UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA MAGISTRALE

Progettazione e sviluppo di un'applicazione per dispositivi mobili dedicata alla fruizione di musica elettronica su nastro magnetico

Laureando:
Riccardo Gasparini
Matricola: 1152986

Relatore:
Prof. Sergio Canazza

Correlatore:
Ing. Niccolò Pretto

Anno Accademico 2017-2018

Sommario

Questa tesi descrive il processo di portabilità e di ammodernamento di un'applicazione per dispositivi Android che permette la fruizione di documenti sonori storici. Viene presentato il progetto REMIND, che si propone di creare una virtualizzazione più fedele possibile di un magnetofono per la fruizione di opere su nastro magnetico digitalizzate. Viene poi discusso lo stato dell'arte in Android per quanto concerne le funzionalità audio ad alte prestazioni e il processo di portabilità nell'ambiente di sviluppo. Infine, verranno presentate delle nuove funzionalità introdotte nell'applicazione prevalentemente con lo scopo di estenderne alcuni aspetti legati alla gestione e consultazione delle opere sonore.

Indice

1 La conservazione dei documenti sonori **1**
1.1 Introduzione 1
1.2 Nastri magnetici 1
1.3 Magnetofono 2
1.4 Velocità ed equalizzazioni 2
1.5 Conservazione 3

2 Il progetto REMIND **5**
2.1 Introduzione 5
2.2 Descrizione dell'applicazione 5
2.3 Ambiente di sviluppo 6
2.3.1 Android 7

3 Stato dell'Arte **9**
3.1 Introduzione 9
3.2 Audio in Android 10
3.2.1 OpenSL ES 11
3.2.2 *AAudio* 11
3.2.3 *Oboe* 13
3.3 Gestione dell'audio 13
3.3.1 Inizializzazione e partenza 13
3.3.2 Riproduzione dei campioni audio 14
3.3.3 Chiusura di uno stream 14

4 Sviluppo del Lavoro **15**
4.1 Introduzione 15
4.2 Applicazione REMIND 15
4.3 Gestione dei permessi 15
4.4 Libreria *AAudio* e *Oboe* 16
4.4.1 Prove di verifica 16
4.4.2 Implementazione nell'applicazione 17
4.5 Gestione campioni audio 20
4.5.1 *Thread* separato 20
4.5.2 Risultati 20
4.6 Disconnessione di un flusso audio 23
4.7 Mixer 24
4.7.1 Struttura 24
4.7.2 Volume 25
4.8 Video offset 26

4.9 Visualizzatore *PDF* 27
4.10 Equalizzazioni 29
4.10.1 Introduzione 29
4.10.2 Implementazione 32
4.11 Opzioni avanzate 35
4.12 Gestione delle immagini 37
4.12.1 Glide 37
4.13 Modifica di un brano 38
4.14 Prestazioni 39
4.15 Risultato finale 40

5 Conclusioni e sviluppi futuri **43**

A Appendice **45**
A.1 Confronto OpenSL ES, *AAudio* e *Oboe* 45
A.1.1 OpenSL ES 45
A.1.2 *AAudio* 47
A.1.3 *Oboe* 48
A.2 Errori comuni 49

Elenco delle figure

1.1 Magnetofono Studer. 2

2.1 Stack software in Android. 8

3.1 Tipico flusso audio in Android. 9
3.2 *callback* audio in Android. 10
3.3 Logo *OpenSL ES*. 11
3.4 Transizione stati di un flusso *AAudio*. 12

4.1 Richiesta dei permessi durante l'installazione (a sinistra) e richiesta a *runtime* (a destra). 16
4.2 Applicazione di prova per la libreria *AAudio* fornita da Google. . . . 17
4.3 Applicazione di prova modificata. 18
4.4 Struttura per l'uso del codice nativo. 19
4.5 Struttura riproduzione audio iniziale. 21
4.6 Struttura riproduzione audio rivista. 21
4.7 Schema mixer con due tracce. 24
4.8 Layout della schermata per il controllo dei volumi. 25
4.9 Timer gestito dalla classe *UILcd*. 26
4.10 Timer gestito dalla classe *UILcdCustom*. 26
4.11 File *PDF* gestito con la classe *PdfRenderer*. 27
4.12 File *PDF* gestito con la libreria *PdfViewer*. 28
4.13 Anteprima file *PDF* con la classe *PdfRenderer*. 28
4.14 Equalizzazione NAB. 31
4.15 Equalizzazione CCIR. 31
4.16 pipeline dell'elaborazione audio. 32
4.17 Schema pipeline elaborazione audio con dettaglio dei *buffer* circolari coinvolti. 33
4.18 *Activity* delle impostazioni generali abilitata. 35
4.19 Azioni per eludere il controllo della password. 36
4.20 Logo della libreria Glide. 37
4.21 *Activity ImportSongActivity* nei due casi. 38
4.22 Prestazioni dell'applicazione catturate usando *Android Profiler*. . . . 39
4.23 Schermate principali dell'applicazione. 41

Capitolo 1

La conservazione dei documenti sonori

1.1 Introduzione

La storia dell'umanità è basata sulla creazione e conservazione di documenti di diverso tipo: da frammenti di manufatti a intere sculture, da piccoli oggetti del passato ai più maestosi monumenti, da piccole semplici raffigurazioni alle più moderne fotografie. Tutto questo fa parte di quello che è comunemente chiamato patrimonio culturale.

Nel momento in cui l'uomo ha iniziato a registrare e memorizzare ciò che udiva e suonava, anche l'eredità musicale è diventata una parte molto importante di questo patrimonio. A partire dai primi esempi di spartiti musicali risalenti al medioevo, passando attraverso i primi esperimenti di registrazione e riproduzione, risalenti al 19-esimo secolo, la quantità di informazioni che ci vengono tramandate è andata via via aumentando. Ad oggi esistono dispositivi che permettono a tutti la registrazione e riproduzione di opere sonore. L'avanzamento tecnologico, seguito dalla diffusione dell'audio digitale e dall'adozione di standard nella sua codifica, hanno separato la componente di riproduzione e l'informazione sonora vera e propria. In questo modo i dispositivi fisici, il software e i mezzi con cui il suono viene memorizzato sono slegati dall'informazione sonora vera e propria.

Una consistente parte del patrimonio musicale, però, è ancora in buona parte dipendente da suoni analogici registrati in supporti fisici risalenti all'epoca in cui il documento sonoro è stato creato. Per questo motivo la vita di questi documenti è inevitabilmente legata al supporto stesso su cui risiedono. Altro fattore importante da tenere in considerazione è che il mezzo fisico impedisce a queste opere di essere fruite facilmente da chi ne fosse interessato.

È il caso, per esempio, dei nastri magnetici usati per la registrazione di opere musicali, registrazioni vocali, ecc. e riprodotti attraverso un magnetofono.

1.2 Nastri magnetici

Le opere su nastro magnetico presentano delle peculiarità che le rendono uniche dal punto di vista della preservazione e che, in alcuni casi, complicano la procedura

di analisi e conservazione. Esistono infatti casi e situazioni uniche [1] per i nastri magnetici quali, ad esempio:

- La manipolazione fisica dei nastri con tagli, unioni, ecc.
- Le eventuali annotazioni importanti per l'opera e scritte direttamente sul nastro.
- La particolare conoscenza dei dispositivi in uso e dei loro limiti che a volte venivano aggirati modificandoli fisicamente.
- La presenza di suoni elettronici assieme a strumenti acustici che rendono difficile la distinzione tra rumori e/o distorsioni volontari (voluti e inseriti dall'artista) e involontari (artefatti derivanti dall'invecchiamento).

1.3 Magnetofono

Figura 1.1: Magnetofono Studer.

Un magnetofono è un dispositivo in grado di riprodurre o registrare del suono su del nastro magnetico. Il nastro è composto da una lamina di materiale flessibile ricoperta con del materiale magnetizzabile. Dal punto di vista meccanico deve far scorrere il nastro magnetico a ridosso delle testine facendo ruotare due bobine, una che contiene il nastro magnetico da cui viene srotolato, e l'altra per raccogliere il nastro fatto passare a ridosso delle testine. Dal punto di vista elettronico, questo strumento deve gestire e processare il segnale che passa dal nastro all'uscita audio, in caso di riproduzione, o dall'entrata audio alle testine in caso di registrazione.

1.4 Velocità ed equalizzazioni

Un ulteriore aspetto affrontato in questo lavoro di tesi è la gestione delle equalizzazioni e delle velocità di riproduzione del nastro.

Il termine *equalizzazione* comprende il processo in cui viene aggiustato l'equilibrio tra le diverse componenti in frequenza di un segnale. Per ragioni tecniche, il segnale registrato in un nastro (il flusso magnetico) non possiede una risposta in frequenza soddisfacente per essere usata direttamente e, per questo motivo, nel corso di decenni si sono formati molti standard [2] di equalizzazione.

In generale il processo di equalizzazione è diviso in due momenti distinti: registrazione e riproduzione [3].

Durante la fase di registrazione su del nastro magnetico, viene applicata una curva di pre-enfasi al segnale audio. Questo fa sì che il dispositivo incaricato di riprodurre il segnale debba possedere una curva post-enfasi compatibile con quella di registrazione e che, in particolare, la annulli in modo da ottenere il segnale originale non alterato.

Un'altra caratteristica in grado di alterare la resa sonora di un suono, oltre alla sua equalizzazione, è la velocità di registrazione e riproduzione. Variare la velocità di un nastro, soprattutto nella fase di registrazione, permette di sacrificare la qualità sonora a dispetto di una maggiore capacità temporale di registrazione o viceversa. Questo è dovuto al fatto che facendo scorrere il nastro più lentamente, il segnale sonoro, convertito in un segnale elettromagnetico per poter essere registrato, viene memorizzato in maniera più compatta e quindi più soggetta a distorsioni.

1.5 Conservazione

Per poter ascoltare delle registrazioni è necessario possedere:

- Un supporto di memorizzazione in buone condizioni.
- Dell'equipaggiamento di riproduzione compatibile e perfettamente funzionante.
- Una conoscenza dell'equipaggiamento, del formato e delle loro caratteristiche.

Tutto ciò è in generale, sempre più difficile col passare del tempo e col progresso tecnologico, infatti i supporti sono sempre più soggetti ad una rapida obsolescenza in ogni aspetto (registrazione del suono, riproduzione e memorizzazione) e da una aspettativa di vita modesta.

Per questi motivi esistono due metodologie di conservazione:

- La conservazione passiva: ha come scopo quello di conservare l'opera originale e proteggerla dagli agenti esterni (sporco, umidità, temperatura, etc).
- La conservazione attiva: prevede il trasferimento delle informazioni necessarie alla fruizione su nuovi supporti.

La conservazione attiva porta con sé però anche il problema della digitalizzazione, infatti, alcuni aspetti negativi fanno propendere per una conservazione passiva e alcuni di questi aspetti sono:

- Rapida evoluzione della tecnologia che causa una rapida obsolescenza dell'hardware, dei formati digitali e dei supporti.
- Mancanza di standard universalmente accettati riguardo il tasso di campionamento, la bit depth e i formati da usare.
- Bassa aspettativa dei supporti digitali.

- L'idea secondo la quale la digitalizzazione è un metodo primario per fornire accesso a materiale raro, in pericolo e distante, non una soluzione per la preservazione.

Verso gli anni 2000 però inizia a prendere piede la mentalità secondo la quale la preservazione a lungo termine dei documenti originali e degli equipaggiamenti (tutti i formati) è senza speranza [4] , per cui la conservazione dei documenti sonori deve essere basata sul trasferimento Analogico/Digitale senza perdita di informazione (quindi loss-less) del loro contenuto. In questo modo, avendo le opere in formato digitale, è possibile distribuire, conservare e copiare facilmente le opere.

Capitolo 2

Il progetto REMIND

2.1 Introduzione

La maggior parte dei documenti sonori presenti negli archivi vengono riprodotti tramite media-player classici senza considerare e preservare le caratteristiche visive, sonore e tattili presenti nei dispositivi di riproduzioni originali. L'applicazione oggetto di questa tesi si colloca nell'ambito della conservazione attiva dei supporti grazie alla simulazione del dispositivo di riproduzione usato in origine per l'ascolto di nastri magnetici. Infatti, non solo le opere musicali ma anche i dispositivi usati per la loro registrazione e riproduzione soffrono della sempre più rapida obsolescenza ed è importante preservarne le funzionalità e caratteristiche quanto più fedelmente possibile.

Il centro di Centro di Sonologia Computazionale [5] dell'università di Padova svolge varie attività in questo ambito tra le quali si trovano i progetti REWIND [6] e REMIND [7].

2.2 Descrizione dell'applicazione

Il progetto REMIND (*Restoring the Experience: a Mobile-based INterface for accesing Digitized recordings*) ha come obiettivo la creazione di applicazioni che permettano la fruizione di documenti sonori digitalizzati preservando quanto più possibile l'esperienza originale. L'applicazione cerca di essere quanto più fedele possibile nella riproduzione dell'esperienza originale sia per quanto riguarda l'opera in sé, sia per quanto riguarda le informazioni contestuali. In particolare cerca di:

- Simulare fedelmente l'esperienza di interazione con il magnetofono.
- Riprodurre in maniera fedele il segnale audio delle opere.
- Mantenere tutti i metadati e le informazioni contestuali del supporto originale come, ad esempio, le annotazioni.

L'applicazione di questa tesi propone un'interfaccia ispirata al magnetofono Studer A810, permettendo la riproduzione di brani immagazzinati in una libreria locale contenente non solo il segnale audio ma anche tutte le informazioni utili ad esse.

I dati audio vengono letti e processati dall'applicazione, simulando la resa sonora del magnetofono analogico. Vengono forniti i vari comandi per il controllo della

riproduzione, un selettore per l'impostazione della velocità del nastro e un selettore per l'equalizzazione del brano. Viene inoltre visualizzato un riquadro in cui è possibile riprodurre il video del nastro che scorre di fronte alle testine, sincronizzato col brano musicale.

La libreria dei brani, con tutte le loro informazioni, è memorizzata sul dispositivo e contiene vari metadati, come autore, anno di composizione, tipologia di nastro, e anche immagini del supporto originale nella sua confezione. L'applicazione e la sua interfaccia sono pensate per un tablet con schermo di diagonale da 10 pollici o più con sistema operativo Android.

2.3 Ambiente di sviluppo

Strumenti software

- Android [8]

  Sistema operativo di Google per dispositivi mobile.

- Android SDK [9]*Software Development Toolkit*

  Kit di sviluppo software multipiattaforma per Android che include una serie completa di strumenti di sviluppo, tra cui un debugger, librerie, un emulatore, documentazione e codici d'esempio.

- Android Studio [10]

  IDE (*Integrated Development Environment*) per lo sviluppo di applicazioni in Android. Negli ultimi anni ha soppiantato Eclipse diventando l'IDE ufficiale.

- Android NDK [11]*Native Development Toolkit*

  Un insieme di strumenti che consente di implementare alcune parti di un'applicazione in codice nativo usando linguaggi come C e C++. In alcuni casi può anche essere utile per riusare librerie già esistenti e scritte in quei linguaggi.

- Git [12]

  Git è un software di controllo versione distribuito utilizzabile da interfaccia a riga di comando, creato da Linus Torvalds nel 2005. Nacque per essere un semplice strumento per facilitare lo sviluppo del kernel Linux ed è diventato uno degli strumenti di controllo versione più diffusi.

- GitLab [13]

  GitLab è una piattaforma web open source che permette la gestione di repository Git e di funzioni trouble ticket.

Strumenti hardware

- Google Nexus 5

  Cellulare con Android Pie (API 28) usato come riferimento per l'uso delle nuove librerie e funzionalità introdotte in Android Oreo

- Samsung Galaxy S Tab 10.5

  Tablet con schermo da 10 pollici con una risoluzione pari a 2560 x 1600 con Android Kitkat (API 19).

2.3.1 Android

Android è uno stack software open source basato su Linux creato per un'ampia gamma di dispositivi e fattori di forma. Il diagramma in figura 2.1 mostra i componenti principali della piattaforma Android.

Il sistema è diviso in 5 macro-aree [14]:

- Kernel Linux: costituisce le fondamenta della piattaforma Android. Ad esempio, Android Runtime (ART) si basa sul kernel di Linux per le funzionalità sottostanti come il *threading* e la gestione della memoria di basso livello.
- Hardware Abstraction Layer (HAL): fornisce interfacce standard che espongono le funzionalità hardware del dispositivo al framework API Java di livello superiore. L'HAL è costituito da più librerie, ognuna delle quali implementa un'interfaccia per un tipo specifico di componente hardware, come la fotocamera o il modulo Bluetooth. Quando un'API effettua una chiamata per accedere all'hardware del dispositivo, il sistema Android carica la libreria per quel componente hardware.
- Android Runtime: per i dispositivi con Android versione 5.0 (livello API 21) o superiore, ciascuna applicazione viene eseguita nel proprio processo e con la propria istanza Android RunTime (ART).
- Librerie C/C++ Native: molti componenti e servizi di base del sistema Android, come ART e HAL, sono costruiti da codice nativo che richiede librerie native scritte in C e C ++. La piattaforma Android fornisce API scritte in Java per esporre le funzionalità di alcune di queste librerie native alle app. Ad esempio, è possibile accedere a OpenGL ES attraverso l'API Java OpenGL del framework Android per disegnare e manipolare elementi grafici 2D e 3D nella propria app.
- Framework Java API: l'intera serie di funzioni del sistema operativo Android è disponibile tramite API scritte in linguaggio Java. Queste API costituiscono gli elementi costitutivi necessari per creare app Android.
- Applicazioni di sistema: Android include una serie di app di base per e-mail, messaggi SMS, calendari, navigazione su Internet, contatti etc. Le app incluse nella piattaforma non hanno uno stato speciale tra le app che l'utente sceglie di installare. Pertanto, un'app di terze parti può diventare il browser Web predefinito, il servizio di messaggistica SMS o persino la tastiera predefinita dell'utente (con alcune eccezioni).

Figura 2.1: Stack software in Android.

Capitolo 3

Stato dell'Arte

3.1 Introduzione

Un DAC (*Digital to Analog Converter* è un Convertitore Digitale Analogico che, come dice il nome, converte flussi binari, generati da del software e resi disponibili in un *buffer*, in valori analogici che guidano gli altoparlanti. Esso è presente in ogni dispositivo in grado di riprodurre dell'audio e, per quanto riguarda l'architettura ad alto livello della gestione dell'audio in Android, è necessario che esista un percorso in grado di far comunicare un'applicazione con il DAC del dispositivo.

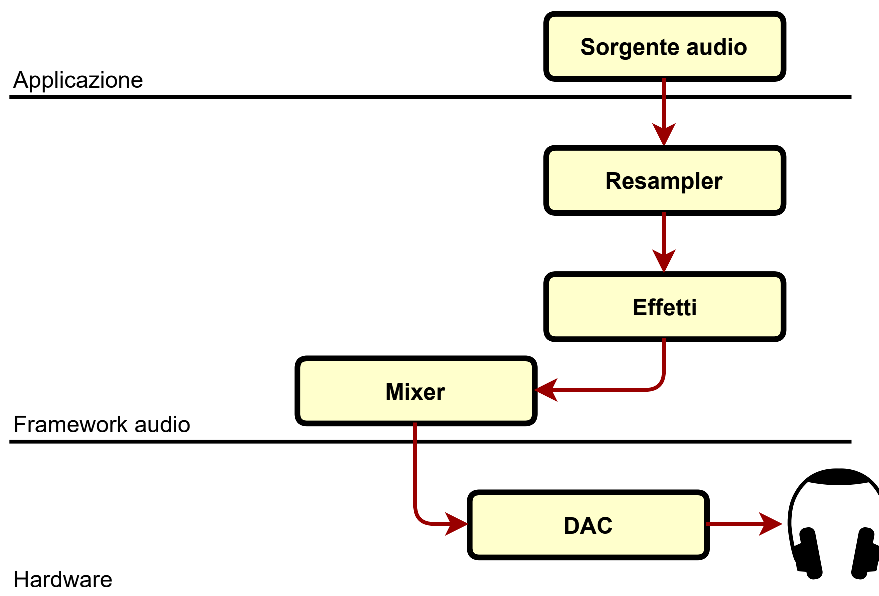


Figura 3.1: Tipico flusso audio in Android.

Un DAC, nel contesto dell'audio in Android, è caratterizzato dal volere dati sonori ad un tasso specifico, ad esempio 48000 campioni al secondo, e raggruppati in frame con una grandezza specifica, ad esempio a gruppi da 192 campioni.

Il DAC deve essere a conoscenza di queste informazioni per riprodurre correttamente i campioni audio e, analogamente, l'applicazione deve conoscere questi parametri per poter rifornire in tempo il DAC di tutti i campioni di cui ha bisogno.

Nel momento in cui il DAC necessita di nuovi campioni audio verrà chiamata una *callback* tramite un *thread* ad alta priorità che avrà il compito di interfacciarsi con l'applicazione per generare e trasmettere nuovi frame audio al DAC.

Ciò che accade dentro la *callback* è di fondamentale importanza per la corretta riproduzione dell'audio. Poiché l'applicazione dovrà generare frame audio ad intervalli fissi e molto specifici, ogni *callback* ha un tempo limite entro il quale deve generare i campioni audio. In generale però il tempo speso all'interno di questa *callback* varierà in base alla complessità dei campioni sonori, alla frequenza di funzionamento del processore e al tipo di dispositivo in cui ci si trova.

Se la deadline non viene rispettata il DAC non avrà campioni audio da consumare e quindi riprodurrà silenzio creando fastidiosi artefatti sonori.

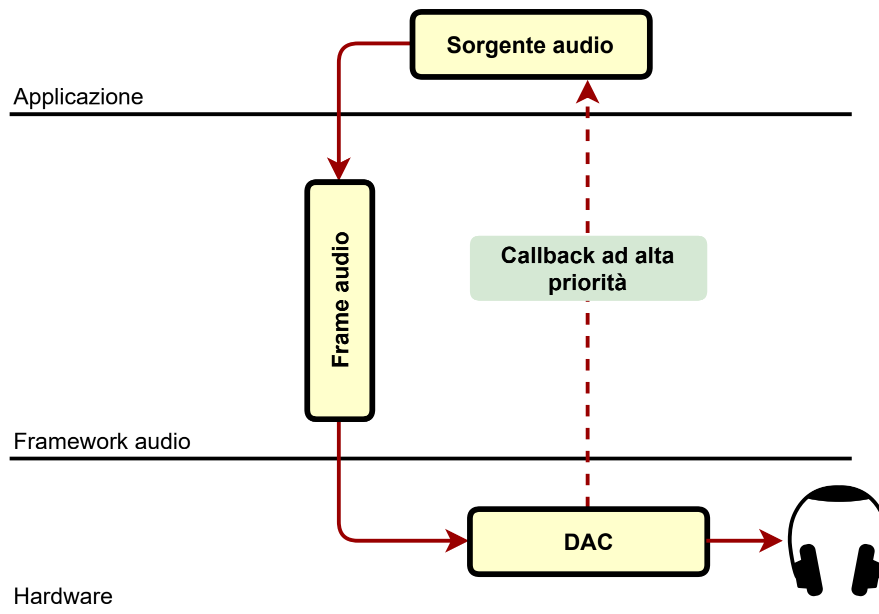


Figura 3.2: *callback* audio in Android.

3.2 Audio in Android

Le applicazioni in ambito mobile che richiedono audio ad alte prestazioni [15] solitamente richiedono molte più funzionalità rispetto alla semplice abilità di riprodurre o registrane suoni. In molti casi si basano su interazioni in tempo reale e su elaborazioni non banali di campioni audio. Alcuni esempi sono:

- Sintetizzatori
- Drum machines
- Videogiochi
- DJ mixing
- Effetti audio

Per venire in contro a queste esigenze Google mette a disposizione in Android due librerie audio con il preciso obiettivo di aiutare a sviluppare applicazioni che richiedono audio ad alte prestazioni. Queste librerie sono *OpenSL* ES e *AAudio*, e sono disponibile tramite l'uso del *Native Development Kit* (NDK), un insieme di strumenti che permette di usare codice C e C++ nello sviluppo di applicazioni in Android.

3.2.1 OpenSL ES

Figura 3.3: Logo *OpenSL ES*.

In Android, OpenSL ES è una specifica implementazione delle rispettive Application programming interface (API) specificate dal Khronos Group. Questa libreria permette di usare codice C e C++ per implementare funzionalità audio altamente performanti e a bassa latenza. Pur essendo basate sulle specifiche ufficiali OpenSL ES 1.0.1, alcune funzionalità potrebbero differire nell'effettiva implementazione nell'NDK, ma esse sono discusse nella documentazione ufficiale fornita da Google.

3.2.2 *AAudio*

AAudio [16] è una libreria sviluppata principalmente per essere leggera e fornire un'alternativa in ambiente nativo Android [17] alla libreria OpenSL ES. Il risultato di tutto ciò è una maggiore facilità di utilizzo e una minor pesantezza soprattutto nella scrittura del codice.

AAudio è stata introdotta nella release 8 di Android Oreo (API 26) rilasciata ufficialmente nell'agosto 2017. L'API è minimale per design e non offre alcune funzioni come:

- Enumerazione di dispositivi audio
- Instradamento automatico tra endpoint audio
- File I/O
- Decodifica di audio compresso

Il suo scopo è quello di aiutare e semplificare lo scambio di dati dall'applicazione all'*endpoint* audio, o viceversa. Per far ciò l'applicazione scambia dati scrivendo o leggendo su flussi audio, sotto forma di *buffer*, messi a disposizione dalla libreria. Uno flusso audio è caratterizzato da:

- un dispositivo audio che può essere una sorgente o destinazione
- una modalità di condivisione che determina se un flusso audio ha, o non ha, un accesso esclusivo a un dispositivo audio
- un formato per i dati audio nel flusso

La creazione di un flusso audio avviene tramite l'uso di uno schema architetturale *Builder*, in questo modo viene separata la complicata costruzione del flusso dalla sua relativamente semplice rappresentazione.

Un flusso *AAudio* è, di solito, sempre in uno dei seguenti stati:

- Open

- Started
- Paused
- Flushed
- Stopped

E le transizioni tra gli stati sono definite dalla figura 3.4.

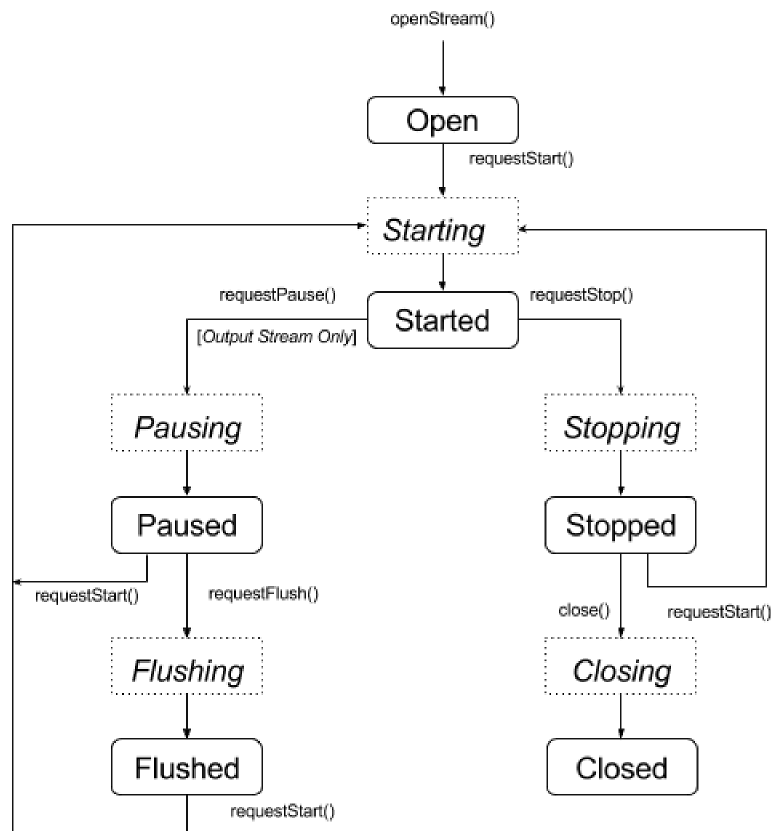


Figura 3.4: Transizione stati di un flusso *AAudio*.

Dopo l'inizio di un flusso esistono due metodi per leggerlo o scriverlo:

Funzione a chiamata Permette di eseguire delle letture o scritture bloccanti e anche non bloccanti.

***callback* ad alta priorità** *AAudio* permette l'esecuzione di una *callback* in un *thread* ad alta priorità, con migliori performance, con cui è possibile leggere o scrivere il flusso audio. Questa modalità è utile per applicazioni che leggono o scrivono dati audio da un *thread* ordinario che può non essere sufficiente a garantire un flusso audio stabile e costante.

Ogni flusso *AAudio* mette inoltre a disposizione 3 modalità di prestazione:

NONE è la modalità di default che bilancia latenza e impiego di energia.

LOW LATENCY usa *buffer* più piccoli e cerca di ottimizzare il percorso dei dati per ridurre la latenza.

POWER SAVING usa *buffer* più grandi e percorsi dei dati che sacrificano la latenza per risparmiare energia.

3.2.3 *Oboe*

Poiché *AAudio* è compatibile solo con dispositivi che usano Android Oreo o superiori, Google ha reso disponibile [18] anche un modo per renderla retrocompatibile, infatti, dopo aver scritto il codice usando *AAudio*, è sufficiente usare *Oboe* per rendere retrocompatibile il codice. *Oboe* [19] è una libreria scritta in C++ e distribuita da Google tramite GitHub che permette di adeguare automaticamente il codice in base alla versione di Android che si sta usando. *Oboe* sceglierà in automatico l'API audio (*AAudio* con API da 26 in poi oppure OpenSL ES con API da 16) in maniera totalmente trasparente al programmatore.

In appendice, a pagina 45, viene presentata l'inizializzazione di un motore audio scritto nelle 3 diverse API appena presentate.

Inoltre, sempre in appendice ma a pagina 49, vengono brevemente presentati alcuni consigli per evitare errori comuni quando si ha a che fare con queste librerie.

3.3 Gestione dell'audio

Nelle librerie *AAudio* e *Oboe* l'audio viene gestito attraverso un oggetto *AudioStream* il cui scopo è quello di inizializzare, iniziare e fermare il flusso dati in un *buffer* audio. Nello specifico, facendo riferimento alla libreria *Oboe*, la gestione classica di un flusso audio inizia con una fase di setup, prosegue con la fase di play e di reperimento dei campioni audio e si conclude fermando e distruggendo lo stream.

3.3.1 Inizializzazione e partenza

Tutto inizia con la creazione di un oggetto *builder*, nello specifico un oggetto *oboe::AudioStreamBuilder* a cui vengono settati i vari parametri come ad esempio l'id del device audio, la direzione (input o output), il tasso di campionamento, il numero di canali, il formato (float o interi), le prestazioni desiderate e, ultima ma non meno importante, la *callback* in cui i campioni audio verranno copiati nel *buffer* di riproduzione (in caso di registrazioni la *callback* copierà i valori dal *buffer* audio in un *buffer* dell'applicazione). Questo meccanismo permette di astrarre e nascondere l'effettiva creazione e configurazione audio del dispositivo liberando il programmatore da molti dettagli e rendendo il tutto molto più semplice e immediato. Successivamente si comunica al *builder* l'intenzione di creare uno stream audio fornendo un riferimento ad un oggetto *oboe::AudioStream*, il builder si occuperà di tutto ciò che è necessario per la creazione e configurazione del flusso sonoro. Per far partire il flusso audio, e quindi permettere l'esecuzione della *callback* audio, è sufficiente eseguire la chiamata al metodo *start()* dell'oggetto *oboe::AudioStream* precedentemente creato e inizializzato.

3.3.2 Riproduzione dei campioni audio

Il *buffer* audio del dispositivo cercherà di essere riempito attraverso la chiamata alla *callback onAudioReady* che deve essere definita ereditando la classe *oboe::AudioStreamCallback*.

I meccanismi adottati per creare, tenere in memoria e rifornire in tempo il *buffer* audio sono descritti più in dettaglio nella sezione 4.4.2

3.3.3 Chiusura di uno stream

La chiusura, in maniera identica all'avvio dello stream, avviene tramite la chiamata al metodo *stop()* dell'oggetto *oboe::AudioStream*. Se lo stream non verrà più usato esiste la chiamata *close()* che, come suggerisce il nome, chiude definitamente il flusso audio.

Capitolo 4

Sviluppo del Lavoro

4.1 Introduzione

I precedenti lavori di tesi sono stati utili per capire i vari ambiti coperti dall'applicazione e in particolare per studiare gli approcci seguiti nella scrittura del codice sorgente. In generale, la tesi [20] è stata consultata per capire l'architettura audio implementata, [21] per l'interfaccia grafica e le varie schermate delle applicazioni, [22] per la grafica e la logica di gestione dell'interfaccia del magnetofono e, infine, [23] per la gestione dei documenti sonori.

4.2 Applicazione REMIND

Dopo aver importato l'applicazione e le sue dipendenze in Android Studio, non è stato possibile configurare un ambiente di sviluppo funzionante con le API 19, che in origine erano state usate per sviluppare il codice sorgente iniziale. Per questo motivo è stato scelto di procedere direttamente all'uso delle API 28 (che al momento della scrittura sono le ultime a disposizione) e di correggere mano a mano eventuali incompatibilità ed errori.

Prima di procedere all'implementazione della nuova libreria audio è stato necessario correggere alcune parti del codice per renderlo nuovamente utilizzabile nelle nuove versioni di Android.

4.3 Gestione dei permessi

Un primo aspetto affrontato durante l'adeguamento dell'applicazione alle nuove API è stata la gestione dei permessi [24].

Da Android 6.0 (API 23 in poi) l'utente può concedere i permessi mentre l'applicazione è in esecuzione e non più solo quando l'applicazione viene installata. Inoltre, un permesso può essere revocato anche dopo averlo concesso, per cui, ad esempio, un'applicazione che ieri ha usato la fotocamera oggi non può assumere di avere ancora il permesso abilitato.

Innanzitutto, i permessi devono essere dichiarata nel file *AndroidManifest.xml* che, oltre che fornire informazioni essenziali riguardo l'applicazione ai tool di sviluppo, al sistema operativo Android e al servizio Google Play, fornisce anche la lista

dei permessi di cui l'applicazione necessita. In più i permessi sono divisi in due principali categorie:

Permessi *normali* Il sistema concede automaticamente questa classe di permessi all'applicazione che li chiede. Esempi di questa classe di permessi sono: l'accesso ad Internet, la vibrazione, l'accesso al Bluetooth, etc.

Permessi *pericolosi* L'utente deve necessariamente essere d'accordo nel concedere questa classe di permessi. Esempi di questa classe di permessi sono: l'accesso al calendario, alla fotocamera, ai contatti, allo storage, etc.

L'applicazione necessita dell'accesso al file system del telefono e questa tipologia di permessi rientra nella classe dei permessi pericolosi per cui ad ogni esecuzione dell'applicazione verrà controllato il permesso e, se non è mai stato concesso oppure è stato negato, verrà mostrata una richiesta a schermo. Per versioni di Android più vecchie la lista dei permessi verrà mostrata appena prima dell'installazione e con l'installazione essi verranno automaticamente concessi.

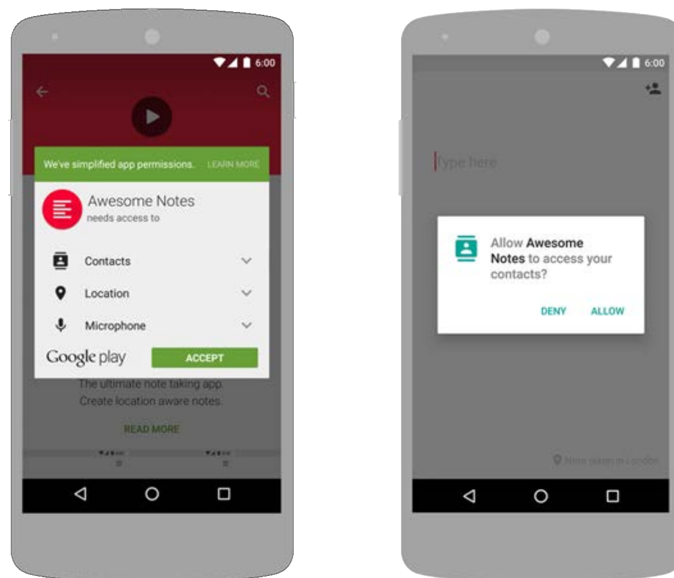


Figura 4.1: Richiesta dei permessi durante l'installazione (a sinistra) e richiesta a *runtime* (a destra).

4.4 Libreria *AAudio* e *Oboe*

4.4.1 Prove di verifica

Il primo passo per acquisire confidenza con l'ambiente di sviluppo e la nuova libreria *AAudio* è stato lo studio delle applicazioni di esempio [25] fornite da Google e distribuite tramite la piattaforma GitHub.

Come base di partenza è stato scelto il progetto *"hello-aaudio"* che implementa due generatori d'onda i quali generano due onde sinusoidali a frequenza diversa e in maniera distinta sul canale destro e sinistro in risposta a un tocco dello schermo. Partendo dal codice sorgente sono state fatte le seguenti modifiche per testare il funzionamento della nuova libreria:

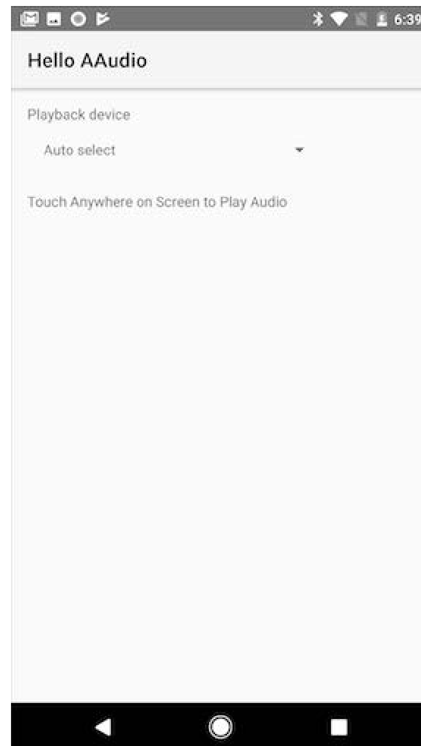


Figura 4.2: Applicazione di prova per la libreria *AAudio* fornita da Google.

- Modifiche ai generatori di segnale: i due generatori sono stati modificati per generare un'onda quadra a frequenza variabile in tempo reale e regolata dal rispettivo slider nell'interfaccia utente. Il loro segnale è poi stato convertito in stereo e mixato per permettere la presenza di entrambi i generatori nello stesso istante e nei canali destro e sinistro.

- Riproduzione di file *.mp3*: per testare un possibile impiego non banale di questa libreria è stata scelta la libreria LAME [26] (*"LAME Ain't an MP3 Encoder"*) per riprodurre un brano *.mp3* in risposta al tocco del pulsante centrale dell'interfaccia utente. LAME è un encoder MPEG Audio Layer III (MP3) ad alta qualità concesso in licenza secondo l'accordo LGPL e attualmente è il più diffuso.

Grazie a questo banco di prova è stato possibile capire il funzionamento della nuova libreria e avere un riferimento funzionante e relativamente avanzato su cui basare il porting dell'applicazione che simula il magnetofono.

4.4.2 Implementazione nell'applicazione

Il cuore audio dell'applicazione risiede nel file *native-player.cpp*. Esso contiene sia il codice che gestisce la parte audio attraverso la libreria OpenSL ES, sia tutte le funzioni che svolgono la funzione di interfaccia tra il codice nativo, scritto in C/C++, e il codice Java, sfruttando la JNI *"Java Native Interface"*. Si è scelto quindi di sperare il codice in due file:

native-player.cpp Contiene la classe *NativePlayer* che, una volta istanziata, permette di gestire l'audio secondo il paradigma OOP *"Object-oriented programming"* in C++.

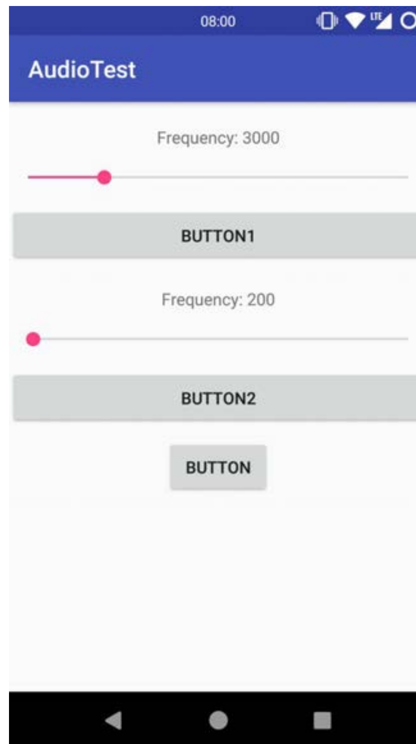


Figura 4.3: Applicazione di prova modificata.

jnibridge.cpp Contiene tutti i metodi che tramite la JNI si interfacciano al codice scritto in Java, e anche un oggetto *NativePlayer* che tramite delle chiamate alle sue funzioni permette di creare, distruggere, partire e fermare flussi audio.

Nella fase successiva il codice per la gestione dell'audio scritto per sfruttare le funzionalità di OpenSL ES è stato riscritto per sfruttare la libreria *AAudio*. Dopo averne verificato il corretto funzionamento la transizione alla libreria *Oboe* è stata semplice data l'estrema somiglianza con *AAudio*.

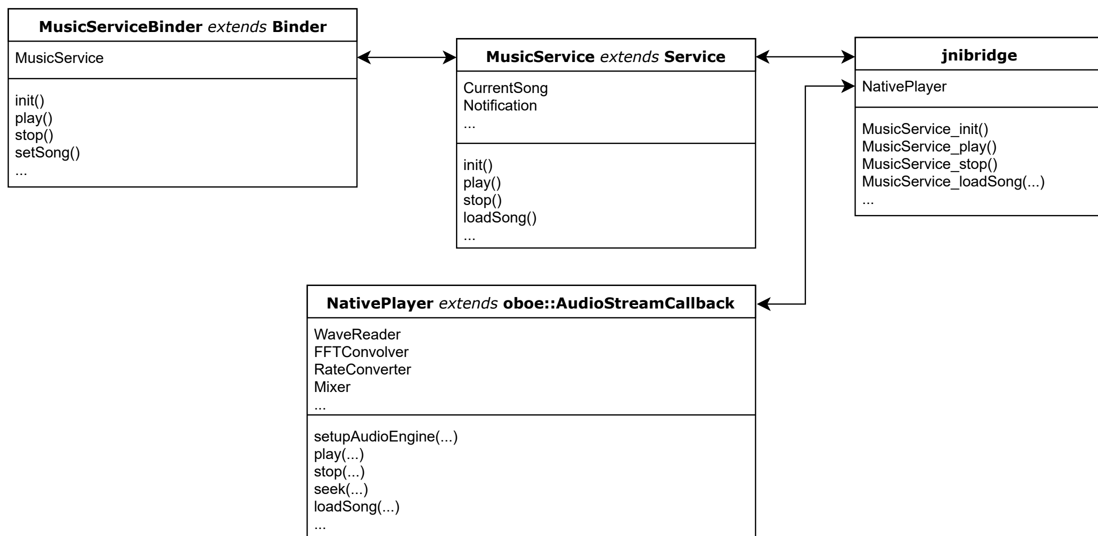


Figura 4.4: Struttura per l'uso del codice nativo.

4.5 Gestione campioni audio

La gestione dei campioni audio durante la riproduzione è stata oggetto di varie modifiche con lo scopo di testare quale fosse la migliore.

4.5.1 *Thread* separato

In questo caso, durante l'esecuzione di un brano, l'applicazione presenta due *thread* principali il cui schema logico è rappresentato in figura 4.5.Il primo *thread* ha il compito di reperire i campioni audio dalla *pipeline* di elaborazione e copiarli in un *buffer* temporaneo. Il rimanente *thread* invece gestisce la chiamata *callback onAudioReady* con lo scopo di fornire i campioni audio al sistema sonoro del dispositivo.

Prima di tutto viene eseguita una configurazione iniziale, prima della riproduzione, con lo scopo di settare correttamente tutte le funzioni e i vari parametri di riproduzione. Successivamente la chiamata al metodo play crea ed esegue il *thread* di lettura dei campioni e avvia lo stream audio. Questo *thread* reperisce i campioni audio dalla pipeline di elaborazione delle tracce, discussa nella sezione 4.10.2, copia i valori ottenuti in un *buffer* temporaneo ed esegue un *lock* su una primitiva di sincronizzazione con lo scopo di restare in attesa. Il *thread* audio, ad intervalli regolari, esegue la chiamata alla *callback onAudioReady* e controlla che ci siano un numero sufficiente di campioni audio nel *buffer* temporaneo, se così non dovesse essere esegue un *unlock* sulla primitiva di sincronizzazione che blocca il *thread* di lettura dei dati. Successivamente copia un numero predeterminato di campioni dal *buffer* temporaneo al *buffer* audio.

Per evitare interruzioni nel flusso audio il *buffer* temporaneo viene riempito di un numero di campioni molto superiore (ma non troppo) al numero di campioni richiesti da una singola *callback onAudioReady*, in questo modo anche se il *thread* di lettura non dovesse essere abbastanza veloce nel reperire i nuovi valori, la *callback* ha comunque a disposizione campioni da riprodurre.

4.5.2 Risultati

Questa architettura separa il *thread* ad alta priorità, che chiama la *callback onAudioReady*, dal *thread* che si occupa di ottenere i campioni audio elaborati. Inoltre, l'uso di oggetti bloccanti, invece di intervalli temporali fissi o, ancora peggio, cicli busy waiting, dovrebbe fornire un'elevata efficienza nell'uso delle risorse.

Dalle prove svolte è però emerso che mentre il funzionamento era corretto durante l'esecuzione nell'emulatore e nel Nexus 5, la riproduzione audio nel tablet Samsung presentava degli artefatti audio. Per risolvere questo problema sono stati tolti gli oggetti bloccanti e, al loro posto, è stato inserito un ciclo *while* nel *thread ReadData* che legge un certo numero di campioni e va in *sleep* per una durata prefissata.

Anche questo sistema ha prodotto gli stessi risultati problematici, per cui, alla fine, si è usato il sistema estremamente semplice di reperire i campioni audio all'interno della *callback*, come mostrato nel codice sottostante.

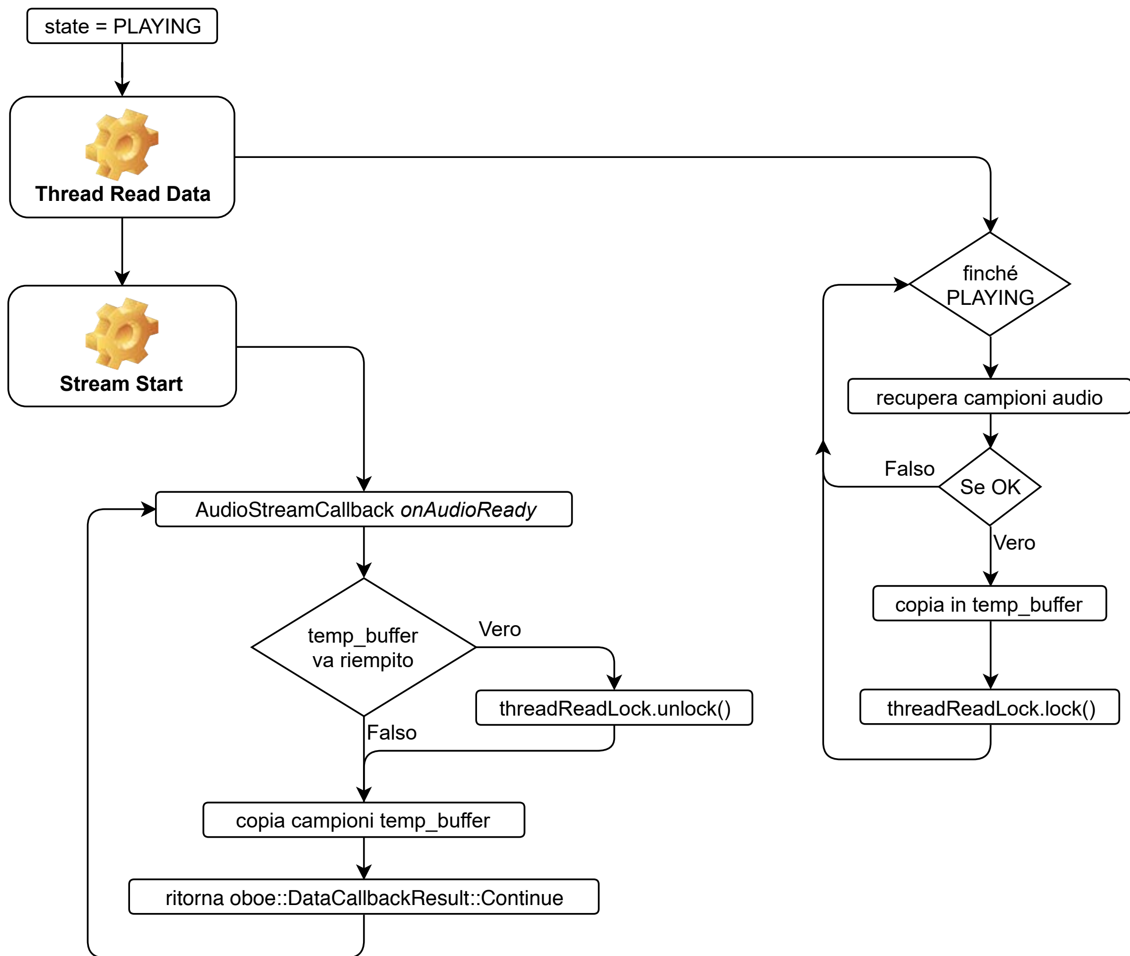


Figura 4.5: Struttura riproduzione audio iniziale.

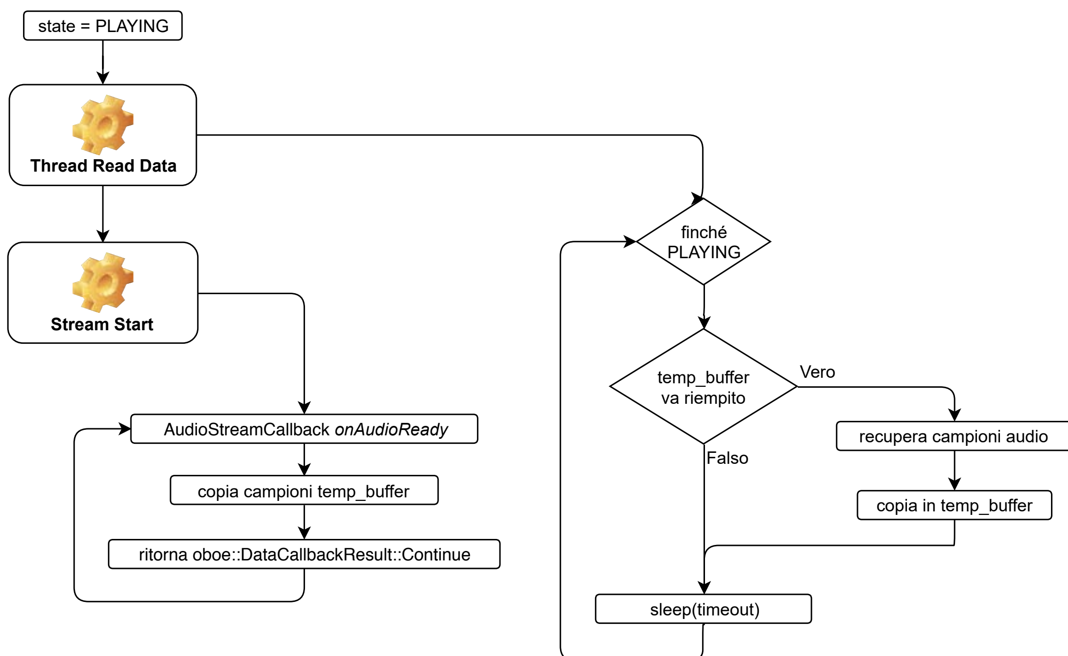


Figura 4.6: Struttura riproduzione audio rivista.

```
oboe::DataCallbackResult NativePlayer::onAudioReady(
        oboe::AudioStream *oboeStream,
        void *audioData,
        int32_t numFrames) {
        [.......
        ........
        ........]
        int16_t *temp = static_cast<int16_t *>(audioData);

        //zeroing the audio buffer for silence if stopped
        if (state == STOPPED) {
                memset(temp, 0, sizeof(int16_t) * channels *
                   numFrames);
                return oboe::DataCallbackResult::Continue;
        }

        audioBufferFillValue = numFrames * 2 * 2;
        while (audioBuffer.size() < audioBufferFillValue &&
           playbackCallback());

        int i;
        for (i = 0; i < numFrames * channels; i++)
                temp[i] = (int16_t) (audioBuffer.at(i));

        audioBuffer.erase(audioBuffer.begin(),audioBuffer.begin
           () + i);

        return oboe::DataCallbackResult::Continue;
}
```

La struttura dettagliata dell'elaborazione audio è descritta nella sezione 4.10.2. Nel dettaglio, la struttura in figura 4.17 permette di prelevare i campioni audio direttamente nella *callback* audio ad alta priorità. Ogni *buffer* circolare (rappresentato nella figura come un anello multicolore) contiene 4 *buffer* da 2048 campioni ciascuno. Ogni oggetto è gestito da un singolo *thread* che, ad intervalli regolari, verifica se una posizione del *buffer* circolare è libera e, in caso affermativo, procede ad eseguire il suo compito e a riempirlo. In particolare, l'oggetto *WaveReader* ha come obiettivo quello di mantenere pieno il suo *buffer* circolare che a sua volta andrà a rifornire l'intera catena, mentre la *callback* audio ha come obiettivo quello di svuotare regolarmente i campioni forniti dal *Mixer* creando quindi spazio per nuovi campioni audio da elaborare nella catena. Il *buffer audioBuffer* crea un ulteriore *buffer* per adeguare la differenza tra il numero di campioni richiesti dalla *callback* audio (variabile a causa di molti fattori) e quelli forniti dal *Mixer* (a blocchi da 2048).

Ipotizzando di avere un flusso audio mono da 48 000 kHz, un *buffer* circolare pieno contiene $1/48000 * 2048 * 4 = 0.170\,\text{s}$ audio, per cui se dovessero esserci dei momentanei rallentamenti in qualche punto della pipeline di elaborazione, la *callback*, e quindi anche l'oggetto *Mixer* da cui vengono prelevati i dati, riescono ad ottenere un numero momentaneamente sufficiente di valori per evitare problemi udibili in riproduzione.

4.6 Disconnessione di un flusso audio

Un flusso audio della libreria *Oboe* può andare nello stato *disconnesso* se si verifica uno dei seguenti casi:

- Il dispositivo audio associato non è più connesso. Questo caso può verificarsi, ad esempio, se le cuffie vengono disconnesse.
- Accade un errore interno.
- Un dispositivo audio non è più il dispositivo audio principale.

Il primo caso risulta essere il più importante in quanto è prevedibile che un utente possa cambiare la sorgente audio durante l'esecuzione. Quando un flusso risulta *disconnesso* ogni tentativo di scrittura risulta impossibile per cui l'unica alternativa è eliminarlo e crearlo di nuovo.

La libreria mette a disposizione il metodo *onErrorAfterClose(stream, error)* che viene invocato dopo aver rilevato un errore e aver chiuso il flusso audio, e che permette di inizializzare un nuovo flusso audio tramite un *thread*.

```
void NativePlayer::onErrorAfterClose(oboe::AudioStream *
    audioStream, oboe::Result error) {
     if (error == oboe::Result::ErrorDisconnected) {
          std::function<void(void)> restartStream = std::bind(&
              NativePlayer::restartStream, this);
          std::thread streamRestartThread(restartStream);
          streamRestartThread.detach();
     }
}

void NativePlayer::restartStream() {
     double time = currentTime;
     stop();
     closeOutputStream();
     setupAudioEngine(currentPlaybackDeviceId,
         currentSampleFormat, currentSampleChannels,
                           currentSampleRate);
     seek(time);
     play();
}
```

4.7 Mixer

4.7.1 Struttura

Per l'audio multi-traccia è stato deciso di eliminare la parte di simulazione della spazialità e implementare un sistema più semplice che permetta di scegliere pan e volume per ogni traccia. A questo scopo lo schema adottato è il seguente: l'applicazione mostra una schermata in cui, tra le varie informazioni, fa scegliere, per ogni traccia e per ogni canale, il volume di riproduzione che verrà poi comunicato al motore audio, che a sua volta passerà i valori al mixer al suo interno (rappresentato dalla classe *mixer.cpp*).

Lo schema adottato per il mixaggio è in figura 4.7, la traccia verde viene trattata come se fosse composta da due tracce identiche all'originale ma una traccia è per il canale destro, col suo volume caratteristico, e l'altra traccia è per il canale sinistro, sempre col suo volume caratteristico. Fatto ciò tutte le tracce destre vengono sommate e il risultato mandato al *buffer* audio destro e tutte le tracce sinistre vengono sommate e il risultato mandato al *buffer* audio sinistro.

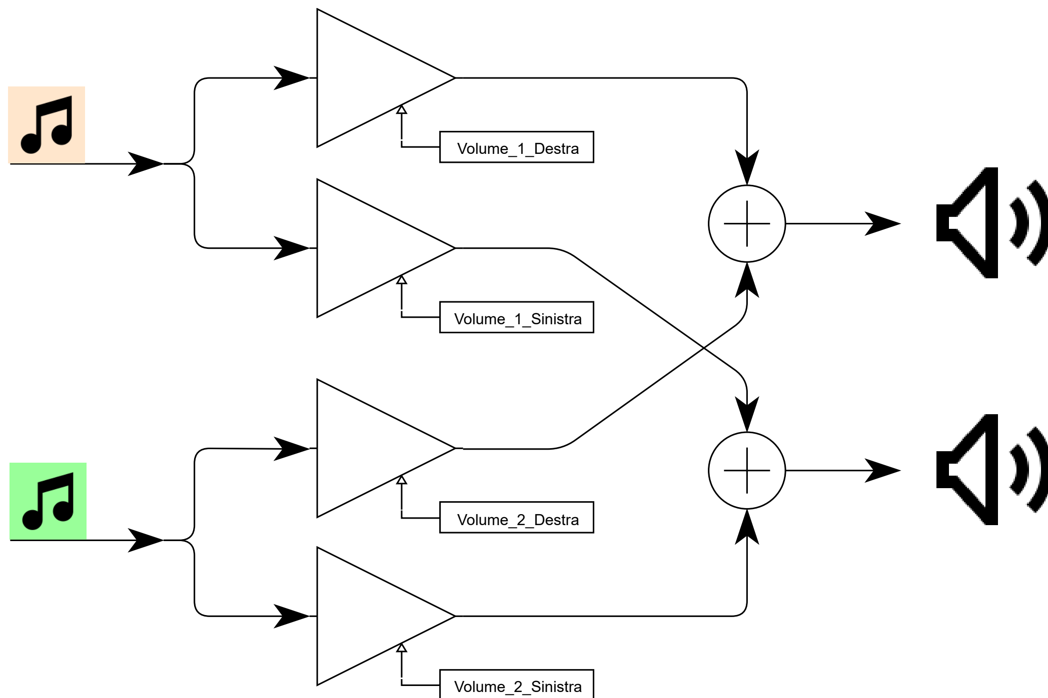


Figura 4.7: Schema mixer con due tracce.

4.7.2 Volume

L'approccio scelto per regolare il volume delle tracce in riproduzione è mostrato nella figura 4.8 e si basa su un controllo pan per ogni traccia.

Ogni traccia possiede una *SeekBar*, una *CheckBox* e un *Button*, le cui funzioni sono rispettivamente: gestione del volume nei canali destro e sinistro, attivazione o disattivazione della traccia e rispristino di un volume neutro (uguale per entrambi i canali).

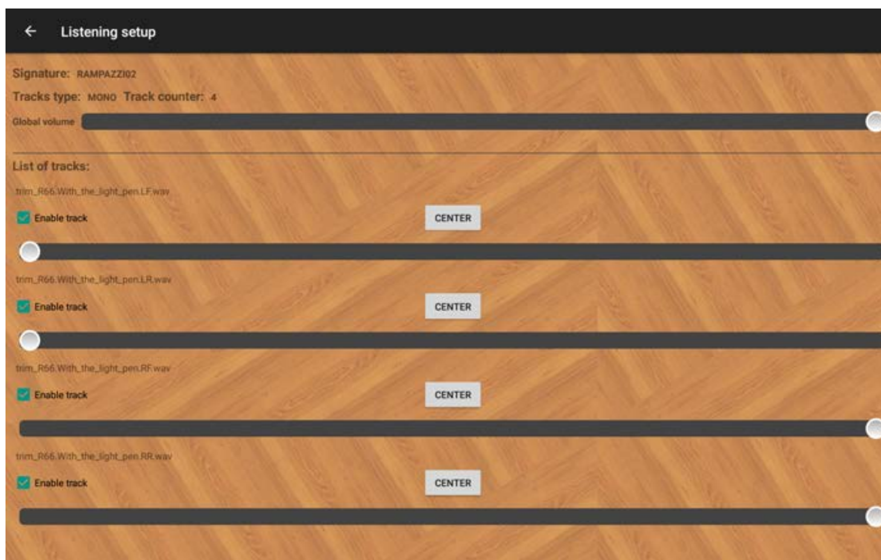


Figura 4.8: Layout della schermata per il controllo dei volumi.

In un primo momento la gestione del valore di uno dei volumi è stata affidata direttamente ad una *SeekBar*, con valori compresi tra 0 e 100. Ad ogni tocco della barra il selettore cambia posizione seguendo le coordinate del tocco e comunicando al codice nativo il nuovo volume desiderato, per i canali destro e sinistro, normalizzato da valori compresi tra 0 e 100 a valori compresi tra 0 e 1. Questo rende la regolazione del volume poco familiare in quanto, a livello percettivo, solo la parte centrale della barra sembra avere un effetto sensibile nella regolazione del volume. Ciò è dovuto al fatto che l'udito è più sensibile a piccole variazioni a volumi bassi e meno sensibile alle variazioni a volumi elevati.

Per correggere questo comportamento ogni volta che si vuole settare un volume, il codice converte il valore dalla scala lineare ad una scala logaritmica e viceversa ogni qualvolta si voglia leggere il valore. Le formule usate sono:

- $\frac{10^{volume}-1}{10}$ per passare da valori lineari a valori esponenziali
- $\log_{10}(10 * volume + 1)$ per passare da valori esponenziali a valori lineari

Internamente i volumi associati all'interfaccia grafica sono poi normalizzati tra loro in maniera tale che, anche con 4 tracce attive con i volumi ai valori massimi, l'audio finale non risulti essere 4 volte più forte.

Per fare ciò la formula usata è la seguente:

$$volume_{mix_i} = \frac{volume_{UI_i}}{\sum_i volume_{UI_i}}$$

4.8 Video offset

Riguardo la gestione del video allegato alla traccia, è stata aggiunta la possibilità di introdurre uno sfasamento tra la sua riproduzione e l'audio. Nel progetto era già presente la classe *UILcd* che ha il compito di gestire il timer temporale del magnetofono simulando il comportamento di un display LCD composto da 5 cifre, una per l'ora, due per i minuti e le rimanenti due per i secondi.

Figura 4.9: Timer gestito dalla classe *UILcd*.

Sfruttando ciò è stata implementata la classe *UILcdCustom* che estende la classe *UILcd* e, in particolare:

- Modifica il costruttore permettendo l'aggiunta di un numero tale di segmenti che permetta la visualizzazione di millisecondi.
- Ridefinisce il metodo *setTime* con lo scopo di poter trattare valori temporali che comprendono i millisecondi.

Inoltre, sono stati creati tre elementi grafici a corredo che gestiscono l'aumento dell'offset, la sua diminuzione e un'immagine a supporto che simula una struttura di montaggio di questi elementi e che rende questa parte dell'interfaccia meno artificiale.

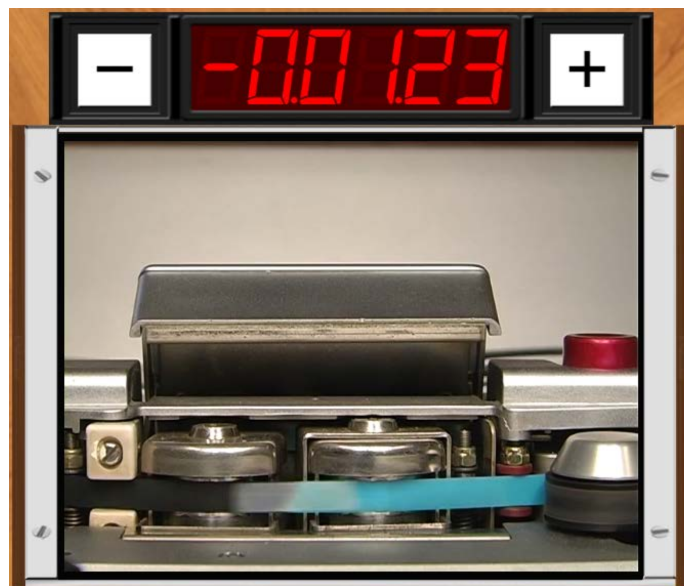


Figura 4.10: Timer gestito dalla classe *UILcdCustom*.

Per rendere piacevole e veloce l'utilizzo di questa funzione i pulsanti + e - gestiscono l'aumento o la diminuzione dell'offset in maniera esponenziale, cioè, la pressione prolungata del pulsante fa aumentare, o diminuire, esponenzialmente il valore di offset. In questo modo è possibile operare piccole variazioni precise con brevi pressioni dei tasti e grandi variazione imprecise con pressioni prolungate.

Dal punto di vista tecnico questo è stato ottenuto tramite la gestione di un *thread* che viene configurato e fatto partire appena uno dei due tasti viene premuto e viene arrestato appena il tasto viene rilasciato.

La gestione del rilascio di un bottone non era stata implementata nel codice a disposizione, per cui è stato necessario modificare ed estendere alcune sezioni delle classi *UIButton*, *UIBaseElement* e *UIComponent*, in particolare la rilevazione del rilascio del tasto e la gestione delle *callback* con questa modifica.

4.9 Visualizzatore *PDF*

Le funzionalità dell'app REMIND comprendono anche la gestione di file *PDF* contenenti metadati e informazioni contestuali relativi al nastro magnetico digitalizzato.

Per fare ciò in un primo momento è stata esplorata la possibilità di usare la componente *PdfRenderer* [27]. Questa classe, presente da Android API 21 in poi, permette la visualizzazione di un file *PDF* in maniera molto semplice. Fin da subito è stata evidente la mancanza di molte funzioni, come lo zoom e lo scrool continuo, che rende la navigazione nel file poco intuitiva.

Per questo motivo si è scelto di usare la libreria *Android PdfViewer* [28], una libreria per visualizzare documenti nel formato *PDF* in Android, con supporto per animazioni, gestures, zoom e doppio tocco. Essa è basata su *PdfiumAndroid* per decodificare i file *PDF* e funziona dalle API 11 (Android 3.0) in poi. Un'altra importante peculiarità è la licenza è l'utilizzo della licenza Apache 2.0 che ne consente una grande flessibilità.

La classe *PdfRenderer* è stata comunque utilizzata per fornire un'anteprima del documento nel menu che elenca le proprietà delle canzoni in libreria

Figura 4.11: File *PDF* gestito con la classe *PdfRenderer*.

1 / 1

1

Indice

1 **List of documents enclosed in the preservation copy** 2

2 **Description of the preservation copy** 2
- 2.1 General information 2
- 2.2 Description of the audio files 2
  - 2.2.1 Audio file 1 2
  - 2.2.2 Audio file 2 3
- 2.3 Description of the image files 4
- 2.4 Description of the video files 4

3 **Description of the source document** 6

4 **State of preservation of the source document** 7

5 **Technical scheme of the transfer system** 9

Figura 4.12: File *PDF* gestito con la libreria *PdfViewer*.

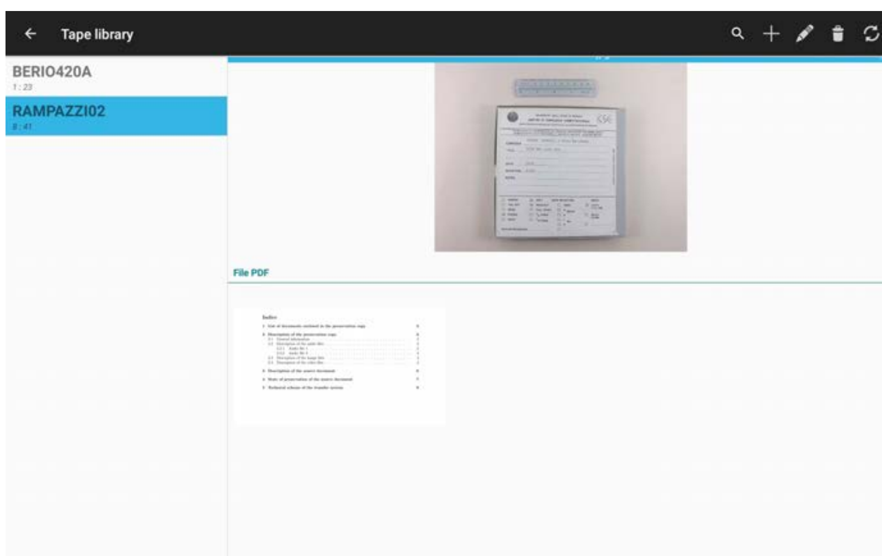


Figura 4.13: Anteprima file *PDF* con la classe *PdfRenderer*.

4.10 Equalizzazioni

In questa sezione verrà discussa la parte di gestione delle equalizzazioni partendo dalla loro descrizione teorica.

4.10.1 Introduzione

Gli standard di equalizzazione sono comunemente identificati dalle iniziali dell'organizzazione che scrisse lo standard e tra le varie a disposizione quelle di nostro interesse nell'applicazione sono:

NAB L'americana National Association of Broadcasters

CCIR La francese Consultative Committee on International Radio. Attorno al 1993 il nome CCIR fu cambiato in ITU-R, International Telecommunication Union - Radiocommunications.

Le equalizzazioni standard presentate fanno riferimento alla seguente formula [29] che esprime il livello del segnale in dB al variare della frequenza.

$$N(dB) = 20log_{10}\omega t_1\sqrt{\frac{1+(\omega t_1)^2}{1+(\omega t_2)^2}}$$

Con $\omega = 2\pi f$.

I parametri che permettono di ottenere la risposta in relazione allo standard sono riportati nella tabella 4.1.

Velocità (*ips*)	30		15		7,5		3,75	
Parametri (*us*)	t_1	t_2	t_1	t_2	t_1	t_2	t_1	t_2
CCIR	∞	17,5	∞	35	∞	70	3180	90
NAB	∞	17,5	3180	50	3180	50	3180	90

Tabella 4.1: Valori dei parametri delle varie equalizzazioni

La formula e i parametri possono essere espressi direttamente in frequenza per fornire un'intuizione migliore del loro effetto sul segnale, ricordando che $\tau = \frac{1}{2\pi F}$.

$$N(dB) = 10log_{10}\frac{1 + (F_{low}/f)^2}{1 + (f/F_{hi})^2}$$

Velocità (*ips*)	30		15		7,5		3,75	
Parametri (*Hz*)	F_{low}	F_{hi}	F_{low}	F_{hi}	F_{low}	F_{hi}	F_{low}	F_{hi}
CCIR	0	9000	0	4500	0	2270	50	1800
NAB	0	9000	50	3150	50	3150	50	1800

Tabella 4.2: Valori dei parametri delle varie equalizzazioni

L'applicazione usa le formule nella loro versione lineare, per cui i valori di equalizzazione del filtro in frequenza si ottengono con:

$$F = \omega t_1 \sqrt{\frac{1 + (\omega t_1)^2}{1 + (\omega t_2)^2}}$$

$$F^{-1} = \frac{1}{\omega t_1 \sqrt{\frac{1+(\omega t_1)^2}{1+(\omega t_2)^2}}}$$

Di seguito vengono rappresentate le curve principali degli standard NAB e CCIR. I valori esatti variano a seconda della velocità desiderata ma, a meno di leggere variazioni, le forme che ne derivano sono principalmente due.

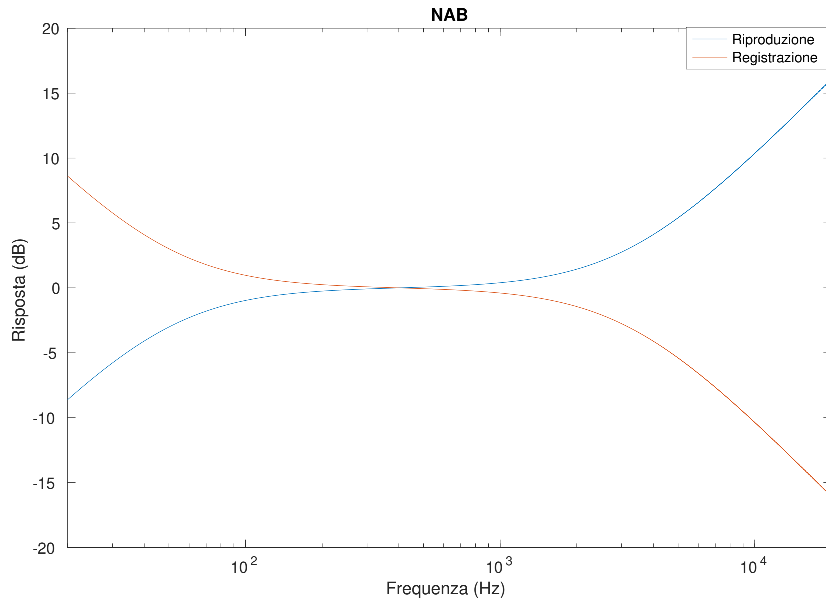


Figura 4.14: Equalizzazione NAB.

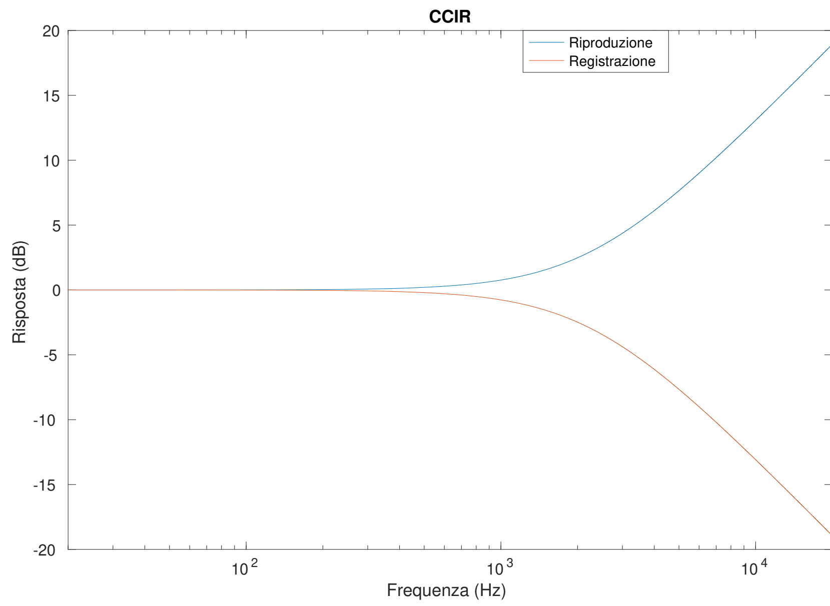


Figura 4.15: Equalizzazione CCIR.

4.10.2 Implementazione

L'applicazione gestisce due equalizzazioni, *NAB* e *CCIR*, con 4 possibili velocità di registrazione/esecuzione: 3,75 , 7,5, 15, 30 *ips*. Questo da vita a 32 possibili combinazioni tramite la velocità di digitalizzazione del brano e la sua equalizzazione, e la velocità di riproduzione e l'equalizzazione desiderata nell'applicazione.

In generale è necessario eliminare l'equalizzazione applicata durante la fase di registrazione, cambiare la velocità d'esecuzione del brano e applicare l'equalizzazione desiderata. Questo viene fatto tramite una catena di elaborazione rappresentata nella figura 4.16 che sfrutta gli oggetti già implementati nel codice nativo e che, in ordine, possiede:

- Un oggetto *WaveReader* legge un file audio nel formato *.wav* e comunica i valori letti a blocchi ad un primo oggetto *FFTConvolver*
- Il primo *FFTConvolver* prende il blocco audio e, tramite la FFT applica un primo filtro in frequenza (se necessario). Successivamente trasforma il risultato tramite la FFT inversa e passa il risultato al *RateConverter*.
- Il *RateConverter* si occupa di cambiare la velocità di una traccia i cui dati arrivano dal primo filtro FFT e sono destinati ad un secondo filtro, il secondo *FFTConvolver*
- Il secondo *FFTConvolver* applica un secondo filtro (se necessario). Il risultato viene poi passato al *Mixer*
- Il *Mixer* prende tutte le tracce elaborate e le elabora in modo da renderle riproducibili al canale destro e sinistro.

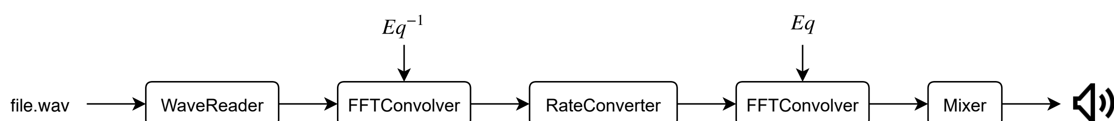


Figura 4.16: pipeline dell'elaborazione audio.

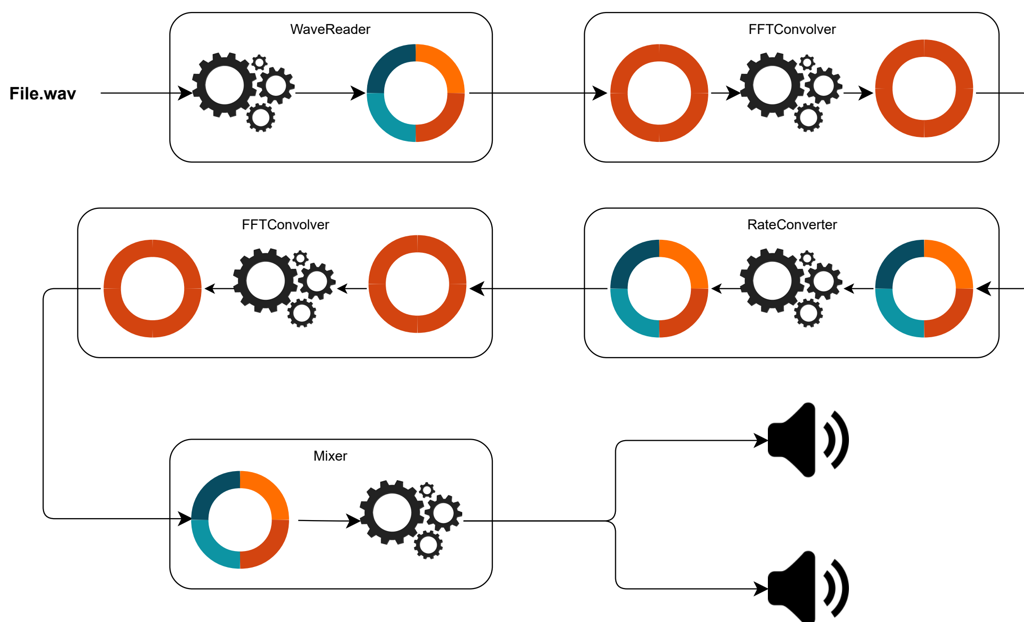


Figura 4.17: Schema pipeline elaborazione audio con dettaglio dei *buffer* circolari coinvolti.

Ad esempio, supponiamo di avere un brano digitalizzato tramite l'equalizzazione *NAB* alla velocità di 3,75 *ips* e che l'applicazione sia settata per riprodurre il brano con l'equalizzazione *CCIR* alla velocità di 7,5 *ips*, il caso corrisponde alla seconda riga della tabella 4.3. I passi da fare sono:

- Togliere l'equalizzazione $NAB_{3,75}$ applicata durante la fase di registrazione, e quindi riportando il brano allo stato in cui è presente sul nastro. Questa fase è svolta dal primo *FFTConvolver*
- Cambiare la velocità di esecuzione della traccia usando l'oggetto *RateConverter*.
- Applicare l'equalizzazione $CCIR_{7,5}$. Questa fase è svolta dal secondo *FFTConvolver*.

$Eq.originale_{vel.orignale}$	$Eq.nuova_{vel.nuova}$	Filtro #1	Cambio vel.	Filtro #2
$NAB_{3,75}$	$CCIR_{3,75}$	$NAB^{-1}_{3,75}$	✗	$CCIR_{3,75}$
$NAB_{3,75}$	$CCIR_{7,5}$	$NAB^{-1}_{3,75}$	✓	$CCIR_{7,5}$
$NAB_{3,75}$	$CCIR_{15}$	$NAB^{-1}_{3,75}$	✓	$CCIR_{15}$
$NAB_{3,75}$	$CCIR_{30}$	$NAB^{-1}_{3,75}$	✓	$CCIR_{30}$
$NAB_{7,5}$	$CCIR_{3,75}$	$NAB^{-1}_{7,5}$	✓	$CCIR_{3,75}$
$NAB_{7,5}$	$CCIR_{7,5}$	$NAB^{-1}_{7,5}$	✗	$CCIR_{7,5}$
$NAB_{7,5}$	$CCIR_{15}$	$NAB^{-1}_{7,5}$	✓	$CCIR_{15}$
$NAB_{7,5}$	$CCIR_{30}$	$NAB^{-1}_{7,5}$	✓	$CCIR_{30}$
NAB_{15}	$CCIR_{3,75}$	NAB^{-1}_{15}	✓	$CCIR_{3,75}$
NAB_{15}	$CCIR_{7,5}$	NAB^{-1}_{15}	✓	$CCIR_{7,5}$
NAB_{15}	$CCIR_{15}$	NAB^{-1}_{15}	✗	$CCIR_{15}$
NAB_{15}	$CCIR_{30}$	NAB^{-1}_{15}	✓	$CCIR_{30}$
NAB_{30}	$CCIR_{3,75}$	NAB^{-1}_{30}	✓	$CCIR_{3,75}$
NAB_{30}	$CCIR_{7,5}$	NAB^{-1}_{30}	✓	$CCIR_{7,5}$
NAB_{30}	$CCIR_{15}$	NAB^{-1}_{30}	✓	$CCIR_{15}$
NAB_{30}	$CCIR_{30}$	NAB^{-1}_{30}	✗	$CCIR_{30}$

Tabella 4.3: Elenco delle elaborazioni da effettuare nel caso di registrazioni da *NAB* a *CCIR*

Alcuni casi potrebbero non richiedere cambi di velocità e/o filtraggi, in questi casi i filtri *FFTConvolver* e il *RateConverter* vengono settati con parametri che non modificano le tracce ma risultano ancora in funzione (questo per non complicare troppo il codice con casi particolari da gestire). Nel caso dei filtri in frequenza, ad esempio, essi sono sempre in funzione, ma, se non richiesti, vengono settati con una risposta piatta in frequenza a 0 *dB*.

4.11 Opzioni avanzate

L'*Activity* delle impostazioni generali possiede uno scopo particolare, il suo primo elemento è una *CheckBoxPreference* che, se selezionata, permette di visualizzare anche le altre *Preference*, altrimenti esse sono tenute nascoste ed inutilizzabili. Quando si cerca di selezionare la *CheckBox*, appare un *Dialog* che richiede una password e solo se essa viene inserita correttamente si possono sbloccare le altre opzioni. Queste riguardano azioni che un utente che ha il solo interesse a consultare le opere non deve poter eseguire e sono illustrate nella figura 4.18. Per questo l'utilizzo di una password permette di nasconderle quando necessario.

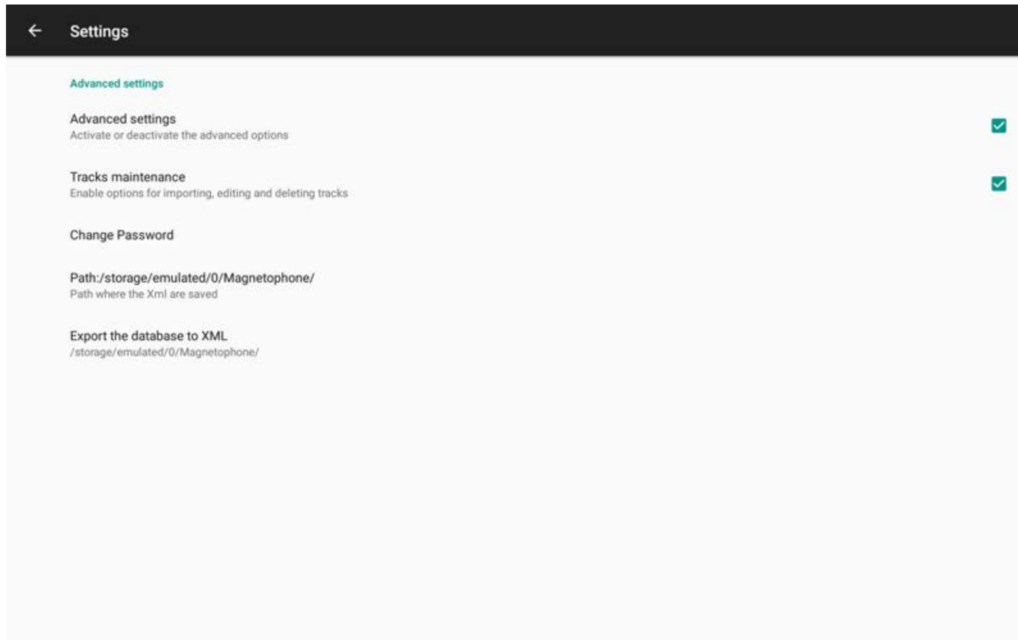


Figura 4.18: *Activity* delle impostazioni generali abilitata.

Nella precedente implementazione era possibile a chiunque di poter aggirare il controllo della password e di sbloccare le opzioni avanzate tramite le seguenti azioni illustrate nella figura 4.19 e qui di seguito descritte:

1. Entrata nel menù delle opzioni avanzate e tocco per attivare le opzioni avanzate. Questa operazione ha due effetti:
    (a) Lancia la *Dialog* per la richiesta della password.
    (b) Cambia lo stato della *CheckBox* come evidenziato dal cerchio verde in figura 4.19.
2. Tocco nell'area evidenziata in rosso della schermata in cui è presente la *Dialog* per l'inserimento della password. In questo modo la *Dialog* viene chiusa senza passare per i controlli di validità della password e si ritorna nella schermata delle opzioni avanzate con la *CheckBox* settata incorrettamente ma con le *Preference* avanzate non ancora caricate.
3. Arrivati a questo punto, è sufficiente tornare indietro alla schermata principale e rientrare nell'*Activity* per trovare le opzioni avanzate abilitate.

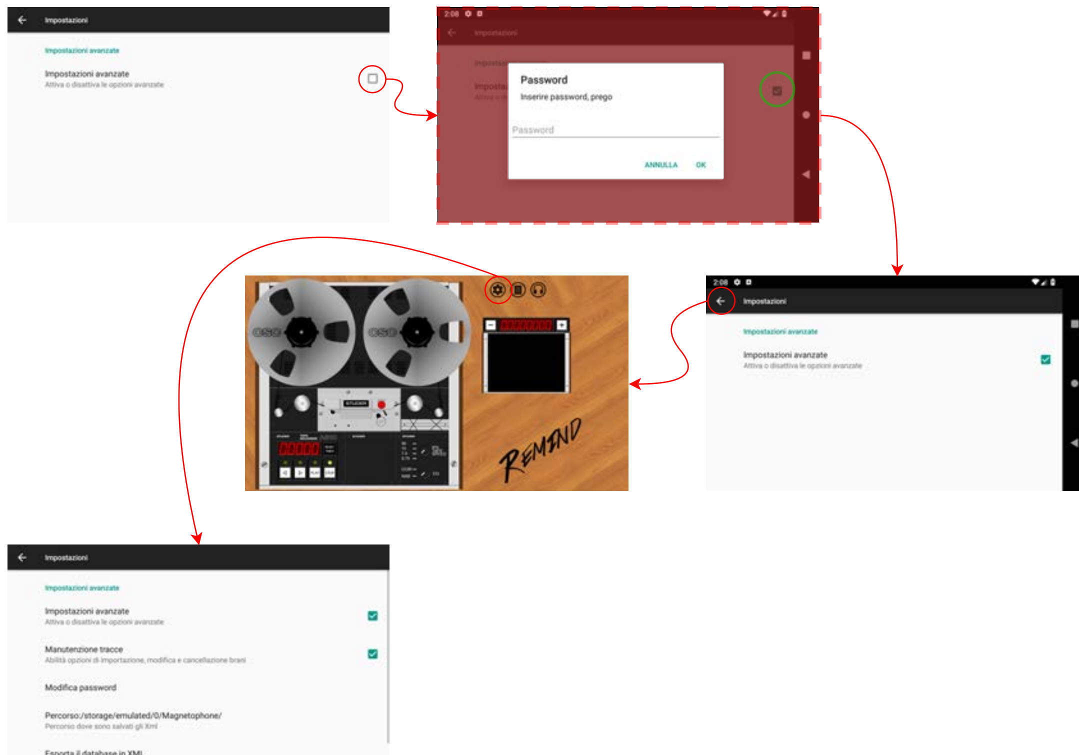


Figura 4.19: Azioni per eludere il controllo della password.

La risoluzione di questo problema è risultata semplice: immediatamente dopo il tocco della *Preference* per attivare le opzioni avanzate, si esegue *setChecked(false)* sulla *Preference*, in questo modo solo l'inserimento della password corretta può risultare nella chiamata *setChecked(true)* sulla *Preference* e attivare di conseguenza le opzioni avanzate.

4.12 Gestione delle immagini

La visualizzazione delle immagini allegate ad un'opera causava, a lungo andare, rallentamenti notevoli dell'applicazione che potevano culminare in un arresto anomalo dell'applicazione causato dall'esaurimento della memoria a disposizione. La documentazione di Android in riferimento alla gestione di grandi immagini Bitmap [30] [31] consiglia in più punti di usare librerie di terze parti per risolvere questo tipo di problemi.

4.12.1 Glide

Figura 4.20: Logo della libreria Glide.

Glide [32] è un framework veloce, efficiente e open source per il caricamento e la gestione di immagini in Android. Si occupa della decodifica dei media, della memoria, della cache su disco e il pooling delle risorse in un'interfaccia semplice e facile da usare.

Il metodo classico carica la bitmap in memoria e crea un riferimento all'oggetto *ImageView* che ospiterà l'immagine, successivamente assegna l'immagine all'oggetto e in questo modo l'immagine viene visualizzata.

```
Bitmap myBitmap = BitmapFactory.decodeFile(ImagePath);
ImageView myImage = rootView.findViewById(R.id.myID);
myImage.setImageBitmap(myBitmap);
```

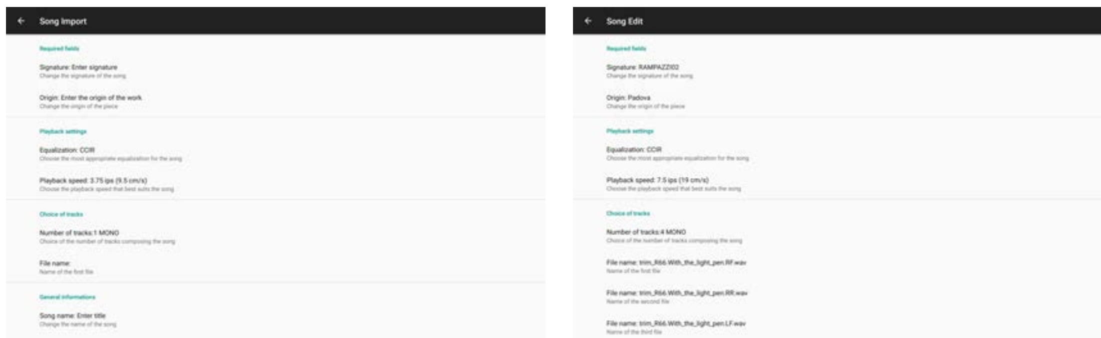
Utilizzare la libreria è stato molto semplice, è bastato creare il riferimento all'oggetto *ImageView* che ospiterà l'immagine e darlo in pasto al metodo *load* della libreria assieme al percorso dell'immagine, la libreria si occupa di tutto il resto (caching, cancellazione, creazione riciclaggio, etc).

```
ImageView myImage = rootView.findViewById(R.id.myID);
Glide.with(this).load(ImagePath).into(myImage);
```

4.13 Modifica di un brano

L'*activity* di modifica di un brano è stata rivista con l'obiettivo di espanderne le capacità e di semplificarne il codice.

Per far ciò è stata modificata la classe *ImportSongActivity*; questa classe è responsabile per l'aggiunta di un nuovo brano nella libreria dell'applicazione. Le modifiche sono state effettuate con lo scopo di fornire due funzionalità (aggiunta e modifica) a seconda dei parametri che riceve durante la creazione, in particolare, se nell'*intent* di avvio di questa classe è presente la variabile booleana *import* a *vero*, la classe procede con il normale caricamento dell'interfaccia ma, in più, procede a leggere le informazioni sul brano di cui si vogliono modificare le proprietà e i cui dati sono stati opportunamente reperiti e gestiti prima del lancio di questa *activity*. Una volta effettuata l'aggiunta o la modifica di un brano il tasto di conferma procede normalmente tranne nel caso in cui l'*activity* sia nella modalità di modifica. In questo caso prima procede a rimuovere il brano nel database in modo da permetterne nuovamente l'inserimento con i dati aggiornati.



(a) Creazione di un brano.

(b) Modifica di un brano.

Figura 4.21: *Activity ImportSongActivity* nei due casi.

4.14 Prestazioni

Usando lo strumento *Android Profler* sono state analizzate le prestazioni e i risultati dell'esecuzione su un Nexus 5 sono mostrati in figura 4.22.

L'utilizzo della CPU presenta picchi massimi pari al 60% soprattutto durante la fase di avvio e, durante l'esecuzione di un brano, l'elaborazione audio occupa circa il 10% a cui va ad aggiungersi il carico dell'elaborazione dell'interfaccia utente (bobine in rotazione, animazioni di manopole, ...) pari a circa un ulteriore 15%.

L'utilizzo della memoria si mantiene stabile sotto i 256 megabyte anche durante la consultazione di immagini di notevole dimensione.

Figura 4.22: Prestazioni dell'applicazione catturate usando *Android Profiler*.

4.15 Risultato finale

L'interfaccia grafica finale dell'applicazione non risulta essere troppo differente dalla versione originale. All'avvio viene caricata la schermata principale divisa visivamente in due sezioni.

La prima metà dello schermo mostra il magnetofono a sinistra, la seconda mostra tre pulsanti e un contatore con al di sotto un elemento visivo per la visione dei filmati associati ad un'opera.

Dei tre pulsanti, il primo permette di accedere alla schermata per la gestione delle opzioni avanzate che permettono di abilitare la manutenzione dei brani (aggiunta, modifica e cancellazione), cambiare la password per attivare queste opzioni, scegliere il percorso di riferimento per le operazioni che l'applicazione deve svolgere e il percorso in cui esportare il database delle opere.

Il secondo pulsante permette di accedere alla libreria delle opere. La schermata permette di visualizzare i dettagli delle opere e di caricarle nel magnetofono per eseguirle inoltre, tramite l'*Action Bar* permette di cercare e di aggiungere, modificare ed eliminare un brano se le opzioni avanzate risultano essere sbloccate, è possibile inoltre visualizzare nel dettaglio le immagini e il file *PDF* allegato ad un'opera, se presenti, premendo sugli elementi dell'interfaccia. Premendo sui tasti di aggiunta e modifica di un'opera si causerà l'apertura di una nuova schermata che guiderà l'utente nell'inserimento o modifica delle informazioni.

Il terzo e ultimo pulsante permette la regolazione dei volumi delle singole tracce presenti in un brano, oltre che la loro presenza o meno nella riproduzione.

Figura 4.23: Schermate principali dell'applicazione.

Capitolo 5

Conclusioni e sviluppi futuri

Il restauro e la conservazione di documenti sonori su supporti analogici tramite la digitalizzazione ne può assicurare la conservazione per un tempo virtualmente infinito inoltre, il trasferimento di un documento digitale da un supporto ad un altro non implica perdita di informazioni o, peggio ancora, degradazione del segnale audio.

La creazione di un'esperienza quanto più fedele possibile all'ascolto originale rimane comunque un compito non banale a causa dei vari aspetti coinvolti nel processo, dalla digitalizzazione di opere e documenti alla creazione di interfacce di riproduzione fedeli all'originale.

L'applicazione oggetto di questa tesi si pone come obiettivo la creazione di un'interfaccia scheumorfica che rappresenti in maniera fedele il comportamento di un magnetofono non solo dal punto di vista estetico, ma anche dal punto di vista tecnico e quindi sonoro.

Per quanto riguarda la simulazione dell'esperienza d'ascolto originale, l'introduzione della libreria *AAudio* in Android ha permesso di semplificare la gestione audio nelle applicazioni e di creare flussi audio con minor latenza aprendo la strada a nuovi tipi di applicazioni sonore avanzate. Il numero enorme di dispositivi e la loro diversità in funzioni, costi e capacità però rappresenta ancora un problema che impatta soprattutto la diffusione dei miglioramenti introdotti in ogni nuova release del sistema operativo.

In questi casi librerie come *Oboe* risultano essere indispensabili per fornire retro compatibilità e non aggravare troppo la complessità del codice sorgente.

L'audio stereo rimane ancora dominante nei dispositivi mobili ma la recente tendenza all'eliminazione della porta audio jack in favore dello standard USB Type C potrebbe portare ad un miglioramento della qualità audio dei dispositivi abilitando potenzialmente la riproduzione ad alta qualità (24 bit, 96 KHz) su un numero crescente di dispositivi.

Appendice A

Appendice

A.1 Confronto OpenSL ES, *AAudio* e *Oboe*

Di seguito viene riportato del codice con l'obiettivo di confrontare i tre modi attualmente disponibili per gestire dell'audio nel codice nativo. L'obiettivo è quello di rappresentare come il codice diventi via via più leggibile e facile da gestire man mano che si passa da *OpenGL ES* alla libreria *Oboe*.

A.1.1 OpenSL ES

```
//create the Engine object
(*engineItf)->CreateOutputMix(engineItf, &output_mix_obj, 0,
    NULL, NULL);
(*output_mix_obj)->Realize(output_mix_obj, SL_BOOLEAN_FALSE);
// configure audio source
SLDataLocator_AndroidSimpleBufferQueue loc_bq = {
    SL_DATALOCATOR_ANDROIDSIMPLEBUFFERQUEUE, 2 };

SLuint32 slSampleRate;
switch (sampleRate)    {
        case 96000:
                slSampleRate = SL_SAMPLINGRATE_96; //!!! non
                    supportata
                break;
        ...
}

SLuint32 slPcmFormat;
switch (bitPerSample){
        case SF_FORMAT_PCM_S8:
                slPcmFormat = SL_PCMSAMPLEFORMAT_FIXED_8;
                break;
        ...
}

SLDataFormat_PCM format_pcm = { SL_DATAFORMAT_PCM, 2,
    slSampleRate, slPcmFormat, slPcmFormat, 0,
    SL_BYTEORDER_LITTLEENDIAN };
```

```
SLDataSource audioSrc = { &loc_bq, &format_pcm };
// configure audio sink
SLDataLocator_OutputMix loc_outmix = { SL_DATALOCATOR_OUTPUTMIX,
    output_mix_obj };Google I/O 2017
SLDataSink audioSnk = { &loc_outmix, NULL };
//create the object
const SLInterfaceID ids[] = { SL_IID_BUFFERQUEUE };
const SLboolean req[] = { SL_BOOLEAN_TRUE };
(*engineItf)->CreateAudioPlayer(engineItf, &player_obj, &
   audioSrc, &audioSnk, 1, ids, req);
(*player_obj)->Realize(player_obj, SL_BOOLEAN_FALSE);
(*player_obj)->GetInterface(player_obj, SL_IID_PLAY, &player);

(*player_obj)->GetInterface(player_obj, SL_IID_BUFFERQUEUE, &
   player_buf_q);

(*player)->SetPositionUpdatePeriod(player, 100);

(*player)->RegisterCallback(player, timeUpdate, NULL);

// register callback on the buffer queue
(*player_buf_q)->RegisterCallback(player_buf_q, playbackCallback
   , NULL);

LOGD("creato audio engine");
```

A.1.2 *AAudio*

```
void NativePlayer::setupAudioEngine(int playbackDeviceId_, int
    sampleFormat_, int sampleChannels_, int sampleRate_) {
AAudioStreamBuilder *builder = createStreamBuilder();
    if (builder != nullptr) {
        setupPlaybackStreamParameters(builder, playbackDeviceId_
            , sampleFormat_, sampleChannels_, sampleRate_);
        aaudio_result_t result = AAudioStreamBuilder_openStream(
            builder, &playStream_);
        if (result == AAUDIO_OK && playStream_ != nullptr) {
            ...
        }
    }
}

AAudioStreamBuilder *NativePlayer::createStreamBuilder() {

    AAudioStreamBuilder *builder = nullptr;
    aaudio_result_t result = AAudio_createStreamBuilder(&builder
        );
    if (result != AAUDIO_OK) {
        LOGE("Error %s", AAudio_convertResultToText(result));
    }
    return builder;
}

void NativePlayer::setupPlaybackStreamParameters(
    AAudioStreamBuilder *builder, int playbackDeviceId_, int
    sampleFormat_, int sampleChannels_, int sampleRate_) {
    AAudioStreamBuilder_setDeviceId(builder, playbackDeviceId_);
    AAudioStreamBuilder_setFormat(builder, sampleFormat_);
    AAudioStreamBuilder_setChannelCount(builder, sampleChannels_
        );
    AAudioStreamBuilder_setSampleRate(builder, sampleRate_);
    AAudioStreamBuilder_setSharingMode(builder,
        AAUDIO_SHARING_MODE_EXCLUSIVE);
    AAudioStreamBuilder_setPerformanceMode(builder,
        AAUDIO_PERFORMANCE_MODE_LOW_LATENCY);
    AAudioStreamBuilder_setDirection(builder,
        AAUDIO_DIRECTION_OUTPUT);
    AAudioStreamBuilder_setDataCallback(builder, ::dataCallback,
         this);
    AAudioStreamBuilder_setErrorCallback(builder, ::
        errorCallback, this);
}
```

A.1.3 *Oboe*

```
void NativePlayer::setupAudioEngineAndPlay(int playbackDeviceId_
    , oboe::AudioFormat sampleFormat_, int sampleChannels_, int
    sampleRate_) {

     oboe::AudioStreamBuilder builder;
     builder.setDeviceId(playbackDeviceId_);
     builder.setDirection(oboe::Direction::Output);
     builder.setSharingMode(oboe::SharingMode::Exclusive);
     builder.setSampleRate(sampleRate_);
     builder.setChannelCount(sampleChannels_);
     builder.setFormat(sampleFormat_);
     builder.setPerformanceMode(oboe::PerformanceMode::LowLatency
         );
     builder.setCallback(this);

     oboe::Result result = builder.openStream(&stream);

     if (result == oboe::Result::OK) {
     ...
```

A.2 Errori comuni

Poiché l'uso delle *callback* nell'applicazione è di fondamentale importanza, in questa sezione verranno discussi alcuni comportamenti da evitare per non incorrere in distorsioni e difetti audio.

Come riferimento si userà il seguente codice in C++, presentato e discusso durante la presentazione ufficiale della libreria *AAudio* al Google I/O 2017 [33] che esprime in maniera schematizzata un insieme istruzioni non adatte ad essere inserite in una *callback* audio.

```
  void badCallback(SLAndroidBufferQueueItf bq, void *context){
1     __android_log_print(LOG, TAG, "soo fancy, such logging");
2     int16_t *newAudioBuffer = new int16_t[LOTS_OF_ITEMS];
3     while(!isOtherThreadReady){
4         audioFile.open();
4         audioFile.read(newAduioBuffer, size);
5         usleep(50);
      }
  }
```

Il codice presenta i 5 errori più comuni e di seguito vengono presentate le linee guida da seguire per evitarli:

1. Evitare di fare logging: esistono strumenti più adatti in Android per queste circostanze, come ad esempio *Systrace*.

2. Non allocare memoria: se si ha bisogno di memoria essa va allocata durante la creazione del flusso audio.

3. Non aspettare altri *thread*: una *callback AAudio* è ad alta priorità, se attendo *thread* a priorità più bassa rischio di creare problemi di priorità inversa in cui il *thread* ad alta priorità verrà effettivamente bloccato da un *thread* a priorità più bassa.

4. Non effettuare operazioni di lettura/scrittura: se dovesse essere necessario è meglio creare un altro *thread* e usare *buffer* o code circolari per trasferire i dati.

5. Non attendere: non dovrebbe esserci nessuna necessità di aspettare in una *callback* di questo tipo.

Bibliografia

[1] C. Fantozzi, F. Bressan, N. Pretto, and S. Canazza, "Tape music archives: from preservation to access," *International Journal of Digital Libraries*, vol. 18, no. 233, 2017.

[2] Choo&U, "Choosing and using mrl calibration tapes for audio tape recorder standardization," Magnetic Reference Laboratory Inc, Tech. Rep., 2016.

[3] M. Camras, *Magnetic Recording Handbook*, 1988.

[4] E. Cohen, "Preservation of audio in folk heritage collections in crisis," vol. Proceedings of Council on Library and Information Resources, Washington, DC, USA, 2001.

[5] Centro di Sonologia Computazionale. [ultimo accesso: 20 ottobre 2018]. [Online]. Available: smc.dei.unipd.it/

[6] N. Pretto and S. Canazza, "Rewind: Simulazione di un'esperienza d'ascolto storicamente fedele di dischi fonografici digitalizzati," in *Proceedings of the XX Colloquium of Musical Informatics, XX CIM*, Roma, October 2014.

[7] S. Canazza, C. Fantozzi, and N. Pretto, "Accessing tape music documents on mobile devices," *ACM Transaction on Multimedia Computing, Communication and Application*, vol. 12, no. 1s, p. 20, 2015.

[8] Sistema operativo Android. [ultimo accesso: 24 settembre 2018]. [Online]. Available: android.com

[9] Android SDK. [ultimo accesso: 20 settembre 2018]. [Online]. Available: developer.android.com/studio

[10] Android Studio. [ultimo accesso: 20 settembre 2018]. [Online]. Available: developer.android.com/studio

[11] Android NDK. [ultimo accesso: 20 settembre 2018]. [Online]. Available: developer.android.com/ndk

[12] Git. [ultimo accesso: 20 ottobre 2018]. [Online]. Available: git-scm.com

[13] GitLab. [ultimo accesso: 20 ottobre 2018]. [Online]. Available: about.gitlab.com

[14] Piattaforma Android. [ultimo accesso: 20 settembre 2018]. [Online]. Available: developer.android.com/guide/platform

[15] High-Performance Audio in Android. [ultimo accesso: 7 agosto 2018]. [Online]. Available: developer.android.com/ndk/guides/audio

[16] Libreria AAudio. [ultimo accesso: 1 agosto 2018]. [Online]. Available: developer.android.com/ndk/guides/audio/aaudio/aaudio

[17] Libreria AAudio nel Native Development Kit. [ultimo accesso: 2 agosto 2018]. [Online]. Available: developer.android.com/ndk/guides/stable_apis

[18] PresentazioneLibreria Oboe. [ultimo accesso: 18 ottobre 2018]. [Online]. Available: android-developers.googleblog.com/2018/10/introducing-oboe-c-library-for-low.html

[19] Libreria Oboe. [ultimo accesso: 20 settembre 2018]. [Online]. Available: github.com/google/oboe

[20] M. Ambrico, "Implementazone su piattaforma mobile di filtri audio digitali per documenti musicali storici," 2016.

[21] L. Bianconi, "Progettazione e sviluppo di un'applicazione mobile per la simlazone dell'ascolto quadrifonico di un documento sonoro," 2014.

[22] D. Colanardi, "Progetto e realizzazione di un'interfaccia utente scheumorfica per la fruizione di documenti sonori storici su dispositivi mobili," 2014.

[23] D. Dosso, "Gestione su dispositivi mobili di un archivio per la fruizione di documenti sonori," 2014.

[24] Gestione dei permessi in Android. [ultimo accesso: 8 agosto 2018]. [Online]. Available: developer.android.com/guide/topics/permissions/overview

[25] Esempio di applicazioni che implementano la libreria AAudio. [ultimo accesso: 2 agosto 2018]. [Online]. Available: github.com/googlesamples/android-audio-high-performance/tree/master/aaudio

[26] Libreria LAME. [ultimo accesso: 23 luglio 2018]. [Online]. Available: lame.sourceforge.net

[27] PdfRenderer Android. [ultimo accesso: 28 agosto 2018]. [Online]. Available: developer.android.com/reference/android/graphics/pdf/PdfRenderer

[28] Libreria AndroidPdfViewer. [ultimo accesso: 28 agosto 2018]. [Online]. Available: github.com/barteksc/AndroidPdfViewer

[29] *NAB Standard: magnetic tape recording and reproducing (reel-to-reel).*

[30] Gestione della memoria. [ultimo accesso: 14 settembre 2018]. [Online]. Available: developer.android.com/topic/performance/graphics/manage-memory

[31] Gestione delle Bitmap. [ultimo accesso: 14 settembre 2018]. [Online]. Available: developer.android.com/topic/performance/graphics/load-bitmap

[32] Libreria Glide. [ultimo accesso: 22 settembre 2018]. [Online]. Available: github.com/bumptech/glide

[33] Google I/O 2017, presentazione libreria AAudio. [ultimo accesso: 9 agosto 2018]. [Online]. Available: youtube.com/watch?v=C0BPXZIvG-Q

[34] E. Micheloni, N. Pretto, and S. Canazza, "A step toward ai tools for quality control and musicological analysis of digitized analogue recordings: recognition of audio tape equalizations," in *Proceedings of the 11th International Workshop on Artificial Intelligence for Cultural Heritage co-located with the 16th International Conference of the Italian Association for Artificial Intelligence (AI*CH 2017) - pages 17–24,Bari, Italy*, November 2017.

[35] S. Verde, N. Pretto, S. Milani, and S. Canazza, "Stay true to the sound of history: Philology, phylogenetics and information engineering in musicology," *Applied Sciences*, vol. 8, no. 2, 2018. [Online]. Available: http://www.mdpi.com/2076-3417/8/2/226

Ringraziamenti

Vorrei ringraziare i miei genitori per avermi sostenuto in questi anni e per avermi permesso di intraprendere questo percorso.

Ringrazio il Prof. Sergio Canazza, l'Ing. Niccolò Pretto e il Dott. Edoardo Micheloni per avermi dato la possibilità di contribuire a questo progetto e avermi guidato durante lo svolgimento di questo lavoro.

Ringrazio infine tutte le amicizie formatesi durante questi anni di studio e un grazie in particolare al Dott. Andrea Pastea Mattiazzo per il suo prezioso contributo nello stressare l'applicazione.