ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea in Ingegneria Elettronica, Informatica e Telecomunicazioni

SVILUPPO DI APPLICAZIONI ANDROID BASATE SULLA SENSORISTICA

Tesi di Laurea di:
RICCARDO BENEDETTI

Relatore:
Prof. ANTONIO NATALI

ANNO ACCADEMICO 2014–2015
SESSIONE I

PAROLE CHIAVE

Android

Sensoristica

SensorFusion

Accelerometro

Giroscopio

Ai miei genitori e a mia sorella

Indice

Introduzione **ix**

1 Storia e caratteristiche di Android **1**
1.1 Nascita e sviluppo 1
1.2 Le versioni 4
1.3 Panoramica sulla sensoristica 12

2 Sviluppo di applicazioni **15**
2.1 Cos'è un applicazione Android 15
2.2 Struttura di un'applicazione 16
2.3 Gli strumenti dello sviluppatore 23
2.4 Il primo progetto su Android Studio 26
2.5 Il file Manifest 28
2.6 Google Play Store 33

3 Il framework dei sensori **35**
3.1 La classe Sensor Manager 35
3.2 Tipi di sensori 36
3.3 Funzionalità del framework 38
3.4 Interpretazione dei dati 39
3.5 Disponibilità di sensori in un dispositivo 39
3.6 Monitorare gli eventi 41
3.7 Documentazione 42

4 Sensor Fusion **43**
4.1 Gli errori nelle misurazioni 43
4.2 Introduzione al Sensor Fusion 46
4.3 Kionix Software Solution 47
4.4 Determinare l'orientamento di un dispositivo 52

5 L'applicazione Device Orientation **57**
5.1 Caratteristiche e funzionalità 57
5.2 Analisi del problema 58
5.3 Fase d'implementazione 61

5.4 DetermineOrientationActivity.java 73

A La fisica dei sensori **83**

A.1 Accelerometro 83
A.2 Giroscopio 86
A.3 Sensore di Prossimità 90
A.4 Magnetometro 93
A.5 Sensore di luminosità 94
A.6 Barometro 96

Introduzione

Oggigiorno qualsiasi persona, in possesso di un dispositivo mobile, utilizza applicazioni che spesso necessitano di interagire con l'ambiente circostante tramite la sensoristica hardware.
L'obiettivo di questa tesi è quello di fornire le informazioni di base che, un aspirante programmatore Android, deve sapere per scrivere applicazioni che facciano uso dei sensori presenti nei moderni telefoni cellulari (accelerometro, giroscopio, sensore di prossimità , ecc...).
La tesi si apre citando qualche aneddoto storico sulla nascita del sistema operativo più famoso al mondo ed elencando tutte le releases ufficiali e le novità che hanno portato dalla 1.0 all'attuale 5.1.1 Lollipop.
Verranno analizzate le componenti fondamentali per costruire un applicazione Android: Activities, Services, Content Providers e Broadcast Receivers.
Verrà introdotto e approfondito il concetto di sensore, sia punto di vista fisico sia dal punto di vista informatico/logico, evidenziando le tre dimensioni più importanti ovvero struttura, interazione e comportamento.
Si analizzeranno tutte i tipi di errori e problematiche reali che potrebbero influire negativamente sui valori delle misurazioni (disturbi, rumori, ecc...) e si propone la moderna soluzione del Sensor Fusion come caso particolare di studio, prendendo spunto dal lavoro di grandi aziende come la Invensense e la Kionix Inc.
Infine, si conclude l'elaborato passando dalle parole al codice: verranno affrontate le fasi di analisi e d'implementazione di un'applicazione esemplificativa capace di determinare l'orientamento del dispositivo nello spazio, sfruttando diverse tecniche Sensor Fusion.
Il vantaggio principale che ci offre il sistema operativo Android è tutto racchiuso nella parola open source ovvero la possibilità di utilizzare strumenti e librerie native senza alcuna limitazione.
Gli esperimenti e i test, effettuati per studiare e approfondire tutti gli argomenti trattati, sono stati svolti su uno smartphone Samsung Galaxy S3 connesso via USB ad un PC portatile.

Capitolo 1

Storia e caratteristiche di Android

1.1 Nascita e sviluppo

La nascita del sistema operativo Android, grazie a Google, ha portato un evoluzione significativa nel mercato dell'informatica.
L'idea dell'informatico statunitense Andy Rubin è racchiusa nella seguente frase:

> "make a device about the size of a small candy bar that cost less than 10 dollars and allowed users to scan objects and unearth information about them on the Internet"

Figura 1.1: Andy Rubin, fondatore di Android

"Is this interesting to Google?" chiese Andy Rubin a Larry Page nella primavera del 2005, presentando il progetto Android ad una delle più grandi aziende mondiali. Un semplice "yes" fu la risposta.

Figura 1.2: Larry Page e Sergey Brin, fondatori di Google

Rubin e Page si erano incontrati 3 anni prima, quando Rubin era in procinto di lanciare uno smartphone chiamato il Sidekick.
Ai tempi Google era solo un up-and-comer e Rubin scelse proprio la compagnia statunitense come motore di ricerca per il suo Sidekick. Quando Page ne venne a conoscenza si lasciò sfuggire uno spontaneo "Cool".

Nel 2002 il mercato registrava quasi 700 milioni di telefoni cellulari venduti ogni anno a fronte di meno di 200 milioni di PC... e il divario si stava allargando.
Il telefono cellulare era sempre più il mezzo preferito dalla gente per comunicare, eppure il settore della telefonia era piuttosto bloccato e attraversava tempi bui.
Rubin aveva la soluzione: Android, una piattaforma mobile open source che qualsiasi programmatore poteva utilizzare e che qualsiasi produttore poteva installare sui propri dispositivi.
Android avrebbe dovuto avere lo spirito di Linux e la portata di Windows. Sarebbe dovuto diventare un sistema operativo globale, aperto ad un futuro dominato dalla tecnologia wireless.
Page in realtà stava già pensando da tempo di progettare un dispositivo che funzionasse come smart-device con funzioni di telefonia incorporata, prendendo spunto dalla Microsoft con Pocket PC 2002 Phone Edition.
Page e Brin cominceranno ad usare il Sidekick di Rubin ma non abbandonano l'ambizione di realizzare un prodotto similare.

Nel 2005 Google investe 2 milioni di dollari sulla progettazione di PC wireless dal costo di circa 100 dollari l'uno, un idea di Nicholas Negroponte, fondatore dei M.I.T. Media Laboratory.
Nello stesso anno Rubin propone a Page la sua visione futura: una serie di dispositivi (smart-phone) che permettessero agli utenti della telefonia mobile di connettersi ad internet ed estendere così le possibilità comunicare.
La start-up Android Inc. rappresentava questa nuova tipologia di Mobile OS open source basata su Kernel Linux con un'interfaccia utente molto semplice e intuitiva.

Figura 1.3: Il logo della start-up Android Inc.

Page fu sorpreso non solo dal prodotto che gli era stato presentato ma, soprattutto, dalla motivazione con la quale gli era stato presentato. Rubin, infatti, precisò che la sua proposta non era dettata da esigenze economiche (Android Inc. non aveva certamente problemi di fondi), ma sperava che Google scegliesse Android come sistema mobile su cui puntare, proponendo anche l'introduzione di una semplice e-mail di pubblico dominio.
Page, però, sapeva bene che "sponsorizzare" una società terza significava dargli troppo potere (vedere il caso di IBM con Microsoft) e così propose a Rubin ciò che quest'ultimo non si aspettava: l'acquisizione di Androind Inc.

Nel 2005 Android Inc. si trasforma nella Google Mobile Division affidata, ovviamente, alla cura di Rubin e degli altri tre co-fondatori (Rich Miner, Nick Sears e Chris White).
Dopo che Google ha acquistato Android nel luglio 2005, Silicon Valley pulsava di pettegolezzi e ipotesi su ciò che il gigante della ricerca stava progettando.
Quando Google ha finalmente rotto il suo silenzio, all'inizio di novembre, non c'era nulla circa un Google Phone stile Apple come molti avrebbero immaginato. Invece, fu presentato un comunicato stampa che diceva che trentaquattro aziende - come Texas Instruments, Intel, T-Mobile e Sprint Nextel - si erano unite a Google per costruire un'interfaccia wireless basata su software open source Linux. Il gruppo si soprannominata la Open Handset Alliance.
I principali concorrenti/protagonisti del mercato non riescono a trattenere i loro commenti: Steve Ballmer (CEO di Microsoft) liquidò la cosa durante una conferenza in Giappone affermando che "...i loro sforzi sono solo parole su carta", mentre il CEO di Nokia definì la cosa come "Un'altra piattaforma Linux".
Una settimana dopo, Google alza la posta. La società mette online l'Android Software Developer's Kit e annuncia la Developer Challenge, con 10 milioni di dollari di premi in denaro da spartire con i creatori delle migliori applicazioni per il nuovo sistema, ad esempio un grande social network o un programma di riconoscimento della scrittura a mano.
La sfida era un invito aperto a tutti ed era una grande occasione per qualunque aspirante programmatore Android.

Coloro che speravano in un nuovo gadget per rivaleggiare contro l'iPhone avevano finalmente capito che Google aveva qualcosa di radicalmente diverso in mente.
Non ci sarebbe stato un gPhone (o Google Phone), ci sarebbero centinaia di gPhones. HTC, Motorola, LG e tanti altri annunciarono l'intenzione di commercializzare nuovi telefoni Android in una moltitudine di forme e dimensioni, ognuno con diverse opzioni software. Android è un sistema completamente personalizzabile. Anche le poche applicazioni che Google stava creando da zero (app email, gestore di contatti, ecc..) potevano essere sostituite con software di terze parti che offrivano lo stesso identico servizio.

Figura 1.4: I più famosi gPhones

Il DNA di Android era lì sotto agli occhi di tutti: tenere il tutto legato a Internet e senza intoppi.
Android ha portato Google a sviluppare un sistema in cui ogni nuova release rappresenta un sostanziale step evolutivo di ciò che oggi è uno dei punti di riferimento per il mercato degli smartphone, giocandosi il primo posto con iOS e Windows Phone.

1.2 Le versioni

La storia delle releases di Android comincia con la versione beta del novembre 2007 cioè con il rilascio del SDK (software development kit).
L'SDK era composto dagli strumenti di sviluppo, le librerie, un emulatore del dispositivo, la documentazione in inglese, esempi di progetti, ecc...
La prima versione commerciale, Android 1.0, è stata rilasciata nel settembre 2008.
Dall'aprile 2009, le versioni di Android, dalla 1.5 in poi, hanno cominciato a prendere il nome di prodotti dolciari e procedono esattamente in ordine alfabetico.
Nel novembre 2014 viene rilasciata Lollipop, quella che al momento (e con "al

momento" si intende luglio 2015, ovvero la data di completamento di questa tesi) è la versione attuale.

Figura 1.5: I loghi delle versioni di Android

Figura 1.6: La versione attuale di Android: Lollipop

Versione	**Caratteristiche (in ordine cronologico)**
Alpha (1.0)	• **API Level 1**
Beta (1.1)	• **API Level 2** • Aggiornamento per risoluzione di alcuni bug
Cupcake (1.5)	• **API Level 3** • **Linux kernel 2.6.27** • Migliorata integrazione con servizi Google (supporto widget)
Donut (1.6)	• **API Level 4** • **Linux kernel 2.6.29** • Aggiunta di ricerca vocale e testuale per i contenuti presenti in locale e sul Web • Introdotta la sintesi vocale e le gesture
Eclair (2.0 - 2.1)	• **API Level 5 (in Eclair 2.0)** • Aggiunte funzionalità per fotocamera • Migliorata la sincronizzazione dell'account Google e aggiunto il supporto agli account Exchange • Aggiunto il supporto al multitouch e ai live wallpaper • User Interface e prestazioni migliorate • **API Level 6 (in Eclair 2.0.1)** • **API Level 7 (in Eclair 2.1)**

Froyo (2.2 - 2.2.3)	• **API Level 8 (in Froyo 2.2)** • **Linux kernel 2.6.32** • Migliore gestione delle risorse hardware (debug JIT) • Aggiunti Tethering USB e Wi-Fi • Integrazione del motore JavaScript V8 di Google Chrome nel browser di sistema • Supporto alla tecnologia Adobe Flash • Miglioramenti alle altre app del sistema • Drastico incremento prestazionale e miglioramento della sicurezza • Patch di sicurezza
Gingerbread (2.3 - 2.3.7)	• **API Level 9 (in Gingerbread 2.3)** • **Linux kernel 2.6.35** • User Interface più user-friendly • Aggiunto il supporto agli schermi XL • Supporto nativo al SIP VoIP e alla tecnologia NFC • Tastiera migliorata e riprogettata • Aggiunta l'app Download Manager • Supporto nativo a giroscopio e barometro • **API Level 10 (in Gingerbread 2.3.3)** • Supporto per la chat video e vocale tramite Google Talk • Migliorate le applicazioni Fotocamera e Gmail • Efficienza energetica migliorata • Supporto a Google Wallet per Nexus S 4G

<table>
<tr><td>Honeycomb (3.0 - 3.2.6)</td><td>
API Level 11 (in Honeycomb 3.0)
Linux kernel 2.6.36
Nuova User Interface (Holo)
Aggiunta barra di sistema e Action Bar
Aggiunto il multi-tab al browser
Supporto per processori multi-core
Possibilità di criptaggio dei dati personali
API Level 12 (in Honeycomb 3.1): miglioramenti a UI e widget, supporto per periferiche USB
API Level 13 (in Honetcomb 3.2)
Possibilità per gli sviluppatori di personalizzare la UI
Aggiornamento di Google Ricerca libri e Android Market
Supporto per i Tablet 3G e 4G dell'opzione Pay as You Go
</td></tr>
<tr><td>Ice Cream Sandwich (4.0 - 4.0.4)</td><td>
API Level 14 (in Ice Cream Sandwich 4.0)
Linux kernel 3.0.1
User Interface rivoluzionata (prestazioni, pulsanti, cartelle, font di sistema...)
Aggiornamento di tutte le app in vista delle nuove API
Possibilità di scattare screenshots
Dettatura in tempo reale
Face Unlock (sbloccare dispositivo tramite riconoscimento facciale)
Possibilità di accedere alle applicazioni direttamente dalla schermata di sblocco
Fotocamera migliorata (ritardo scatto nullo, modalità panorama e zoom video)
</td></tr>
</table>

...	• App Contatti con integrazione con i social network • Android Beam (scambio di dati tramite NFC) • Wi-Fi Direct • **API Level 15 (in Ice Cream Sandwich 4.0.3)**:migliorata accessibilità, multitasking, rotazione schermo, app Camera...
Jelly Bean (4.1 - 4.3.1)	• **API Level 16 (in Jelly Bean 4.1)** • **Linux kernel 3.0.31** • Miglioramenti nel riconoscimento del tocco e digitazione del testo • Miglioramenti in gestione widget • Project Butter (miglioramenti in fluidità) • Ottimizzazione fotocamera • Nuove funzionalità per la condivisione di foto e video tramite NFC • Miglioramento di Android Beam • Gesture avanzate per le notifiche • Nuovo servizio Google Now • Progressi nella sintesi vocale • Aggiornamento Google Play Store • Riconoscimento vocale avanzato (GSV) • Abbandono tecnologia Adobe Flash • **API Level 17 (in Jelly Bean 4.2)** • **Linux kernel 3.4.0** • Implementazione nativa della funzione swipe nella tastiera

...	• Ulteriore miglioramento di Google Now e Android Beam • Introdotta modalità fotografica a 360° (PhotoSphere) • **API Level 18 (in Jelly Bean 4.3)** • OpenGL ES 3.0 • Nuova fotocamera • Wi-Fi always-on • Dialer con la ricerca dei contatti • Accesso alle notifiche da parte delle app • Connessione Bluetooth con i dispositivi a basso consumo • Possibilità di introdurre contenuti DRM • Introdotte le impostazioni per il multiutente (Restricted Profile) che permettono di scegliere quali contenuti e quali applicazioni gli utenti possono visualizzare
KitKat (4.4 - 4.4.4)	• **API Level 19 (in KitKat 4.4)** • Rinnovamento dell'interfaccia e introduzione del full screen completo • Hangouts: client ufficiale per SMS e MMS • Nuove funzioni di chiamata • Possibilità di registrare video mp4 su ciò che avviene sul display • Supporto nativo alla stampa di foto • Possibilità di indirizzare documenti e web-page verso stampanti compatibili (Google Cloud Print, HP ePrint...) • Nuova voce per la Home nelle impostazioni per lo switch tra launcher

...	• Aggiunte le emoji (emoticon avanzate) • Rinnovamento app Download (ordinamento e visualizzazione) • Supporto per 3 nuovi sensori (vettore di rotazione geomagnetica, rilevatore e contatore di passi) e, di conseguenza, aggiunta la funzione contapassi • Riproduzione audio a risparmio energetico • Introduzione e sperimentazione del nuovo compilatore ART (Android RunTime) • Introdotte barre trasparenti nel blocco dello schermo • “Foto” sostituisce l’app Galleria • Aggiornamento Fotocamera (messa a fuoco in condizioni di scarsa luminosità, bilanciamento del bianco, chiusura dell’otturatore e introdotta la possibilità di effettuare pinch to zoom sullo smartphone in modalità HDR+)
Lollipop (5.0 - 5.0.2)	• **API Level 21 (in LolliPop 5.0)** • Nuova User Interface (la Material Design) • Aggiunte 5000 API • Rinnovato il multitasking • Rinnovamento gestione audio • Google Fit (rilevamento attività fisiche) • **Linux Kernel 3.10.x** • Introduzione della runtime ART • Supporto nativo ai 64 bit • Bluetooth 4.1 • USB Audio

...	• Burst Mode • Gestione più efficiente della batteria (Battery Saver) • Più prestazioni grafiche (OpenGL ES 3.1) • Aggiunte API per migliorare fotocamera • Notifiche Heads Up in App Fullscreen • Personal Unlocking • Introduzione del multi-utente e modalità ospite • Nuova LookScreen con notifiche classificabili • Google Play Service 6.0

La precedente tabella si riferisce alle versioni pubblicate solo per smartphone e tablet. L'API Level 20 è mancante poichè ricade nelle versioni pubblicate per Android Wear.

1.3 Panoramica sulla sensoristica

La maggior parte dei dispositivi Android sono dotati di sensori che misurano il movimento, l'orientamento e varie caratteristiche ambientali.
Questi sensori sono in grado di fornire dati con elevata precisione e accuratezza e sono utili se si desiderano monitorare gli spostamenti, dal punto di vista tridimensionale, o i cambiamenti delle condizioni dell'ambiente nelle vicinanze.
Ad esempio: un gioco potrebbe eseguire più letture del sensore di gravità in funzione di movimenti più o meno complessi compiuti dall'utente come l'inclinazione, l'agitazione, la rotazione, ecc... Allo stesso modo, si potrebbe usare il sensore di temperatura e umidità per effettuare rilevazioni meteorologiche e calcolare parametri come il punto di rugiada... o altrimenti, un navigatore potrebbe utilizzare il magnetometro e l'accelerometro per simulare il funzionamento della bussola.

La piattaforma Android supporta tre grandi categorie di sensori:

- **I sensori di movimento**
  Misurano, lungo tre assi, le forze che provocano l'accelerazione e la rotazione. Questa categoria comprende accelerometri, sensori di gravità, giroscopi e il vettore di rotazione.

Figura 1.7: Amiigo: l'applicazione per il fitness che utilizza la sensoristica Android

- **I sensori ambientali**
  Misurano vari parametri ambientali, come la temperatura ambiente, la pressione dell'aria, la luminosità e l'umidità. Questa categoria comprende barometri, fotometri, e termometri.

- **I sensori di posizione**
  Rilevano la posizione fisica di un dispositivo. Questa categoria comprende sensori di orientamento e magnetometri.

È possibile accedere ai sensori disponibili sul dispositivo e acquisire i dati tramite il framework Android. Il framework fornisce diverse classi e interfacce che consentono di eseguire un'ampia varietà di operazioni sui sensori connessi. Si può utilizzare il framework per effettuare le seguenti operazioni:

- Determinare quali sensori sono disponibili su un dispositivo.
- Determinare le capacità di un singolo sensore, come la sua portata massima, il produttore, i requisiti di alimentazione e la risoluzione.
- Acquisire dati grezzi del sensore e impostare la frequenza di rilevamento dei dati
- Registrarsi o deregistrarsi ai "listener" che monitorano i cambiamenti del sensore, in modo da percepire o meno gli eventi emessi.

Questa sezione fornisce una panoramica della sensoristica disponibile sulla piattaforma Android e fornisce inoltre un'introduzione al framework.
Il framework Android permette di accedere ai tanti tipi di sensori. Alcuni di questi sono basati su hardware e alcuni sono basati su software. Quelli basati su hardware sono componenti fisici integrati in un dispositivo portatile e producono

dati misurando direttamente proprietà ambientali specifiche come l'accelerazione, intensità del campo magnetico terrestre o la variazione angolare. I sensori basati su software non sono dispositivi fisici, ma imitano quelli basati su hardware. Essi producono dati sfruttando uno o più dei sensori basati su hardware e sono anche chiamati sensori virtuali o sintetici. Il sensore di accelerazione lineare e quello di gravità sono esempi di sensori basati su software.

Capitolo 2

Sviluppo di applicazioni

Questo capitolo vi spiegherà come sviluppare la vostra prima applicazione Android.
Innanzitutto verrà esaminato il comportamento di un'applicazione all'interno dell'S.O. e verranno descritte le componenti fondamentali che caratterizzano la struttura (activity, service, broadcast receiver, ecc...).
In secondo luogo verranno mostrati i software principali necessari per lavorare con la piattaforma Android (JDK, Android Studio, Android SDK, ecc...) e vi verrà spiegato come installarli sul vostro pc.
Infine, con riferimento all'IDE Android Studio, vi guideremo alla creazione del primo progetto Android.

2.1 Cos'è un applicazione Android

Le applicazioni Android vengono scritte in linguaggio Java. Gli strumenti dell'SDK compilano tutto il codice e producono un file APK (Android Package) e, grazie a tale file, l'applicazione può essere installata sul dispositivo.
Una volta installata, l'app vive all'interno di un ambiente di sicurezza che nella terminologia originale è soprannominato "sandbox".
Il S.O. Android è una versione multi-utente del S.O. Linux in cui ogni applicazione è rappresentata da un processo, ben distinto dagli altri, che gira in modo isolato da tutte le altre applicazioni. I processi vengono creati quando uno o più dei componenti dell'app vengono eseguiti e terminati quando il S.O. non ne ha più bisogno. In questo modo la memoria RAM viene gestita in modo intelligente e viene liberata per consentire l'esecuzione di nuovi processi.
Android implementa il principio del minimo privilegio, cioè, ogni applicazione ha accesso ad un numero limitato di risorse tali da permettergli di fare il suo lavoro e non di più. Tutto ciò implica maggior sicurezza perchè ai processi applicativi viene impedito di accedere alle parti più delicate e vitali del sistema operativo, il cui

danneggiamento potrebbe comprometterne in modo irreversibile il funzionamento. Tuttavia, esistono meccanismi per condividere dati tra applicazioni e per utilizzare i servizi di sistema. Un modo per fare ciò consiste nell'inizializzare due applicazioni con lo stesso Linux ID ed, eventualmente, farle girare nello stesso processo e condividere la stessa "sandbox" (ma solo se le due applicazioni sono state firmate con lo stesso certificato).
A quali file si potrebbe accedere? Contatti utente, SMS, supporti rimovibili (SD Card), fotocamera, Bluetooth, ecc... Tutte le autorizzazioni devono essere approvate in fase di installazione dell'app.

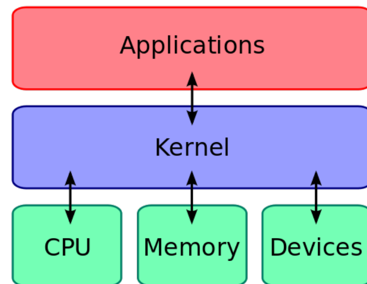

Figura 2.1: Interazione tra applicazioni, dati e servizi

In questo paragrafo si è vista l'applicazione da un punto di vista esterno, ovvero, come un'entità che interagisce con altri componenti software in maniera disciplinata, grazie al Sistema Operativo.
Nel prossimo paragrafo verrà analizzata la sua struttura interna e le sue parti vitali che ne caratterizzano il comportamento.

2.2 Struttura di un'applicazione

Le applicazioni Android sono costruite basandosi su cinque elementi fondamentali:

- **Activity**
- **Intent**
- **Broadcast Receiver**
- **Service**
- **Content Provider**

Activity. Derivate dalla classe *android.app.Activity*, rappresentano l'elemento più importante della programmazione Android.
Le Activity rappresentano blocchi logici dell'applicazione ed sono responsabili dell'interazione diretta con l'utente mediante i dispositivi di input dello specifico device.

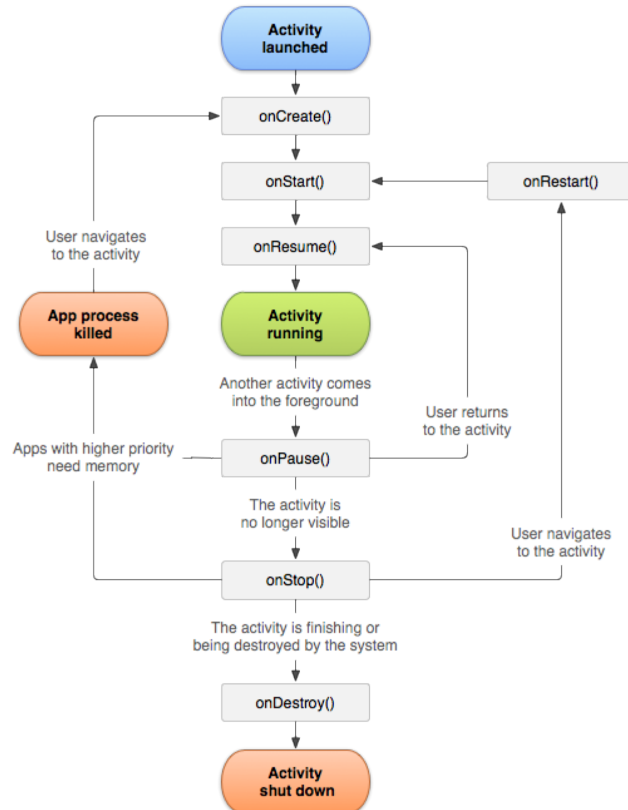

Figura 2.2: Activity: ciclo di vita

E' un errore molto comune associare tutte le activities alle user interface ma è anche vero che la maggior parte delle schermate che compaiono sul nostro dispositivo sono delle activities (ad esempio: la schermata principale di Whatsapp è un'activity; se si entra in una chat room viene eseguita una nuova activity e sospesa la precedente; se si tocca il campo di input testuale per scrivere un messaggio appare una tastiera virtuale che però non è un'activity ma un service).
Nello switch tra un'activity e l'altra è possibile fare in modo che la prima restituisca un particolare valore alla seconda (ad esempio: se su Whatsapp si vuole inviare

un video ad un contatto, l'activity che permette la navigazione della galleria per la ricerca del video restituirà un riferimento al file selezionato all'activity che si occuperà dell'invio).
Le activity sospese sono conservate nell'activity stack, ovvero una pila di tipo FIFO (first-in-first-out). E' dispendioso conservare molte activity nello stack ed è compito del sistema operativo provvedere a rimuovere quelle inutili.
A partire dalla versione Android 3.0 sono stati introdotti i **Fragments**. È possibile combinare più Fragment in una singola activity, per costruire una interfaccia utente multi-pannello, e riutilizzare un Fragment in molteplici Activities.
I Fragments hanno un ciclo di vita uguale o minore del ciclo di vita dell'Activity che li ospita, possono rilevare eventi di input e possono essere aggiunti (o rimossi) dinamicamente mentre l'Activity è in esecuzione. Non solo il ciclo di vita dei Fragments è fortemente influenzato dall'Activity, ma anche il loro stato (se l'Activity è in pausa i Fragments sono in pausa, se l'Activity viene distrutta i Fragments vengono distrutti).
Android ha introdotto i Fragment principalmente per sostenere i design di interfacce utenti su grandi schermi, come i tablet, per renderle più dinamiche e flessibili. Poiché lo schermo di un tablet è molto più grande di quello di uno smartphone, c'è più spazio per combinare e interscambiare i componenti della U.I..
I Fragments permettono di gestire tali design senza la necessità di di effettuare cambiamenti complessi alla struttura della vista. Suddividendo il layout di un'Activity in Fragments, si è in grado di modificare l'aspetto della propria Activity in fase di esecuzione e di separare la parte grafica da quella decisionale, in modo da adattare l'applicazione al dispositivo in uso a runtime (ad esempio: in un tablet un'applicazione per un quotidiano può usare un Fragment per visualizzare un elenco di articoli sulla sinistra e un altro Fragment per visualizzare l'intero articolo sulla destra, il tutto dentro un unica Activity. Si veda la figura successiva).

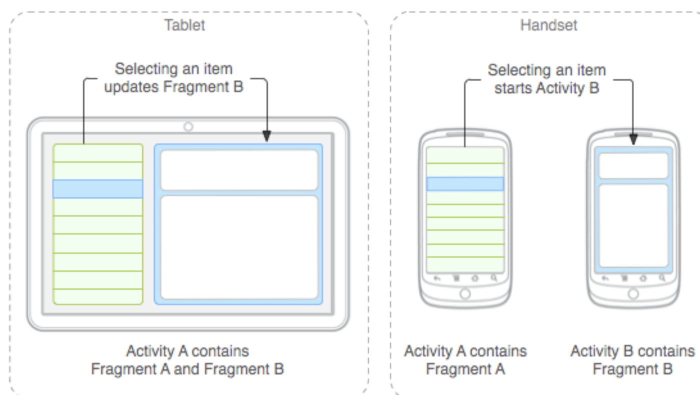

Figura 2.3: Fragments: l'app "quotidiano" in tablet e smartphone

Intent. Sono dei messaggi che forniscono una struttura per eseguire il late binding in diverse applicazioni. Questi messaggi (definiti in *android.content.Intent*) hanno il compito di attivare i tre elementi fondamentali della programmazione Android: le Activities, i Services e i Broadcast Receiver. La struttura dati dell'Intent è composta da:

- **action:** l'azione da eseguire (es. ACTION_VIEW, ACTION_DIAL, ACTION_MAIN, ACTION_EDIT, ecc...).
- **data:** i dati su cui eseguire l'azione, come ad esempio un record "Persona" tra i contatti della rubrica.

Gli Intent possono essere di due tipi: espliciti o impliciti.
In quelli espliciti si specifica, all'interno del costruttore, l'oggetto da eseguire (solitamente il nome dell'Activity)

```
1 Intent esplIntent = new Intent(this, SubActivity.class);
2 startActivity(esplIntent);
```

In quelli impliciti si specifica l'azione da eseguire senza sapere a chi effettivamente sarà delegata l'esecuzione. Nel codice successivo l'Intent avvierà un'Activity che aprirà il sito dell'Ansa:

```
1 Uri ansaUri = Uri.parse("http://www.ansa.it");
2 Intent implIntent = new Intent(Intent.ACTION_VIEW, ansaUri);
3 startActivity(implIntent);
```

Spesso, però, gli Intent hanno bisogno di permessi per eseguire certi tipi di operazioni. Questi permessi sono dichiarati nel file Manifest descritto nel prossimo paragrafo.
Ricapitolando, gli Intent sono pensati per essere gli intermediari responsabili dello switch tra Activities attraverso il metodo *startActivity(myIntent);*. Grazie a questo metodo la nuova Activity viene informata e lanciata.
E' possibile, per un'applicazione, mandare un'Intent ad un'altra applicazione per farle eseguire una determinata azione? Per far ciò esistono i PendingIntent.
Le istanze di questa classe sono create con:

- *public static PendingIntent getActivity (Context context, int requestCode, Intent intent, int flags);*
- *public static PendingIntent getBroadcast (Context context, int requestCode, Intent intent, int flags);*
- *public static PendingIntent getService (Context context, int requestCode, Intent intent, int flags);*

e consentono al componente che riceve tale PendingIntent di eseguire una nuova Activity/BroadcastReceiver/Service tramite il metodo *startActivity(myIntent);* citato prima.

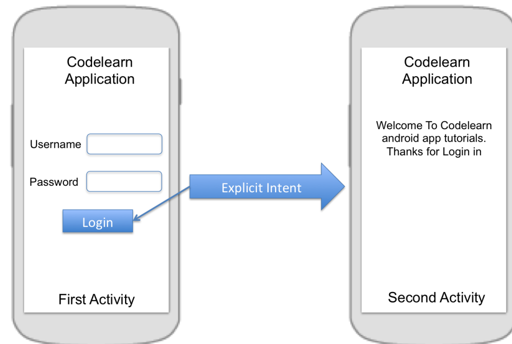

Figura 2.4: Intent: switch tra Activities

Broadcast Receiver. La classe *android.content.BroadcastReceiver* permette di eseguire del codice in risposta ad un evento esterno improvviso (ad esempio: chiamata entrante, batteria scarica, la connessione dati diventa assente/disponibile ecc...). Il vantaggio principale è che l'applicazione contenente il codice da eseguire non deve necessariamente essere in esecuzione ma viene azionata dal Broadcast Receiver stesso e il suo output si sovrappone all'output dell'activity in esecuzione al momento.
I Broadcast Receiver non possiedono una propria interfaccia grafica ma avvisano l'utente tramite messaggi utilizzando il metodo *NotificationManager()*.
Come per gli Intent, anche la registrazione del BR all'interno dell'*AndroidManifest.xml* è obbligatoria, ma in questo caso si può fare anche runtime, grazie al metodo *Context.registerReceiver()*. Qualunque applicazione poi può lanciare BroadcastReceiver nel sistema con il metodo *Context.sendBroadcast()*.

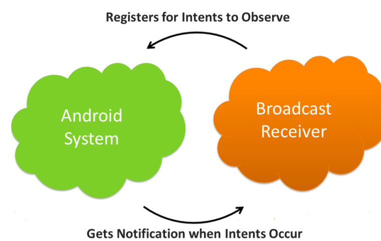

Figura 2.5: Broadcast Receiver

Service. Mentre le Activities solitamente vengono utilizzate quando c'è bisogno di un'interazione diretta con l'utente tramite un interfaccia grafica, i Services rappresentano l'esecuzione di attività autonome in background e sono implementati in *android.app.Service*.
Un esempio di Service consiste nella riproduzione di un file audio nel proprio smarphone. In questo caso chiunque sa che, anche se il dispositivo sta utilizzando le proprie risorse per riprodurre la musica, tutto ciò avviene in sottofondo ed è comunque consentito utilizzare altre applicazioni durante l'ascolto (e quindi avviare altre Activities).

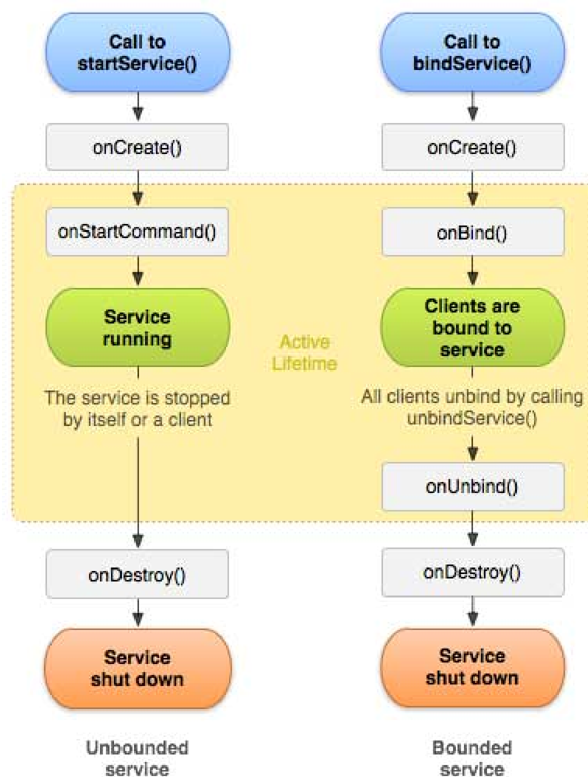

Figura 2.6: Services: ciclo di vita

E' importante notare che un Service NON rappresenta un processo o un thread separato dall'applicazione, ma è eseguito all'interno dello stesso processo dell'applicazione.
I Services vengono utilizzati dall'applicazione per comunicare al sistema che una particolare operazione verrà eseguita in background anche quando l'utente non sta

interagendo in modo esplicito con essa.
Invocando una *Context.bindService()* viene stabilita una connessione tra l'app e il richiedente del Service ed è proprio tramite questo metodo che un'applicazione mette a disposizione le proprie funzionalità alle altre.

Content Provider. L'ultimo componente che verrà esaminato è il Content Provider, definito nel package *android.content*.
Il Content Provider nasce dall'esigenza delle applicazioni di memorizzare i propri dati e di mantenerne la persistenza e la consistenza anche a fronte di una eventuale terminazione del processo o di una failure del sistema.
Per fare ciò le app utilizzano il file-system, i database SQLite o il web. Per quanto riguarda la condivisione dei dati con le altre applicazioni, queste strutture non sono più sufficienti e pertanto bisogna ricorrere alla classe ContentProvider. Le operazioni principali di ContentProvider permettono alle applicazione di salvare, cancellare, leggere i dati.
Tutti i Content Providers vanno dichiarati nel Manifest come elementi di tipo <provider>

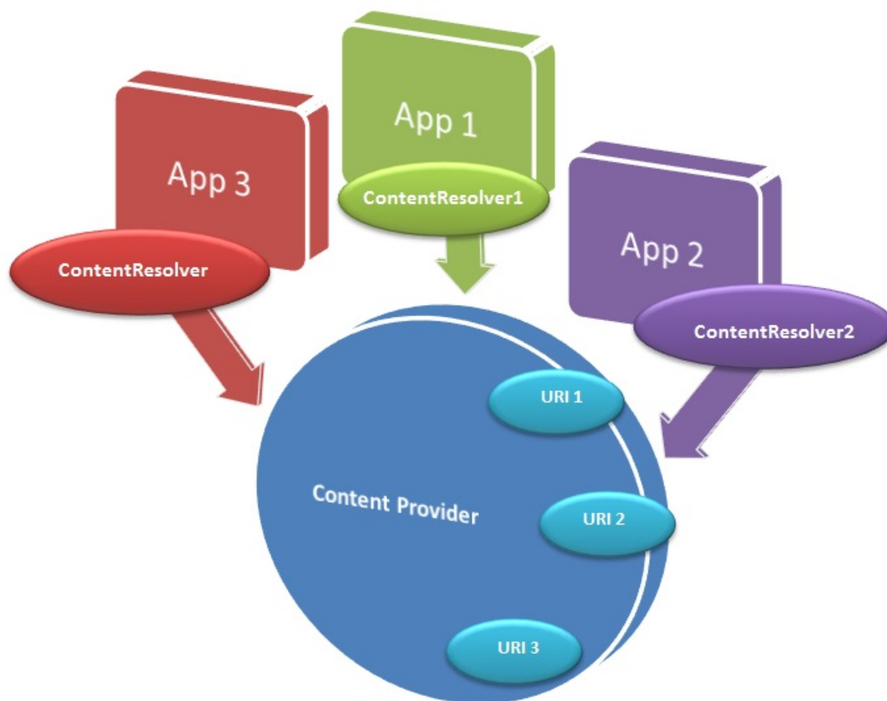

Figura 2.7: Content Provider e applicazioni

2.3 Gli strumenti dello sviluppatore

Il seguente paragrafo è rivolto a chi possiede un computer con installato il sistema operativo Windows 7/8, ma nel caso si usi Linux o Mac è possibile trovare sul web altre guide molto dettagliate.
Non è fondamentale, anche se consigliato, possedere un dispositivo Android poichè, l'installazione dei seguenti strumenti comprende anche un emulatore.

JDK. Il JDK (Java Development Kit) è l'insieme degli strumenti per sviluppare programmi Java ed è un prodotto della Oracle scaricabile dal sito ufficiale semplicemente accettando le condizioni e scegliendo la versione adatta al pc. Il link per la versione 8 è `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`.
Dopo il download, l'installazione è il passo più semplice ma non l'ultimo.
E' consigliato controllare che l'installazione sia andata a buon fine aprendo il prompt dei comandi, digitando in input il comando "javac -version" e verificare che in output venga effettivamente stampata la versione che è appena stata installata.

Figura 2.8: Java Development Kit

In seguito, per un corretto avvio degli eseguibili Java, aggiungere la cartella bin del JDK nella variabile d'ambiente PATH di Windows. Per fare ciò bisogna individuare la locazione JAVA_HOME (cioè il compilatore javac.exe e il runtime java.exe). Tale cartella può variare da pc a pc in base alle scelte effettuate in fase di installazione; si suppone che tale percorso sia: C:\Program Files\Java\jdk1.8.0_40\bin.
Per trovare la variabile PATH bisogna cliccare su START, Pannello di Controllo, Sistema e Sicurezza, Sistema, Impostazioni di Sistema Avanzate e Variabili d'Ambiente. Infine, occorre modificare PATH (in variabili di sistema), aggiungendo la

JAVA_HOME alla stringa che ne rappresenta il valore.

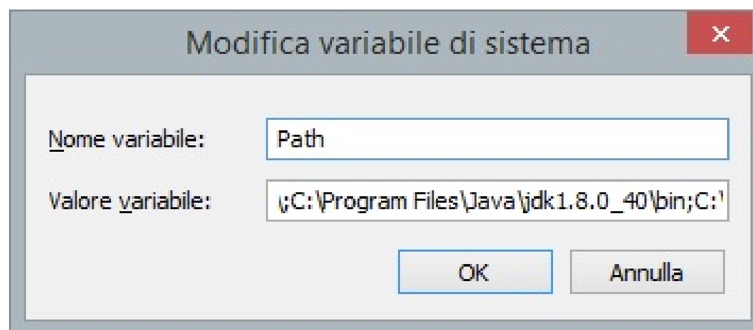

Figura 2.9: Modifica della variabile d'ambiente PATH

Android Studio. Il secondo passo consiste nel procurarsi un ambiente di sviluppo. In questa tesi si è scelto l'Android Studio IDE (integrated development environment) ovvero un ambiente di sviluppo integrato per gli sviluppatori Android.
Ideato da Google, che ne ha rilasciato la prima versione stabile nel dicembre 2014, Android Studio è disponibile per Windows, Mac OS X e Linux, e prende il posto di Eclipse (ADT) come IDE principale per lo sviluppo di applicazioni native Android.

Figura 2.10: Android Studio 1.1.0: l'attuale ultima versione

Per quanto riguarda il download, seguendo il link `http://developer.android.com/sdk/index.html`, è possibile trovare l'ultima versione disponibile.
Avviando l'eseguibile, è necessario scegliere il workspace, ovvero la cartella dove saranno contenuti i progetti creati, e installare l'Android SDK (Software Development Kit), che verrà illustrato nel prossimo paragrafo.

Android SDK. Un Software Development Kit (pacchetto di sviluppo per applicazioni) è un insieme di strumenti che consentono lo sviluppo e la documentazione di software. Un SDK è dotato di un compilatore, librerie standard con relative API, documentazione, licenze, ecc... Lo specifico SDK concepito per Android è compreso nell'installazione di Android Studio.

Figura 2.11: Android SDK

Una volta avviato Android Studio, nella schermata iniziale cliccare su Configure e, successivamente, su SDK Manager.
Infine, scegliere le versioni da installare e accettare le condizioni d'uso. L'installazione può richiedere tanto tempo se è fatta per la prima volta. Una volta completato anche questo passo, è bene ricordare che le versioni possono essere aggiunte/rimosse liberamente in secondi momenti.

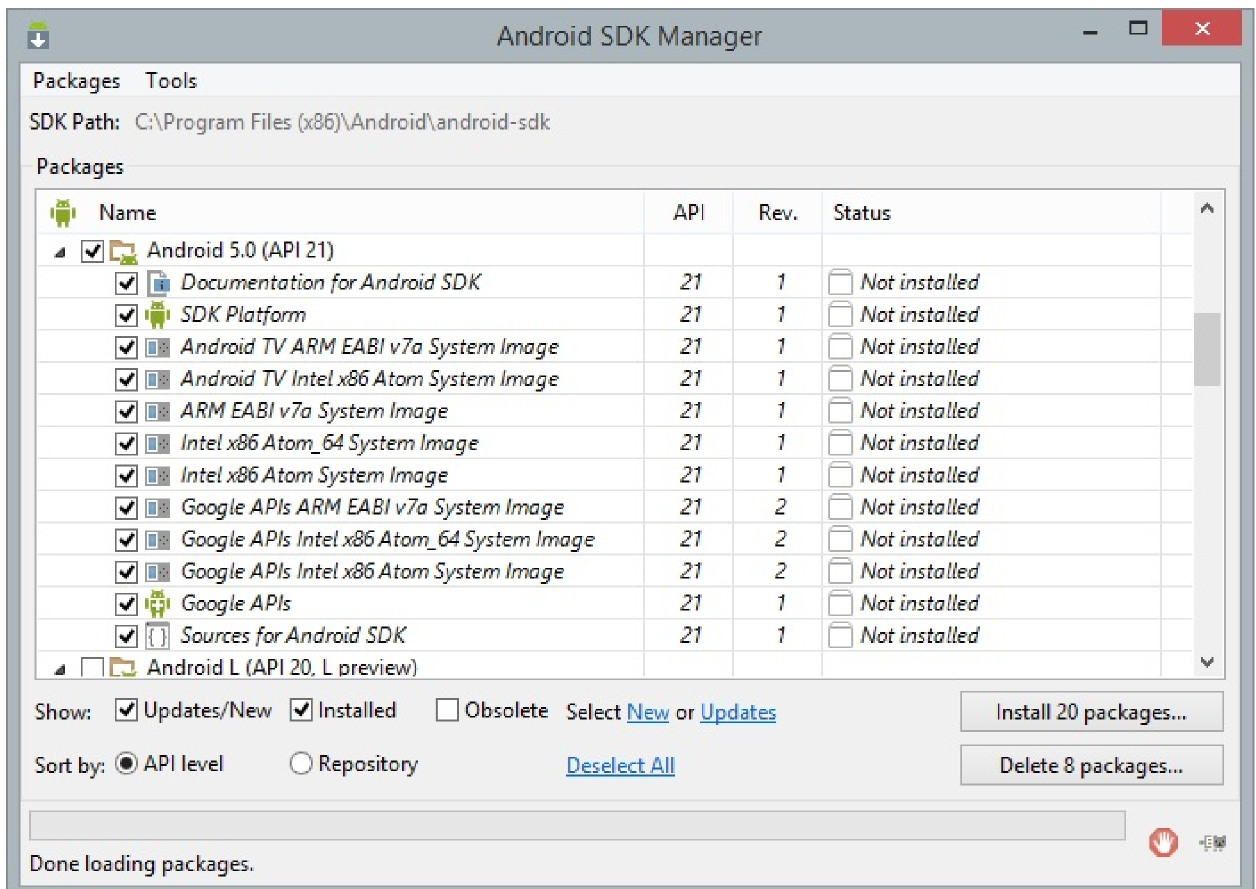

Figura 2.12: SDK Manager

2.4 Il primo progetto su Android Studio

Creare il primo progetto Android è molto semplice, basta cliccare in alto a sinistra su File e New Project. La schermata che si aprirà chiede il nome e la locazione del progetto all'interno del file system.
Il secondo passo è molto importante perchè consiste nella scelta della piattaforma sulla quale girerà la nostra app (in questa tesi ci si focalizza sugli smartphone). Viene chiesto inoltre di indicare la versione (API) di Android di minima compatibilità. Se si scelgono le ultime versioni, si avranno più novità da sfruttare ma meno dispositivi compatibili con l'applicazione, e viceversa se si opta per le versioni più antiche.
Infine l'ultimo step chiede di impostare un layout predefinito per la MainActivity.

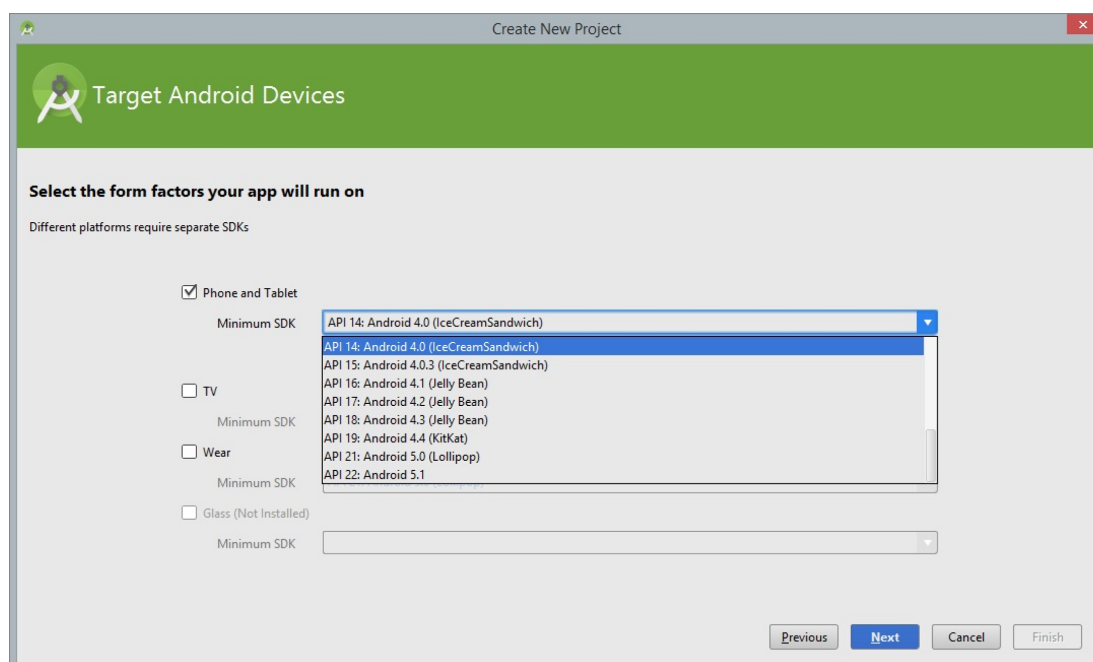

Figura 2.13: Android Studio: scelta piattaforma e API

Caricato il progetto, esaminiamo la struttura di cartelle create di default da Android Studio. Queste cartelle sono comuni a tutte le applicazioni Android e suddividono in maniera ordinata i file necessari a far girare l'applicazione.

- *java*: Contiene tutto il codice sorgente dell'applicazione, il codice dell'interfaccia grafica, le classi etc. Inizialmente Android Studio genererà in automatico il codice della prima Activity.

- *res*: Contiene tutti i file e le risorse necessarie del progetto: immagini, file xml per il layout, menù, stringe, stili, colori, icone ecc...

- *manifest*: Contiene il file *AndroidManifest.xml* (esaminato nel dettaglio nella prossima sezione) ovvero la definizione XML degli aspetti principali dell'applicazione, come ad esempio la sua identificazione (nome, versione, logo), i suoi componenti (Views, messaggi) o i permessi necessari per la sua esecuzione.

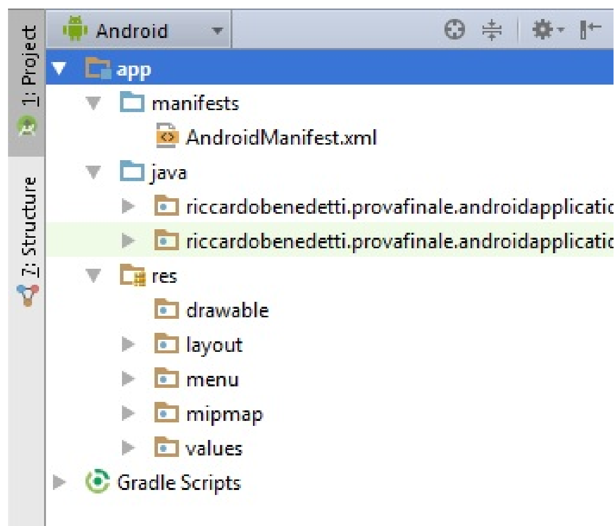

Figura 2.14: Android Studio: struttura cartelle

2.5 Il file Manifest

Qualsiasi applicativo Android deve essere provvisto di un file detto *AndroidManifest.xml* nella sua directory principale.
Il file Manifest può essere pensato come il biglietto da visita con il quale un'applicazione si presenta al Sistema Operativo e le informazioni che contiene sono essenziali ancor prima di poter eseguire qualsiasi codice.
Altre caratteristiche del file Manifest sono elencate di seguito:

- Contiene il nome del Java package principale come identificatore univoco per l'applicazione.
- Descrive i principali componenti dell'app ovvero le activities, i services, i broadcast receiver e i content providers di cui l'applicazione è composta. Tiene traccia dei nomi delle classi che implementano ciascuno di essi e , ad esempio, i messaggi Intent che possono gestire. Queste dichiarazioni permettono al S.O., oltre che avere una lista di tutti i componenti, di sapere sotto quali condizioni possono essere lanciati.
- Indica quali processi ospiteranno i componenti dell'applicazione.
- Dichiara quali permessi deve avere l'applicazione per potere accedere alle parti protette del S.O. e per interagire con altre applicazioni.
- Dichiara quali permessi devono avere le altre applicazioni per interagire con i componenti dell'applicazione in questione.

- Elenca gli Instrumentation classes (si veda http://developer.android.com/reference/android/app/Instrumentation.html) che contengono informazioni su come l'applicazione è in esecuzione. Queste dichiarazioni sono presenti nel Manifest solo in fase di sviluppo e di test e vengono rimosse quando l'app viene pubblicata.
- Dichiara la versione (API) di Android minima richiesta per far girare l'applicazione
- Elenca le librerie linkate all'app.

Il codice Xml che segue mostra la struttura generale del file Manifest con tutti gli elementi che può contenere.
Per maggiori dettagli e/o informazioni su ogni singolo elemento si rimanda alla documentazione online.

```
1  <?xml version="1.0" encoding="utf-8"?>
2
3  <manifest>
4
5      <uses-permission />
6      <permission />
7      <permission-tree />
8      <permission-group />
9      <instrumentation />
10     <uses-sdk />
11     <uses-configuration />
12     <uses-feature />
13     <supports-screens />
14     <compatible-screens />
15     <supports-gl-texture />
16
17     <application>
18
19         <activity>
20             <intent-filter>
21                 <action />
22                 <category />
23                 <data />
24             </intent-filter>
25             <meta-data />
26         </activity>
27
28         <activity-alias>
29             <intent-filter> . . . </intent-filter>
30             <meta-data />
```

```
31          </activity-alias>
32
33          <service>
34              <intent-filter> . . . </intent-filter>
35              <meta-data/>
36          </service>
37
38          <receiver>
39              <intent-filter> . . . </intent-filter>
40              <meta-data />
41          </receiver>
42
43          <provider>
44              <grant-uri-permission />
45              <meta-data />
46              <path-permission />
47          </provider>
48
49          <uses-library />
50
51      </application>
52
53  </manifest>
```

In pratica, gli unici elementi che possono essere inseriti nel file Manifest sono (in ordine alfabetico):

- <action>
- <activity>
- <activity-alias>
- <application>
- <category>
- <data>
- <grant-uri-permission>
- <instrumentation>
- <intent-filter>
- <manifest>
- <meta-data>
- <permission>

- <permission-group>
- <permission-tree>
- <provider>
- <receiver>
- <service>
- <supports-screens>
- <uses-configuration>
- <uses-feature>
- <uses-library>
- <uses-permission>
- <uses-sdk>

Per un'eventuale manipolazione del file Manifest è utile tenere presente alcune regole:

- **Elementi.** I <manifest> e gli <application> sono obbligatori e devono essere presenti una sola volta, mentre tutti gli altri elementi possono comparire zero, una o più volte.
  Tutti i valori degli elementi sono impostati tramite gli attributi.
  Gli elementi possono essere inseriti nel manifest in qualsiasi ordine, eccezione fatta per l'<activity-alias> che deve per forza seguire l'<activity> (di cui è l'alias).

- **Attributi.** In pratica sono tutti opzionali ma per alcuni elementi l'assenza di attributi può rendere l'elemento inutile. E' comunque sempre consigliato leggere la documentazione a seconda delle esigenze.
  In alcuni casi l'assenza di attributi viene assunta implicitamente dal compilatore come la presenza di quelli di default.
  Tranne per quelli del <manifest>, la sintassi di tutti gli altri attributi comincia col prefisso ANDROID: (ad esempio android:name=...).

- **Dichiarare i nomi delle classi.** La maggior parte degli elementi hanno una corrispondenza diretta con un oggetto di Java come gli stessi <application> e i componenti principali, cioè <activity>, <service>, <receiver>, <provider>.
  Se si definisce una subclass bisogna utilizzare l'attributo NAME includendo la designazione completa del package. L'esempio per un Service:

```
<manifest . . . >
    <application . . . >
        <service android:name=”com.example.project.SecretService” . . . >
            . . .
        </service>
        . . .
    </application>
</manifest>
```

Quando si avvia un componente, il sistema crea un istanza della specifica subclass (se la subclass non è specificata, crea un'istanza della classe base).

- **Valori multipli.** Nel caso di valori multipli, la regola dice di replicare l'elemento anziche i valori. Ad esempio per un intent filter:

```
<intent−filter . . . >
    <action android:name=”android.intent.action.EDIT” />
    <action android:name=”android.intent.action.INSERT” />
    <action android:name=”android.intent.action.DELETE” />
    . . .
</intent−filter>
```

- **Valori di tipo risorsa.** Alcuni attributi possono avere dei valori che saranno poi visibili agli utenti. Questi valori devono essere settati da una risorsa o da un tema.
  Per le risorse il formato è:

```
@[package:]type:name
```

dove il package può essere omesso se la risorsa si trova nello stesso package dell'applicazione, il type rappresenta il tipo di risorsa (es. stringa o immagine) e name la identifica. Esempio:

```
<activity android:icon=”@drawable/smallPic” . . . >
```

Per i temi la sintassi è analoga ma al posto del '@' si utilizza '?':

```
?[package:]type:name
```

- **Valori di tipo stringa.** Se l'attributo necessita di una striga come valore si usa il doppio backslash ('\\') per i caratteri di escape - esempio: '\\$n$' per andare a capo o '\\$uxxxx$' per un carattere Unicode.

2.6 Google Play Store

Google Play è l'applicazione di prima scelta per la distribuzione di app Android. Quando si pubblica su Google Play, il proprio lavoro viene messo a disposizione di enorme numero di clienti, in più di 190 paesi di tutto il mondo.

Figura 2.15: Google Play Store

Il Play Store è un elemento centrale dell'esperienza Android. Ogni utente può personalizzare il proprio dispositivo installando applicazioni, giochi e molti altri contenuti.
Esiste una grandissima scelta tra app sia gratuite sia a pagamento e scaricarle è estremamente semplice e conveniente se si possiede un account Google.
I potenziali utenti, prima di scaricare/acquistare un'app, guardano valutazioni e recensioni come parametri chiave della qualità del prodotto.

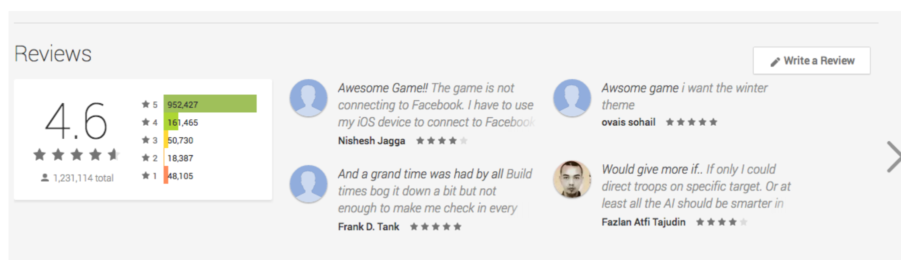

Figura 2.16: Valutazioni e recensioni su un'applicazione

Una volta che l'app è stata ultimata, la si può pubblicare su Google Play Store registrandosi presso la **Google Play Developer Console**, accessibile tramite l'indirizzo https://play.google.com/apps/publish/signup/.

Capitolo 3

Il framework dei sensori

Sui dispositivi Android sono installati una serie di sensori che gli sviluppatori possono utilizzare allo scopo di permettere l'interazione tra le applicazioni e l'ambiente circostante. L'accesso ai sensori non è diretto: sono state infatti messe a disposizioni dei programmatori delle API (Application Programming Interface) per ottenerne i dati. Queste Api non hanno il totale accesso al sensore fisico ma offrono la possibilità di ricevere una notifica quando il sensore cambia i suoi valori. Android, tramite il framework, offre diverse classi e metodi per utilizzare la sensoristica disponibile. Tali concetti saranno affrontati in questo capitolo.

3.1 La classe Sensor Manager

La classe *SensorManager* è contenuta nel package *android.hardware* e consente l'accesso ai sensori disponibili sulla piattaforma Android.
SensorManager gestisce TUTTI i tipi di sensori Android, a differenza di alcuni dispositivi che non dispongono di alcuni, quindi, per questi dispositivi una parte di SensorManager risulterà inutile.
La classe definisce anche diverse costanti che rappresentano caratteristiche specifiche dei sensori, tra cui:

- **Il tipo di sensore**: definisce tutti i tipi di sensori (accelerometro, giroscopio, sensore luminoso, di prossimità, di campo magnetico, ecc...)
- **Frequenza di campionamento**: sono definite delle frequenze di campionamento di default, le quali sono da interpretare come convenzioni e sono 0 μs (assenza di ritardo), 20000 μs (per i giochi), 60000 μs (per l'interfaccia utente) e 200000 μs (frequenza di campionamento normale).
- **Precisione**: su una scala di quattro livelli: alta, media, bassa e inaffidabile.

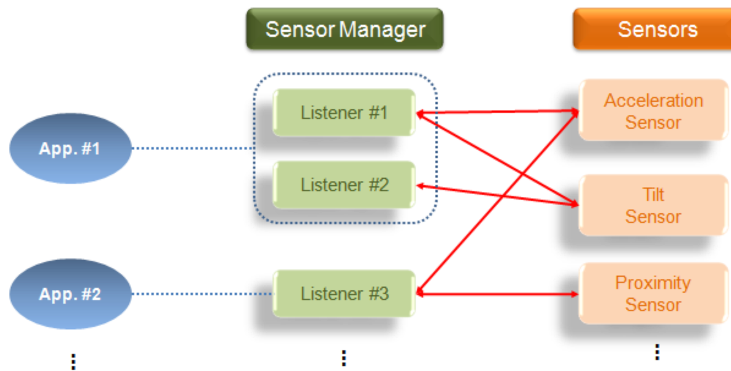

Figura 3.1: Interazione tra applicazione e sensore fisico tramite le classe SensorManager

3.2 Tipi di sensori

Come già accennato nel capitolo 1, il framework Android consente l'accesso a molti tipi di sensori, alcuni di questi sono basati su hardware (sensori reali) e alcuni sono sono basati sul software (sensori virtuali). La tabella successiva riassume i sensori che sono supportati dalla piattaforma Android.
Pochi dispositivi Android possiedono tutti i tipi di sensori. La maggior parte dei dispositivi cellulari e tablet hanno un accelerometro e un magnetometro, altri devices hanno anche hanno barometri e termometri. Inoltre, un dispositivo può avere più varianti di uno stesso sensore, come ad esempio due sensori di gravità ma con una frequenza di misura diversa.

Sensore	**Tipo**	**Descrizione**	**Possibili applicazioni**
TYPE_ACCELEROMETER	Accelerometro (hardware)	Misura l'accelerazione causata da una forza esterna (considerando la forza di gravità) applicata ad un dispositivo, in m/s^2, sui 3 assi cartesiani (x, y, z)	Rilevamento di movimenti (vibrazioni, inclinazione, ecc...)
TYPE_AMBIENT _TEMPERATURE	Termometro (hardware)	Misura la temperatura dell'ambiente in gradi Celsius (° C)	Monitoraggio temperatura dell'aria

TYPE_GRAVITY	Gravimetro (hardware e/o software)	Misura la forza di gravità in m/s^2 che viene applicata ad un dispositivo sui 3 assi cartesiani (x, y, z)	Rilevamento della posizione del dispositivo rispetto alla direzione della forza di gravità
TYPE_GYROSCOPE	Giroscopio (hardware)	Misura la velocità angolare di un dispositivo in rad/s attorno ai 3 assi cartesiani (x, y, z)	Rilevamento di rotazione
TYPE_LIGHT	Fotometro (hardware)	Misura il livello di luce ambientale (illuminazione) in lux	Adattare la luminosità dello schermo
TYPE_LINEAR _ACCELERATION	Accelerometro e Gravimetro (Software)	Misura l'accelerazione causata da una forza esterna (escludendo la forza di gravità) applicata ad un dispositivo, in m/s^2, sui 3 assi cartesiani (x, y, z)	Rilevamento di movimenti (vibrazioni, inclinazione, ecc...)
TYPE_MAGNETIC _FIELD	Magnetometro (hardware)	Misura del campo magnetico esterno sui 3 assi cartesiani (x, y, z) in microtesla (uT)	Simulazione di una bussola
TYPE_ORIENTATION	Gravimetro e Magnetometro (software)	Al livello API 3 si può ottenere la matrice di inclinazione e la matrice di rotazione per un dispositivo mediante il sensore di gravità e il sensore di campo magnetico in combinazione con il metodo getRotationMatrix ()	Determinazione della posizione del dispositivo
TYPE_PRESSURE	Barometro (hardware)	Misura la pressione atmosferica in hPa o mbar	Monitoraggio variazioni della pressione dell'aria

TYPE_PROXIMITY	Sensore di Prossimità (hardware)	Misura la distanza tra un oggetto, posto davanti al display, e il display stesso, in cm	Determinare se il cellulare è tenuto accostato all'orecchio dell'interlocutore
TYPE_RELATIVE _HUMIDITY	Igrometro (hardware)	Misura l'umidità relativa dell'ambiente in percentuale (%)	Monitoraggio del punto di rugiada relativo e dell'umidità
TYPE_ROTATION _VECTOR	Giroscopio (software e/o hardware)	Misura l'orientamento di un dispositivo fornendo i tre elementi del vettore di rotazione	Rilevamento dei movimenti e delle rotazioni
TYPE_TEMPERATURE	Termometro (hardware)	Misura la temperatura del dispositivo in gradi Celsius (°C). Recentemente è stato sostituito con il sensore TYPE _AMBIENT _TEMPERATURE a livello di API 14	Monitoraggio della temperatura ambientale

3.3 Funzionalità del framework

Anche il framework Android è parte del package *android.hardware*. Questo sottosistema comprende l'interfaccia tra il driver hardware e le altre classi del framework (sensor Hardware Abstraction Layer o sensor HAL). Inoltre possiede le interfacce che permettono ai programmatori di:

- **Individuare i sensori e le loro proprietà.** Ciò significa che le applicazioni che hanno bisogno di particolari sensori identificano, prima di tutto, quelli effettivamente presenti sul dispositivo. In base ai risultati, disabilitano le funzionalità che sfruttano sensori non presenti.

- **Monitorare eventi per l'acquisizione di dati grezzi dai sensori.** Ogni n nanosecondi, se un sensore rileva un cambiamento di qualche valore notifica il tutto inviando un "evento" con 4 tipi di informazioni: nome sensore, timestamp dell'istante in cui è avvenuto l'evento, precisione dell'evento e il nuovo valore.

Queste due operazioni possono essere svolte usando classi e interfacce come:

- *SensorManager*. Classe che crea un istanza del servizio e fornisce metodi per accedere, calibrare e monitorare i sensori, registrando e deregistrando i listeners. Definisce anche delle costanti per determinare la frequenza di acquisizione dati.
- *Sensor*. Classe che crea un istanza del sensore specifico, fornendo metodi per determinarne le proprietà.
- *SensorEvent*. Classe che crea un evento contenente le 4 informazioni citate in precedenza.
- *SensorEventListener*. Interfaccia che definisce due metodi di callback, i quali vengono chiamati quando cambiano i valori o l'accuratezza di un sensore.

3.4 Interpretazione dei dati

Il framework utilizza un sistema di coordinate a 3 assi per esprimere i valori provenienti dai sensori.
Le coordinate X, Y e Z sono rappresentate rispettivamente dagli indici 0, 1 e 2 dell'array "values" presente nell'oggetto *SensorEvent*. Questo vale, ad esempio per sensori quali accelerometri e giroscopi mentre altri, come il sensore di prossimità, di luminosità, di pressione atmosferica forniscono singoli valori rappresentati dal solo values[0], rendendo insignificanti quelli di indice 1 e 2. Per *TYPE_ACCELEROMETER, TYPE_GRAVITY, TYPE_GYROSCOPE, TYPE_LINEAR_ACCELERATION e TYPE_MAGNETIC_FIELD* il sistema di coordinate è definito relativamente alla posizione del dispositivo nel suo orientamento predefinito: l'asse X ha direzione orizzontale e punta a destra, l'asse Y ha direzione verticale e punta verso l'alto ed il verso dell'asse Z esce dal display con direzione perpendicolare.
Gli assi non vengono scambiati quando l'orientamento del display del dispositivo cambia.

3.5 Disponibilità di sensori in un dispositivo

Il framework Android fornisce diversi metodi che rendono semplice determinare a runtime quali sensori si trovano su un dispositivo e le proprietà di ciascuno, come ad esempio la portata massima, la risoluzione, i requisiti di alimentazione, la frequenza minima e il fornitore. Prima di tutto, per identificare i sensori, è necessario ottenere un riferimento al servizio, creando un'istanza della classe *SensorManager* con il metodo *getSystemService()* utilizzando *SENSOR_SERVICE* come parametro:

```
1 private SensorManager sensorManager;
2 sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

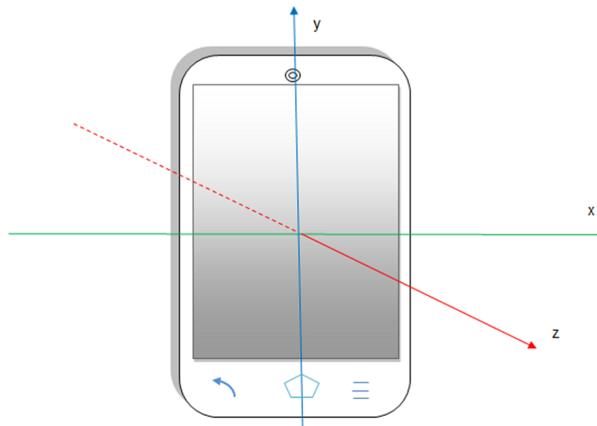

Figura 3.2: L'orientamento degli assi in uno smartphone

Ora, chiamando il metodo *getSensorList()* e utilizzando la costante *TYPE_ALL* come parametro, *SensorManager* ritorna la lista contenente ogni sensore presente nel dispositivo:

```
List<Sensor> availableSensors = sensorManager.getSensorList(Sensor.TYPE_ALL);
```

Utilizzando, invece della costante *TYPE_ALL*, altre costanti della classe *Sensor* quali *TYPE_GYROSCOPE*, *TYPE_LINEAR_ACCELERATION*, *TYPE_ GRAVITY*, ecc.., *SensorManager* restituirà la lista di tutti i sensori presenti che corrispondono a quel tipo.
È anche possibile determinare se uno specifico tipo di sensore esiste su un dispositivo mediante il metodo *getDefaultSensor()* con il tipo come parametro. Se vengono trovati più di un sensore di quel tipo, uno di essi deve essere stato progettato per essere quello di default. Se il sensore di default non esiste, il metodo restituisce null (il che significa che il dispositivo non ne dispone). Il seguente codice controlla se è presente un magnetometro:

```
private SensorManager sensorManager;
mSensorManager = (SensorManager) getSystemService( Context.SENSOR_SERVICE);

if(sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD) != null){
        //Routine da eseguire in caso di disponibilità del magnetometro
}
else{
        //Routine da eseguire in caso di assenza del magnetometro
}
```

Oltre a elencare i sensori su un dispositivo, è possibile, sempre utilizzando i metodi pubblici della classe *Sensor*, determinare proprietà e attributi di ciascuno. Tutto ciò è fondamentale se un'applicazione deve avere un comportamento diverso a seconda delle caratteristiche dei sensori disponibili. Esistono metodi per ottenere la risoluzione e la massima portata di misura (*getResolution()* e *getMaximumRange()*) o il fabbisogno di potenza (con *getPower()*). Altri due metodi particolarmente utili sono *getVendor()* (per sapere il produttore) e *getVersion()* (per ottenere la versione). Ad esempio: per ottimizzare un'applicazione che utilizza la versione 3 di un sensore di gravità fornito da Google Inc., prima si controlla se quel particolare sensore è presente sul dispositivo; se non lo è, l'applicazione prova a ripiegare sull'accelerometro.
getMinDelay() restituisce l'intervallo di tempo minimo, in microsecondi, tra due dati rilevati. Ogni sensore che restituisce un valore diverso da zero è detto "streaming sensor" - introdotti da Android 2.3 Gingerbread (API Level 9) in poi - mentre se restituisce zero, significa che non è streaming e riporta i dati solo quando vi è un cambiamento effettivo. Questo metodo consente di determinare la massima velocità con cui si è in grado di acquisire i dati.

3.6 Monitorare gli eventi

L'interfaccia *SensorEventListener* introduce due metodi di callback, che devono essere implementati per monitorare l'emissione di dati da parte del sensore. Questi metodi sono *onAccuracyChanged()* e *onSensorChanged()*.
Il primo viene invocato quando la precisione cambia. I parametri sono il riferimento all'oggetto *Sensor* che ha subito la modifica e il suo nuovo livello accuratezza. La precisione è rappresentata tramite una delle quattro costanti definite nella classe *SensorManager*:
- *SENSOR_STATUS_ACCURACY_LOW*
- *SENSOR_STATUS_ACCURACY_MEDIUM*
- *SENSOR_STATUS_ACCURACY_HIGH*
- *SENSOR_STATUS_ACCURACY_UNRELIABLE*

Il metodo *onSensorChanged()*, invece, viene richiamato quando un sensore deve segnalare un nuovo valore. Ciò viene fatto tramite un oggetto *SensorEvent* che contiene 4 informazioni: il tipo di sensore che ha generato i nuovi dati, il valore dei nuovi dati, la precisione e il timestamp dell'istante in cui l'evento è accaduto.

```
1 private class SensorEventsListener implements SensorEventListener{
2
3         @Override
4         public void onSensorChanged(SensorEvent event){
5                 // Routine che, ad esempio, stampa i valori X, Y e X
6                 textX.setText(""+event.values[0]);
```

```
7                 textY.setText(""+event.values[1]);
8                 textZ.setText(""+event.values[2]);
9 }
10
11         @Override
12         public void onAccuracyChanged(Sensor sensor, int accuracy){
13                 // Routine che fa qualcosa quando cambia l'accuratezza
14         }
15 }
```

3.7 Documentazione

Per ulteriori approfondimenti riguardo a tutte classi e le interfacce descritte fin'ora, si rimanda il lettore alla documentazione online attraverso i seguenti link:

- http://developer.android.com/reference/android/hardware/Sensor.html
- http://developer.android.com/reference/android/hardware/SensorManager.html
- http://developer.android.com/reference/android/hardware/SensorEvent.html
- http://developer.android.com/reference/android/hardware/SensorEventListener.html

Capitolo 4

Sensor Fusion

4.1 Gli errori nelle misurazioni

Le misure effettuate dai sensori non sono sempre perfettamente affidabili. La causa di ciò può essere attribuita all'esistenza di disturbi ambientali o dal lento deterioramento dell'hardware nel tempo. Esistono, però, tecniche e algoritmi utili per affrontare queste problematiche e che consentono di ridurre tali errori.
Prima di vedere questi algoritmi e tecniche, occorre sapere quali sono le tipologie di errori che possono essere riscontrati.
Un'ulteriore parentesi va aperta per quanto riguarda la misura della accuratezza e della precisione di un sensore. Per far ciò sono necessari due numeri: il valore effettivo che il sensore sta cercando, o dovrebbe, misurare (es. campo magnetico, umidità, ecc...) e il valore che il sensore sta effettivamente misurando. Ad esempio se avessimo a disposizione un termometro molto affidabile e preciso potremmo usarlo per misurare la temperatura dell'ambiente e confrontare il valore ottenuto (che assumiamo come privo di qualsiasi errore) con quello rilevato dal sensore.
L'accuratezza del sensore si valuta proprio calcolando la differenza tra questi due valori. Più tale differenza tende a zero, più il sensore ha un alto grado di accuratezza.
L'alta precisione, invece, si ha quando tutte le misure sono strettamente raggruppate attorno ad un valore particolare, che non è necessariamente quello reale.

I tipi di errori che possono essere riscontrati durante le misurazioni sono:

- **Errori umani.** Sono errori di distrazione commessi dall'utente che effettua la misura (es. lettura errata di un valore da un grafico) e ovviamente non esistono algoritmi per evitarli. Occorre solo maggior attenzione.

- **Errori sistematici.** Riguardano l'accuratezza della misurazione e hanno uno scostamento costante dal valore reale (es. misurazioni di un magnetometro con presenza di un magnete nelle immediate vicinanze). Possono essere preveduti e rimossi con calibrazioni del sensore oppure modificando direttamente le misurazioni.

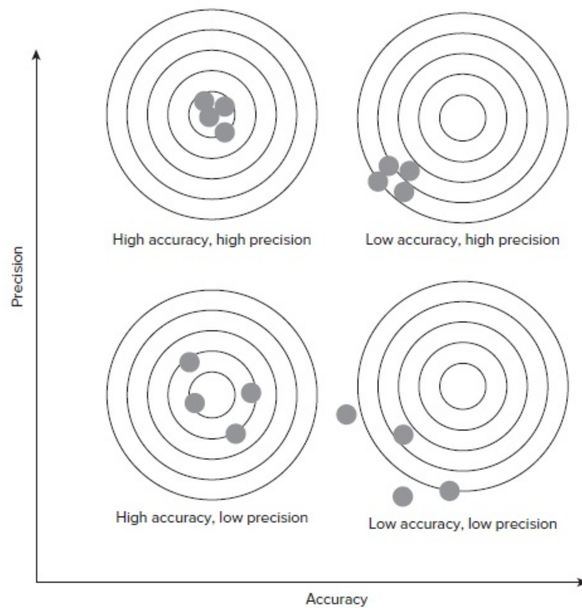

Figura 4.1: Diverse condizioni di accuratezza/precisione

- **Errori casuali.** Come gli errori sistematici, anche questi sono causati da disturbi esterni che interferiscono durante fase di misurazione, ma, essendo molto più imprevedibili, le tecniche sopracitate non saranno efficaci.
- **Rumori.** Provocano una fluttuazione casuale del valore misurato e si dividono in rumori bianchi, marroni, ecc... Anche in questo caso esistono tecniche per limitarli (es. filtri passa-basso) ma nessuna di esse riguarda la programmazione del software. Per questo motivo non verranno trattati in questa tesi.
- **Deriva.** Consiste in una lenta divergenza, dei valori misurati, dal valore reale, che aumenta sempre di più col tempo. Sono causati dal deterioramento del sensore o da un'eventuale integrazione dei dati. Quest'ultima può causare un piccolo errore che si somma ad ogni iterazione dell'integrazione.
- **Zero offset (o "offset" o "bias").** Si ha un Zero offset quando il valore di uscita del sensore è inizialmente diverso da zero (mentre in realtà dovrebbe essere proprio zero). Ad esempio, se un accelerometro fermo su una superficie orizzontale non produce esattamente i valori (0; 0; -9.80665 $\frac{m}{s^2}$) allora l'accelerometro è affetto da un offset. Allo stesso modo, se un giroscopio nella sua fase stazionaria non misura un angolo (0; 0; 0) $\frac{rad}{s}$, l'offset che ne deriva può influenzare le successive integrazioni dell'angolo.

- **Ritardi di tempo e dati scartati.** Android non è un sistema operativo real-time quindi eventuali ritardi nella elaborazione e trasmissione dei dati possono generare timestamp errati. Inoltre, alcuni dati potrebbero venire scartati quando il dispositivo si trova in un "busy state". La soluzione a tutto ciò esiste ma non verrà trattata poichè questi tipi di errori non sono particolarmente di intralcio ai programmatori Android.

- **Integrazione dei dati.** La funzione principale del giroscopio è ricavare la velocità angolare del dispositivo. In certi casi, però, può essere utile calcolare anche la variazione angolare. L'unica soluzione è l'integrazione dei valori misurati, ma come sappiamo già, l'integrazione genera errori di deriva e zero offset, i quali, svilupperanno nel tempo una divergenza tale che le successive rilevazioni produrranno valori numerici enormi senza alcun significato fisico. Per risolvere il problema si ricorre ad un Sensor Fusion con l'accelerometro. Una delle aziende che ha sviluppato l'algoritmo è la Kionix Inc. che verrà trattata nella prossima sezione.

Il significato matematico di integrazione dovrebbe essere ben chiaro al lettore, ma il compito del programmatore è quello di tradurre il concetto matematico in linguaggio informatico. Di seguito si riporta il codice che consente di effettuare l'integrazione dei dati di un giroscopio, allo scopo di determinare la rotazione angolare, in radianti, del dispositivo (vedi commenti al codice):

```
1  //NS2S converte nanosecondi in secondi
2  private static final float NS2S = 1.0f / 1000000000.0f;
3  private float timestamp;
4
5  public void onSensorChanged(SensorEvent event)
6  {
7          //event.values: velocità angolare al secondo (sugli assi x, y e z)
8          float[] valuesClone = event.values.clone();
9          if (timestamp != 0)
10         {
11                 //dT: intervallo di tempo tra due letture consecutive del sensore (in secondi)
12                 final float dT = (event.timestamp − timestamp) ∗ NS2S;
13                 //...da velocità angolare a variazione angolare...
14                 angle[0] += valuesClone[0] ∗ dT; //asse x
15                 angle[1] += valuesClone[1] ∗ dT; //asse y
16                 angle[2] += valuesClone[2] ∗ dT; //asse z
17         }
18         timestamp = event.timestamp;
19 }
```

Fin'ora sono state elencate e spigate tutte le possibili problematiche che si possono incontrare in generale nell'ambito della programmazione basata sulla sensoristica hardware.

Le tecniche che consentono di risolvere, o limitare, questi problemi sono essenzialmente tre; due che saranno citate spiegate brevemente per completezza di informazione, mentre la terza, la Sensor Fusion, è per l'appunto l'argomento principale di questo capitolo:

- **Re-Zeroing.** Se l'uscita di un sensore è affetta da un offset dovuto a disturbi ambientali, può essere utile effettuare l'azzeramento del sensore.
  Per capire meglio, immaginate di dover pesare un liquido su una bilancia. Inizialmente si appoggia sul piatto il contenitore vuoto, se ne legge il peso, si effettua l'azzeramento del display, si riempie il contenitore con il liquido e si legge il peso netto.
  In generale, questa calibrazione può essere effettuata fisicamente dall'utente senza bisogno di dover mettere mano al codice.

- **Filtri.** I filtri possono essere di vari tipi.
  I filtri passa-basso permettono solo il passaggio di certe frequenze sotto una determinata soglia, detta frequenza di taglio, filtrando qualsiasi rumore ad alta frequenza.
  I filtri passa-alto, invece, eliminano lente derive o offset dei dati, favorendo le variazioni ad alta frequenza (superiori a quella di taglio).
  I filtri passa-banda consistono in un approccio misto tra il passa-basso e il passa-alto, ovvero, rifiutano sia basse frequenze che alte frequenze (in pratica si hanno due frequenze di taglio) mantenendo i dati entro un range prestabilito.

- **Sensor Fusion.** Questa tecnica importantissima sfrutta la sinergia tra sensori in modo da mitigare i punti deboli di ognuno.

4.2 Introduzione al Sensor Fusion

Sensor Fusion è una tecnica che consiste nel combinare più sensori per ottenere migliore precisione, prestazioni e efficienza.
Prelevando i dati provenienti dai più sensori si riescono a correggere le carenze dei singoli e si possono calcolare la posizione e l'orientamento esatti del dispositivo.
Ad esempio, l'accelerometro risponde velocemente ai cambiamenti ma è molto sensibile ai rumori. Filtrare i dati dell'accelerometro può risolvere il problema del rumore ma introdurrebbe un ritardo nella risposta.
Le misure del giroscopio, invece, vengono integrate nel tempo. Il significato di 'integrate nel tempo' è strettamente matematico ovvero si calcola la variazione di un angolo infinitesimo in un tempo infinitesimo allo scopo di ricavare un'approssimazione della velocità angolare. E' proprio da questa approssimazione che nasce l'errore (detto deriva) perchè si presuppone continua una misurazione che è in realtà digitale, con una certa frequenza. La misura dell'angolo viene quindi fornita con un basso livello di rumore ma col tempo la deriva aumenta sempre più e

ha come conseguenza che l'integrazione di tali dati rischia di diventare unphysical (ovvero che non corrisponde all'orientamento reale del dispositivo).
Pertanto, un Sensor Fusion System può risolvere questi problemi leggendo i dati integrati del giroscopio e limitarne la deriva confrontandoli costantemente coi dati dell'accelerometro (non soggetti a deriva).

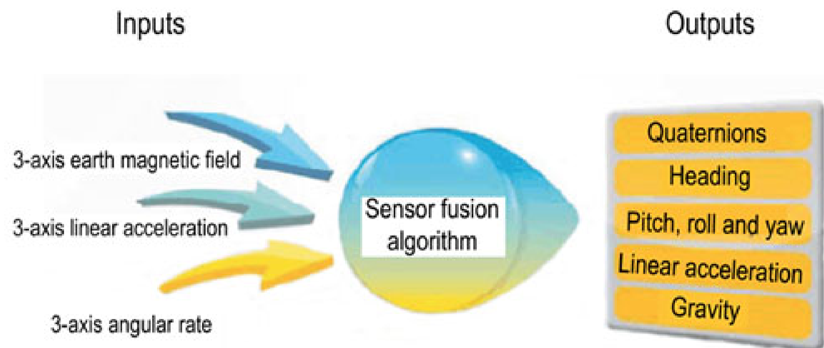

Figura 4.2: Sensor fusion: input e output

Oggigiorno le aziende del settore investono molto sul Sensor Fusion. Invensense è un'azienda produttrice di accelerometri e giroscopi ed è proprietaria dell'algoritmo ROTATION_VECTOR ottenuto proprio dall'integrazione tra questi due sensori. Molti dispositivi, come il Samsung Galaxy Tab 10.1, l'HTC Sensation, l'EVO 3D e il Galaxy Nexus, funzionano utilizzando proprio l'algoritmo dell'Invensense. Nel caso il lettore fosse interessato a provare questo algoritmo su uno di questi device, si consiglia prima di tutto di verificare che il produttore del giroscopio sia effettivamente l'Invensense, tramite il metodo *Sensor.getVendor()*. Appurato ciò, è molto probabile che l'algoritmo sia implementato e disponibile per eventuali test e/o applicazioni.

La tecnica Sensor Fusion è il futuro delle applicazioni basate sulla sensoristica ed è destinata ad essere perfezionata sempre di più nel tempo. Negli ultimi tempi, infatti, sono sempre più in disuso gli algoritmi che si interfacciano direttamente col sensore fisico.

4.3 Kionix Software Solution

Kionix Inc. è un produttore di MEMS (Micro Electro-Mechanical Systems) con sede a Ithaca, NY, USA. Kionix offre accelerometri a bassa potenza, giroscopi e sensori combinati a 6 assi garantendo alte prestazioni e mettendo a disposizione

una serie di librerie complete che supportano una vasta gamma di tipologie di Sensor Fusion.

Figura 4.3: Kionix logo

Negli ultimi anni, i sensori MEMS sono diventati elementi costitutivi di cruciale importanza all'interno dei dispositivi moderni. In particolare i dispositivi mobili supportano tutti una serie di MEMS che consentono alle applicazioni di rilevare la rotazione dello schermo, riconoscere i gesti, conteggiare i passi, ecc...
Il successo della Kionix è dovuto proprio alla valorizzazione dei prodotti sfruttando il Sensor Fusion.

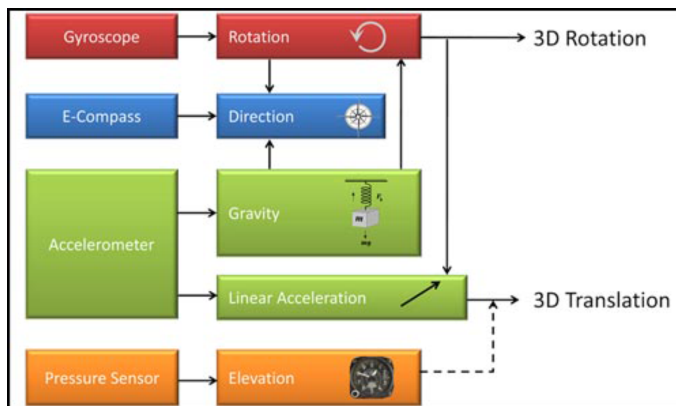

Figura 4.4: Sensor Fusion schema

La soluzione Kionix Sensor Fusion è estremamente flessibile e offre sia una configurazione hardware, dove il software gira su un microcontrollore integrato, sia una configurazione software, che invece sfrutta l'Application Processor. Tutto ciò offre una scalabilità senza pari per quanto riguarda le implementazioni multisensore e, proprio per questo, Kionix Sensor Fusion supporta varie combinazioni di diversi tipi di MEMS (accelerometro-magnetometro, accelerometro-giroscopio, accelerometro-magnetometro-giroscopio).
La scalabilità della Kionix Sensor Fusion si ha anche nell'ottica dei Sistemi Ope-

rativi, dai microcontrollori RTOS a virgola fissa a quelli a virgola mobile fino ai sistemi mobile a 32-bit come Android e Windows 8.
Potenza e prestazioni sono garantite grazie a sofisticate tecniche power-management che aiutano i progettisti a gestire l'interazione coi sensori e l'elaborazione dei dati con un minimo overhead.
KSF supporta anche software sviluppato da terzi per l'elaborazione dei movimenti e altre applicazioni avanzate, fornendo codice sorgente completo ai clienti qualificati, così come l'assistenza diretta, i BPS (board support packages) e, in futuro, circuiti integrati a radiofrequenze.
I progettisti che utilizzano i prodotti KSF saranno quindi in grado di calibrare, compensare e correggere eventuali errori o anomalie, gestire l'assorbimento di potenza, incrementando la durata della batteria, ed evitare interferenze tra i diversi sensori.

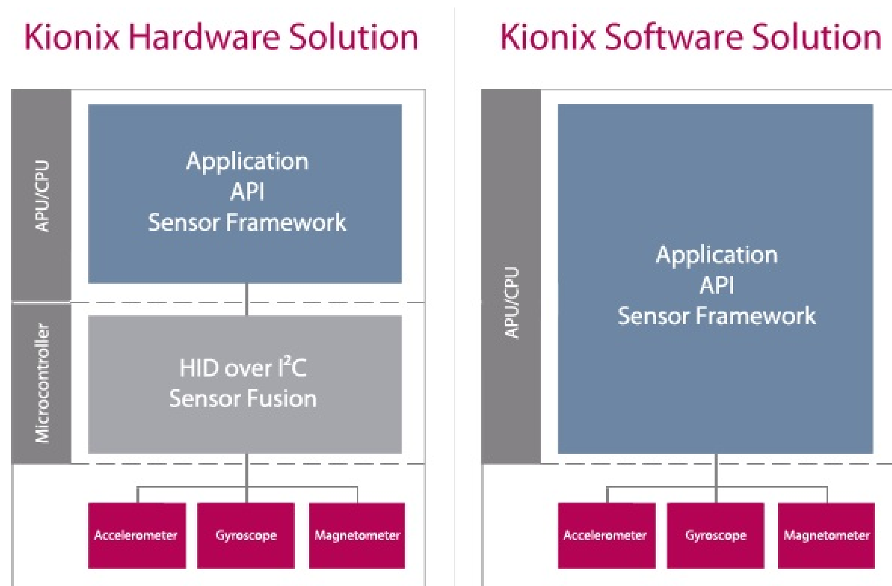

Figura 4.5: La soluzione Kionix Sensor Fusion

Le librerie Kionix e le loro principali funzionalità sono:

- **Accelerometro.** Configurazione minima con MIPS bassi, orientamento del dispositivo (tramite la scatola a 6 lati), contatore dei passi, ecc..

- **Accelerometro/Magnetometro.** Configurazione principale per la maggior parte degli smartphone, magnetometro gravity-compensated, 600 uA di potenza per sensori con 3 MIPS di calcolo, ecc..

- **Accelerometro/Magnetometro/Giroscopio** Configurazione per high-end smartphone, tablets e Win8 slates, alta frequenza di campionamento per migliore precisione, giroscopi immuni da disturbi ambientali, 6 mA di potenza per sensori con 8-10 MIPS di calcolo, ecc...

La Kionix's Android Sensor Library fornisce una implementazione dell'HAL (hardware abstraction layer) dei sensori, definita da Google per sistemi Android. La HAL costituisce il middleware, cioè il collante, tra i driver del kernel (basso livello) e il framework Java/Android (alto livello). Questa libreria implementa il Sensor Fusion combinando gli input dell'accelerometro, del magnetometro e del giroscopio e restituendo in uscita l'orientamento 3D del dispositivo. La libreria include anche meccanismi per l'ottimizzazione della batteria e per fare ciò pone semplicemente i sensori, che non sono momentaneamente utilizzati, in una modalità di basso consumo.
La Kionix Software Solution consiste in una libreria C++ con i relativi header files associati. E' incluso anche un "working sensor HAL implementation" per supportare i sensori fisici e che può essere utilizzato out-of-the-box per fornire ad un dispositivo Android uno specifico orientamento 3D.
L'accesso ai sensori fisici e a quelli virtuali (Sensor Fusion) è garantito grazie a un'interfaccia di API object-oriented. In particolare i sensori virtuali, o sintetici, comprendono quelli che, nel Capitolo precedente, sono stati definiti come: *TYPE_ORIENTATION, TYPE_ROTATION_VECTOR, TYPE_LINEAR_ ACCELERATION* e *TYPE_GRAVITY*. Questa interfaccia, però, non si limita solamente a questi quattro tipi, anzi, la soluzione Kionix può essere anche adattata a sensori luminosi (*TYPE_LIGHT*), sensori di temperatura (*TYPE_TEMPERATURE*) e di pressione (*TYPE_PRESSURE*).
La KSS è compatibile con la gestione eventi di Android e con l'attuale CCD "Android 4.0 Compatibility Definition" che include quanto segue:

- Tipi di sensori supportati:
  - *SENSOR.TYPE_ACCELEROMETER* (fisico, in $\frac{m}{s^2}$)
  - *SENSOR.TYPE_MAGNETIC_FIELD* (fisico, in micro-Tesla)
  - *SENSOR.TYPE_GYROSCOPE* (fisico, in $\frac{rad}{s}$)
  - *SENSOR.TYPE_GRAVITY* (sintetico, in $\frac{m}{s^2}$)
  - *SENSOR.TYPE_LINEAR_ACCELERATION* (sintetico, in $\frac{m}{s^2}$)
  - *SENSOR.TYPE_ROTATION_VECTOR* (sintetico, adimensionale)
  - *SENSOR.TYPE_ORIENTATION* (sintetico, in gradi)
- L'output per tutti i sensori supportati corrisponde al sistema di coordinate definito dall'API SensorEvent.
- Tutti i tipi di sensori (fisici e sintetici) supportano il campo di precisione (Accuracy event field).
- Tutti i tipi di sensori (fisici e sintetici) supportano il campo Timestamp.

  - Per tutti i sensori fisici, l'implementazione principale utilizza il meccanismo degli interrupt, fornendo un Timestamp dell'interrupt molto preciso.
  - L'accelerometro e il giroscopio supportano lo streaming periodico dei dati, che fornisce informazioni utili per convalidare la precisione del Timestamp.

- Dalla sezione 7.3 dell'"Android 4.0 Compatibility Definition":
  - L'implementazione del software fornisce un elenco dei sensori supportati.
  - E' previsto un valore di ritorno, TRUE o FALSE, a seguito di ogni eventuale tentativo di registrazione da parte di un Listener.
  - I SensorListener non vengono notificati se i corrispondenti sensori non sono presenti.
  - Per ottimizzare la gestione energetica, i sensori che non sono registrati in Listener attivi vengono spenti o messi in "sleep mode".

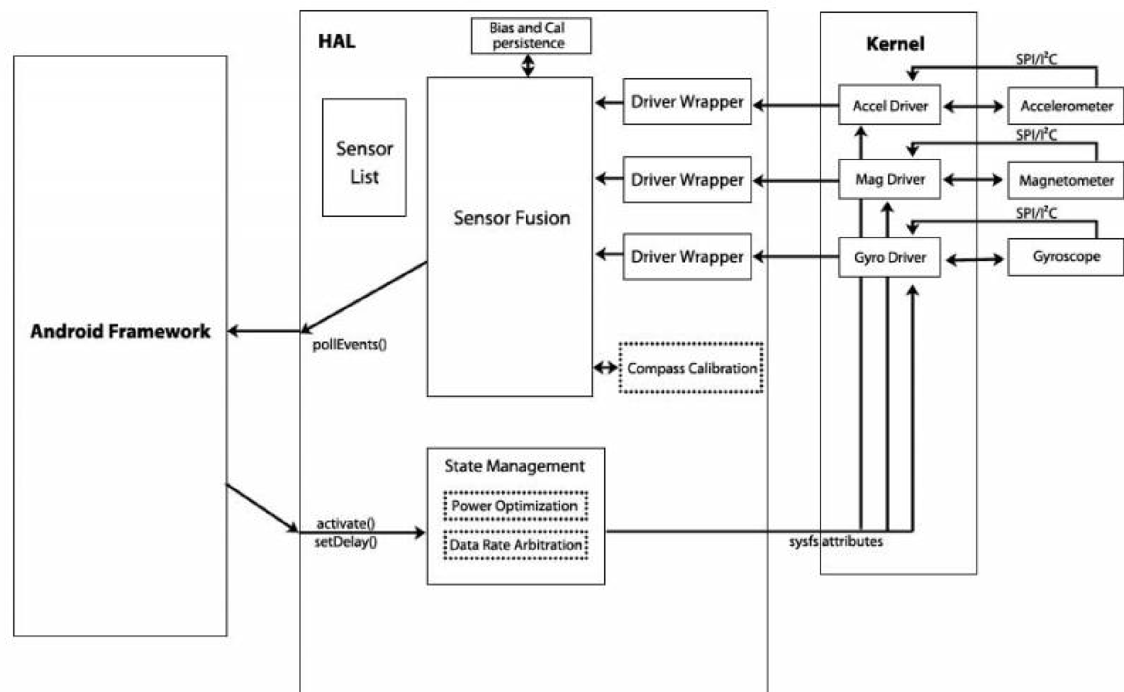

Figura 4.6: Kionix HAL Architecture

Dalla figura precedente di nota che la Kionix's Sensor Library si interfaccia con i kernel drivers relativi all'accelerometro, al giroscopio e al magnetometro. Il funzionamento principale dei kernel driver consiste in un input event driver dotato di attributi sysfs per il controllo (ad esempio, enable, data rate e bias). Questa tipologia di driver è standard per tutti i sensor drivers presenti nel Linux Kernel. I vantaggi nell'utilizzo comprendono minore complessità, alta affidabilità e maggiore trasparenza. Inoltre, se un input event driver non è disponibile per un determinato sensore fisico, la Kionix's Sensor Library può essere estesa per interfacciarsi col nuovo hardware.
Gli unici Integration Models supportati sono di due tipi: AP based (dove lo streaming dei dati grezzi è trattato interamente dall'Application Processor) e HUB based (dove il Sensor Fusion è trattato su un piccolo microprocessore HUB dedicato).

4.4 Determinare l'orientamento di un dispositivo

Tantissime applicazioni necessitano di sapere l'orientamento preciso del dispositivo così come la velocità angolare di rotazione.

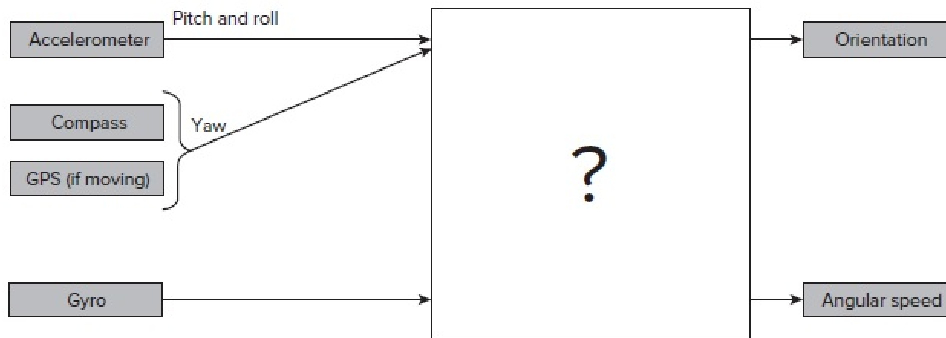

Figura 4.7: Problema: determinare l'orientamento di un dispositivo

La prima soluzione ovvia è quella di mappare le uscite del sensore sintetico per fornire in output proprio queste informazioni. A tal proposito, sarebbe utile disporre di uscite che indichino il pitch (rotazione attorno ad asse trasversale o beccheggio), il roll (rotazione attorno ad asse longitudinale o rollìo) e lo yaw (rotazione attorno ad asse verticale o imbardata).

Ma anche se riuscissimo ad avere a disposizione tutto ciò, il problema non sarebbe comunque risolto perchè i sensori in questione, cioè l'accelerometro e la

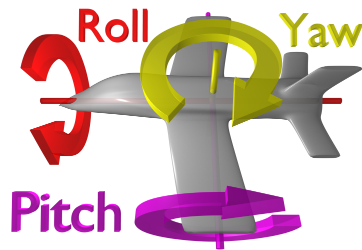

Figura 4.8: Significato di pitch, roll e yaw per un aereo

bussola, sono di per se intrinsicamente rumorosi e fornirebbero dati di scarsa qualità e affidabilità.
Se, peggio ancora, il dispositivo si trovasse su un veicolo in movimento sarebbe necessario e indispensabile anche un GPS da integrare con la bussola.

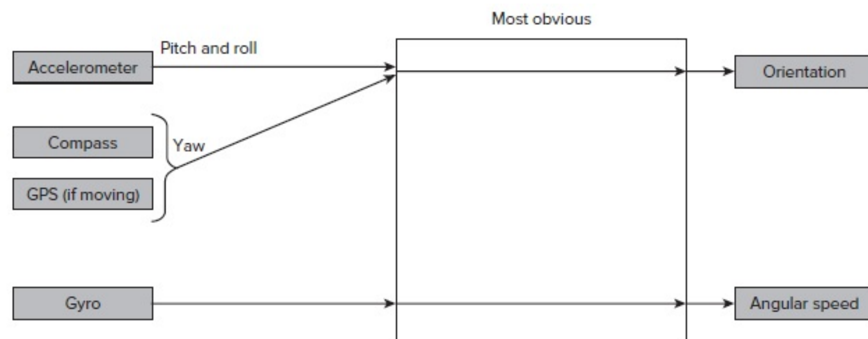

Figura 4.9: Soluzione 1: mappatura I/O

A partire dall'API level 9, fortunatamente, la classe *SensorManager* fornisce il metodo *getOrientation()* e i dispositivi supportano un sensore sintetico detto "Rotation Sensor" (riferito al tipo *Sensor.TYPE_ROTATION*). Il Rotation Sensor, ottenuto grazie ad algoritmi Sensor Fusion, permette di determinare facilmente l'orientamento del dispositivo. Attualmente, però, l'implementazione del Rotation Sensor può differire a seconda della tipologia dell'hardware e del produttore. Se per esempio prendiamo i dispositivi del 2010 che montavano giroscopi (e anche accelerometri) dell'Invensense, come il Galaxy Nexus, il Samsung Galaxy Tab 10.1, e l'HTC EVO 3D, possiamo osservare che il sensore sintetico funziona essenzialmente con l'algoritmo Sensor Fusion della medesima azienda. Lo scopo di questo algoritmo, come già ribadito, è quello di ridurre gli errori causati dai dati integrati del giroscopio facendone il confronto col pitch, roll e yaw forniti dall'accelerometro

e dalla bussola, i quali non sono soggetti a deriva.
Gli algoritmi scritti da aziende come l'Invensense sono detti "proprietary" e purtroppo non sono disponibili per gli sviluppatori. Gli sviluppatori Android possono solo limitarsi ad utilizzare il sintetico *Sensor.TYPE_ROTATION*. Tuttavia, il successo degli algoritmi Invensense e Kionix ha indotto altre aziende a seguirne l'esempio e ciò alimenta sempre di più la speranza che in futuro qualche altro produttore realizzi algoritmi Sensor Fusion simili, rendendoli open source.
Se possibile, è buona norma preferire l'utilizzo del metodo *getOrientation()* o del Rotation Sensor, piuttosto che complicarsi la vita lavorando coi dati grezzi dell'accelerometro e della bussola. Proprio per questo gli algoritmi Sensor Fusion sono destinati a diventare la nuova frontiera della programmazione Android, perchè permettono agli sviluppatori di utilizzare la sensoristica in modo relativamente semplice, senza preoccuparsi della complessità e delle problematiche dell'hardware sottostante.
L'utilizzo di *SensorManager.getOrientation()* crea però altri problemi. Questo metodo necessita di un parametro, ovvero la matrice di rotazione, ottenibile in diversi modi. Si consiglia di evitare l'uso di *SensorManager.getRotationMatrix()* perchè, per un corretto output, l'orientamento dovrebbe essere relativamente statico. L'applicazione solitamente vuole sapere in che direzione sta puntando lo smartphone senza tener conto di eventuali scuotimenti o vibrazioni del dispositivo. Questo metodo, invece, si basa sulla lettura dei dati dell'accelerometro e della bussola che, come si sa, sono influenzati da questi tipi di disturbi.
Siccome la maggior parte degli sviluppatori non hanno la possibilità di utilizzare gli algoritmi Sensor Fusion perchè non sono open source, nasce la necessità di cercare altre strade per determinare l'orientamento (vedi figura successiva).

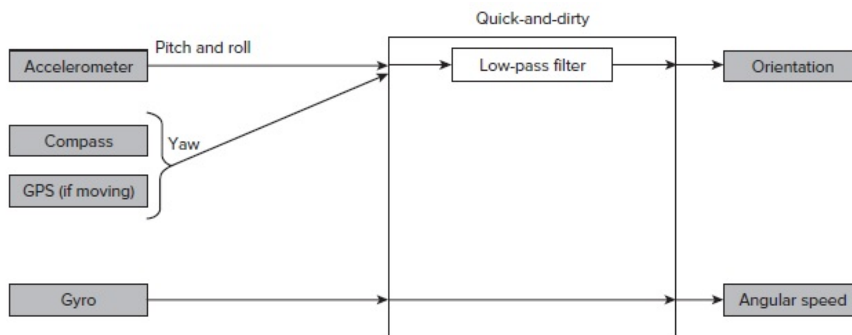

Figura 4.10: Soluzione 2: determinare l'orientamento utilizzando brutalmente un filtro passa-basso sui dati grezzi

Riguardo al giroscopio, invece, teoricamente, conoscendo la posizione iniziale esatta del dispositivo si può calcolare quella finale grazie all'integrazione dei dati.

Nella pratica, però, il giroscopio in fase stazionaria non riesce a leggere esatta-

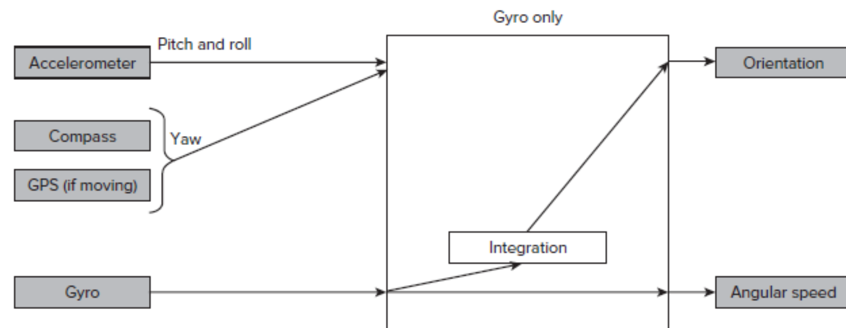

Figura 4.11: Soluzione 3: determinare l'orientamento con l'integrazione dei dati del giroscopio

mente il valore zero e le conseguenze sono ormai note: deriva, divergenza dei dati e risultati inaccettabili.

Arriviamo quindi alla soluzione finale: il Balance Filter o Filtro Complementare. Questa soluzione è stata proposta da Shane Colton al MIT:

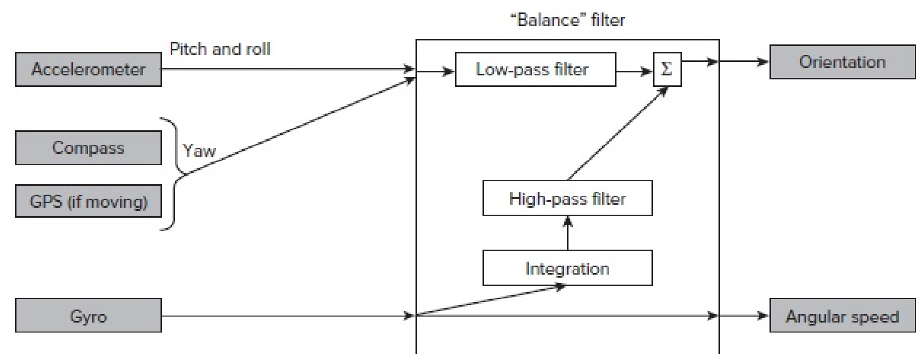

Figura 4.12: Soluzione finale: determinare l'orientamento col Balance Filter

Il Balance Filter integra i dati del giroscopio e ne elimina la deriva grazie ad un filtro passa-alto. I dati in uscita dal filtro passa-alto sono quindi smussati da eventuali derive e vengono uniti a quelli dell'accelerometro/bussola (anch'essi trattati da un filtro) formando un tutt'uno. Il risultato finale è un'uscita che rappresenta una vera e propria stima dell'orientamento del dispositivo.

Nello specifico: immaginiamo di ricevere un aggiornamento della velocità angolare

dal giroscopio (in $\frac{rad}{s}$) e di memorizzarlo nella variabile “gyro”; di ricevere anche una misura dell’ultimo angolo dall’accelerometro e di memorizzarla in “angle_acc”; infine “dt” rappresenta l’intervallo di tempo trascorso dall’ultima misurazione dell’accelerometro.
Con questi dati posso determinare il nuovo angolo:

angle = b * (angle + giro*dt) + (1 - b)*(angle_acc);

Si può inizialmente provare a porre ad esempio b=0.98, per valorizzare maggiormente i dati del giroscopio rispetto a quelli dell’accelerometro. Sarebbe anche utile disporre di un giroscopio veloce nelle misurazione, ovvero avere un basso “dt” per limitare le variazioni dell’angolo di massimo due gradi tra una misurazione e l’altra.
In tutti i caso anche il Balance Filter, sebbene semplice e utile da implementare, non incarna l’approccio ideale di Sensor Fusion.
E’ probabile che l’Invensense riesca a determinare un orientamento di altissima qualità facendo uso di algoritmi intelligenti e di un altro particolare tipo di filtro detto il Kalman Filter (del quale non entreremo nei dettagli).

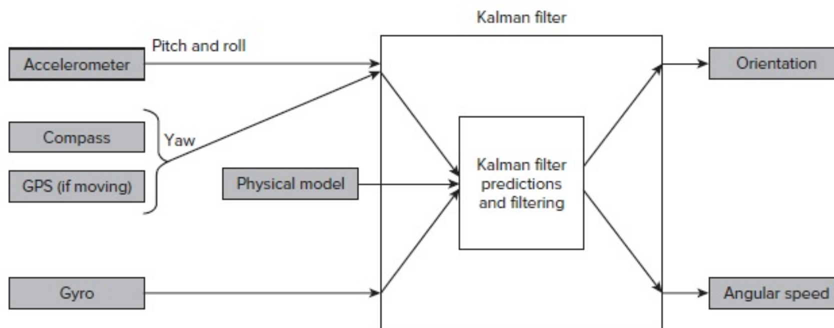

Figura 4.13: Soluzione alternativa: algoritmi intelligenti e Kalman Filter

Capitolo 5

L'applicazione Device Orientation

Come il lettore può aver già intuito dai capitoli precedenti, il problema di determinare l'orientamento di un dispositivo è una delle questioni trattate più frequentemente in questa tesi. Il motivo di ciò è che tantissime applicazioni del Play Store necessitano di questa funzionalità per realizzare molteplici scopi. Alcuni giochi, ad esempio, sfruttano l'inclinazione e la rotazione del dispositivo come forma di interazione tra l'utente e l'applicativo.
I tipi di sensori che consentono di rilevare questo tipo di movimenti sono già stati ampiamente presentati e descritti. Con questo capitolo si passa dalla trattazione teoria di questi argomenti al loro utilizzo pratico. Vi mostreremo come realizzare un'applicazione completa che chiameremo "Device Orientation", partendo dai requisiti e dalle funzionalità, fino ad arrivare allo sviluppo dell'interfaccia utente e alla implementazione del codice sorgente.

5.1 Caratteristiche e funzionalità

Una delle funzionalità principali dell'applicazione è quella di determinare se un dispositivo è orientato verso l'alto o verso il basso. Per ricavare queste informazioni ed eventuali modifiche dell'orientamento, il codice che andremo ad implementare in seguito farà uso di più sensori.
Inoltre, per notificare questi cambiamenti, useremo il Text-To-Speech (TTS) ovvero una classe implementata nel package *android.speech.tts.TextToSpeech* che consente di leggere e sintetizzare un testo scritto allo scopo di produrre un file audio riproducibile.
Tali notifiche saranno anche visualizzate sul display insieme ai valori provenienti dai singoli sensori. Ciò consentirà all'utente di osservare come variano i dati in

uscita dai sensori al variare dell'orientamento dello smartphone.

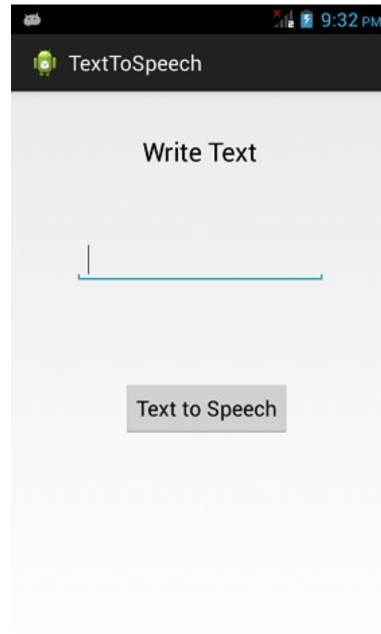

Figura 5.1: Esempio di un'applicazione Text-To-Speech

Inoltre, dato che esistono più tecniche per realizzare queste funzionalità, l'applicazione sarà provvista di una serie di opzioni/pulsanti per scegliere quali specifici sensori saranno coinvolti durante l'esecuzione.
Infine, è stato inserito un pulsante virtuale che consente di disabilitare/abilitare le notifiche audio TTS poichè in certi ambienti potrebbero risultare fastidiose. L'intensità del volume sarà regolabile tramite i tasti standard del dispositivo.

5.2 Analisi del problema

Le tecniche scelte per determinare l'orientamento del dispositivo sono essenzialmente quattro e l'utente avrà la possibilità di decidere quale utilizzare grazie ad un apposito menù di RadioButton. Ognuna di queste tecniche di esse sfrutta diverse combinazioni di diversi tipi di sensori:

- Gravity Sensor
- Accelerometro + Magnetometro

- Gravity Sensor + Magnetometro
- Rotation Sensor

Il lettore potrebbe chiedersi come mai non sia presente il TYPE_ORIENTATION e il motivo è che questo tipo di sensore è stato deprecato.

Gravity Sensor. Il Gravity Sensor consente di determinare se il dispositivo è rivolto verso l'alto o verso il basso, il suo utilizzo è relativamente semplice ma non fornisce molte informazioni sull'orientamento.
La forza di gravità è misurata su tre assi coordinati X, Y e Z:

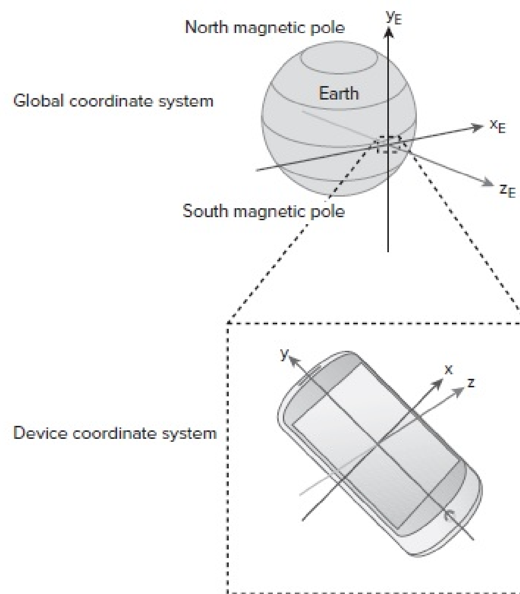

Figura 5.2: Orientamento degli assi coordinati in uno smartphone

L'orientamento dei tre assi è standard per tutti gli smartphone, mentre per alcuni tipi di tablet può cambiare.
Le frecce degli assi rappresentano le direzioni positive. Così quando il dispositivo si trova "a faccia in su", i valori di Z sono positivi e viceversa quando il dispositivo è voltato "a faccia in giù". Allo stesso modo, l'asse X è rivolto verso il nord e l'asse Y è rivolto verso est.
In pratica, il Gravity Sensor determina la posizione del dispositivo semplicemente misurando su quale asse agisce la forza di gravità. Se, ad esempio, lo smartphone fosse disteso a "faccia in giù" su un tavolo, il Gravity Sensor misurerebbe la terna di valori (0; 0; -9.8) $\frac{m}{s^2}$ riferiti rispettivamente a X, Y e Z (è dato per scontato

che il tavolo si trovi sul pianeta terra, altrimenti l'intensità della forza di gravità risulterebbe diversa).
E' normale aspettarsi che durante il test dell'applicazione il valore misurato non sia esattamente 9,8 $\frac{m}{s^2}$. Questo perchè, come già detto in precedenza, il sensore è molto sensibile ai rumori.

Accelerometro + Magnetometro. Un altro modo per determinare l'orientamento del dispositivo consiste in un Sensor Fusion tra accelerometro e magnetometro.
I dati provenienti da questi due tipi di sensori possono essere utilizzati per ricavare la matrice di rotazione tramite il metodo *SensorManager.getRotationMatrix()*.
Come già spiegato nel precedente capitolo, la matrice di rotazione dovrà poi essere passata come argomento del metodo *SensorManager.getOrientation()*, il quale restituirà la rotazione del dispositivo attorno ai soliti 3 assi coordinati in termini di azimuth (asse Z), pitch (asse X) e roll (asse Y).
Le misure del magnetometro, però, possono essere influenzate dall'eventuale presenza di magneti nelle vicinanze e ciò potrebbe compromettere l'accuratezza dei dati misurati.
Per determinare gli orientamenti "faccia in su e faccia in giù" saranno necessari quindi solo i valori di X e Y mentre l'asse Z terrà conto di come il dispositivo è posizionato rispetto al polo nord. Più precisamente, l'asse X (Y) misura, in termini di rotazione, un eventuale dislivello tra i due lati corti (lunghi) dello smartphone.
Quindi se X = $\pm\frac{\pi}{2}$, ciò significa che il dispositivo è in piedi perpendicolarmente al terreno (il segno positivo o negativo indica rispettivamente se è a "testa in su o testa in giù"). Analogamente, per Y = $\pm\frac{\pi}{2}$, il dispositivo è disteso sul lato lungo, destro per il valore positivo e sinistro per il negativo (col display sempre perpendicolare al terreno). La posizione "faccia in su/giù" è data dai valori zero o π.
In conclusione, i dati dell'accelerometro producono la matrice di rotazione ma questa strategia è fortemente sconsigliata perchè eventuali vibrazioni o agitazioni del dispositivo manderebbero in tilt il sensore compromettendo il funzionamento corretto dell'applicazione (si ricordi le considerazioni fatte nel capitolo precedente riguardo i metodi *getOrientation()* e *getRotationMatrix()*).
Per ovviare a questi inconvenienti, nella prossimo metodo che andremo a descrivere, l'accelerometro verrà sostituito dal Gravity Sensor.

Gravity Sensor + Magnetometro. C'è poco da spiegare riguardo a questa metodologia. Il procedimento è lo stesso della precedente.
Si tratta ancora di ricavare la matrice di rotazione che poi daremo in pasto al metodo *SensorManager.getOrientation()*. L'unica differenza è che i valori passati a *SensorManager.gerRotationMatrix()* questa volta provengono dal Gravity Sensor e non più dall'accelerometro.
Questo metodo propone quindi una modifica al precedente, allo scopo di cercare di risolverne/limitarne le problematiche causate da eventuali disturbi esterni.

Rotation Sensor. Come avrete potuto notare, ogni metodo descritto propone qualche miglioramento rispetto al precedente.
Il Rotation Sensor, già visto, è un sensore sintetico che sfrutta il Sensor Fusion tra accelerometro, magnetometro e giroscopio.
Il significato fisico delle sue uscite X, Y e Z è lo stesso delle precedenti metodologie (vedi accelerometro + magnetometro).
Il sensore produce un Rotation Vector che può essere convertito in una corrispondente matrice di rotazione grazie al metodo *SensorManager.getRotationMatrixFromVector()*.
Con la matrice di rotazione che otterremo possiamo, come al solito, fare uso del metodo *SensorManager.getOrientation()* per determinare l'orientamento.
Questa strategia è più semplice e preferibile rispetto alla strategia "Accelerometro + Magnetometro", poichè l'implementazione del Rotation Sensor (di proprietà di terze parti, es. Invensense) nasconde allo sviluppatore tutte le complessità dovute all'integrazione tra i tre sensori fisici.

Concludiamo l'analisi del problema facendo un ultima considerazione. Per questa applicazione, l'angolo prodotto dal metodo *SensorManager.getOrientation()* fornisce una rappresentazione accettabile dell'orientamento del dispositivo, ma purtroppo ha alcuni limiti. La rappresentazione Euclidea della rotazione non può essere sufficiente per applicazioni più complesse dove possono verificarsi problematiche come il blocco del giunto cardanico del giroscopio.
Per questi motivi, è preferibile una rappresentazione a quattro valori della rotazione. A tal scopo, si consiglia l'utilizzo del metodo *SensorManager.getQuaternionFromVector()*.

5.3 Fase d'implementazione

L'interfaccia utente per *DetermineOrientationActivity* è composta da una serie di Radio Buttons, i quali consentiranno di selezionare i tipi di sensori che verranno impiegati per effettuare le misurazioni. Le letture di tali sensori (sui 3 assi) sono stampate nella medesima schermata.

Il listato successivo mostra il layout Xml che implementa la grafica (*determine_orientation.xml*):

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:layout_width="match_parent"
4     android:layout_height="match_parent"
5     android:orientation="vertical" >
6
7     <RadioGroup android:id="@+id/sensorSelector"
8         android:layout_width="match_parent"
9         android:layout_height="wrap_content"
```

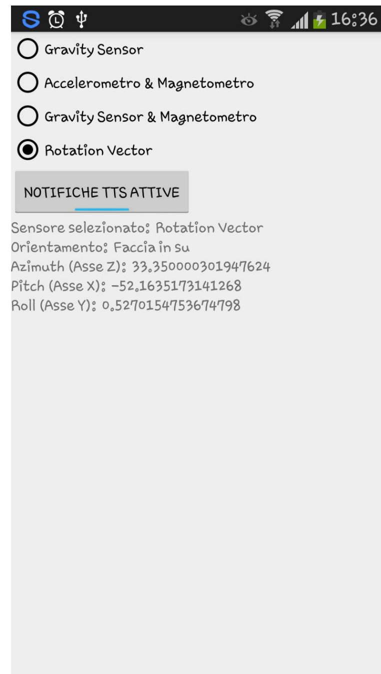

Figura 5.3: Interfaccia utente dell'applicazione DeviceOrientation

```
10      android:layout_alignParentTop="true" >
11
12      <RadioButton android:id="@+id/gravitySensor"
13          android:layout_width="match_parent"
14          android:layout_height="wrap_content"
15          android:text="@string/gravitySensorLabel"
16          android:checked="true"
17          android:onClick="onSensorSelectorClick" />
18
19      <RadioButton
20          android:id="@+id/accelerometerMagnetometer"
21          android:layout_width="match_parent"
22          android:layout_height="wrap_content"
23          android:checked="false"
24          android:onClick="onSensorSelectorClick"
25          android:text="@string/accelerometerMagnetometerLabel" />
26
27      <RadioButton android:id="@+id/gravityMagnetometer"
28          android:layout_width="match_parent"
29          android:layout_height="wrap_content"
30          android:text="@string/gravityMagnetometerLabel"
```

```
31              android:checked="false"
32              android:onClick="onSensorSelectorClick" />
33
34          <RadioButton android:id="@+id/rotationVector"
35              android:layout_width="match_parent"
36              android:layout_height="wrap_content"
37              android:text="@string/rotationVectorLabel"
38              android:checked="false"
39              android:onClick="onSensorSelectorClick" />
40      </RadioGroup>
41
42      <ToggleButton android:id="@+id/ttsNotificationsToggleButton"
43          android:layout_width="wrap_content"
44          android:layout_height="wrap_content"
45          android:text="@string/speakOrientationLabel"
46          android:checked="true"
47          android:layout_below="@id/sensorSelector"
48          android:textOn="@string/ttsNotificationsOn"
49          android:textOff="@string/ttsNotificationsOff"
50          android:onClick="onTtsNotificationsToggleButtonClicked" />
51
52      <TextView android:id="@+id/selectedSensorLabel"
53          android:layout_width="wrap_content"
54          android:layout_height="wrap_content"
55          android:text="@string/selectedSensorLabel"
56          android:layout_below="@id/ttsNotificationsToggleButton"
57          android:layout_marginRight="5dip" />
58
59      <TextView android:id="@+id/selectedSensorValue"
60          android:layout_width="wrap_content"
61          android:layout_height="wrap_content"
62          android:layout_toRightOf="@id/selectedSensorLabel"
63          android:layout_alignTop="@id/selectedSensorLabel"
64          android:layout_alignBottom="@id/selectedSensorLabel" />
65
66      <TextView android:id="@+id/orientationLabel"
67          android:layout_width="wrap_content"
68          android:layout_height="wrap_content"
69          android:text="@string/orientationLabel"
70          android:layout_below="@id/selectedSensorValue"
71          android:layout_marginRight="5dip" />
72
73      <TextView android:id="@+id/orientationValue"
74          android:layout_width="wrap_content"
75          android:layout_height="wrap_content"
76          android:layout_toRightOf="@id/orientationLabel"
77          android:layout_alignTop="@id/orientationLabel"
78          android:layout_alignBottom="@id/orientationLabel" />
79
```

```
80      <TextView android:id="@+id/sensorXLabel"
81          android:layout_width="wrap_content"
82          android:layout_height="wrap_content"
83          android:layout_below="@id/orientationValue"
84          android:layout_marginRight="5dip" />
85
86      <TextView android:id="@+id/sensorXValue"
87          android:layout_width="wrap_content"
88          android:layout_height="wrap_content"
89          android:layout_toRightOf="@id/sensorXLabel"
90          android:layout_alignTop="@id/sensorXLabel"
91          android:layout_alignBottom="@id/sensorXLabel" />
92
93      <TextView android:id="@+id/sensorYLabel"
94          android:layout_width="wrap_content"
95          android:layout_height="wrap_content"
96          android:layout_below="@id/sensorXLabel"
97          android:layout_marginRight="5dip" />
98
99      <TextView android:id="@+id/sensorYValue"
100         android:layout_width="wrap_content"
101         android:layout_height="wrap_content"
102         android:layout_toRightOf="@id/sensorYLabel"
103         android:layout_alignTop="@id/sensorYLabel"
104         android:layout_alignBottom="@id/sensorYLabel" />
105
106     <TextView android:id="@+id/sensorZLabel"
107         android:layout_width="wrap_content"
108         android:layout_height="wrap_content"
109         android:layout_below="@id/sensorYLabel"
110         android:layout_marginRight="5dip" />
111
112     <TextView android:id="@+id/sensorZValue"
113         android:layout_width="wrap_content"
114         android:layout_height="wrap_content"
115         android:layout_toRightOf="@id/sensorZLabel"
116         android:layout_alignTop="@id/sensorZLabel"
117         android:layout_alignBottom="@id/sensorZLabel" />
118
119 </RelativeLayout>
```

La classe *DetermineOrientationActivity* ha essenzialmente due compiti:

- Ottenere un riferimento a *SensorManager*
- Inizializzare il Text-To-Speech per notificare l'utente sull'orientamento del dispositivo

Il listato successivo mostra l'implementazione di tali operazioni, effettuate nel metodo *DetermineOrientationActivity.onCreate()*:

```java
@Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        super.setContentView(R.layout.determine_orientation);

        // Mantiene acceso il display per tutta la durata dell'applicazione
        getWindow().addFlags(WindowManager.LayoutParams.
                                FLAG_KEEP_SCREEN_ON);

        // Imposta lo stream per il Text-To-Speech
        ttsParams = new HashMap<String, String>();
        ttsParams.put(Engine.KEY_PARAM_STREAM,
                        String.valueOf(TTS_STREAM));

        // Imposta il controllo del volume tramite i tasti laterali
        // in modo da regolare l'intensità delle notifiche TTS
        this.setVolumeControlStream(TTS_STREAM);

        // Ottiene un riferimento al servizio SensorManager
        sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);

        // Inizializza i riferimenti ai componenti dell'interfaccia grafica
        // i quali verranno aggiornati più avanti nel codice
        sensorSelector = (RadioGroup) findViewById(R.id.sensorSelector);
        selectedSensorValue = (TextView) findViewById(R.id.selectedSensorValue);
        orientationValue = (TextView) findViewById(R.id.orientationValue);
        sensorXLabel = (TextView) findViewById(R.id.sensorXLabel);
        sensorXValue = (TextView) findViewById(R.id.sensorXValue);
        sensorYLabel = (TextView) findViewById(R.id.sensorYLabel);
        sensorYValue = (TextView) findViewById(R.id.sensorYValue);
        sensorZLabel = (TextView) findViewById(R.id.sensorZLabel);
        sensorZValue = (TextView) findViewById(R.id.sensorZValue);
        ttsNotificationsToggleButton =
                (ToggleButton) findViewById(R.id.ttsNotificationsToggleButton);

        // Recupera le preferenze memorizzate
        preferences = getPreferences(MODE_PRIVATE);
        ttsNotifications =
                preferences.getBoolean(TTS_NOTIFICATION_PREFERENCES_KEY,
                                        true);
    }
```

Appena completata l'inizializzazione, il passo successivo è la registrazione dei sensori in base alla scelta dell'utente.
Osservando la struttura dell'interfaccia si può facilmente intuire che è possibile cambiare dinamicamente a runtime la scelta del tipo di sensori. Per realizzare ciò, è necessario un metodo che effettui la deregistrazione agli eventi del sensore corrente e la registrazione a quelli del nuovo sensore scelto.
Questo metodo è *DetermineOrientationActivity.updateSelectedSensor()* e viene richiamato sia da *DetermineOrientationActivity.onResume()* sia dal codice che gestisce la selezione dei Radio Buttons.

Il listato successivo mostra l'implementazione del metodo *DetermineOrientationActivity.updateSelectedSensor()*:

```
1  private void updateSelectedSensor()
2      {
3          // Elimina tutte le registrazioni correnti
4          sensorManager.unregisterListener(this);
5
6          // Determina quale Radio Button è stato selezionato
7          // e abilita i relativi sensori
8          selectedSensorId = sensorSelector.getCheckedRadioButtonId();
9          if (selectedSensorId == R.id.accelerometerMagnetometer)
10         {
11             sensorManager.registerListener(this,
12                     sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
13                     RATE);
14
15             sensorManager.registerListener(this,
16                     sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD),
17                     RATE);
18         }
19         else if (selectedSensorId == R.id.gravityMagnetometer)
20         {
21             sensorManager.registerListener(this,
22                     sensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY),
23                     RATE);
24
25             sensorManager.registerListener(this,
26                     sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD),
27                     RATE);
28         }
29         else if ((selectedSensorId == R.id.gravitySensor))
30         {
31             sensorManager.registerListener(this,
32                     sensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY),
33                     RATE);
```

```
34          }
35          else
36          {
37              sensorManager.registerListener(this,
38                      sensorManager.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR),
39                      RATE);
40          }
41
42          // Aggiorna la Label in base al sensore selezionato
43          RadioButton selectedSensorRadioButton =
44                  (RadioButton) findViewById(selectedSensorId);
45          selectedSensorValue.setText(selectedSensorRadioButton.getText());
46      }
```

Poichè *updateSelectedSensor()* regista l'istanza corrente di *DetermineOrientationActivity*, la classe deve implementare *SensorEventListener* e quindi i suoi metodi *onSensorChanged()* e *onAccuracyChanged()*.
Questa applicazione, però, non ha bisogno di sapere il livello di accuratezza dei sensori perciò *onAccuracyChanged()* non fa niente di particolare, se non stampare un messaggio di log.
Il metodo *onSensorChanged()*, invece, deve gestire i *SensorEvents* delle diverse tipologie di sensori.
La fonte dei dati dipende dalla selezione dell'utente, dato che le registrazioni vengono aggiornate quando ogni qual volta si interagisce coi Radio Buttons. In pratica, *onSensorChanged()* si limita a gestire il *SensorEvent* che gli viene passato in input, a prescindere dalla sua provenienza.
Per quanto riguarda i dati, essi vengono prelevati dall'array *SensorEvent.values* ma a causa della loro diversa natura (che varia a seconda del sensore che li genera) è necessario individuarne la fonte prima di elaborarli.

Elaborazione dati Gravity Sensor. In questo caso l'array *SensorEvent.values* conterrà i valori della gravità sugli assi X, Y e Z rispettivamente nelle posizioni [0], [1] e [2].
Per determinare se il dispositivo è a "a faccia in su o a faccia in giù", è necessario solo quello sull'asse Z (cioè perpendicolare al display) ovvero *SensorEvent.values[2]*.
Di conseguenza, quando lo smartphone è appoggiato sul dorso, la componente Z della forza di gravità sarà pari a "g" (9,8 $\frac{m}{s^2}$), il cui valore è memorizzato nella costante *SensorManager.STANDARD_GRAVITY*. Al contrario, cioè faccia verso il basso, Z deve valere "-g" cioè -1 * *SensorManager.STANDARD_GRAVITY*.
Tuttavia si ricorda che, a causa del rumore generato dal sensore, si potrebbero avere delle fluttuazioni indesiderate dei dati. Per evitare questo inconveniente, l'app utilizza un valore di soglia pari alla metà di *SensorManager.STANDARD_GRAVITY*, in modo da determinare il verso del dispositivo anche quando quest'ultimo non sia

perfettamente parallelo al terreno.

Il listato successivo mostra come realizzare tutto ciò:

```
1  private static final double GRAVITY_THRESHOLD =
2  SensorManager.STANDARD_GRAVITY / 2;
3  ...
4  if (selectedSensorId == R.id.gravitySensor)
5  {
6          if (event.values[2] >= GRAVITY_THRESHOLD)
7      {
8          onFaceUp();
9      }
10     else if (event.values[2] <= (GRAVITY_THRESHOLD * -1))
11     {
12                 onFaceDown();
13         }
14 }
15 ...
```

Elaborazione dati Accelerometro + Magnetometro. I valori dell'accelerometro e del magnetometro vengono passati in input al metodo *SensorManager.getRotationMatrix()*, il quale a sua volta genererà un matrice di rotazione che verrà passata in input al metodo *SensorManager.getOrientation()*, il quale infine determinerà l'orientamento del dispositivo.
Poichè sono necessari i valori di entrambi i sensori, *DetermineOrientationActivity* memorizzerà i più recenti. Per ragioni di sincronismo tra la lettura dei dati e la chiamata a *onSensorEvent()* è importante copiare i dati invece di assegnargli un altro riferimento. Questo perchè l'array *event.values* potrebbe venire sovrascritto da Android durante l'esecuzione di *onSensorEvent()*.

Il listato successivo mostra l'implementazione per *generateRotationMatrix()*. Questo metodo utilizza i valori di accelerometro e magnetometro se e solo se entrambi i set di valori sono stati riempiti coi i dati aggiornati.

```
1  private float[] generateRotationMatrix()
2      {
3          float[] rotationMatrix = null;
4
5          if (accelerationValues != null && magneticValues != null)
6          {
7              rotationMatrix = new float[16];
8              boolean rotationMatrixGenerated;
```

```
            rotationMatrixGenerated =
                    SensorManager.getRotationMatrix(rotationMatrix,
                    null,
                    accelerationValues,
                    magneticValues);

            if (!rotationMatrixGenerated)
            {
                Log.w(TAG, getString(R.string.rotationMatrixGenFailureMessage));

                rotationMatrix = null;
            }
        }

        return rotationMatrix;
    }
```

Dopo aver verificato che *accelerationValues* e *magneticValues* (aggiornati in *onSensorChanged()*) siano non-nulli, viene chiamato *SensorManager.getRotationMatrix()*. Questo metodo necessita di 4 parametri: i primi due conterranno la matrice di rotazione e la matrice di inclinazione (quest'ultima inutile per la nostra app e quindi posta a null), gli altri due sono i valori di accelerometro e magnetometro.
Prima di gestire la matrice di rotazione ottenuta è opportuno controllare il valore di ritorno di *getRotationMatrix()*. Nel caso sia false, significa che le matrici di uscita sono state lasciate intatte (allo stato antecedente la chiamata) e verrà stampato il messaggio relativo al tipo di errore che si è verificato.

Elaborazione dati Rotation Vector. Il procedimento è molto simile al precedente. Le principali differenze sono che si ha un solo sensore da gestire invece che due, eliminando la necessità di clonare eventuali array, e la matrice di rotazione è generata in modo diverso.

Il listato successivo mostra, in *onSensorChanged()*, la gestione dei *SensorEvents* che riguardano accelerometro, magnetometro e sensore di gravità:

```
@Override
    public void onSensorChanged(SensorEvent event)
    {
        float[] rotationMatrix;

        switch (event.sensor.getType())
        {
            case Sensor.TYPE_GRAVITY:
                sensorXLabel.setText(R.string.xAxisLabel);
                sensorXValue.setText(String.valueOf(event.values[0]));
```

```
11
12                  sensorYLabel.setText(R.string.yAxisLabel);
13                  sensorYValue.setText(String.valueOf(event.values[1]));
14
15                  sensorZLabel.setText(R.string.zAxisLabel);
16                  sensorZValue.setText(String.valueOf(event.values[2]));
17
18                  sensorYLabel.setVisibility(View.VISIBLE);
19                  sensorYValue.setVisibility(View.VISIBLE);
20                  sensorZLabel.setVisibility(View.VISIBLE);
21                  sensorZValue.setVisibility(View.VISIBLE);
22
23                  if (selectedSensorId == R.id.gravitySensor)
24                  {
25                      if (event.values[2] >= GRAVITY_THRESHOLD)
26                      {
27                          onFaceUp();
28                      }
29                      else if (event.values[2] <= (GRAVITY_THRESHOLD * -1))
30                      {
31                          onFaceDown();
32                      }
33                  }
34                  else
35                  {
36                      accelerationValues = event.values.clone();
37                      rotationMatrix = generateRotationMatrix();
38
39                      if (rotationMatrix != null)
40                      {
41                          determineOrientation(rotationMatrix);
42                      }
43                  }
44
45                  break;
46              case Sensor.TYPE_ACCELEROMETER:
47                  accelerationValues = event.values.clone();
48                  rotationMatrix = generateRotationMatrix();
49
50                  if (rotationMatrix != null)
51                  {
52                      determineOrientation(rotationMatrix);
53                  }
54                  break;
55              case Sensor.TYPE_MAGNETIC_FIELD:
56                  magneticValues = event.values.clone();
57                  rotationMatrix = generateRotationMatrix();
58
59                  if (rotationMatrix != null)
```

```
60                  {
61                      determineOrientation(rotationMatrix);
62                  }
63                  break;
64              case Sensor.TYPE_ROTATION_VECTOR:
65
66                  rotationMatrix = new float[16];
67                  SensorManager.getRotationMatrixFromVector(rotationMatrix,
68                          event.values);
69                  determineOrientation(rotationMatrix);
70                  break;
71          }
72      }
```

A prescindere dal fatto che l'evento sia stato generato dall'accelerometro, dal magnetometro o dal Rotation Vector, il metodo *onSensorChanged()* si conclude con una chiamata a *determineOrientation()*, il quale, data in ingresso la matrice di rotazione, permette di determinare l'orientamento del dispositivo.

Il listato successivo mostra l'implementazione del metodo *determineOrientation()*:

```
1  private void determineOrientation(float[] rotationMatrix)
2      {
3          float[] orientationValues = new float[3];
4          SensorManager.getOrientation(rotationMatrix, orientationValues);
5
6          double azimuth = Math.toDegrees(orientationValues[0]);
7          double pitch = Math.toDegrees(orientationValues[1]);
8          double roll = Math.toDegrees(orientationValues[2]);
9
10         sensorXLabel.setText(R.string.azimuthLabel);
11         sensorXValue.setText(String.valueOf(azimuth));
12
13         sensorYLabel.setText(R.string.pitchLabel);
14         sensorYValue.setText(String.valueOf(pitch));
15
16         sensorZLabel.setText(R.string.rollLabel);
17         sensorZValue.setText(String.valueOf(roll));
18
19         sensorYLabel.setVisibility(View.VISIBLE);
20         sensorYValue.setVisibility(View.VISIBLE);
21         sensorZLabel.setVisibility(View.VISIBLE);
22         sensorZValue.setVisibility(View.VISIBLE);
23
24         if (pitch <= 10)
```

```
25         {
26             if (Math.abs(roll) >= 170)
27             {
28                 onFaceDown();
29             }
30             else if (Math.abs(roll) <= 10)
31             {
32                 onFaceUp();
33             }
34         }
35     }
```

SensorManager.getOrientation() necessita di due parametri: due array di float, di cui uno contenente la matrice di rotazione e l'altro il risultato della chiamata. Questo risultato, *orientationValues*, consiste in tre valori misurati in radianti, i quali rappresentano azimuth, pitch e roll.
Successivamente, questi valori verranno convertiti in gradi e stampati nell'interfaccia grafica. In questo modo l'utente potrà osservare il loro cambiamento al variare dell'orientamento del dispositivo.
Inoltre, gli stessi verranno utilizzati anche per determinare il face up/down. Per far ciò sono necessari solo pitch e roll (il pitch dovrebbe essere zero se il dispositivo è in posizione perpendicolare al terreno). Tuttavia, come nel caso del sensore di gravità, è necessario che l'algoritmo gestisca eventuali rumori introducendo un valore di soglia per il pitch pari a 10 gradi. La stessa cosa viene fatta per il roll ma, dato che all'app non interessa la direzione verso cui il dispositivo viene ruotato, si confronta il valore di soglia con il valore assoluto (face up quando il valore assoluto di roll è inferiore o uguale a 10 gradi e face down nel caso sia maggiore o uguale a 170).

A questo punto rimane solo da affrontare la questione di come notificare l'utente quando cambia l'orientamento.

Notifiche Text-To-Speech. Tutte le volte che l'applicazione riuscirà a determinare se il dispositivo è rivolto verso l'alto o verso il basso, l'utente verrà notificato dal Text-To-Speech (TTS) con un messaggio vocale, in modo che non sia necessario andare a controllare il display quando viene voltato verso il terreno. Questa funzionalità è stata implementata per rendere l'applicazione più comoda ma, a seconda delle preferenze, può essere disabilitata tramite un pulsante sull'interfaccia grafica.

Il listato successivo mostra l'implementazione dei metodi *faceUp()* e *faceDown()*:

```
1 /**
2      * Handler relativo all'orientamento "faccia in su"
3      */
4     @SuppressWarnings("deprecation")
5         private void onFaceUp()
6     {
7         if (!isFaceUp)
8         {
9             if (tts != null && ttsNotificationsToggleButton.isChecked())
10            {
11                tts.speak(getString(R.string.faceUpText),
12                        TextToSpeech.QUEUE_FLUSH,
13                        ttsParams);
14            }
15
16            orientationValue.setText(R.string.faceUpText);
17            isFaceUp = true;
18        }
19    }
20
21    /**
22     * Handler relativo all'orientamento "faccia in giù"
23     */
24    @SuppressWarnings("deprecation")
25        private void onFaceDown()
26    {
27        if (isFaceUp)
28        {
29            if (tts != null && ttsNotificationsToggleButton.isChecked())
30            {
31                tts.speak(getString(R.string.faceDownText),
32                        TextToSpeech.QUEUE_FLUSH,
33                        ttsParams);
34            }
35
36            orientationValue.setText(R.string.faceDownText);
37            isFaceUp = false;
38        }
39    }
```

5.4 DetermineOrientationActivity.java

Il listato successivo riassume in un unica classe, *DetermineOrientationActivity.java*, tutto il codice e i metodi analizzati nella sezione precedente:

```
public class DetermineOrientationActivity extends
        SpeechRecognizingAndSpeakingActivity implements SensorEventListener
{
    private static final String TAG = "DetermineOrientationActivity";
    private static final int RATE = SensorManager.SENSOR_DELAY_NORMAL;
    private static final int TTS_STREAM = AudioManager.STREAM_NOTIFICATION;
    private static final String TTS_NOTIFICATION_PREFERENCES_KEY =
            "TTS_NOTIFICATION_PREFERENCES_KEY";
    private static final double GRAVITY_THRESHOLD =
            SensorManager.STANDARD_GRAVITY / 2;

    private SensorManager sensorManager;
    private float[] accelerationValues;
    private float[] magneticValues;
    private TextToSpeech tts;
    private boolean isFaceUp;
    private RadioGroup sensorSelector;
    private TextView selectedSensorValue;
    private TextView orientationValue;
    private TextView sensorXLabel;
    private TextView sensorXValue;
    private TextView sensorYLabel;
    private TextView sensorYValue;
    private TextView sensorZLabel;
    private TextView sensorZValue;
    private HashMap<String, String> ttsParams;
    private ToggleButton ttsNotificationsToggleButton;
    private SharedPreferences preferences;
    private boolean ttsNotifications;
    private int selectedSensorId;

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        super.setContentView(R.layout.determine_orientation);

        // Mantiene acceso il display per tutta la durata dell'applicazione
        getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

        // Imposta lo stream per il Text-To-Speech
        ttsParams = new HashMap<String, String>();
        ttsParams.put(Engine.KEY_PARAM_STREAM, String.valueOf(TTS_STREAM));

        // Imposta il controllo del volume tramite i tasti laterali
        // in modo da regolare l'intensità delle notifiche TTS
        this.setVolumeControlStream(TTS_STREAM);

```

```
49          // Ottiene un riferimento al servizio SensorManager
50          sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
51
52          // Inizializza i riferimenti ai componenti dell'interfaccia grafica
53          // i quali verranno aggiornati più avanti nel codice
54          sensorSelector = (RadioGroup) findViewById(R.id.sensorSelector);
55          selectedSensorValue = (TextView) findViewById(R.id.selectedSensorValue);
56          orientationValue = (TextView) findViewById(R.id.orientationValue);
57          sensorXLabel = (TextView) findViewById(R.id.sensorXLabel);
58          sensorXValue = (TextView) findViewById(R.id.sensorXValue);
59          sensorYLabel = (TextView) findViewById(R.id.sensorYLabel);
60          sensorYValue = (TextView) findViewById(R.id.sensorYValue);
61          sensorZLabel = (TextView) findViewById(R.id.sensorZLabel);
62          sensorZValue = (TextView) findViewById(R.id.sensorZValue);
63          ttsNotificationsToggleButton =
64                  (ToggleButton) findViewById(R.id.ttsNotificationsToggleButton);
65
66          // Recupera le preferenze memorizzate
67          preferences = getPreferences(MODE_PRIVATE);
68          ttsNotifications =
69                  preferences.getBoolean(TTS_NOTIFICATION_PREFERENCES_KEY, true);
70      }
71
72      @Override
73      protected void onResume()
74      {
75          super.onResume();
76
77          ttsNotificationsToggleButton.setChecked(ttsNotifications);
78          updateSelectedSensor();
79      }
80
81      @Override
82      protected void onPause()
83      {
84          super.onPause();
85
86          // Deregistra gli aggiornamenti dai sensori
87          sensorManager.unregisterListener(this);
88
89          // Spegne le notifiche TTS
90          if (tts != null)
91          {
92              tts.shutdown();
93          }
94      }
95
96      @Override
97      public void onSensorChanged(SensorEvent event)
```

```
98      {
99          float[] rotationMatrix;
100
101         switch (event.sensor.getType())
102         {
103             case Sensor.TYPE_GRAVITY:
104                 sensorXLabel.setText(R.string.xAxisLabel);
105                 sensorXValue.setText(String.valueOf(event.values[0]));
106
107                 sensorYLabel.setText(R.string.yAxisLabel);
108                 sensorYValue.setText(String.valueOf(event.values[1]));
109
110                 sensorZLabel.setText(R.string.zAxisLabel);
111                 sensorZValue.setText(String.valueOf(event.values[2]));
112
113                 sensorYLabel.setVisibility(View.VISIBLE);
114                 sensorYValue.setVisibility(View.VISIBLE);
115                 sensorZLabel.setVisibility(View.VISIBLE);
116                 sensorZValue.setVisibility(View.VISIBLE);
117
118                 if (selectedSensorId == R.id.gravitySensor)
119                 {
120                     if (event.values[2] >= GRAVITY_THRESHOLD)
121                     {
122                         onFaceUp();
123                     }
124                     else if (event.values[2] <= (GRAVITY_THRESHOLD * -1))
125                     {
126                         onFaceDown();
127                     }
128                 }
129                 else
130                 {
131                     accelerationValues = event.values.clone();
132                     rotationMatrix = generateRotationMatrix();
133
134                     if (rotationMatrix != null)
135                     {
136                         determineOrientation(rotationMatrix);
137                     }
138                 }
139
140                 break;
141             case Sensor.TYPE_ACCELEROMETER:
142                 accelerationValues = event.values.clone();
143                 rotationMatrix = generateRotationMatrix();
144
145                 if (rotationMatrix != null)
146                 {
```

```
147                     determineOrientation(rotationMatrix);
148                 }
149                 break;
150             case Sensor.TYPE_MAGNETIC_FIELD:
151                 magneticValues = event.values.clone();
152                 rotationMatrix = generateRotationMatrix();
153 
154                 if (rotationMatrix != null)
155                 {
156                     determineOrientation(rotationMatrix);
157                 }
158                 break;
159             case Sensor.TYPE_ROTATION_VECTOR:
160 
161                 rotationMatrix = new float[16];
162                 SensorManager.getRotationMatrixFromVector(rotationMatrix,
163                         event.values);
164                 determineOrientation(rotationMatrix);
165                 break;
166         }
167     }
168 
169     @Override
170     public void onAccuracyChanged(Sensor sensor, int accuracy)
171     {
172         Log.d(TAG,
173                 String.format("Accuracy for sensor %s = %d",
174                 sensor.getName(), accuracy));
175     }
176 
177     /**
178      * Genera un matrice di rotazione utilizzando i valori di
179      * accelerationValues e magneticValues.
180      *
181      */
182     private float[] generateRotationMatrix()
183     {
184         float[] rotationMatrix = null;
185 
186         if (accelerationValues != null && magneticValues != null)
187         {
188             rotationMatrix = new float[16];
189             boolean rotationMatrixGenerated;
190             rotationMatrixGenerated =
191                     SensorManager.getRotationMatrix(rotationMatrix,
192                     null,
193                     accelerationValues,
194                     magneticValues);
195 
```

```
196             if (!rotationMatrixGenerated)
197             {
198                 Log.w(TAG, getString(R.string.rotationMatrixGenFailureMessage));
199
200                 rotationMatrix = null;
201             }
202         }
203
204         return rotationMatrix;
205     }
206
207     /**
208      * Utilizza l'ultima lettura dell'accelerometro e del Gravity sensor
209      * per determinare se il dispositivo è a "faccia in su" o a "faccia in giù"
210      *
211      */
212     private void determineOrientation(float[] rotationMatrix)
213     {
214         float[] orientationValues = new float[3];
215         SensorManager.getOrientation(rotationMatrix, orientationValues);
216
217         double azimuth = Math.toDegrees(orientationValues[0]);
218         double pitch = Math.toDegrees(orientationValues[1]);
219         double roll = Math.toDegrees(orientationValues[2]);
220
221         sensorXLabel.setText(R.string.azimuthLabel);
222         sensorXValue.setText(String.valueOf(azimuth));
223
224         sensorYLabel.setText(R.string.pitchLabel);
225         sensorYValue.setText(String.valueOf(pitch));
226
227         sensorZLabel.setText(R.string.rollLabel);
228         sensorZValue.setText(String.valueOf(roll));
229
230         sensorYLabel.setVisibility(View.VISIBLE);
231         sensorYValue.setVisibility(View.VISIBLE);
232         sensorZLabel.setVisibility(View.VISIBLE);
233         sensorZValue.setVisibility(View.VISIBLE);
234
235         if (pitch <= 10)
236         {
237             if (Math.abs(roll) >= 170)
238             {
239                 onFaceDown();
240             }
241             else if (Math.abs(roll) <= 10)
242             {
243                 onFaceUp();
244             }
```

```
245         }
246     }
247
248     /**
249      * Handler relativo all'orientamento "faccia in su"
250      */
251     @SuppressWarnings("deprecation")
252         private void onFaceUp()
253     {
254         if (!isFaceUp)
255         {
256             if (tts != null && ttsNotificationsToggleButton.isChecked())
257             {
258                 tts.speak(getString(R.string.faceUpText),
259                         TextToSpeech.QUEUE_FLUSH,
260                         ttsParams);
261             }
262
263             orientationValue.setText(R.string.faceUpText);
264             isFaceUp = true;
265         }
266     }
267
268     /**
269      * Handler relativo all'orientamento "faccia in giù"
270      */
271     @SuppressWarnings("deprecation")
272         private void onFaceDown()
273     {
274         if (isFaceUp)
275         {
276             if (tts != null && ttsNotificationsToggleButton.isChecked())
277             {
278                 tts.speak(getString(R.string.faceDownText),
279                         TextToSpeech.QUEUE_FLUSH,
280                         ttsParams);
281             }
282
283             orientationValue.setText(R.string.faceDownText);
284             isFaceUp = false;
285         }
286     }
287
288     /**
289      * Aggiorna l'interfaccia grafica quando viene selezionato un altro sensore
290      */
291     private void updateSelectedSensor()
292     {
293         // Elimina tutte le registrazioni correnti
```

```
294         sensorManager.unregisterListener(this);
295 
296         // Determina quale Radio Button è stato selezionato
297         // e abilita i relativi sensori
298         selectedSensorId = sensorSelector.getCheckedRadioButtonId();
299         if (selectedSensorId == R.id.accelerometerMagnetometer)
300         {
301             sensorManager.registerListener(this,
302                     sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
303                     RATE);
304 
305             sensorManager.registerListener(this,
306                     sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD),
307                     RATE);
308         }
309         else if (selectedSensorId == R.id.gravityMagnetometer)
310         {
311             sensorManager.registerListener(this,
312                     sensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY),
313                     RATE);
314 
315             sensorManager.registerListener(this,
316                     sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD),
317                     RATE);
318         }
319         else if ((selectedSensorId == R.id.gravitySensor))
320         {
321             sensorManager.registerListener(this,
322                     sensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY),
323                     RATE);
324         }
325         else
326         {
327             sensorManager.registerListener(this,
328                     sensorManager.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR),
329                     RATE);
330         }
331 
332         // Aggiorna la Label in base al sensore selezionato
333         RadioButton selectedSensorRadioButton =
334                 (RadioButton) findViewById(selectedSensorId);
335         selectedSensorValue.setText(selectedSensorRadioButton.getText());
336     }
337 
338     /**
339      * Handler per il click event relativo al sensore selezionato
340      *
341      */
342     public void onSensorSelectorClick(View view)
```

```
343      {
344          updateSelectedSensor();
345      }
346
347      /**
348       * Handler per il click event relativo al TTS Button
349       *
350       */
351      public void onTtsNotificationsToggleButtonClicked(View view)
352      {
353          ttsNotifications = ((ToggleButton) view).isChecked();
354          preferences.edit()
355              .putBoolean(TTS_NOTIFICATION_PREFERENCES_KEY, ttsNotifications)
356              .commit();
357      }
358
359      @Override
360      public void onSuccessfulInit(TextToSpeech tts)
361      {
362          super.onSuccessfulInit(tts);
363          this.tts = tts;
364      }
365
366      @Override
367      protected void receiveWhatWasHeard(List<String> heard, float[] confidenceScores)
368      {
369          // no-op
370      }
371  }
```

In questo capitolo ci siamo limitati a trattare solo la parte dell'applicazione che riguarda strettamente la sensoristica Android.
In realtà l'applicazione è più complessa e implementa altri requisiti meno funzionali che non verranno spiegati nel dettaglio (mantenere lo schermo costantemente acceso, regolare il volume del TTS coi tasti laterali dello smartphone, estendere la classe *SpeechRecognizingAndSpeakingActivity* per supportare più lingue, ecc...).

Appendice A

La fisica dei sensori

A.1 Accelerometro

L'accelerometro è un sensore di movimento che consiste in un contenitore in cui è situata una massa agganciata a una o più molle.

Figura A.1: Schema di un Accelerometro

Se si usano tre accelerometri orientati a seconda delle tre dimensioni spaziali (rappresentate di solito dalle coordinate cartesiane X, Y e Z), si possono determinare gli spostamenti nello spazio dell'oggetto cui sono collegati gli accelerometri.

Il sensore, obbedendo alla seconda legge di Newton, misura l'accelerazione come rapporto tra forza elastica e massa della sfera, tenendo presente che la forza

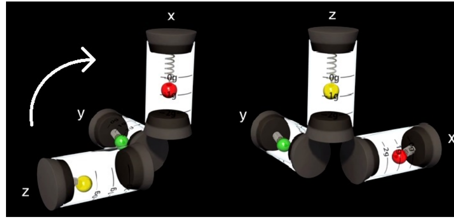

Figura A.2: Schema di un Accelerometro orientato

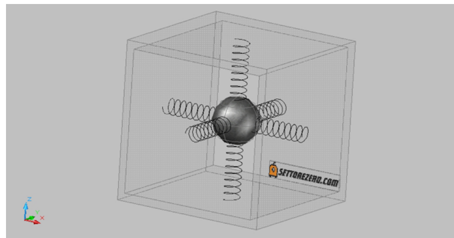

Figura A.3: Schema alternativo di un Accelerometro orientato

elastica è direttamente proporzionale all'allungamento/compressione della molla (Legge di Hooke).
A seconda dell'allungamento/compressione delle molle, il sensore può inviare specifici segnali a qualsiasi dispositivo.

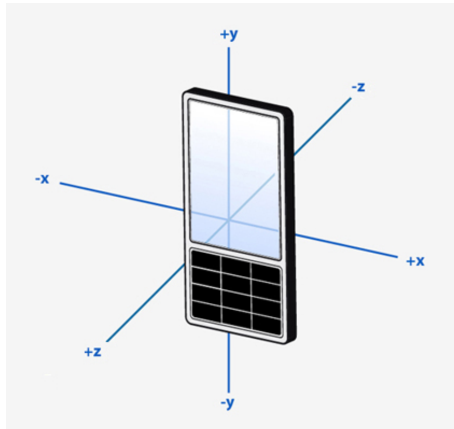

Figura A.4: Samsung Galaxy S3: orientamento assi xyz

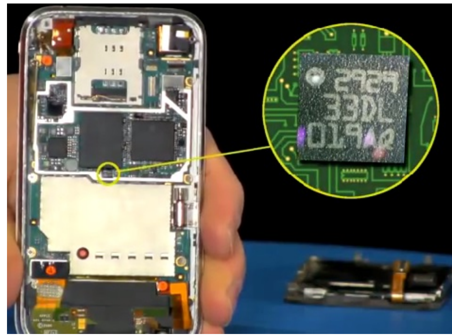

Figura A.5: Locazione di un accelerometro all'interno di uno smartphone

Dal punto di vista puramente fisico, sono più interessanti le conseguenze che l'accelerazione ha sulla posizione e sulla velocità del dispositivo in un certo periodo di tempo. Ciò ci porta a considerare altri tipi di equazioni, oltre alla già citata seconda legge di Newton (a = F/m), ovvero:

$$v_x(t) = v_{0x} + a_x t$$
$$v_y(t) = v_{0y} + a_y t$$
$$v_z(t) = v_{0z} + a_z t$$

dove, ad esempio, $v_x(t)$ rappresenta la velocità dell'oggetto in funzione della variabile temporale t sull'asse x, mentre v_{0x} è la costante additiva che ne indica la velocità iniziale.
Per quanto riguarda la variazione della posizione rispetto ad un asse, basta integrare le equazioni precedenti:

$$x(t) = x_0 + v_{0x}t + \frac{1}{2}a_x t^2$$
$$y(t) = y_0 + v_{0y}t + \frac{1}{2}a_y t^2$$
$$z(t) = z_0 + v_{0z}t + \frac{1}{2}a_z t^2$$

Nella maggior parte dei casi si suppone che l'oggetto si trovi inizialmente in uno stato di quiete ponendo v_0=0 e che in tale stato la posizione corrisponda all'origine del sistema di coordinate xyz. Le equazioni precedenti si semplificano in:

$$v_x(t) = a_x t$$
$$v_y(t) = a_y t$$
$$v_z(t) = a_z t$$
$$x(t) = \frac{1}{2}a_x t^2$$
$$y(t) = \frac{1}{2}a_y t^2$$
$$z(t) = \frac{1}{2}a_z t^2$$

Occorre anche considerare la forza di gravità, che che agisce allo stesso modo su tutti gli oggetti che si trovano in prossimità della superficie terrestre indipendentemente dalla loro massa. In tal caso l'accelerazione a assume il valore g (dove g varia tra 9,78 e 9.82 in funzione della latitudine) e, se si suppone che z sia l'asse verticale dove agisce la gravità, si trasformano le equazioni $v_z(t)$ e $z(t)$ in:

$v_z(t) = gt$
$z(t) = \frac{1}{2}gt^2$

A.2 Giroscopio

Il giroscopio è un sensore di movimento che consiste in un rotore che ruota attorno ad un asse. L'asse di rotazione tende a mantenersi orientato in una direzione fissa come succede in una trottola. Due giunti cardanici ingabbiano la trottola e premettono ad essa di ruotare liberamente nelle tre direzioni dello spazio vincolando l'asse a mantenersi sempre verticale rispetto alla terra, indipendentemente dall'inclinazione del sensore

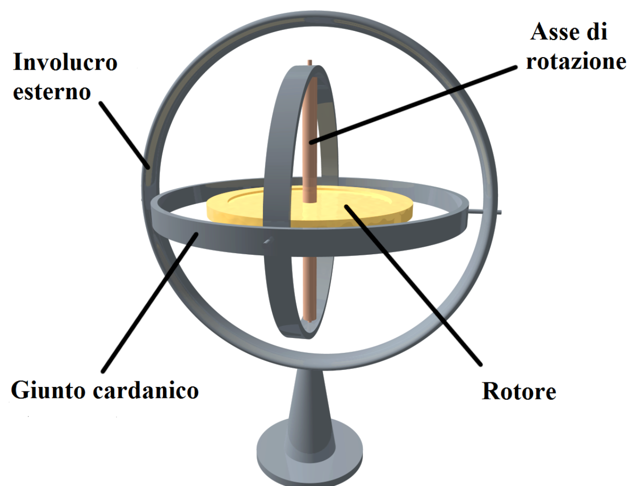

Figura A.6: Schema di un Giroscopio

All'interno degli smartphone moderni è sempre presente almeno un chip hardware che funziona da giroscopio e permette di determinare le rotazioni nello spazio del dispositivo inviando specifici segnali alle applicazioni.

Figura A.7: Locazione di un giroscopio all'interno di uno smartphone

L'orientamento degli assi segue le stesse direzioni viste per l'accelerometro con i versi determinati utilizzando la regola della mano destra.

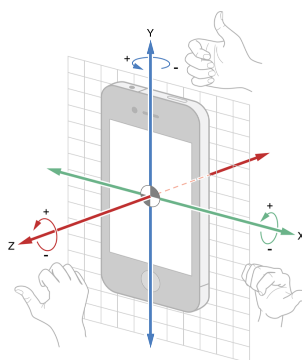

Figura A.8: Giroscopio: orientamento assi in uno smartphone

Per comprendere il comportamento fisico del giroscopio bisogna avere familiarità con tutto ciò che riguarda i moti rotatori, in particolare con grandezze quali il momento angolare, il momento d'inerzia, il momento torcente, la velocità e l'accelerazione angolare, ecc...
Si consideri un trottola come caso semplificato del moto del giroscopio e la si ponga in rotazione attorno al suo asse di simmetria con l'estremo coincidente con l'origine O di un sistema di riferimento inerziale. La trottola si muove attorno all'asse descrivendo un cono (moto di precessione).
Mentre il vettore rappresentante la velocità angolare $\overrightarrow{\omega}$ è sempre orientato secondo l'asse di rotazione, il vettore rappresentante il momento angolare $\overrightarrow{L}$ lo è solo se il corpo è dotato di simmetria.
Indicando con $\overrightarrow{\omega}_{\mathrm{p}}$ la velocità angolare di precessione, possiamo supporre, nel caso della trottola, $\overrightarrow{\omega}$ e $\overrightarrow{\omega}_{\mathrm{p}}$ coassiali, cioè ad una lenta precessione corrisponde un lento cambio di direzione dell'asse.

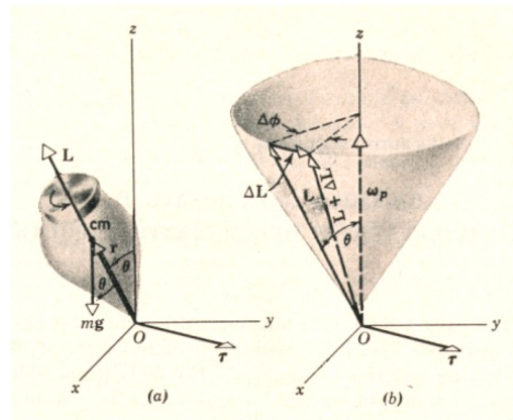

Figura A.9: Il moto della trottola

L'angolo compreso tra l'asse della trottola (dove si trova anche $\vec{L}$) e l'asse perpendicolare al piano orizzontale è indicato con θ. La risultante delle forze agenti sulla trottola è nulla perchè peso (mg) e reazione vincolare sono opposte e uguali in modulo. Rispetto all'origine O il momento torcente $\vec{\tau}$ (o momento delle forze), invece, non è nullo a causa della forza peso:

$$\vec{\tau} = \vec{r} \wedge \mathrm{m}\vec{g}$$

dove $\vec{r}$ è il vettore posizionale del centro di massa rispetto al perno.
Il momento torcente esprime la tendenza di una forza a imprimere una rotazione ad un corpo, è per definizione perpendicolare al piano individuato dai vettori $\vec{r}$ e m$\vec{g}$ e, se non è nullo, il momento angolare varia secondo la seguente legge:

$$\vec{\tau} = \mathrm{d}\vec{L}/\mathrm{dt} \Rightarrow \mathrm{d}\vec{L} = \vec{\tau}\mathrm{dt}$$

dove d$\vec{L}$ indica una variazione del vettore, in particolare direzione e verso, mentre il modulo rimarrà costante.
La punta del vettore momento angolare disegnerà un cerchio orizzontale e in questo modo la trottola si muove attorno alla verticale causando la precessione.
La trottola, come già accennato, può essere vista come un caso particolare del moto giroscopico in cui momento torcente e momento angolare hanno stessa direzione, mentre il giroscopio può essere messo in condizioni tali che ciò non si verifichi.
Un altro modo di riprodurre il moto giroscopico consiste nel mettere in moto a velocità angolare $\vec{\omega}$ la ruota di una bicicletta attorno al suo asse. Un'estremità di tale asse deve essere poi fissata a un supporto in modo da essere libera di orientarsi liberamente. Lasciando andare l'estremità libera dell'asse, la ruota non cadrà a terra ma descriverà una traiettoria circolare attorno al supporto originando un

moto di precessione a velocità angolare $\overrightarrow{\omega}_{\mathrm{p}}$.

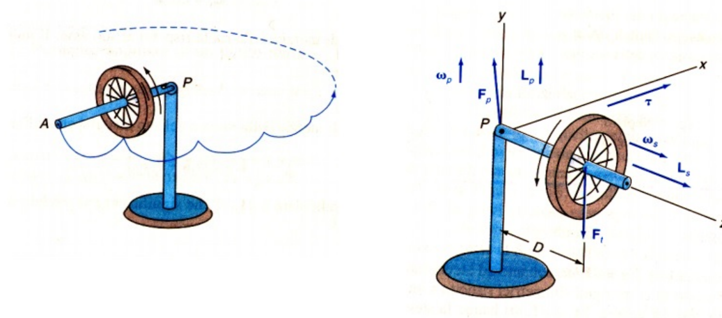

Figura A.10: Moto giroscopico: esperimento con la ruota

Le oscillazioni del moto di precessione sono dovute all'attrito del perno e costituiscono il moto di nutazione.
Nella figura successiva si può osservare in 6 fasi il moto giroscopico di un vero e proprio giroscopio.

Figura A.11: Moto giroscopico: esperimento con il giroscopio

A.3 Sensore di Prossimità

Il sensore di prossimità appartiene alla categoria dei sensori di posizione. Il suo funzionamento consiste nell'emettere radiazioni elettromagnetiche ed eventualmente ricevere l'onda riflessa. In generale, questi sensori producono dati di tipo booleano a indicare o meno la presenza di oggetti entro la loro portata nominale. Tuttavia è possibile realizzare al livello software dei protocolli che, tenendo conto della velocità di propagazione dell'onda e dal tempo di andata e ritorno, possono calcolare la distanza a cui è stato rilevato l'oggetto.

Figura A.12: Sensore di prossimità

Negli smartphone moderni i touchscreen capacitivi sono molto sensibili e durante una chiamata il nostro orecchio può causare la pressione indesiderata dei tasti sul display. La funzione del sensore di prossimità è proprio quella di disabilitare il display non appena viene rilevato che la distanza tra il display e l'orecchio sia inferiore a circa 5 cm.

Figura A.13: Sensore di prossimità all'interno di uno smartphone

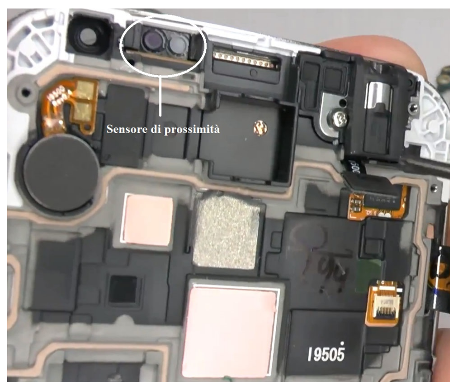

Figura A.14: Sensore di prossimità visto dall'esterno di uno smartphone

In realtà il mondo dei sensori di prossimità non si limita ai soli telefoni cellulari ma ne esistono diversi modelli realizzati con diversi tipi di tecnologie:

- sensori induttivi
- sensori capacitivi
- sensori ottici
- sensori ad ultrasuoni
- sensori magnetici

Quelli induttivi sono costituiti da un circuito oscillante in cui quale una bobina genera un campo elettromagnetico ad alta frequenza che induce correnti parassite in azionatori metallici. Questi tipi di sensore rilevano solo oggetti di materiale ferromagnetico e hanno una portata di pochi millimetri. La presenza di tali oggetti interferisce nel campo magnetico provocando una variazione di riluttanza fino a causare la commutazione dell'uscita.
I sensori di prossimità capacitivi hanno una portata più elevata (1 o 2 cm) e sfruttano la capacità elettrica di un condensatore. Le armature del condensatore sono costituite dal lato sensibile del sensore e dal bordo dell'oggetto conduttore rilevato. Una volta rilevata la capacità tra le armature, i circuiti interni commuteranno l'uscita.
Gli ottici sono molto economici e emettono fasci luminosi di infrarossi cercando poi di rilevare un eventuale riflessione sull'oggetto colpito. Tale radiazione viene difficilmente disturbata dalle fonti di luce presenti nell'ambiente anche se sarebbe meglio cercare di evitare forti proiezioni sul riflettore catadiotrico del sensore per evitare il rischio di danneggiarlo. Il campo sensibile cresce enormemente rispetto

Figura A.15: Sensori di prossimità di tutti i tipi

agli induttivi e ai capacitivi e può raggiungere l'ordine dei metri (da 1 a 50) ma può causare qualche problema per distanze troppo ravvicinate (da 0 a 10 cm).
I sensori a ultrasuoni emettono impulsi sonori a frequenze da 40 a 200 kHz e, come quelli ottici, sfruttano la riflessione di tali onde sull'oggetto, in particolare l'eco. Sapendo che il suono viaggia approssimativamente a 340 m/s si possono implementare algoritmi software che, misurando il tempo di andata e ritorno, possono calcolare la distanza come prodotto tra velocità e tempo. Sono immuni ai disturbi elettromagnetici, hanno una portata di circa 10 metri e la loro applicazione pratica più frequente è fungere da sensori di parcheggio delle automobili.
Infine, l'ultima categoria comprende quelli magnetici, i quali necessitano che l'oggetto da rilevare generi un campo magnetico (in genere attraverso un magnete installato su esso). La portata è proporzionale all'intensità del campo e il funzionamento di tale sensore può essere fortemente compromesso da fonti di disturbo elettromagnetiche nelle vicinanze. Esempi di applicazioni: determinare la posizione del pistone magnetico in un cilindro pneumatico, sostituire interruttori meccanici a leva o a pulsante, ecc...

A.4 Magnetometro

Il magnetometro è un sensore di posizione che permette di avere una vera e propria bussola sul proprio smartphone. Quando usiamo l'applicazione Navigatore vediamo che la mappa e il cursore, che rappresenta la nostra posizione, ruotano e si muovono a seconda dei nostri spostamenti.

Figura A.16: Magnetometro

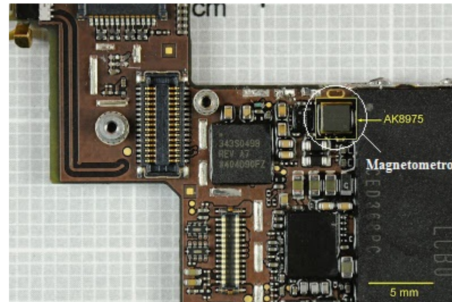

Figura A.17: Locazione di un magnetometro dentro ad uno smartphone

I sensori di campo magnetico possono operare in diversi modi a seconda della loro architettura o dal produttore. Possono sfruttare l'effetto Hall, tramite le proprietà dei materiali magnetoresistivi, oppure la forza di Lorenz. In fisica, in particolare in elettromagnetismo, l'effetto Hall è la formazione di una differenza di potenziale, detto potenziale di Hall, sulle facce opposte di un conduttore elettrico dovuta a un campo magnetico perpendicolare alla corrente elettrica che scorre in esso. I magnetometri basati sull'effetto Hall attualmente sono i più diffusi nel

mercato e, come già spiegato, lavorano semplicemente facendo passare della corrente attraverso un filo. I sensori che optano per la forza di Lorenz sono simili ma misurano una deformazione meccanica del filo piuttosto che una tensione tra i capi.
Indipendentemente dal meccanismo fisico con cui funzionano, i magnetometri misurano il campo magnetico in x, y e z (avendo tre sensori separati, allineati ognuno lungo un asse). Chi ha avuto modo di utilizzare un magnetometro potrà notare che la lettura dei valori è meno precisa e meno stabile rispetto a tutti gli altri sensori già visti. Il settore dei magnetometri senza dubbio continuerà a fare progressi da questo punto di vista, infatti, l'eliminazione del rumore, la sensibilità, la precisione e il basso prezzo rappresentano per la MEMS (Micro Electro-Mechanical Systems) problemi ancora insoluti.

Figura A.18: Trasformare il telefono cellulare in un navigatore con un magnetometro

I campi magnetici sono misurati in microtesla (μT). Il tipico range di valori va da 0 a 2000 μT, la risoluzione del sensore è di 0,1 μT e il campo magnetico terrestre varia da 30 a 60 μT. Tuttavia, come già detto, il valore assoluto è irrilevante, infatti tutte le misurazioni sono stime la cui precisione dipende fortemente dall'ambiente circostante (vicinanza ad altri metalli, effetti isteretici, la tendenza dei valori a cambiare nel tempo, ecc...).

A.5 Sensore di luminosità

Il sensore di luminosità è un sensore ambientale spesso visibile sulla faccia del dispositivo, sotto una piccola apertura sul vetro. È semplicemente un fotodiodo, che opera sullo stesso principio fisico di un LED (light-emitting diode), ma in senso inverso cioè invece di generare luce quando viene applicata una tensione, genera una tensione quando la luce è incidente su di esso. Il sensore di luce riporta i valori

in lux, e ha un range dinamico tipico tra 1 e 30.000 lux con una risoluzione di 1 lux.

Figura A.19: Sensore di luminosità

Un valore all'incirca di 0,25 lux rappresenta la luminosità indiretta percepita in una serata di luna piena, cioè l'ambiente è abbastanza luminoso per consentire all'occhio umano di vedere ma una macchina fotografica standard senza flash non riuscirebbe a catturare abbastanza luce per scattare una fotografia. In una giornata nuvolosa vengono misurati circa 10.000 lux, in caso di piena luce diurna (sole indiretto) si arriva a circa 20.000 lux fino a raggiungere circa 110.000 lux con la luce del sole riflessa direttamente sul sensore. Questi valori abbracciano una vasta gamma e non si può definire con precisione un valore per indicare, ad esempio, "un giorno nuvoloso" (che può variare in luminosità a seconda dello spessore della copertura nuvolosa, l'altezza del sole nel cielo e così via).
Tuttavia, i seguenti numeri possono dare l'idea:

- assenza di luna: 0.001 lux
- luna piena: 0.25 lux
- molto nuvoloso: 100 lux
- crepuscolo: 400 lux
- leggermente nuvoloso: 10000 lux
- ombroso: 20000 lux
- soleggiato: 110000 lux
- fortemente soleggiato: 120000 lux

Il sensore di luce è in gran parte utilizzato per regolare la luminosità dello schermo in base alla luce ambientale. La luminosità dello schermo è una proprietà gestita dal sistema operativo e dalle impostazioni di Android e in genere non è qualcosa che gli sviluppatori necessitano di accedere. Alcuni dispositivi in passato non avevano un sensore di prossimità, e quindi alcuni sviluppatori hanno scritto programmi per utilizzare il sensore di luminosità come un sensore di prossimità allo scopo di bloccare il display durante le chiamate.

Figura A.20: Nel Samsung Galaxy S3 i sensori di luminosità e di prossimità sono un unico sensore

A.6 Barometro

Il barometro è un sensore ambientale che misura la pressione dell'aria.

Figura A.21: Barometro

Figura A.22: Barometro: chip hardware

Il suo impiego primario è quello di determinare l'altitudine in luoghi dove il GPS non può arrivare, come ad esempio la posizione all'interno di un edificio.
Questo sensore, a differenza degli altri, non è disponibile su tutti i dispositivi mobile.
Fisicamente appare come una "drum skin" (pelle di tamburo) avvolta in una camera avente una pressione nota all'interno. A seconda delle variazioni di pressione esterne, la pelle di tamburo tende a deformarsi verso l'interno o verso l'esterno. Tutto ciò è legato alla densità dell'aria, la quale a sua volta è legata alla pressione atmosferica, riferita a una data temperatura.
In condizioni normali, la pressione può subire variazioni di circa 0,5 millibar (mBar) nell'arco di un ora, ma nel caso di arrivo di una tempesta intensa tali variazioni possono arrivare fino a 1 mBar. I cicli di pressione consistono in innalzamenti e abbassamenti circa due volte al giorno a causa di maree atmosferiche e altre cause, quali i cambiamenti di temperatura.

Altitudine assoluta. A livello software, l'altitudine può essere calcolata in base alla pressione misurata p e alla pressione al livello del mare p_0:

$$h(p_0, p) = \frac{T_0}{L}\left(1 - \left(\frac{p}{p_0}\right)^{\frac{\mathrm{RL}}{\mathrm{gM}}}\right) = 44330 * \left(1 - \left(\frac{p}{p_0}\right)^{\frac{1}{5.255}}\right)$$

in questa equazione, h è l'altitudine, T_0 è la temperatura standard a livello del mare, L rappresenta il gradiente termico verticale (lapse rate) ovvero il tasso con cui la temperatura cambia al variare della quota, R è la costante universale dei gas, g è l'accelerazione di gravità e M è il peso molecolare medio dell'aria secca. Tutte le costanti sopracitate sono riportate coi relativi valori nella parte destra dell'equazione.

Altitudine relativa. Utilizzando la formula precedente, è possibile calcolare le differenze di altitudine relative come i dislivelli tra i piani di un centro commerciale. Poiché la pressione non rimane costante nel tempo, l'applicazione software dovrebbe cercare le differenze relative che si verificano nel corso di un breve, ma sufficiente, lasso di tempo (come il tempo che impiegherebbe una persona a salire una rampa di scale). È importante prima calcolare le altitudini e poi sottrarle per ottenere le differenze relative, piuttosto che cercare di confrontare le pressioni. Questo metodo funziona abbastanza bene anche se purtroppo le quote assolute non sono precise al 100%. Per esempio, ad una variazione di pressione di 1010-1011 mBar corrisponde un calo di altitudine di 8,34 m, e invertendo la formula, ad un dislivello di 10 m corrisponde una variazione di pressione di 1,2 mBar a livello del mare.

Nonostante la pressione sia quasi sempre riportata in millibar (mBar), può capitare di imbattersi in unità di misura: 1 mbar (millibar) = 0.001 bar = 0.1 kPa (kilopascal) = 1 hPa (ettopascal) = 1.000 dyn / cm2 (dyne per cm quadrato) = 0,000987 atm (atmosfere) = 0,0295 inHg (pollici di mercurio) = 0,750 mmHg (millimetri di mercurio) = 0,0145 psi (libbre per pollice quadrato).
Una range tipico per un sensore di pressione per smartphone è 300-1100 mBar con una risoluzione di 0,01 mBar.
Anche se il GPS fornisce i dati di altitudine, il segnale non è sempre disponibile. Inoltre il GPS si limita a dirci dentro quale edificio ci troviamo mentre il sensore di pressione riesce a determinare il piano misurando le differenze di pressione (tipicamente sono dell'ordine di 0,3-0,4 mBar le differenze tra piani di un tipico palazzo residenziali e mentre sono più grandi per edifici come centri commerciali; il tutto è calcolabile con la formula citata in precedenza).
A seconda della posizione, la pressione può variare nel corso di un anno da circa 995-1030 mBar con un valore medio di 1013 mbar. Tuttavia, la pressione di solito cambia di meno di 1 mbar nel corso di un'ora.

Bibliografia

[1] Greg Milette - Adam Stroud:
Android Sensor Programming
Wrox Programmer to Programmer

[2] Carlo Antenucci:
Android Sensors Support

[3] Massimo Carli:
Sviluppare applicazioni per Android
Apogeo

[4] Halliday - Resnick - Krane:
Fisica 1
(5a edizione) Casa Editrice Ambrosiana

[5] Halliday - Resnick - Krane:
Fisica 2
(5a edizione) Casa Editrice Ambrosiana

[6] http://developer.android.com

[7] http://www.kionix.com/