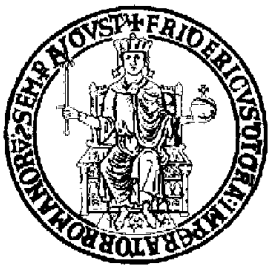
Università degli Studi di Napoli Federico II



Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Classe di Laurea in Ingegneria dell'Informazione, Classe n. L.8

Corso di Laurea in Ingegneria Elettronica

Elaborato di Laurea

ANDROID NDK:
SVILUPPO DI APPLICAZIONI NATIVE CON ANDROID NDK

ANDROID NDK:
DEVELOPMENT OF NATIVE APPLICATIONS WITH ANDROID NDK

Relatore:
Ch.mo Prof. PORFIRIO TRAMONTANA

Candidato:
GIOVANNI CAMPOLATTANO
Matr. N43000980

Anno Accademico
2016/2017

INTRODUZIONE…….3-4

CAPITOLO I

STORIA ED ARCHITETTURA DI ANDROID

1. Un po' di storia……..…....5
2. L'architettura di Android…….…….....6-11

CAPITOLO II

ANDROID NATIVE DEVELOPMENT KIT

1. Prestazioni NDK……...……12
2. Aspetti costitutivi dell'Android NDK…….…....……13-14
3. Componenti di un'applicazione…….…….....15-16

CAPITOLO III

SENSORI DEI DISPOSITIVI MOBILI: L'ACCELEROMETRO

1. Sensori hardware e sensori software……17-19
2. Sistema di coordinate dei sensori……..……....20
3. Accelerometro……...21

CAPITOLO IV

SVILUPPO DI UN APPLICAZIONE NATIVA: SENSOR-GRAPH

1. Introduzione all'applicazione……22
2. File manifest……..23
3. AccelerometerGraphActivity.java…….....24
   3.1. AccelerometerGraphJNI.java…….25
4. Codice cpp……...……26-30
5. Risultati e considerazioni finali…..…….....31-34

BIBLIOGRAFIA……...35

SITOGRAFIA……35

INTRODUZIONE

Il seguente lavoro di tesi descrive l'analisi dell'aspetto e dei contenuti dell'Android *Native Development Kit.* Le motivazioni che mi hanno spinto ad approfondire tale tematica hanno una duplice natura: l'interesse nei confronti dei linguaggi di programmazione, maturato durante l'esperienza universitaria e, in particolare, la volontà di scoprire le potenzialità dei dispositivi mobili che, al giorno d'oggi, ci permettono di compiere molte operazioni che prima erano eseguibili solo attraverso un normale Computer. Per sfruttare al meglio le potenzialità dei dispositivi mobili sono stati introdotti vari sistemi operativi come: Android, Ios e Windows Phone. Attualmente il sistema operativo per dispositivi mobili più diffuso ed utilizzato è Android.

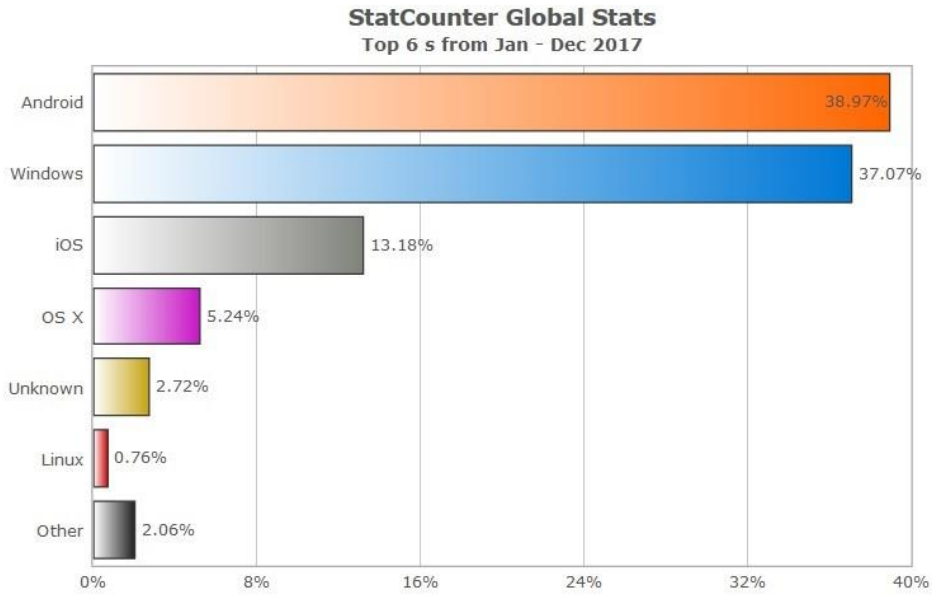


Figura 1- sistemi operativi più diffusi

Nel primo capitolo analizzo dettagliatamente l'architettura Android, per iniziare a familiarizzare con tale sistema operativo, per poi, nel capitolo successivo, introdurre e comprendere a fondo il kit di sviluppo nativo Android e la struttura di tali applicazioni. Dopo aver analizzato questi aspetti delle applicazioni Android, nel terzo capitolo descrivo i sensori disponibili sui dispositivi mobili,

per comprendere le tipologie, le caratteristiche e i metodi con cui essi vengono implementati; tale capitolo ha lo scopo di poter concludere il lavoro di tesi con l'analisi dello sviluppo dell'applicazione Sensor-Graph. Infatti, nel terzo capitolo analizzo, in modo particolare, l'accelerometro, per poi collegarlo al capitolo finale nel quale presento il codice di questa applicazione. Essa consiste in una rappresentazione grafica che, in tempo reale, gestisce le variazioni d'accelerazione generate dal movimento di un dispositivo mobile. Ho scelto di analizzare lo sviluppo di tale applicazione, in quanto essa mette in evidenza la relazione esistente tra il codice C/C++ ed il codice Java e, inoltre, dimostra l'utilizzo del kit di sviluppo nativo di Android per realizzare applicazioni riguardanti la gestione dei sensori di un dispositivo mobile.

CAPITOLO I

STORIA ED ARCHITETTURA DI ANDROID

1. Un po' di storia [3]

Per analizzare l'NDK e comprenderne i vantaggi è bene introdurre il sistema operativo Android, citando i passi storici principali e l'architettura. Android inizialmente fu sviluppato nel 2003 da una piccola azienda californiana di nome Android Inc., fondata da Andy Rubin, Rich Miner, Nick Sears e Chris White. Nel 2005 l'azienda venne acquistata dalla Google Inc., diventando la Google Mobile Division. Nel 2007 venne fondata la Open Handset Alliance (OHA), un consorzio comprendente 84 membri tra cui produttori di hardware e di software, con lo scopo di realizzare tutto il necessario per la diffusione del sistema. Qualche giorno dopo verrà rilasciato anche il primo Software Development Kit (SDK) per gli sviluppatori, che include: gli strumenti di sviluppo, le librerie, un emulatore del dispositivo, la documentazione, tutorial e altro. Nel 2008 venne lanciata la versione Android 1.0 eseguibile solamente sul dispositivo HTC G1 e, in pochi anni, si è affermato come uno dei sistemi operativi per dispositivi mobili più diffuso al mondo, con la creazione di varie versioni, fino alla versione rilasciata nel 2017: *Android 8 Oreo.*

Figura 2- simbolo Android 8 Oreo

2. L’architettura di Android [3]

Android è un insieme di componenti software, comprendente un sistema operativo, un *middleware* e un insieme di applicazioni basilari, utilizzato in smartphones, tablets, smart watches, ma potrebbe essere esteso a supportare qualsiasi dispositivo, ipoteticamente anche un PC. Android è compatibile con un numero elevato di dispositivi e sistemi, ed è un sistema open source. Con il termine open si vogliono intendere diverse cose:

- È open in quanto il suo codice è liberamente consultabile, migliorabile, ma soprattutto non è necessario pagare nessuna royalty per utilizzarlo sui dispositivi;
- È open in quanto le librerie e le *application programming interface* (*API*) utilizzate per la sua realizzazione sono le stesse che possono essere sfruttate per la creazione di nostre applicazioni;
- È open in quanto si possono utilizzare diversi linguaggi di programmazione, anche se il più usato e supportato è Java.

Esso è uno *stack* software, cioè un set di sottosistemi software, basato su *kernel Linux* e che è composto da applicazioni Java eseguite su uno speciale *framework*, anch’esso basato su Java e orientato agli oggetti, a sua volta eseguito su un nucleo costituito da librerie Java eseguite tramite una macchina virtuale, *Dalvik*. L’architettura di Android è di tipo gerarchico, con una complessità crescente dal basso verso l’alto. Tra i vari strati abbiamo un sistema operativo, un insieme di librerie native per le funzionalità core della piattaforma, una implementazione della *Virtual Machine* ed un insieme di librerie Java.

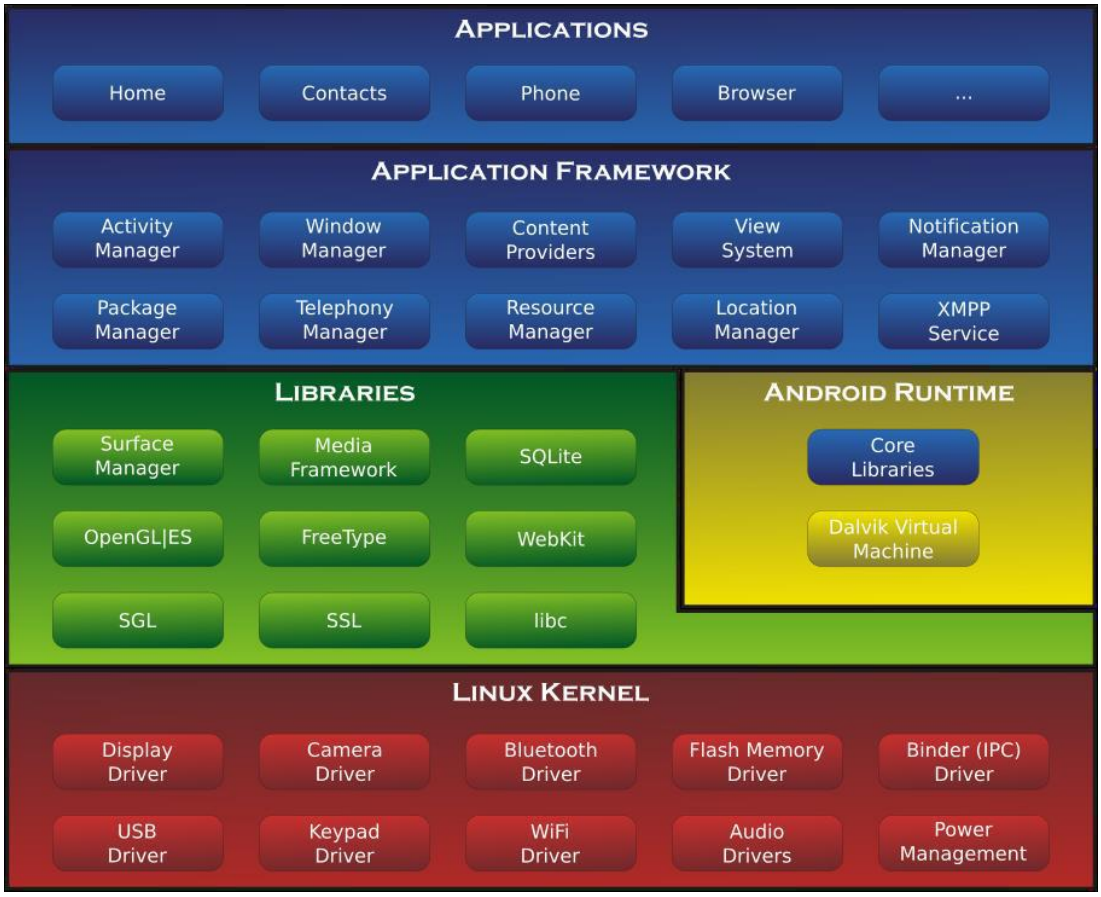


Figura 3-architettura Android

Dalla figura notiamo i seguenti strati:

- *Linux Kernel*

Alla base dello *stack* Android troviamo un *kernel Linux*. La scelta di una simile configurazione è nata dalla necessità di disporre di un vero e proprio sistema operativo che fornisse gli strumenti di basso livello per la virtualizzazione dell'hardware sottostante attraverso l'utilizzo di diversi driver. Esso agisce da strato di astrazione fra l'hardware sottostante (che comprende il GPS, la fotocamera, il touchscreen, il wifi) e il resto dello *stack* software. A differenza di un *kernel Linux* standard per Android sono stati aggiunti ulteriori moduli come:

- ✓ *Binder IPC Driver*: è un driver dedicato che permette la comunicazione tra processi con un costo computazionale minore e un relativo minore consumo di batteria;

- ✓ *Low Memory Killer*: è un modulo che si preoccupa di terminare i processi liberando così spazio nella memoria centrale per soddisfare le richieste di altri processi. Mediante un sistema di ranking vengono terminati i processi con punteggio più alto; ad esempio, un processo che controlla la *User Interface* (*UI*) di un'applicazione visibile sarà sicuramente più basso di quello relativo ad un'applicazione non visibile sullo schermo;
- ✓ *Ashmem*: è un sistema di memoria condiviso anonimo (*Anonimous Shared Memory*) che definisce interfacce che consentono ai processi di condividere zone di memoria attraverso un nome. Il vantaggio è che viene fornito al *kernel* uno strumento per recuperare dei blocchi di memoria non utilizzati;
- ✓ *RAM Console e Log devices*: è un modulo che serve per agevolare il *debugging*. Android fornisce la capacità di memorizzare i messaggi di *log* generati dal *kernel* in un *buffer RAM*;
- ✓ *Android Debug Bridge*: è uno strumento che permette di gestire in maniera versatile un'istanza dell'emulatore o eventualmente di un dispositivo reale;
- ✓ *Power Management*: è una sezione che permette alla CPU di adattare il proprio funzionamento al fine di non consumare energia se nessuna applicazione ne fa richiesta.

- *Android Runtime*

Le *Core Libraries* insieme alla *Dalvik Virtual Machine* (*DVM*) appartengono all'*Android Runtime* cioè una parte dello *stack* Android dedicata all'esecuzione delle applicazioni. Le *Core Libraries* di fatto mettono a disposizione la maggior parte delle funzionalità del linguaggio di programmazione Java: strutture dati, *utility*, librerie di rete, etc. Un programma Java per essere eseguito necessita di una *Virtual Machine* (*JVM*) in grado di interpretare il *bytecode* (file di estensione *.jar*) ed eseguirlo su ogni macchina. In realtà la *Virtual Machine* utilizzata per Android è la *DVM* in

grado di eseguire il codice contenuto all'interno di un file di estensione *.dex*, ottenuto a partire dal *bytecode* Java. Questa scelta è dovuta al fatto che vi è un risparmio di spazio, in quanto un file *.dex* ha dimensioni nettamente inferiori rispetto ad un *.jar*. Inoltre, la *DVM* è ottimizzata per macchine con risorse ridotte e quindi risulta ad-hoc per i dispositivi mobili. Una caratteristica molto importante della *DVM* è quella di consentire un'efficace esecuzione di più processi contemporaneamente; infatti, ciascuna applicazione è in esecuzione nel proprio processo *Linux* e questo comporta dei vantaggi dal punto di vista delle performance. In generale, è possibile eseguire applicazioni Android sia su dispositivi reali che virtuali. In entrambi i casi l'accesso alla macchina è operato tramite *l'utility ADB* (*Android Debug Bridge*) che consente di installare, eseguire e monitorare l'esecuzione di applicazioni. L'utilizzo di dispositivi reali consente una esecuzione più fedele e completa dell'applicazione. Tuttavia, grazie agli emulatori Android, è possibile testare le applicazioni che si vogliono sviluppare in varie condizioni e risoluzioni senza richiedere un dispositivo costoso.

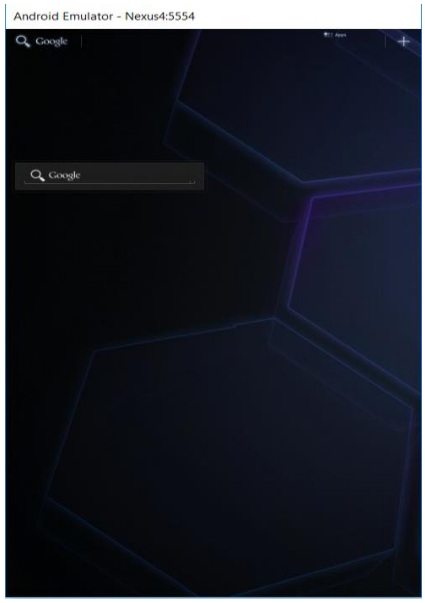


Figura 4-emulatore Android

- *Libraries*

Le librerie hanno un'importantissima caratteristica: non sono sviluppate in Java ma perlopiù in C/C++. Ebbene sì, le applicazioni Android si sviluppano principalmente in Java ma la maggior parte del codice che verrà eseguito sarà codice nativo in C/C++. Questo per il semplice fatto che molti dei componenti della piattaforma sono semplicemente delle interfacce Java di componenti nativi. In particolare, tra di esse troviamo:

- ✓ *Surface Manager (SM)*: gestisce le componenti dell'interfaccia grafica, controllando le diverse finestre visibili a schermo e impedendo, ad esempio, la sovrapposizione disordinata in caso di più finestre aperte contemporaneamente;
- ✓ *OpenGL ES e Scalable Graphics Library (SGL)*: gestiscono rispettivamente la grafica 3D e 2D all'interno di una stessa applicazione;
- ✓ *Media Framework*: contiene le librerie necessarie per gestire vari formati audio e video quali MPEG4, MP3, JPG, e PNG, etc;
- ✓ *FreeType*: è utilizzato per gestire i caratteri;
- ✓ *SQLite*: è una libreria software scritta in C che permette di registrare i dati sul dispositivo;
- ✓ WebKit: è un *browser-engine*, open source, che può essere integrato in varie applicazioni sotto forma di finestra browser;
- ✓ *SSL*: gestisce i *Secure Socket Layer* e quindi tutti i problemi legati alla sicurezza;
- ✓ *Libc*: rappresenta l'implementazione della libreria standard C, ottimizzata per i dispositivi basati su versioni *Linux embedded*.

- *Application Framework*

Lo strato al di sopra delle librerie viste finora è composto da un insieme di componenti di alto livello che costituiscono l'*Application Framework.* Esso mette a disposizione degli sviluppatori componenti riutilizzabili per lo sviluppo delle proprie applicazioni. In questa sezione sono presenti i seguenti moduli:

- ✓ *Activity Manager*: gestisce il ciclo di vita delle *activity*, entità associate ad una schermata, che rappresentano l'interfaccia verso l'utente. Il compito dell'*Activity Manager* è quello di organizzare le varie *activity* sul display in uno *stack*, in base all'ordine di visualizzazione sullo schermo;
- ✓ *Package Manager*: gestisce il ciclo di vita delle applicazioni sul dispositivo;
- ✓ *Telephony Manager*: gestisce l'interazione con le funzioni tipiche di un cellulare;
- ✓ *Content Provider*: gestisce la condivisione di informazioni tra i vari processi attivi, è una sorta di cartella condivisa;
- ✓ *Resource Manager*: gestisce le informazioni di un'applicazione con l'obiettivo di ottimizzare le risorse;
- ✓ *View System*: gestisce l'insieme delle opzioni utilizzate nella costruzione dell'interfaccia verso l'utente (bottoni, griglie, etc);
- ✓ *Location Manager*: gestisce una serie di *API* relativi alla localizzazione;
- ✓ *Notification* Manager: gestisce le informazioni di notifica tra il dispositivo e l'utente.

- *Applications*

Il livello più alto dello *stack* è costituito dalle applicazioni sia native (gestione dei contatti, calendario, calcolatrice, etc) che provenienti da terze parti, in quanto Android non differenzia le due tipologie di applicazioni, garantendo gli stessi privilegi ad entrambe.

CAPITOLO II

ANDROID NATIVE DEVELOPMENT KIT

1. Prestazioni NDK

I dispositivi mobili non possono tollerare la presenza di applicazioni lente, in quanto l'utente si attende da essi risposte immediate e, per ottenere dispositivi mobili con prestazioni sempre più elevate, servono tecniche per sviluppare programmi in grado di fornire tali prestazioni. La prima tecnica da utilizzare consiste nel programmare la parte grafica, se essa costituisce un aspetto rilevante nell'applicazione, tramite librerie OpenGL le cui istruzioni possono essere elaborate dalla GPU del dispositivo, mentre la CPU elabora il *core* del programma, parallelizzando il lavoro del dispositivo. Ovviamente, bisogna tener conto del fatto che, sfruttando in parallelo per lungo tempo GPU e CPU, si avrà un elevato consumo della batteria, indipendentemente dalla qualità del dispositivo mobile. Ci sono poi una serie di operazioni per le quali Java risulta poco performante e, per ovviare a tale problema, è stato rilasciato un pacchetto aggiuntivo opzionale, l'*NDK (Native Development Kit)* che consente la programmazione di porzioni di codice in linguaggio nativo, utilizzando C++ ed altri linguaggi orientati al miglioramento delle prestazioni del dispositivo. Dunque, *NDK* è utile per i casi in cui è necessario eseguire una o più delle seguenti operazioni:

- Spremere le prestazioni extra da un dispositivo per ottenere una bassa latenza o eseguire applicazioni ad alta intensità di calcolo, come giochi o simulazioni fisiche;
- Riutilizzare le proprie librerie C/C ++, andando ad aumentare la velocità, perché le librerie di base di C/C++ sono più veloci in quanto, un eseguibile in C++ è eseguibile direttamente sul processore del dispositivo mobile, invece, un eseguibile in java è eseguibile su di una macchina virtuale che, a sua volta, si contende le risorse con il resto dei processi

che, a loro volta, sono degli eseguibili che girano sul processore. Quindi praticamente è più lenta;

- Ridurre notevolmente il processo di sviluppo del progetto per applicazioni con molte righe di codice C/C++.

2. Aspetti costitutivi dell'Android NDK [1],[2]

Si può, dunque, introdurre il *Native Development Kit* (*NDK*), cioè un set di strumenti che consente di utilizzare il codice C/C ++ con Android e fornisce librerie di piattaforme che è possibile utilizzare per gestire attività native e accedere ai componenti fisici del dispositivo, come sensori e input tattili. Utilizzando Android Studio 2.2 e versioni successive, si può utilizzare *NDK* per compilare il codice C/C ++ in una libreria nativa e inserirlo nel proprio *APK* (file Android *package*) utilizzando *Gradle*, il sistema di *build* integrato dell'*IDE*. Il codice Java può quindi chiamare le funzioni contenute nelle librerie native attraverso il *framework JNI* (*Java Native Interface*). L'*NDK* di Android punta ad iniettare codice ad alte prestazioni sul dispositivo mobile sfruttandone la velocità massima. Consente di scrivere il codice velocemente per attività intensive e di esportarlo su Android. Esso è uno dei più efficienti sistemi operativi per multimedia e giochi.

3. Componenti di un'applicazione

Per realizzare un'applicazione Android bisogna fondamentalmente basarsi sui seguenti elementi:

- ✓ *Activity*: rappresenta i blocchi logici dell'applicazione ed è responsabile dell'interazione diretta con l'utente mediante i dispositivi di input. Il ciclo di vita di un'*activity* è una struttura a pila dove l'attività in cima è sempre quella attiva.

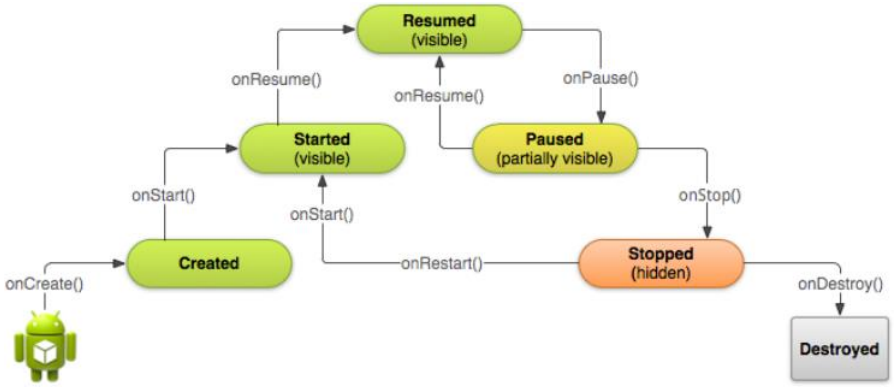


Figura 5- ciclo di vita di un'activity

L'ingresso o l'uscita da uno di questi stati viene notificato con l'invocazione di un metodo di *callback* da parte del sistema. Quando *un'Activity* va in esecuzione, per interagire direttamente con l'utente, vengono obbligatoriamente invocati tre metodi:

- *OnCreate*: l'*activity* viene creata. Il programmatore deve assegnare le configurazioni di base e definire quale sarà il layout dell'interfaccia;
- *OnStart*: l'*activity* diventa visibile. È il momento in cui si possono attivare funzionalità e servizi che devono offrire informazioni all'utente;
- *OnResume*: l'*activity* diventa la destinataria di tutti gli input dell'utente.

Android pone a riposo l'*activity* nel momento in cui l'utente sposta la sua attenzione su un'altra attività del sistema, ad esempio riceve una telefonata o semplicemente viene attivata un'altra *activity*. Anche questo percorso, passa per tre metodi di *callback*:

- *OnPause*: notifica la cessata interazione dell'utente con l'*activity*;

- *OnStop*: segna la fine della visibilità dell'*activity*;
- *OnDestroy*: segna la distruzione dell'*activity*.

- ✓ *Intent*: è un meccanismo di messaggi che può attivare tre dei componenti di base di un'applicazione: *Activity*, *Service* e *Broadcast Receiver*. Esso contiene informazioni per il componente che riceve l'*intent* (come l'azione da svolgere) e per Android (come istruzioni sul lancio dell'attività prescelta). Gli *intent* possono essere espliciti, nei quali si specifica il nome del componente che svolgerà l'attività, o impliciti, nei quali non viene indicato nessun componente;
- ✓ *Broadcast Receiver*: permette di eseguire un codice in seguito ad un evento esterno improvviso, ad esempio una chiamata in arrivo. Non hanno una vera e propria interfaccia grafica, ma avvisano l'utente tramite messaggi utilizzando il metodo NotificationManager();
- ✓ *Service*: svolge un ruolo opposto *all'activity*, infatti rappresenta un lavoro che viene svolto interamente in background senza bisogno di interazione diretta con l'utente. I s*ervice* hanno un'importanza basilare nella programmazione proprio perché spesso preparano i dati che le *activity* devono mostrare all'utente, permettendo una reattività maggiore nel momento della visualizzazione;
- ✓ *Content Provider*: è utilizzato per la condivisione di dati tra applicazioni, permettendo di condividere contenuti custoditi in un database, su file o in rete. Tali contenuti potranno essere usati da altre applicazioni senza invadere lo spazio di memoria.

CAPITOLO III

SENSORI DEI DISPOSITIVI MOBILI: L'ACCELEROMETRO

1. Sensori hardware e sensori software [2]

La maggior parte dei dispositivi Android ha sensori integrati che misurano il movimento, l'orientamento e varie condizioni ambientali. Questi sensori sono in grado di fornire dati con precisione elevata e sono utili se si desidera monitorare il movimento o il posizionamento di un dispositivo tridimensionale o se si desidera monitorare i cambiamenti nell'ambiente circostante al dispositivo. Alcuni di questi sensori sono basati su hardware e alcuni sono basati su software. I sensori basati su hardware sono componenti elettronici fisicamente inseriti in un dispositivo mobile; essi ricavano i loro dati misurando direttamente specifiche proprietà ambientali, quali accelerazione o intensità del campo geomagnetico. I sensori basati su software non sono dispositivi fisici, sebbene imitino i sensori basati su hardware. Essi derivano i dati da uno o più sensori basati sull'hardware e sono talvolta denominati virtuali o sintetici. I sensori supportati dalla piattaforma Android sono di seguito elencati:

- Accelerometro: è un sensore hardware che misura la forza d'accelerazione (m/s^2) applicata ad un dispositivo su tutti e tre gli assi cartesiani, inclusa la forza di gravità. Tra gli usi comuni vi è la rilevazione del movimento (vibrazione, inclinazione, etc);
- Temperatura: è un sensore hardware che misura la temperatura in gradi celsius (°C);
- Gravità: è un sensore hardware o software che misura la forza di gravità (m/s^2) che è applicata ad un dispositivo su tutti e tre gli assi;
- Giroscopio: è un sensore hardware che misura la velocità di rotazione del dispositivo (rad/s) intorno a ciascuno dei tre assi. Tra gli usi comuni abbiamo il rilevamento di una rotazione;

- Luminosità: è un sensore hardware che misura il livello di luminosità ambientale (lux) ed è utilizzato per controllare la luminosità dello schermo;
- Accelerazione lineare: è un sensore hardware o software che misura la forza d' accelerazione (m/s^2) applicata ad un dispositivo su tutti e tre gli assi fisici, esclusa la forza di gravità. Esso monitora l'accelerazione lungo un solo asse;
- Campo magnetico: è un sensore hardware che misura il campo geomagnetico ambientale (μT) in tutti e tre gli assi. È utilizzato per realizzare la bussola;
- Orientamento: è un sensore software che misura i gradi di rotazione che un dispositivo crea attorno a tutti e tre gli assi fisici. Determina la posizione del dispositivo;
- Pressione: è un sensore hardware che misura la pressione dell'ambiente in bar;
- Prossimità: è un sensore hardware che misura la prossimità di un oggetto in cm rispetto allo schermo di visualizzazione del dispositivo. Esso viene in genere utilizzato per determinare se un ricevitore viene tenuto vicino all'orecchio di una persona;
- Umidità: è un sensore hardware che misura la percentuale di umidità nell'ambiente;
- Vettore di rotazione: è un sensore hardware o software che misura l'orientamento di un dispositivo, fornendone i tre elementi del vettore di rotazione.

Pochi dispositivi Android dispongono di ogni tipo di sensore. Ad esempio, la maggior parte dei dispositivi ha un accelerometro e un magnetometro, ma meno dispositivi dispongono di barometri o termometri. Inoltre, un dispositivo può avere più di un sensore di un determinato tipo. Ad esempio, può avere due

sensori di gravità, ciascuno con un intervallo diverso. Di seguito si presentano i metodi principali che si utilizzano per lavorare con i sensori:

- GetSensorList(): serve ad identificare i sensori presenti su un dispositivo;
- GetDefaultSensor(): serve per verificare se esiste un tipo specifico di sensore su un dispositivo;
- GetResolution(): serve per ottenere la risoluzione di un sensore;
- GetMaximumRange(): serve per ottenere l'intervallo di misurazione massimo;
- GetPower(): serve per ottenere i requisiti di alimentazione di un sensore;
- GetMinDelay(): restituisce l'intervallo di tempo minimo (in μs) che un sensore può utilizzare per rilevare i dati.

2. Sistema di coordinate dei sensori [2]

In generale, i sensori di un dispositivo mobile utilizzano un sistema di coordinate a 3 assi per esprimere i valori dei dati. Per la maggior parte dei sensori, il sistema di coordinate è definito in relazione allo schermo del dispositivo quando esso è nell'orientamento predefinito.

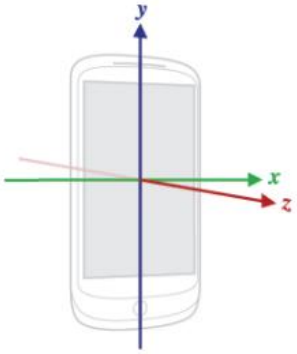


Figura 6- sistema di coordinate dei sensori di un dispositivo mobile

Quando un dispositivo viene tenuto nel suo orientamento predefinito, l'asse X è orizzontale e punta a destra, l'asse Y è verticale e punta verso l'alto e l'asse Z punta verso l'esterno della faccia dello schermo. Questo sistema di coordinate viene utilizzato dai seguenti sensori: accelerazione, gravità, giroscopio, accelerazione lineare e campo geomagnetico. Il punto più importante da comprendere su questo sistema di coordinate è che gli assi non vengono scambiati quando cambia l'orientamento dello schermo del dispositivo, ovvero, il sistema di coordinate del sensore non cambia mai quando il dispositivo si muove. Questo comportamento è uguale al comportamento del sistema di coordinate OpenGL. Un altro punto da capire è che l'applicazione non deve presumere che l'orientamento naturale (predefinito) di un dispositivo sia verticale. L'orientamento naturale per molti dispositivi tablet è orizzontale. E il sistema di coordinate del sensore si basa sempre sull'orientamento naturale di un dispositivo.

3. Accelerometro [2]

L’accelerometro è un sensore di movimento composto da una parte meccanica ed una parte elettronica. Esso sfrutta le variazioni di accelerazione per determinare se ci si sta spostando o meno, in quale direzione ci si sta spostando e altre informazioni riguardanti la posizione e il movimento del dispositivo mobile o di chi lo indossa e lo sta utilizzando. L'accelerometro trova applicazione nei settori più svariati: automobilistico, scientifico, aereospaziale, etc. In particolare, gli accelerometri, trovano spazio all'interno dei laptop per proteggerne il disco rigido. In caso di caduta, infatti, l'accelerometro “capta” il movimento verso il basso e spegne immediatamente l'hard disk, così da interrompere il movimento delle testine ed evitare che i dischi possano riportare danni di varia natura [4]. Fondamentalmente, l’accelerometro attiva il proprio funzionamento sfruttando le proprietà fisiche dei cristalli piezoelettrici o affidandosi a circuiti capacitativi. Può comporsi di elementi che misurano gli spostamenti su due o tre assi. Nel primo caso, i movimenti saranno registrati esclusivamente su un piano orizzontale (o verticale) e dunque bidimensionale; nel secondo caso si avrà la misurazione della variazione di accelerazione tridimensionale e l'accelerometro registra gli spostamenti dell'oggetto in qualunque direzione.

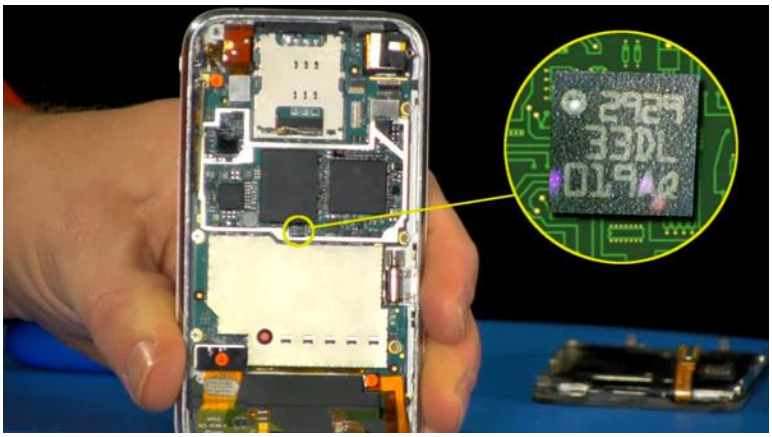


Figura 7- locazione dell’accelerometro in un dispositivo mobile

CAPITOLO IV

SVILUPPO DI UN'APPLICAZIONE NATIVA:
SENSOR-GRAPH [2]

1. Introduzione all'applicazione

Come già detto in precedenza, Android è un sistema open source in quanto si possono utilizzare diversi linguaggi di programmazione che possono interagire fra loro; per tale motivo si analizza ora lo sviluppo di applicazioni Android con codice C++ integrato al codice Java. In particolare, si analizza l'applicazione Sensor-Graph, scritta in java e C++, che legge in tempo reale i valori dell'accelerometro, captando i movimenti di uno smartphone e li rappresenta graficamente tramite OpenGL. Il progetto è contenuto in una cartella denominata accelerometer, che rappresenta il modulo di base e che include quattro file principali: file manifest, cartella con codice java, cartella con codice C++, cartella res (contenente risorse per lo più realizzate in xml). Dopo tale modulo vi è la sezione Gradle Scripts, dove ci sono i file di *build*, che saranno utilizzati da *gradle* per trasformare il progetto in un'applicazione funzionante.

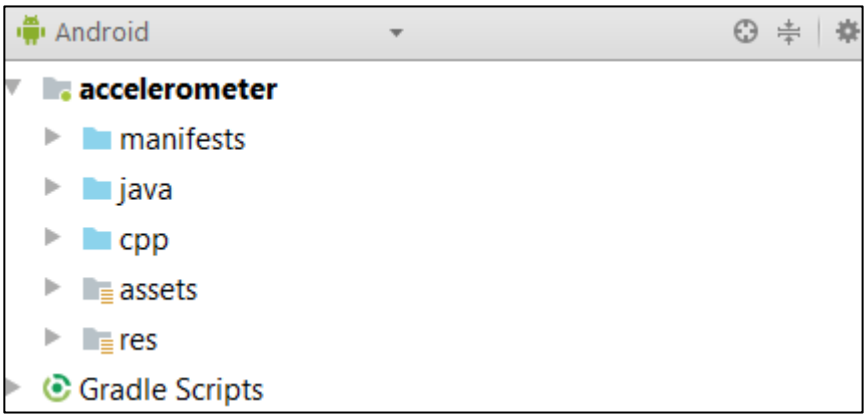


Figura 8- progetto Sensor-Graph

2. File manifest

```
<?xml version="1.0" encoding="utf-8"?>
<!--...-->

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.accelerometergraph">
  <uses-feature android:glEsVersion="0x00020000" android:required="true"/>
    <application
        android:allowBackup="false"
        android:fullBackupContent="false"
        android:icon="@mipmap/ic_launcher"
        android:label="Accelerometer Graph">
      <activity android:name=".AccelerometerGraphActivity"
                android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
                android:launchMode="singleTask"
                android:configChanges="orientation|keyboardHidden">
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
    </application>
</manifest>
```

Figura 9- file manifest

Il file manifest è un file di configurazione dell'applicazione; ogni applicazione deve avere un file manifest nella directory principale in quanto esso fornisce informazioni essenziali dell'applicazione al sistema Android. Questo file descrive i componenti dell'applicazione (*activity*, *intent*, *service*, *broadcast receiver* e *content provider*) ed assegna un nome alle classi che implementano ciascuno dei componenti e stabilisce le loro funzionalità, ad esempio i messaggi che *l'intent* può gestire. La prima cosa da notare è il *tag application*, utilizzato per definire le impostazioni della nostra applicazione. Tra le varie impostazioni configurate abbiamo android:label, cioè un'etichetta leggibile dall'utente, predefinita per ciascun componente dell'applicazione. L'etichetta deve essere impostata come riferimento ad una risorsa stringa, in modo che possa essere localizzata come altre stringhe nell'interfaccia utente. All'interno del *tag application*, troviamo l'*activity*. In questo progetto l'*activity* principale è Accelerometer Graph, in quanto nell'androidmanifest.xml, a tale *activity* è collegato l'*intent-filter*, cioè il costrutto che serve ad indicare che quest'*activity* è la *main activity* del progetto, ossia l'interfaccia che accoglierà l'utente all'ingresso dell'applicazione.

3. AccelerometerGraphActivity.java

```
17  package com.android.accelerometergraph;
18
19  import ...
25  public class AccelerometerGraphActivity extends Activity {
26      GLSurfaceView mView;
27
28      @Override protected void onCreate(Bundle icicle) {
29          super.onCreate(icicle);
30          mView = new GLSurfaceView(getApplication());
31          mView.setEGLContextClientVersion(2);
32          mView.setRenderer(new GLSurfaceView.Renderer() {
33              @Override
34              public void onSurfaceCreated(GL10 gl, EGLConfig config) {
35                  AccelerometerGraphJNI.surfaceCreated();
36              }
37              @Override
38              public void onSurfaceChanged(GL10 gl, int width, int height) {
39                  AccelerometerGraphJNI.surfaceChanged(width, height);
40              }
41              @Override
42              public void onDrawFrame(GL10 gl) { AccelerometerGraphJNI.drawFrame(); }
45          });
46          mView.queueEvent(() -> { AccelerometerGraphJNI.init(getAssets()); });
52          setContentView(mView);
53      }
54
55      @Override protected void onPause() {
56          super.onPause();
57          mView.onPause();
58          mView.queueEvent(() -> { AccelerometerGraphJNI.pause(); });
64      }
65
66      @Override protected void onResume() {
67          super.onResume();
68          mView.onResume();
69          mView.queueEvent(() -> { AccelerometerGraphJNI.resume(); });
75      }
76  }
```

Figura 10- codice java

AccelerometerGraphActivity è una *public class*, cioè è possibile accedere ad essa da qualsiasi classe. Al suo interno viene implementato l'*override* del metodo onCreate, onPause, onResume ecc. L'*override* si utilizza quando si sostituisce un metodo, in quanto permette di trarre vantaggio dal controllo del compilatore, per assicurarsi di aver effettivamente superato il precedente metodo e, inoltre, permette una maggior comprensione del codice. Gli *override* definiscono il ciclo di vita delle *activity*. Alla riga 26 troviamo GLSurfaceView, uno standard industriale per la programmazione grafica 2D/3D sui dispositivi mobili.

3.1. AccelerometerGraphJNI.java

```
package com.android.accelerometergraph;

// Wrapper for native library

import android.content.res.AssetManager;

public class AccelerometerGraphJNI {

    static {
        System.loadLibrary( libname: "accelerometergraph");
    }

    public static native void init(AssetManager assetManager);
    public static native void surfaceCreated();
    public static native void surfaceChanged(int width, int height);
    public static native void drawFrame();
    public static native void pause();
    public static native void resume();
}
```

Figura 11- codice java

Nel secondo file .java si nota la presenza del collegamento tra il codice java ed il codice nativo (C++) tramite *java native interface* (*JNI*), cioè un *framework* del linguaggio Java che gli consente di richiamare il codice nativo. Il codice java può anche essere richiamato dal codice nativo (C++ nel nostro caso).

4. Codice cpp

```cpp
#include <dlfcn.h>
#include <jni.h>
#include <GLES2/gl2.h>

#include <android/log.h>
#include <android/asset_manager_jni.h>
#include <android/sensor.h>

#include <cstdint>
#include <cassert>
#include <string>

#define  LOG_TAG    "accelerometergraph"
#define  LOGI(...)  __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)

const int LOOPER_ID_USER = 3;
const int SENSOR_HISTORY_LENGTH = 100;
const int SENSOR_REFRESH_RATE_HZ = 100;
constexpr int32_t SENSOR_REFRESH_PERIOD_US = int32_t(1000000 / SENSOR_REFRESH_RATE_HZ);
const float SENSOR_FILTER_ALPHA = 0.1f;

#include <dlfcn.h>
const char*  kPackageName = "com.android.accelerometergraph";
ASensorManager* AcquireASensorManagerInstance(void) {
    typedef ASensorManager *(*PF_GETINSTANCEFORPACKAGE)(const char *name);
    void* androidHandle = dlopen("libandroid.so", RTLD_NOW);
    PF_GETINSTANCEFORPACKAGE getInstanceForPackageFunc = (PF_GETINSTANCEFORPACKAGE)
        dlsym(androidHandle, "ASensorManager_getInstanceForPackage");
    if (getInstanceForPackageFunc) {
        return getInstanceForPackageFunc(kPackageName);
    }
    typedef ASensorManager *(*PF_GETINSTANCE)();
    PF_GETINSTANCE getInstanceFunc = (PF_GETINSTANCE)
        dlsym(androidHandle, "ASensorManager_getInstance");
    // by all means at this point, ASensorManager_getInstance should be available
    assert(getInstanceFunc);
    return getInstanceFunc();
}
```

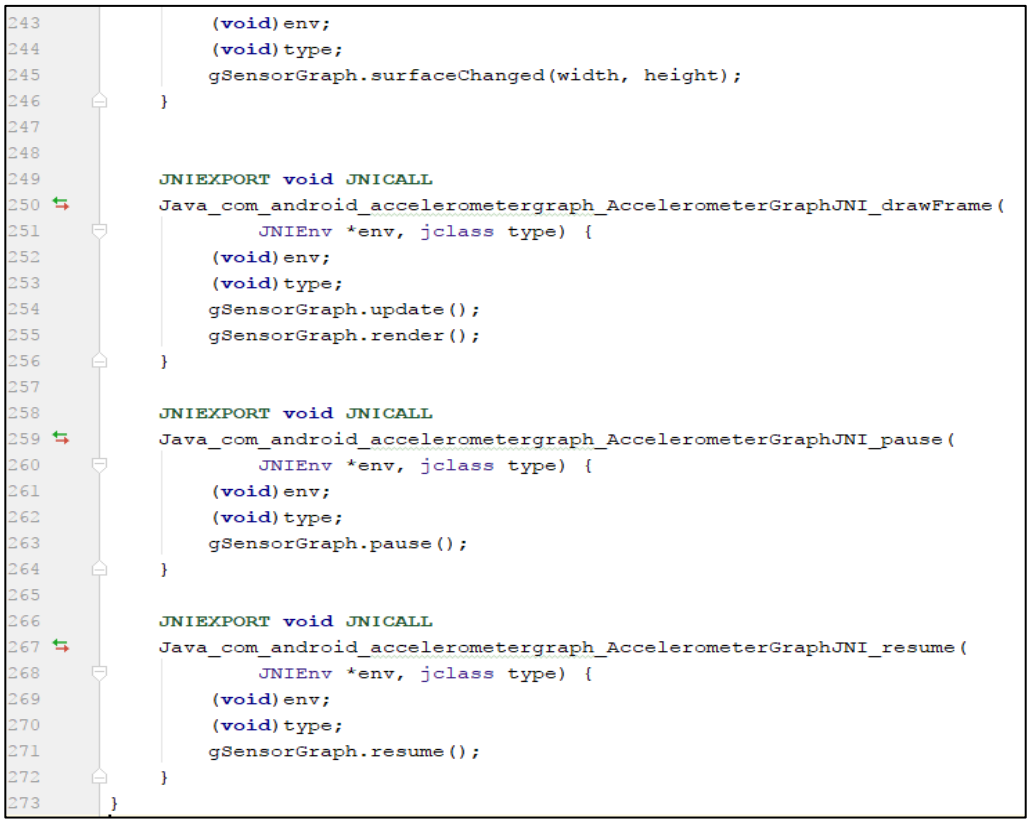
```cpp
class sensorgraph {
    std::string vertexShaderSource;
    std::string fragmentShaderSource;
    ASensorManager *sensorManager;
    const ASensor *accelerometer;
    ASensorEventQueue *accelerometerEventQueue;
    ALooper *looper;

    GLuint shaderProgram;
    GLuint vPositionHandle;
    GLuint vSensorValueHandle;
    GLuint uFragColorHandle;
    GLfloat xPos[SENSOR_HISTORY_LENGTH];

    struct AccelerometerData {
        GLfloat x;
        GLfloat y;
        GLfloat z;
    };
    AccelerometerData sensorData[SENSOR_HISTORY_LENGTH*2];
    AccelerometerData sensorDataFilter;
    int sensorDataIndex;

 public:
    sensorgraph() : sensorDataIndex(0) {}

    void init(AAssetManager *assetManager) {
        AAsset *vertexShaderAsset = AAssetManager_open(assetManager, "shader.glslv",
                                                       AASSET_MODE_BUFFER);
        assert(vertexShaderAsset != NULL);
        const void *vertexShaderBuf = AAsset_getBuffer(vertexShaderAsset);
        assert(vertexShaderBuf != NULL);
        off_t vertexShaderLength = AAsset_getLength(vertexShaderAsset);
        vertexShaderSource = std::string((const char*)vertexShaderBuf,
                                         (size_t)vertexShaderLength);
        AAsset_close(vertexShaderAsset);

        AAsset *fragmentShaderAsset = AAssetManager_open(assetManager, "shader.glslf",
                                                         AASSET_MODE_BUFFER);
        assert(fragmentShaderAsset != NULL);
        const void *fragmentShaderBuf = AAsset_getBuffer(fragmentShaderAsset);
```

```
82          assert(fragmentShaderBuf != NULL);
83          off_t fragmentShaderLength = AAsset_getLength(fragmentShaderAsset);
84          fragmentShaderSource = std::string((const char*)fragmentShaderBuf,
85                                             (size_t)fragmentShaderLength);
86          AAsset_close(fragmentShaderAsset);
87
88          sensorManager = AcquireASensorManagerInstance();
89          assert(sensorManager != NULL);
90          accelerometer = ASensorManager_getDefaultSensor(sensorManager, ASENSOR_TYPE_ACCELEROMETER);
91          assert(accelerometer != NULL);
92          looper = ALooper_prepare(ALOOPER_PREPARE_ALLOW_NON_CALLBACKS);
93          assert(looper != NULL);
94          accelerometerEventQueue = ASensorManager_createEventQueue(sensorManager, looper,
95                                                                    LOOPER_ID_USER, NULL, NULL);
96          assert(accelerometerEventQueue != NULL);
97          auto status = ASensorEventQueue_enableSensor(accelerometerEventQueue,
98                                                       accelerometer);
99          assert(status >= 0);
100         status = ASensorEventQueue_setEventRate(accelerometerEventQueue,
101                                                 accelerometer,
102                                                 SENSOR_REFRESH_PERIOD_US);
103         assert(status >= 0);
104         (void)status;    //to silent unused compiler warning
105
106         generateXPos();
107     }
108
109     void surfaceCreated() {
110         LOGI("GL_VERSION: %s", glGetString(GL_VERSION));
111         LOGI("GL_VENDOR: %s", glGetString(GL_VENDOR));
112         LOGI("GL_RENDERER: %s", glGetString(GL_RENDERER));
113         LOGI("GL_EXTENSIONS: %s", glGetString(GL_EXTENSIONS));
114
115         shaderProgram = createProgram(vertexShaderSource, fragmentShaderSource);
116         assert(shaderProgram != 0);
117         GLint getPositionLocationResult = glGetAttribLocation(shaderProgram, "vPosition");
118         assert(getPositionLocationResult != -1);
119         vPositionHandle = (GLuint)getPositionLocationResult;
120         GLint getSensorValueLocationResult = glGetAttribLocation(shaderProgram, "vSensorValue");
121         assert(getSensorValueLocationResult != -1);
122         vSensorValueHandle = (GLuint)getSensorValueLocationResult;
```

```
123         GLint getFragColorLocationResult = glGetUniformLocation(shaderProgram, "uFragColor");
124         assert(getFragColorLocationResult != -1);
125         uFragColorHandle = (GLuint)getFragColorLocationResult;
126     }
127
128     void surfaceChanged(int w, int h) {
129         glViewport(0, 0, w, h);
130     }
131
132     void generateXPos() {
133         for (auto i = 0; i < SENSOR_HISTORY_LENGTH; i++) {
134             float t = static_cast<float>(i) / static_cast<float>(SENSOR_HISTORY_LENGTH - 1);
135             xPos[i] = -1.f * (1.f - t) + 1.f * t;
136         }
137     }
138
139     GLuint createProgram(const std::string& pVertexSource, const std::string& pFragmentSource) {
140         GLuint vertexShader = loadShader(GL_VERTEX_SHADER, pVertexSource);
141         GLuint pixelShader = loadShader(GL_FRAGMENT_SHADER, pFragmentSource);
142         GLuint program = glCreateProgram();
143         assert(program != 0);
144         glAttachShader(program, vertexShader);
145         glAttachShader(program, pixelShader);
146         glLinkProgram(program);
147         GLint programLinked = 0;
148         glGetProgramiv(program, GL_LINK_STATUS, &programLinked);
149         assert(programLinked != 0);
150         glDeleteShader(vertexShader);
151         glDeleteShader(pixelShader);
152         return program;
153     }
154
155     GLuint loadShader(GLenum shaderType, const std::string& pSource) {
156         GLuint shader = glCreateShader(shaderType);
157         assert(shader != 0);
158         const char *sourceBuf = pSource.c_str();
159         glShaderSource(shader, 1, &sourceBuf, NULL);
160         glCompileShader(shader);
161         GLint shaderCompiled = 0;
162         glGetShaderiv(shader, GL_COMPILE_STATUS, &shaderCompiled);
```

```
        assert(shaderCompiled != 0);
        return shader;
    }

    void update() {
        ALooper_pollAll(0, NULL, NULL, NULL);
        ASensorEvent event;
        float a = SENSOR_FILTER_ALPHA;
        while (ASensorEventQueue_getEvents(accelerometerEventQueue, &event, 1) > 0) {
            sensorDataFilter.x = a * event.acceleration.x + (1.0f - a) * sensorDataFilter.x;
            sensorDataFilter.y = a * event.acceleration.y + (1.0f - a) * sensorDataFilter.y;
            sensorDataFilter.z = a * event.acceleration.z + (1.0f - a) * sensorDataFilter.z;
        }
        sensorData[sensorDataIndex] = sensorDataFilter;
        sensorData[SENSOR_HISTORY_LENGTH+sensorDataIndex] = sensorDataFilter;
        sensorDataIndex = (sensorDataIndex + 1) % SENSOR_HISTORY_LENGTH;
    }

    void render() {
        glClearColor(0.f, 0.f, 0.f, 1.0f);
        glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);

        glUseProgram(shaderProgram);

        glEnableVertexAttribArray(vPositionHandle);
        glVertexAttribPointer(vPositionHandle, 1, GL_FLOAT, GL_FALSE, 0, xPos);

        glEnableVertexAttribArray(vSensorValueHandle);
        glVertexAttribPointer(vSensorValueHandle, 1, GL_FLOAT, GL_FALSE, sizeof(AccelerometerData),
                              &sensorData[sensorDataIndex].x);

        glUniform4f(uFragColorHandle, 1.0f, 1.0f, 0.0f, 1.0f);
        glDrawArrays(GL_LINE_STRIP, 0, SENSOR_HISTORY_LENGTH);

        glVertexAttribPointer(vSensorValueHandle, 1, GL_FLOAT, GL_FALSE, sizeof(AccelerometerData),
                              &sensorData[sensorDataIndex].y);
        glUniform4f(uFragColorHandle, 1.0f, 0.0f, 1.0f, 1.0f);
        glDrawArrays(GL_LINE_STRIP, 0, SENSOR_HISTORY_LENGTH);

        glVertexAttribPointer(vSensorValueHandle, 1, GL_FLOAT, GL_FALSE, sizeof(AccelerometerData),
```

```
                              &sensorData[sensorDataIndex].z);
        glUniform4f(uFragColorHandle, 0.0f, 1.0f, 1.0f, 1.0f);
        glDrawArrays(GL_LINE_STRIP, 0, SENSOR_HISTORY_LENGTH);
    }

    void pause() {
        ASensorEventQueue_disableSensor(accelerometerEventQueue, accelerometer);
    }

    void resume() {
        ASensorEventQueue_enableSensor(accelerometerEventQueue, accelerometer);
        auto status = ASensorEventQueue_setEventRate(accelerometerEventQueue,
                                                     accelerometer,
                                                     SENSOR_REFRESH_PERIOD_US);
        assert(status >= 0);
    }
};


sensorgraph gSensorGraph;

extern "C" {
JNIEXPORT void JNICALL
    Java_com_android_accelerometergraph_AccelerometerGraphJNI_init(
                      JNIEnv *env, jclass type, jobject assetManager) {
        (void)type;
        AAssetManager *nativeAssetManager = AAssetManager_fromJava(env, assetManager);
        gSensorGraph.init(nativeAssetManager);
    }

    JNIEXPORT void JNICALL
    Java_com_android_accelerometergraph_AccelerometerGraphJNI_surfaceCreated(JNIEnv *env, jclass type) {
        (void)env;
        (void)type;
        gSensorGraph.surfaceCreated();
    }

    JNIEXPORT void JNICALL
    Java_com_android_accelerometergraph_AccelerometerGraphJNI_surfaceChanged(
            JNIEnv *env, jclass type, jint width, jint height) {
```



```
243             (void)env;
244             (void)type;
245             gSensorGraph.surfaceChanged(width, height);
246         }
247
248
249         JNIEXPORT void JNICALL
250         Java_com_android_accelerometergraph_AccelerometerGraphJNI_drawFrame(
251                 JNIEnv *env, jclass type) {
252             (void)env;
253             (void)type;
254             gSensorGraph.update();
255             gSensorGraph.render();
256         }
257
258         JNIEXPORT void JNICALL
259         Java_com_android_accelerometergraph_AccelerometerGraphJNI_pause(
260                 JNIEnv *env, jclass type) {
261             (void)env;
262             (void)type;
263             gSensorGraph.pause();
264         }
265
266         JNIEXPORT void JNICALL
267         Java_com_android_accelerometergraph_AccelerometerGraphJNI_resume(
268                 JNIEnv *env, jclass type) {
269             (void)env;
270             (void)type;
271             gSensorGraph.resume();
272         }
273     }
```

Figura 12- codice C++

Questo è il codice C++ che si collega, tramite *JNI*, al codice java per realizzare l'applicazione. La prima cosa da notare è l'inclusione di alcune librerie (.h), come dlfcn.h che definisce le macro funzioni da utilizzare nella costruzione di un argomento in modalità dlopen(); tra queste macro funzioni vi sono:

- RTLD_LAZY: le ricollocazioni vengono eseguite in un tempo dipendente dall'implementazione;
- *RTLD_NOW:* le ricollocazioni vengono eseguite quando l'oggetto viene caricato.

Dopodiché si dichiarano le costanti contenenti informazioni come la durata dell'attività del sensore, la frequenza di aggiornamento del sensore (in Hz) e il periodo di campionamento (in µs). Saranno proprio i valori di queste costanti che potranno essere variati per ottenere un andamento più o meno rapido, in base alle proprie esigenze, delle caratteristiche che rappresentano la variazione d'accelerazione. Tra la riga 55 e la riga 59 si nota la presenza di una struct che serve per memorizzare i valori delle tre dimensioni in cui si analizza

l’accelerazione provocata dal movimento dello smartphone. Gli attributi x, y e z sono di tipo GLfloat, che è un tipo simile al float ma che si utilizza per le multi-piattaforme, per evitare conflitti con le altre piattaforme. Nella parte di codice tra la riga 64 e la riga 107, viene impostato il modo in cui comparirà sullo schermo del dispositivo il grafico. Infatti, si nota la presenza di alcuni *shader* utilizzati dalla libreria grafica OpenGL che sfruttano le capacità di shading delle GPU delle schede video. In particolare, si notano le seguenti *shader*:

- ✓ *VertexShader*: gestisce la posizione dei vertici di un oggetto e quindi ne può alterare la forma. In tale applicazione viene utilizzato per gestire i vertici delle caratteristiche che rappresentano la variazione d’accelerazione;
- ✓ *FragmentShader*: gestisce operazioni sui pixel dell’oggetto rasterizzato.

Dopodiché, alla riga 88, si dichiara la classe SensorManager che permette l’accesso ai sensori del dispositivo. Tra la riga 115 e la riga 126, si crea una coda di eventi del sensore e si abilita il sensore selezionato alla frequenza di campionamento predefinita. Alla riga 167 si crea la funzione update con la quale, in seguito ai movimenti dello smartphone, ci sarà la variazione delle caratteristiche raffiguranti l’ accelerazione lungo gli assi x, y e z. Nelle righe finali si esporta e si collega il file.cpp al codice java. Extern C serve per riferirsi ad una variabile esterna prima della sua dichiarazione. Il prefisso *extern* è una dichiarazione e non una definizione e, quindi, nessun blocco di memoria verrà riservato per una variabile esterna. All’interno si trovano varie funzioni JNIEXPORT, ognuna delle quali ha un preciso scopo: creare il grafico, modificarlo in larghezza ed in altezza, aggiornarlo, metterlo in pausa e riavviarlo.

5. Risultati e considerazioni finali

Ho dunque analizzato il codice C++ che, collegato al resto del progetto, porterà alla creazione dell'applicazione. Nel codice C++, dalla riga 17 alla 21, sono stati inseriti i valori riguardanti la frequenza di aggiornamento del grafico ed il relativo periodo. Con i valori inseriti nel codice (frequenza di aggiornamento = 100 Hz, periodo = 1000000/frequenza) ed eseguendo l'applicazione su un dispositivo mobile reale (Samsung Galaxy S7 edge), otteniamo la seguente interfaccia grafica:

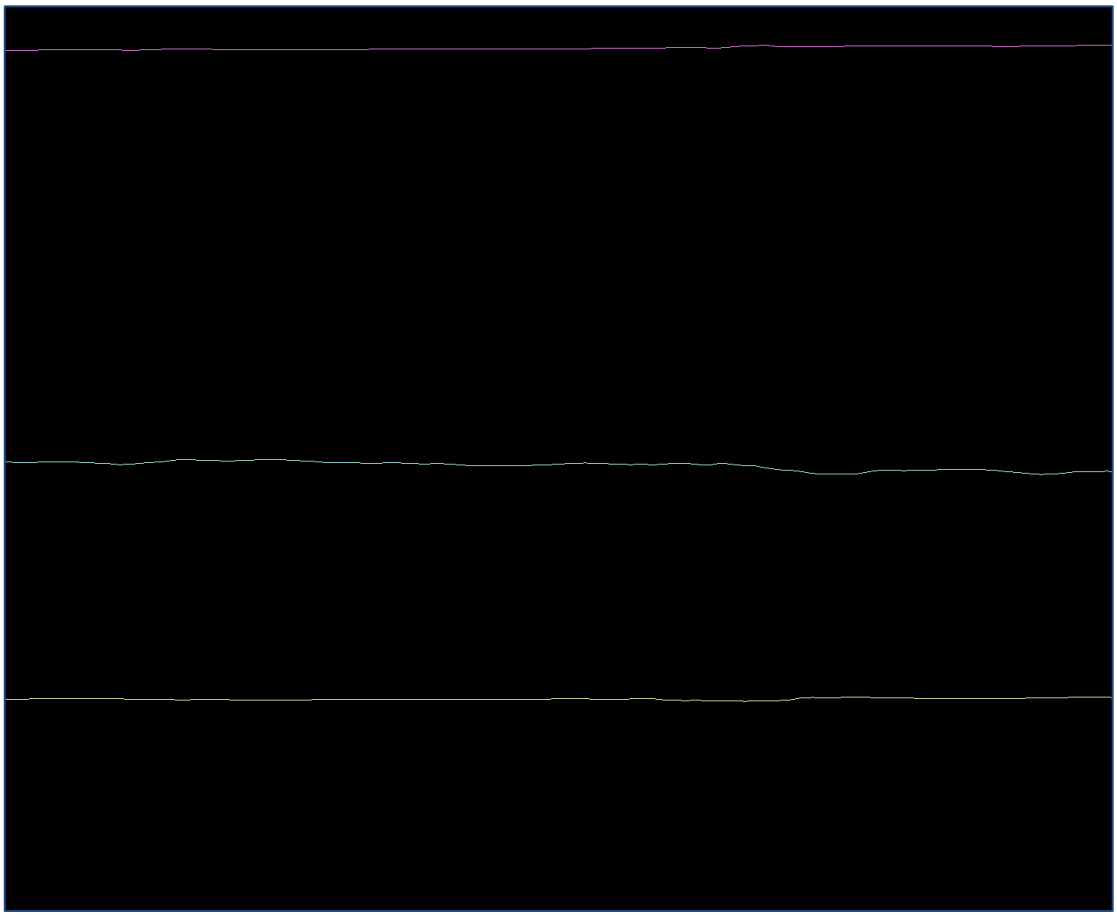


Figura 13- schermata iniziale con dispositivo statico

Nella prima schermata si notano tre caratteristiche che viaggiano quasi parallelamente, per poi notare degli sbalzi (più o meno ampi in base a quanta accelerazione applico al dispositivo) in verticale (lungo l'asse y, caratteristica viola) e/o in orizzontale (lungo l'asse x, caratteristica gialla) e/o in profondità (lungo l'asse z, caratteristica grigia).

Di seguito si riportano i grafici relativi alla variazione dell'accelerazione lungo i singoli assi cartesiani.

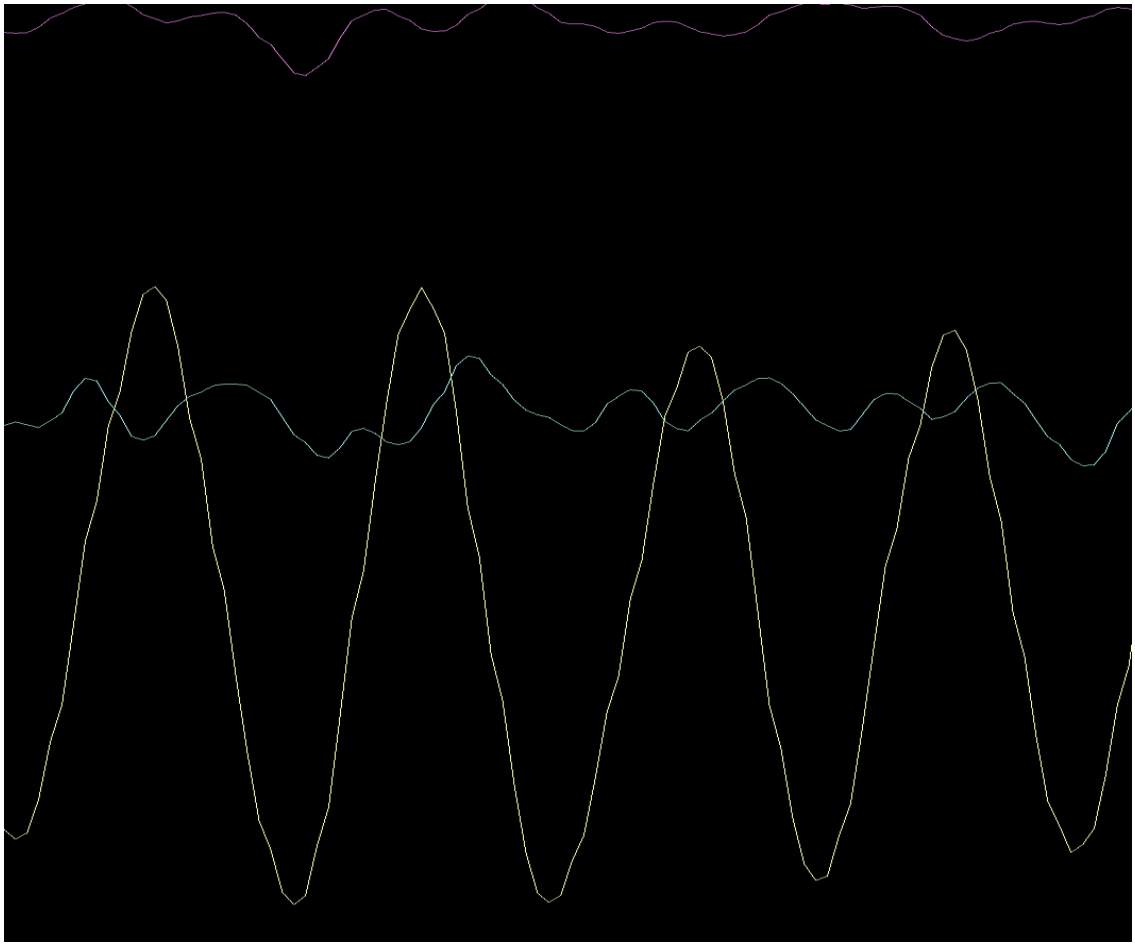


Figura 14- accelerazione lungo l'asse x

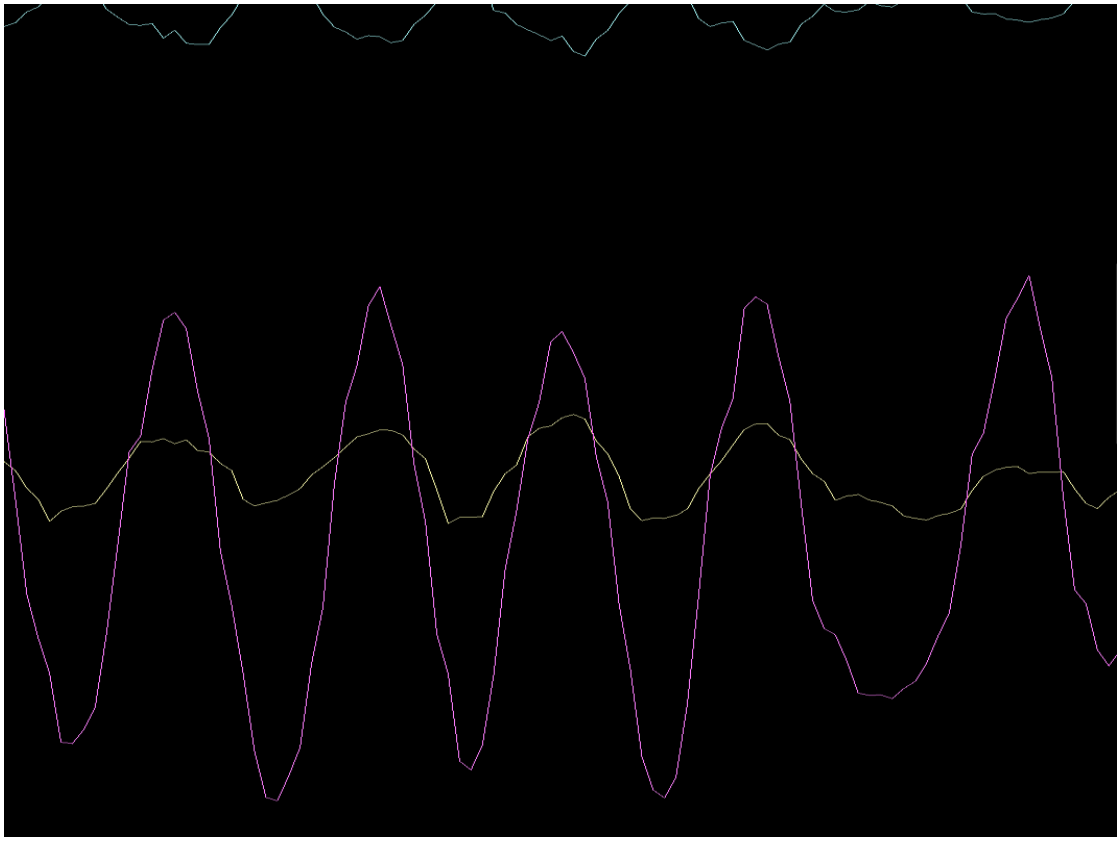


Figura 15- accelerazione lungo l’asse y

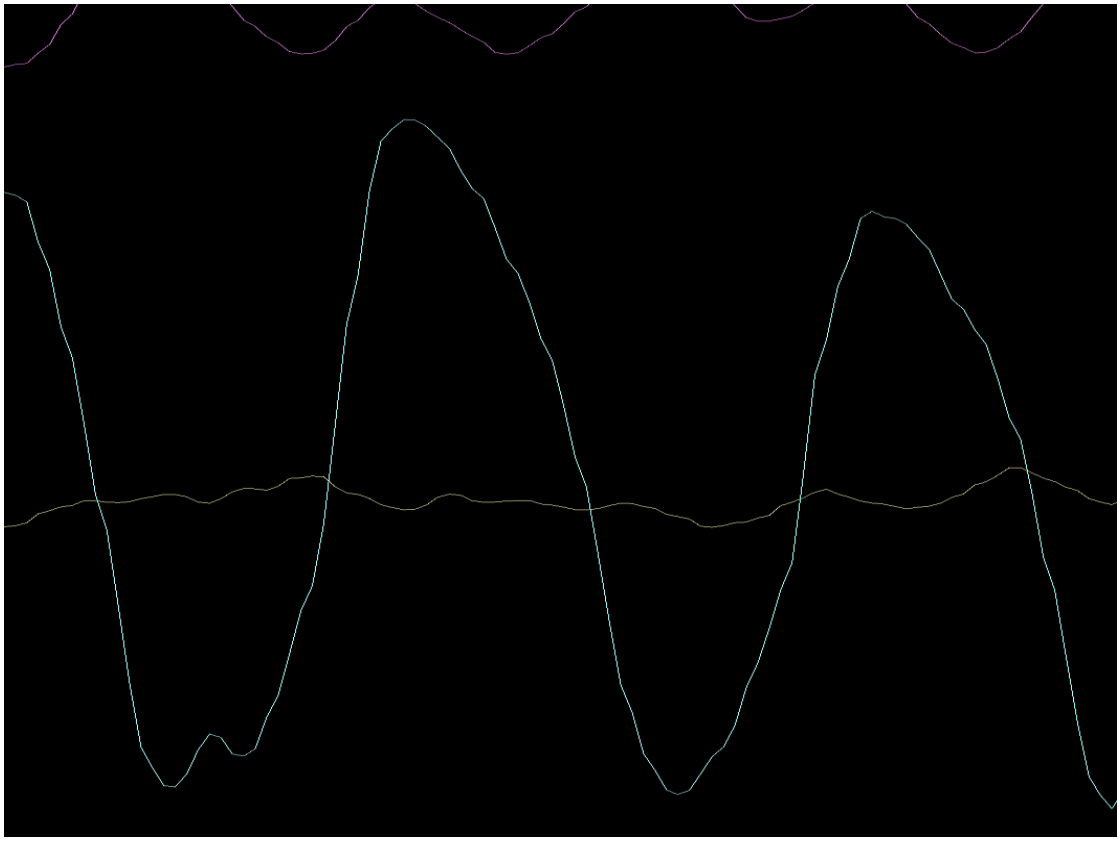


Figura 16- accelerazione lungo l’asse z

Ovviamente, variando i valori di frequenza e periodo, si possono ottenere delle modifiche del grafico, in quanto le variazioni d'accelerazione possono essere acquisite in tempi più o meno veloci, in base ai valori che si inseriscono. Infatti, diminuendo la frequenza di aggiornamento di una grandezza (10 Hz), si ottiene un andamento a gradino del grafico relativo alla variazione d'accelerazione, dovuto al relativo aumento del periodo a causa della sua inversa proporzionalità con la frequenza. Di seguito si riporta il grafico che rappresenta tale andamento.

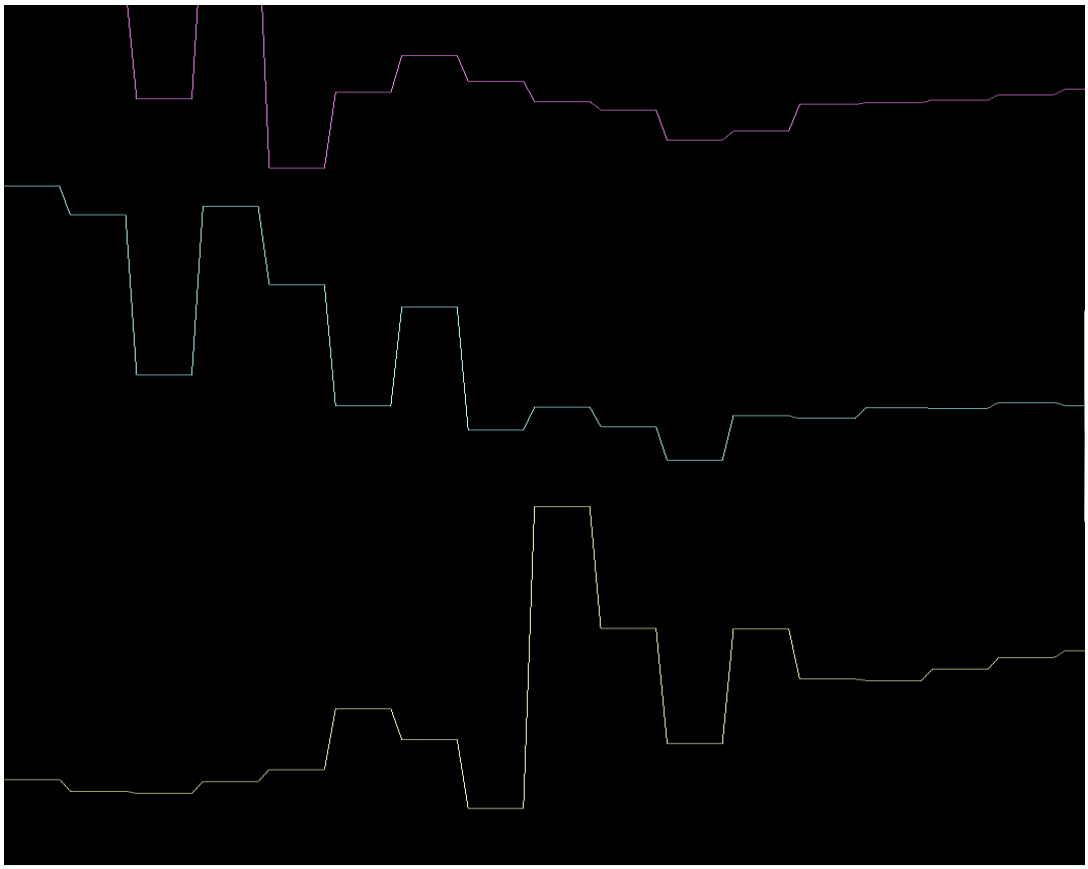


Figura 17- variazione d'accelerazione con andamento a gradino

In conclusione, analizzando lo sviluppo dell'applicazione Sensor-Graph, si è potuta dimostrare l'utilità del kit di sviluppo nativo di Android, in quanto, nel codice C++ di tale applicazione, sono state utilizzate le tecniche native che permettono di migliorare le prestazioni, come per esempio la parallelizzazione del lavoro del dispositivo utilizzando le librerie OpenGL per la parte grafica dell'applicazione.

BIBLIOGRAFIA

1. RATABOUIL S., (2015). *Android NDK beginner's guide*. Packt Publishing Ltd., Birmingham, UK.

SITOGRAFIA

2. developer.android.com
3. docentiunina.it (slides del corso ingegneria del software)
4. www.toshiba.it