il quaternione che rappresenta la posizione iniziale (`calibration`) del Gaze, con altri dispositivi. Tale qua-

ternione infatti, modella l'orientamento relativo al sistema di coordinate del rotation vector e quindi al WCS, che si basa su coordinate geografiche. Settando a più dispositivi lo stesso quaternione come calibrazione, significa fare in modo che le loro Viewport vengano posizionate tutte nello stesso punto, relativo al WCS e quindi a coordinate geografiche.
In questo modo è possibile anche pensare alla realizzazione di sistemi distribuiti basati sul framework. Il poter condividere un proprio workspace con altre persone, potrebbe avere numerose applicazioni interessanti sia in ambito lavorativo, che sul mercato.

Ringraziamenti

Ringrazio il Professor Ricci e il Dottor Croatti, che mi hanno accompagnato e supportato durante questi ultimi mesi e, soprattutto, durante la stesura di questa tesi.

Ringrazio Federico Marinelli, con il quale ho collaborato nel corso della realizzazione del framework.

Ringrazio inoltre, tutte le persone che ho conosciuto in questi tre anni, che hanno fatto parte di questo percorso e che oggi ho il piacere di chiamare amici.

Un grazie va infine alla mia famiglia, che mi ha permesso di intraprendere questa carriera universitaria e mi ha sostenuto sotto ogni punto di vista, non facendomi mai mancare niente.

Grazie a tutti voi.

Bibliografia

[1] J.M. Carrol, *"The Evolution of Human Computer Interaction"*, introduction to "Human Computer Interaction in the new Millenium, Addison Wesley Professional, November 2001.

[2] J.M. Carrol, *"Human Computer Interaction - brief intro" from "The Encyclopedia of Human-Computer Interaction, 2nd Ed."*, Interaction Design Foundation, 2014.

[3] N. Roussel, *"Looking back, a very brief history of HCI"*, pp. 2, Inria Lille - Nord Europe, January 2014.

[4] D. Saha, A. Mukherjee, *"Pervasive Computing: A Paradigm for the 21st Century"*, pp. 26-27, Computers and Society, March 1998.

[5] M. Weiser, *"The Computer for the 21st Century"*, pp. 94-104, Scientific Am., September 1991.

[6] W. Barfield, *"Fundamentals of Wearable Computing and Augmented Reality 2nd Ed."*, CRC Press, 2015.

[7] T. Starner, *"The Challenges of Wearable Computing: Part 1"*, IEEE Computer Society, 2001.

[8] T. Starner, *"The Challenges of Wearable Computing: Part 2"*, IEEE Computer Society, 2001.

[9] P. Brey, *"Human Enhancement and Personal Identity"*, New Waves in Philosophy of Technology, 2008.

[10] BI Intelligence, *"The Wearables Report: Growth trends, consumer attitudes, and why smartwatches will dominate"*, Business Insider UK, May 2015.

[11] P. Milgram, H. Takemura, A. Utsumi, F. Kishino, *"Augmented Reality: A class of displays on the reality-virtuality continuum"*, ATR Communication Systems Research Laboratories, 1994.

[12] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, B. MacIntyre, *"Recent Advances in Augmented Reality"*, IEEE, 2001.

[13] J. Carmigniani, B. Furht, M. Anisetti, P. Ceravolo, E. Damiani, M. Ivkovic, *"Augmented reality technologies, systems and applications"*, Springer Science+Business Media, 2010.

[14] J.P. Rolland, H. Fuchs, *"Optical Versus Video See-Through Head-Mounted Displays in Medical Visualization"*, Massachusets Institute of Technology, June 2000.

[15] T. Kawashima, K.Imamoto, H. Kato, K. Tachibana, M. Billinghurst, *"Magic Paddle: A Tangible Augmented Reality Interface for Object Manipulation"*, ISMR2001, 2001.

[16] S. Kasahara, V. Heun, A.S. Lee, H. Ishii, *"Second Surface: Multi-user Spatial Collaboration System based on Augmented Reality"*, 2012.

[17] Kircher Electronics, *"Gyroscope Explorer"*, http://www.kircherelectronics.com/gyroscopeexplorer/gyroscopeexplorer, 2015.

[18] P. Lawitzki, *"Android Sensor Fusion Tutorial"*, http://plaw.info/2012/03/android-sensor-fusion-tutorial/, 2012.

[19] P. Bernhardt, *"How I learned to Stop Worrying and Love Quaternions"*, http://developerblog.myo.com/quaternions/, 2015.

[20] A. Pacha, *"Sensor Fusion Demo"*, https://bitbucket.org/apacha/sensor-fusion-demo, 2016.