UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

IMAGESOUND: STUDIO, PROGETTO E REALIZZAZIONE DI UN'APP BASATA SU MACHINE LEARNING PER CONVERTIRE IMMAGINI IN SUONI E VICEVERSA

IMAGESOUND: STUDY, DESIGN AND IMPLEMENTATION OF A MACHINE LEARNING BASED APP TO CONVERT IMAGES INTO SOUNDS AND VICE VERSA

Relatore: Chiar.mo
Prof. Aldo Franco Dragoni

Tesi di Laurea di:
Luca Sanchioni

A.A. 2021/2022

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Contesto app ImageSound

In questi ultimi anni è sempre più diffusa la pratica di scattare foto o registrare video tramite smartphone ad avvenimenti piacevoli che ci succedono, elementi che in quel momento attraggono la nostra attenzione al fine di conservare una traccia dell'accaduto nel futuro oppure semplicemente per condividere l'esperienza di quel momento o facilitare la comunicazione con le persone che conosciamo. Considerazioni simili possono essere fatte anche per quanto riguarda il suono ed in particolar modo a quello acquisito tramite microfono del dispositivo.

Lo scopo della tesi è quello di realizzare un'applicazione Android chiamata ImageSound in cui viene definita una **relazione bidirezionale diretta fra informazione uditiva e visiva**, solitamente considerati due flussi concettualmente distinti e cooperanti tra loro, al fine di unire le due tipologie eterogenee di informazione in un unico aspetto complessivo. Questo per valorizzare ulteriormente il momento temporale catturato ed incentivare maggiormente all'acquisizione delle informazioni audiovisive anche in situazioni che normalmente, con i due flussi separati, riteniamo poco interessanti e quindi non meritevoli di tempo da dedicare. Infatti si vuol far riscoprire la realtà con un punto di vista completamente diverso rispetto la concezione standard, facendo sentire l'utilizzatore dell'applicazione come se fosse un artista mobile nel mondo in procinto di realizzare la sua creazione.

Per favorire ulteriormente la creatività durante la creazione vengono anche definite funzionalità aggiuntive (strumenti a disposizione dell'artista) che variano i risultati prodotti come la classificazione delle immagini in base allo stile o elemento architetturale presente, la possibilità di utilizzare note musicali preferite e la classificazione dell'audio in base all'evento che accade o al genere musicale.

Tutto questo fa si che gli **ambiti di utilizzo** principali dell'applicazione siano molteplici come la finalità prettamente ludica o di divertimento, oppure nel settore culturale come i musei per rendere ancora più interattiva ed evoluta digitalmente l'esperienza (attirando anche maggiormente il pubblico più giovane), naturalmente in ambito musicale

utilizzando l'applicazione come se fosse un nuovo strumento musicale digitale, ecc. . Inoltre l'applicazione può anche essere utilizzata come base di partenza per la definizione di algoritmi in ambiti molto più astratti e complessi come dare una semantica visiva a versi o rumori di animali, svolgere azioni automatiche opportune a seconda dell'evento audio rilevato oppure come strumento per rendere più coinvolgente e facile l'apprendimento dei concetti ai bambini.

1.2 Stato dell'arte app ImageSound

In questa sezione viene fatta una panoramica sullo stato dell'arte riguardante le principali tecniche utilizzate per la conversione da immagine a suono e viceversa (da suono ad immagine). Conclusa l'analisi viene illustrato in modo generale e sintetico l'algoritmo utilizzato per la realizzazione dell'applicazione Android ImageSound, svolgendo anche un confronto dell'app con l'insieme di metodi analizzati. L'analisi dettagliata dell'algoritmo utilizzato nell'app verrà fatta poi successivamente nei relativi capitoli dedicati della tesi.

Per i principali processi di conversione da immagine a suono utilizzati nello stato dell'arte sono state definite in modo opportuno **4 diverse categorie logiche di algoritmi** [1] (mostrando per ciascuna almeno due metodi):

- **Mapping diretto base dei pixel dell'immagine in suono:** si produce il suono utilizzando le informazioni dei pixel e l'immagine di mapping deve essere nel modello colore RGB o in scala di grigi (abbreviato in **mapping pixel base**);
- **Mapping diretto avanzato dei pixel dell'immagine in suono:** si produce il suono utilizzando le informazioni dei pixel e l'immagine di mapping deve utilizzare un modello colore che fa uso della tonalità H come HSV o HSL (abbreviato in **mapping pixel avanzato**);
- **Mapping focalizzato solo su aspetti sonori:** per la conversione si utilizzano solamente concetti che riguardano il suono (abbreviato in **mapping sonoro**);
- **Analisi dell'immagine per l'illustrazione sonora dei risultati:** si ispeziona e processa l'immagine al fine di restituire gli output testuali ottenuti sotto forma di risultati sonori, ad esempio con text to speech TTS (abbreviato in **analisi immagine TTS**).

Inoltre alcuni dei metodi vengono utilizzati per aiutare le persone ipovedenti [2] nella percezione dell'ambiente circostante al fine di semplificare la navigazione lungo il percorso. Siccome è una caratteristica comune in metodi di categoria anche diversa tra loro, non è stata definita una categoria specifica per tale aspetto ma lasciata come funzionalità fornita o meno dallo specifico approccio.

Nella Tabella 1.1 sono elencati i principali metodi di conversione da immagine a suono determinati nello stato dell'arte (indicando anche se viene definita la metodologia per l'operazione inversa da suono ad immagine).

Trovare singolarmente metodi di conversione da suono ad immagine nello stato dell'arte risulta invece molto più complesso rispetto al caso inverso precedente (da immagine a

Categoria logica di algoritmo	Processo reversibile	Ausilio ipovedenti	Descrizione metodologia
Mapping pixel base	Si	No	Conversione dei pixel a 24 bit dell'immagine (modello colore RGB) in **frequenze standard** note musicali e con queste si genera una sequenza finale di campioni audio [3]
Mapping pixel base	No	Si	Conversione a blocchi (scansione per colonne) dei pixel dell'immagine (conversione da modello colore RGB a scala di grigi) in segnali sinusoidali (frequenze e ampiezze che dipendono dai valori dei pixel) e applicazione trasformata veloce di Fourier inversa (**IFFT**) alle sinusoidi (si considera solo la parte reale) per ottenere una sequenza finale di campioni audio [4]
Mapping pixel base	Si	No	La luminosità di ciascuno dei pixel dell'immagine (in scala di grigi a 8 bit e con valori nell'intervallo da 0 a 255) viene **scalata linearmente** nell'intervallo da -1.0 a +1.0 dei campioni audio [5]
Mapping pixel avanzato	No	Si	Conversione a blocchi (scansione per righe) dei pixel dell'immagine (modello colore HSV) in un insieme di parametri audio secondo il mapping che fa corrispondere la tonalità ad una **nota musicale**, la saturazione al **timbro** (andamento della forma d'onda con cui si genera il suono, tra sinusoidale o quadrata), il valore al **volume** dell'audio (tra 0 e 1), la posizione dei pixel come **azimut** (posizione sorgente audio sul piano orizzontale) e generazione del suono finale con tali parametri [6] [7] [8]
Mapping pixel avanzato	No	Si	Conversione a blocchi (scansione per righe) dei pixel dell'immagine (modello colore HSL) in suono con la codifica ternaria in cui mediante la tonalità H si associa uno dei sette **strumenti musicali** possibili, con la saturazione S si sceglie fra 4 possibili note musicali e con la luminosità L si sceglie fra 8 possibili frequenze (4 basse e 4 alte) [9]
Mapping sonoro	Si	No	Conversione utilizzando il concetto di **cromatismo** del suono (colore assegnato ad un suono in base alle distanze intervallari tra le note e non in base al valore specifico assunto dalle note stesse) [10] [11]
Mapping sonoro	No	No	Si fa coincidere lo **spettrogramma** del suono (analisi dettagliata sulle strutture ritmiche del suono e relative variazioni) con l'immagine da cui si vuole ottenere il suono [12]
Analisi immagine TTS	No	Si	Calcolo dall'immagine, nel modello colore HSV, dell'**istogramma delle luminosità V** per interrogare un database e determinare la classe di elemento più similare all'istogramma di input, restituendo una voce audio TTS che illustra i risultati ottenuti [13]
Analisi immagine TTS	No	Si	Riconoscimento con una rete neurale degli **oggetti presenti** nell'immagine, restituendo una voce audio TTS che illustra i risultati ottenuti [14]

Tabella 1.1: Stato dell'arte dei metodi di conversione da immagine a suono

suono). Comunque è stato trovato un metodo rilevante che a partire dallo spettrogramma dell'audio di input restituisce un'immagine che rappresenta concettualmente la sorgente che ha prodotto il suono [15]. Questo mediante il confronto di similarità del dato di input con un database contenente le associazioni tra spettrogramma audio ed immagine della relativa sorgente.

Nell'applicazione Android ImageSound il processo di conversione da immagine a suono rientra in una **categoria logica ibrida fra quella del mapping pixel avanzato e**

dell'analisi immagine TTS. Infatti nel mapping tra pixel e suono sono utilizzate solo immagini (foto scattate) nel modello colore HSL (conversione da RGB, più accurato per aspetti colorimetrici) ed in caso di attivazione di funzionalità aggiuntive che comprendono la classificazione (utilizzo di una rete neurale per le immagini), vengono restituiti i risultati ottenuti non in modo sonoro con TTS ma solamente in modo testuale. Inoltre i risultati ottenuti dalla rete neurale influenzano anche il tipo di mapping utilizzato per i passaggio dai pixel dell'immagine al suono finale. Dei metodi di conversione da immagine a suono presenti nello stato dell'arte, l'app ImageSound non segue in modo specifico nessuno di questi ma consiste in minima parte nell'unione di alcuni concetti chiave appartenenti a metodi diversi, aggiungendo multipli aspetti completamente nuovi ed originali per la restante parte significativa.

Nel mapping da pixel immagine a frequenza standard delle note musicali (ad esempio per la nota musicale A_4 si utilizza la frequenza 440Hz) per quanto riguarda la componente tonalità H viene utilizzata la **relazione di Newton** [16] tra i 7 colori dell'arcobaleno (7 diverse tonalità in cui può rientrare un pixel della foto) e le 7 note musicali (pitch). Per le restanti due componenti di HSL si associa il valore di luminosità L con l'ottava musicale (ottava più alta al crescere del valore di luminosità) e la saturazione S con l'alterazione musicale (tra nota musicale senza alterazione, bemolle o diesis rispettivamente per valori di saturazione medi, bassi e alti, forzando a non utilizzare l'alterazione in caso di coppia pitch e alterazione non esistente). Viene anche definita la relazione inversa per il mapping da frequenza standard delle note musicali a pixel immagine HSL.

Per quanto riguarda l'ordine di scansione dei pixel con cui andare a generare le frequenze standard non viene utilizzata una scansione lineare standard (per righe o per colonne) ma viene utilizzata la **curva di Hilbert** [17]. La curva di Hilbert permette il passaggio da uno spazio 2D (immagine con dimensioni larghezza e altezza) a 1D (lista di elementi audio) attraverso la definizione di uno scorrimento che attraversa tutti gli elementi 2D (i pixel) in modo tale da ottimizzare il fatto che elementi vicini nello spazio di partenza lo saranno anche nello spazio di arrivo. Inoltre la relazione è reversibile permettendo anche il passaggio da spazio 1D a 2D.

Quindi riassumendo in breve (ed astraendo da alcuni dettagli implementativi spiegati in seguito) il processo per passare da immagine a suono nell'app ImageSound è il seguente:

1. Conversione della foto scattata da RGB a HSL;
2. Applicazione della curva di Hilbert all'immagine per passare da spazio 2D a 1D (lista di elementi ognuno definito come terna delle componenti HSL);
3. Applicazione ad ogni elemento della lista 1D del mapping definito tra componenti HSL e frequenza standard delle note musicali (in cui vi è la relazione di Newton) al fine di ottenere una lista di frequenze;
4. Mediante la lista di frequenze si definisce il suono finale in output come sinusoide a frequenza variabile nel tempo (sweep) definita mediante un accumulatore di fase.

Il processo inverso da suono ad immagine viene svolto facilmente grazie alla reversibilità della curva di Hilbert, seguendo per la maggior parte i medesimi passi concettuali in

modo inverso. L'unica differenza è che anziché avere come suono in input la sinusoide (output processo da immagine a suono) si hanno campioni audio PCM (informazioni audio numeriche in un range intero o float di valori definiti) acquisiti tramite microfono del dispositivo Android. Questo richiede uno step differente per convertire la lista di campioni PCM in lista di frequenze standard delle note musicali. Per la conversione dei campioni PCM sono utilizzate 2 tecniche ovvero lo zero crossing (tecnica standard) e la tecnica che è stata chiamata divisione (simile ad uno scalo lineare dei valori in un range).

Dalle considerazioni fatte fino ad ora l'app Android ImageSound è in grado di:

- Convertire una foto scattata dal dispositivo in suono di durata finita;
- Convertire i campioni audio acquisiti tramite microfono del dispositivo, in un range di secondi, in immagine;
- Convertire continuamente una sequenza di frame immagine (video) acquisiti dal dispositivo in suono non limitato in durata;
- Convertire continuamente i campioni audio acquisiti tramite microfono del dispositivo in video;
- Possibilità di attivare tra 2 funzionalità aggiuntive nella conversione da immagine a suono ovvero tra il riconoscimento dello stile o elemento architetturale [18] presente nell'immagine (variando di conseguenza il suono prodotto mediante il concetto di corrispondenza cross modale tra oggetti nello spazio e colori [19], il quale si basa sulla complessità visuale [20] [21] [22] e la forma degli oggetti) oppure la possibilità di comporre musica (variando i parametri audio quali nota e/o ottava per la generazione del suono);
- Possibilità di attivare la funzionalità aggiuntiva nella conversione da suono ad immagine per il riconoscimento nei campioni sonori acquisiti tramite microfono dell'evento audio che accade o del genere musicale.

1.3 Organizzazione della tesi

La **tesi è stata suddivisa** concettualmente in due parti ovvero nella **prima parte si affrontano i concetti teorici di base** necessari per lo sviluppo dell'app mentre nella **seconda parte vengono definiti in modo accurato i dettagli di implementazione** della parte sperimentale dell'applicazione.

Si inizia quindi con una breve introduzione al sistema operativo Android nel capitolo 2 per poi passare al capitolo 3 in cui vengono fornite le informazioni sui due principali dati digitali elaborati in input e in output dall'app (immagine e audio). Nel capitolo 4 viene presentata la libreria utilizzata per la gestione dell'audio in Android mentre nel capitolo 5 viene invece mostrata la libreria impiegata per la gestione della fotocamera.

Nel capitolo 6 si passa alla parte sperimentale della tesi in cui viene mostrata l'architettura complessiva utilizzata nella conversione da immagine della fotocamera a suono (chiamato processo Image-Sound) mentre nel capitolo 7, in modo inverso, viene mostrata

l'architettura complessiva utilizzata nel passaggio da audio acquisito tramite microfono ad immagine (processo Sound-Image). Segue nel capitolo 8 la presentazione delle 4 modalità di elaborazione utilizzabili nell'app (due di tipo Image-Sound e due di tipo Sound-Image), i dettagli sulle funzionalità aggiuntive nelle due modalità di tipo Image-Sound (capitolo 9) e nelle due modalità di tipo Sound-Image (capitolo 10). Infine nel capitolo 11 si passa a mostrare le schermate di esecuzione dell'app mentre nel capitolo 12 vengono fatte le considerazioni finali riguardanti il progetto sviluppato.

Capitolo 2

Il sistema operativo Android

2.1 Introduzione ad Android

Il sistema operativo open source Android è basato su Linux ed è stato creato principalmente per dispositivi mobili come smartphone e tablet. Le componenti principali della **piattaforma Android** [23], mostrate nella Figura 2.1, sono:

- **Kernel Linux:** rappresenta la componente più rilevante dell'intera architettura, permette ad Android di beneficiare delle sue principali funzionalità di sicurezza e facilita ai produttori di dispositivi lo sviluppo di driver hardware;
- **Hardware Abstraction Layer (HAL):** insieme di librerie in cui ognuna implementa un'interfaccia verso un componente hardware specifico (come il modulo bluetooth) al fine di far accedere, il framework API Java, alle funzionalità hardware del dispositivo attraverso il caricamento da parte del sistema Android delle opportune librerie;
- **Android Runtime:** la tipologia di Android Runtime utilizzato dipende dalla versione di Android installata nel dispositivo ed in particolare se la versione è precedente alla 5.0 (livello API 21) si utilizza il Dalvik altrimenti l'ART (ogni applicazione di questa tipologia viene eseguita nel proprio processo e con la propria istanza di ART). ART esegue file DEX (formato bytecode ottenuto dagli strumenti di compilazione mediante la compilazione di sorgenti Java) oppure codice macchina (in versioni di Android uguali o maggiori alla 9.0 mediante la conversione dei file DEX per migliorare le prestazioni) al fine di riuscire ad eseguire più macchine virtuali sui dispositivi utilizzando una quantità di memoria limitata. Inoltre ART fa uso del kernel Linux per alcune delle sue funzionalità come il threading e la gestione della memoria di basso livello;
- **Librerie C/C++ Native:** in applicazioni che utilizzano il codice C/C++ (come le componenti principali del sistema Android tra cui ART e HAL) si accede direttamente alle librerie C/C++ native attraverso l'Android NDK [24] mentre in applicazioni

che utilizzano il codice Java si accede alle librerie C/C++ native attraverso l'API Framework Java (overhead);

- **Framework API Java:** API scritte in linguaggio Java per poter utilizzare tutte le possibili funzioni messe a disposizione dal sistema operativo Android al fine di creare proprie app;
- **Applicazioni di sistema:** serie di app predefinite incluse in Android per soddisfare specifiche necessità distinte (e-mail, messaggi SMS, navigazione internet, ecc.) e la cui app predefinita per una specifica necessità può anche essere cambiata (con alcune eccezioni) con una di terze parti scelta dall'utente.

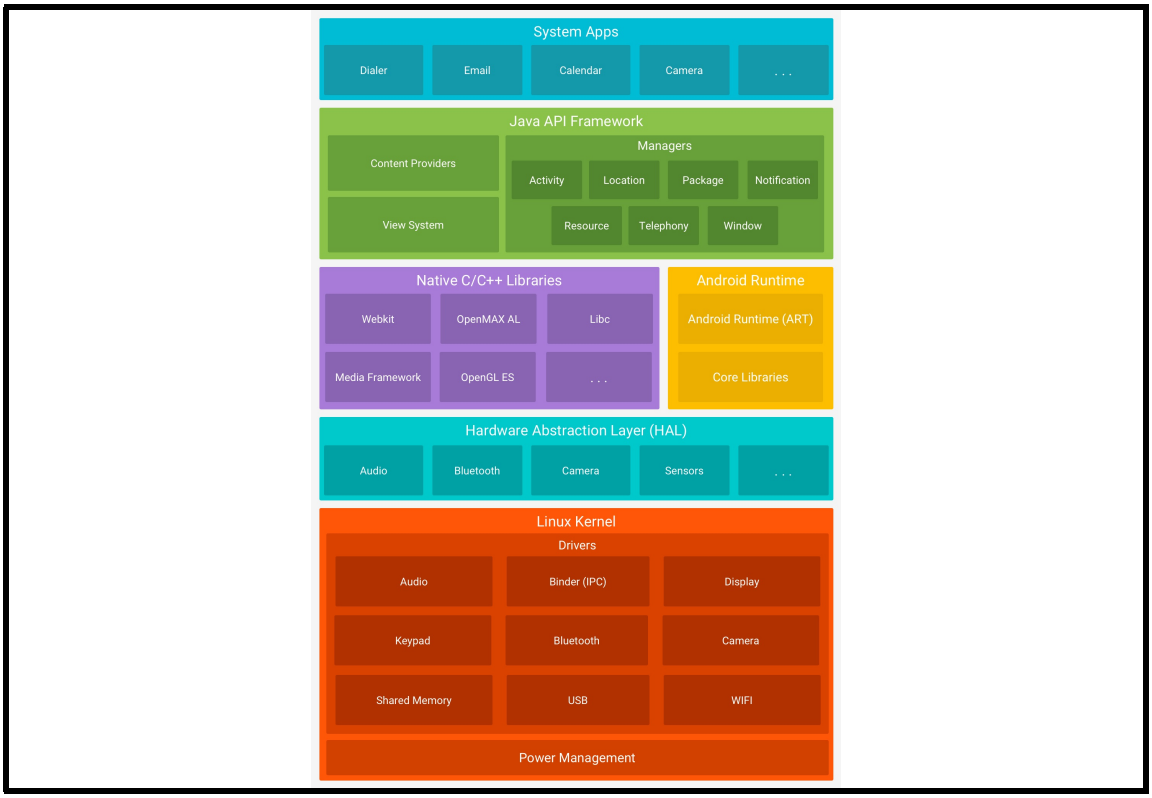

Figura 2.1: Architettura software Android

2.2 Ambiente di sviluppo dell'applicazione Android: Android Studio

Per lo sviluppo dell'applicazione Android ImageSound è stato utilizzato l'IDE (Integrated Development Environment) ufficiale chiamato **Android Studio** [25]. Questo mette a

disposizione una suite completa di funzionalità per facilitare la creazione di applicazioni Android tra cui:

- **Android SDK (Software Development Toolkit):** kit completo di strumenti di sviluppo per codice scritto in Java;
- **Android NDK (Native Development Toolkit):** simile al SDK ma per codice scritto in linguaggio nativo C/C++;
- **Debugger:** per identificare la posizione e la tipologia di errore nel codice al fine di risolvere la problematica;
- **Emulatore:** possibilità di creare dispositivi Android virtuali, con relativa scelta della versione di Android, su cui andare a testare le app sviluppate.

2.3 Versioni di Android e versione minima dell'app ImageSound

Al fine di compilare l'app mediante l'IDE Android Studio è necessario prima specificare la versione minima di Android su cui si desidera poter installare l'applicazione (*minSdkVersion*). Infatti esistono multiple varianti di SDK [26] in cui ad ognuna è associato una specifica versione di Android o livello API e le varie varianti di SDK possono essere scaricate in Android Studio mediante il relativo SDK Manager. Le varianti di SDK disponibili si trovano nel range che va da Android 2.3 (livello API 9) ad Android 13 (livello API 33).

Per quanto riguarda l'app ImageSound è stata scelta la **versione di Android 5.0 (livello API 21)** sia per offrire il giusto compromesso tra compatibilità con dispositivi più datatati e funzionalità complessive disponibili sia per rispettare le versioni minime richieste dalle varie librerie utilizzate nel progetto.

2.4 Classificazione in dispositivi mobili: TensorFlow Lite

La classificazione consiste nel suddividere gli elementi di un dominio in gruppi o classi in cui tutti gli elementi di una data classe presentano una specifica relazione di appartenenza in comune. Mediante la libreria TensorFlow [27] è possibile creare **modelli addestrati di machine learning** per dispositivi generici (previa eventuale conversione di formato), i quali possono essere utilizzati per svolgere le inferenze (assegnazione della classe tra quelle definite ad un dato di input).

Per svolgere le inferenze in dispositivi Android è necessario effettuare prima la conversione del modello addestrato TensorFlow in **TensorFlow Lite** [28] (file in formato tflite) al fine di rendere il modello più portatile, efficiente e con una minore occupazione di memoria.

2.5 Linguaggi di programmazione utilizzati nell'app ImageSound

Per lo sviluppo dell'app ImageSound sono stati utilizzati **3 linguaggi di programmazione**, ognuno dei quali è associato ad una determinata porzione logica dell'applicazione, ovvero:

- **Java:** utilizzato per gli elementi di interfaccia grafica dell'app Android;
- **C++ (tramite NDK):** utilizzato per tutte le principali elaborazioni dell'app, computazionalmente più intense, in modo tale da ottenere le migliori prestazioni dal dispositivo tra cui una bassa latenza;
- **Python:** utilizzato per addestrare i modelli TensorFlow mediante la libreria Keras [29] (API Python di interfacciamento a TensorFlow) ed effettuare la conversione dei modelli addestrati in TensorFlow Lite.

Inoltre viene utilizzata anche l'interfaccia **JNI** (Java Native Interface) la quale permette di definire metodi Java la cui vera implementazione viene fatta in linguaggio nativo C++. Questo per ottenere migliori prestazioni di esecuzione rispetto al solo utilizzo del Java ma inferiori prestazioni rispetto al solo utilizzo del linguaggio nativo C++ (dovuto al passaggio di dati aggiuntivi tra Java e C++).

Capitolo 3

Principali dati digitali elaborati

3.1 Immagine digitale

Le immagini digitali sono la tipologia di immagini visualizzate, prodotte o manipolate dai dispositivi elettronici di consumo come smartphone, fotocamere, computer, etc. . Le immagini digitali vengono definite mediante una delle seguenti **tecniche di rappresentazione**:

- **Raster:** si utilizza una matrice di elementi puntuali chiamati **pixel**, associando ad ognuno di questi l'informazione di colore o intensità con un valore numerico scalare, rappresentato digitalmente mediante un certo numero di bit di informazione (come 8 o 24 bit, definendo il concetto di profondità dell'immagine ovvero il range di possibili valori numerici utilizzabili con tale quantità di bit). Ad esempio in immagini in scala di grigi viene utilizzata la componente intensità a 8 bit con range monocromatico che va dal nero al bianco. Le dimensioni della matrice AxB (con A righe e B colonne) dipendono rispettivamente dalle dimensioni di larghezza e altezza dell'immagine utilizzata. Inoltre le immagini vengono memorizzate in formati file immagine come ad esempio il JPEG (compressione di tipo lossy ovvero riduzione della quantità di dati utilizzati con perdita di informazione), GIF, PNG (entrambi con compressione di tipo lossless ovvero senza perdita di informazione);
- **Vettoriale:** si definisce l'immagine come **insieme di primitive geometriche** quali punti, linee o poligoni. Rispetto al raster si facilitano operazioni come lo scalo o la rotazione dell'immagine (nessuna perdita di dettaglio) in quanto le funzioni sono applicate ad elementi geometrici anziché a matrici. Uno dei formati file tipicamente utilizzati è l'EPS.

Le immagini utilizzate nell'applicazione sono solo di tipo raster in quanto come dato di input vengono utilizzate foto scattate mediante la fotocamera del dispositivo e prodotte in output immagini costituite da un insieme di pixel.

3.2 Modelli colore delle immagini

L'informazione associata a ciascuno dei pixel dell'immagine dipende dal modello colore [30] utilizzato ovvero un modello matematico astratto, impiegato per la **descrizione numerica dei colori** e generalmente è costituito da almeno tre componenti. L'insieme di tutte le possibili combinazioni numeriche assunte da queste componenti definisce uno spazio multidimensionale di colori. Inoltre i modelli colore possono fare uso di una sintesi additiva, la quale ragiona sulla somma dei colori (risulta adeguato ad esempio per i monitor), producendo il colore bianco nel caso tutte le componenti presentano il valore massimo oppure di una sintesi sottrattiva, la quale ragiona sulla sottrazione dei colori (risulta adeguato ad esempio per dispositivi di stampa), producendo il colore nero nel caso tutte le componenti presentano il valore massimo.

Quindi considerando i principali modelli colore questi si possono suddividere logicamente in:

- **Tipo RGB (sintesi additiva):** viene definito in uno spazio cubico, si concentra su **informazioni colorimetriche** e nel RGB a 24 bit sono utilizzati valori a 8 bit per ciascuna delle 3 componenti colore (rosso, verde e blu) mentre nel RGBA a 32 bit viene aggiunta una quarta componente a 8 bit che è la trasparenza dell'immagine;
- **Tipo HSB (sintesi additiva):** viene definito in uno spazio conico o cilindrico o sferico, si **separano le informazioni di intensità da quelle di colore**, viene utilizzata la componente tonalità H (tinta dei colori rappresentata con un angolo), la saturazione S (purezza del colore) e l'ultima componente per rappresentare la luminosità o intensità dei colori (con nome componente V per HSV, I per HSI oppure L per HSL);
- **Tipo CMY (sintesi sottrattiva):** considerazioni simili al RGB ma adatto per **dispositivi di stampa**, nel CMY a 24 bit sono utilizzate le componenti ciano, magenta e giallo mentre nel CMYK a 32 bit viene aggiunta la componente a 8 bit nero (per ottenere nella stampa il nero puro);
- **Tipo CIE:** simile al tipo HSB (separazione delle informazioni di intensità da quelle di colore) ma si focalizza sulla **percezione umana dei colori**, fa uso della componente luminosità L e di due componenti opposte tra loro per le informazioni di colore (a e b nel modello CIELAB oppure u e v nel modello CIELUV).

Nell'applicazione sono utilizzati i modelli colore RGB e RGBA per le foto acquisite con la fotocamera o immagini prodotte in output mentre l'HSL (conversione dal tipo RGB) viene impiegato per il mapping bidirezionale fra immagine e suono.

3.3 Audio digitale

L'audio digitale viene ottenuto dalla conversione di suoni dal formato analogico (ad esempio il suono emesso da una persona che canta) ad una forma digitale, facendo in modo che

con la conversione l'audio sia percepito dall'ascoltatore come identico a quello originale. Uno dei metodi più utilizzati per svolgere questo passaggio da analogico a digitale (conversione A/D o analog to digital converter ADC) e viceversa (conversione D/A o digital to analog converters DAC) è la modulazione a impulsi codificati **PCM**. Nello specifico la conversione A/D del suono con PCM, definita nella **modulazione**, prevede i seguenti passi:

1. **Trasduzione del suono analogico in segnale elettrico:** quando un microfono viene colpito da un'onda meccanica sonora continua produce un segnale elettrico analogico, con valore di tensione compreso tra un valore minimo e massimo (range dinamico del segnale), preservando il più possibile la forma dell'onda acustica originaria;

2. **Campionamento del segnale elettrico:** si leggono i valori di tensione continui del segnale elettrico (campioni) con una frequenza temporale tale da rispettare il teorema del campionamento di Nyquist-Shannon (per non avere perdite di informazione, dove la **frequenza di campionamento** in Hz deve essere almeno il doppio della massima frequenza del segnale da campionare);

3. **Quantizzazione e codifica dei campioni:** la quantizzazione suddivide il range dinamico del segnale elettrico in un numero finito di intervalli, codificando ognuno degli intervalli con un valore digitale (solitamente si sceglie un numero n di intervalli che sia potenza di 2 al fine di utilizzare una **codifica a n bit**). Con un maggiore numero di bit utilizzati dalla codifica si aumenta l'accuratezza del segnale campionato, riducendo quindi l'imprecisione della quantizzazione (rumore di quantizzazione). Mediante opportuni algoritmi si associa ad ogni campione l'intervallo più vicino e quindi la relativa codifica;

4. **Compressione dei dati:** si riduce il numero totale dei bit necessari dalla codifica del passo precedente (di tipo raw) attraverso la compressione delle informazioni in sequenze numeriche più brevi (come lo standard MP3).

La conversione D/A del suono con PCM viene svolta nel processo di modulazione inverso ovvero la **demodulazione**.

3.4 Frequenza audio da terna nota, ottava e alterazione

Nel sistema musicale (tipicamente occidentale) vengono utilizzati **12 semitoni per ottava**, dove un semitono è il minimo intervallo musicale che ci può essere tra due suoni. I 12 semitoni sono costituiti dalle **7 note musicali pitch** in notazione letterale C, D, E, F, G, A, B (o DO, RE, MI, FA, SOL, LA, SI) mentre i restanti 5 elementi sono dati dalla composizione di alcune delle precedenti note musicali con le **due alterazioni** (permette l'aumento o il decremento di un semitono) diesis (♯) oppure bemolle (♭). Nel dettaglio i 5 elementi con alterazione sono C♯/D♭ (equivalenza tra nota diesis e bemolle), D♯/E♭, F♯/G♭, G♯/A♭, A♯/B♭.

Per convenzione si associa al semitono A di ottava 4 (A_4) la **frequenza fondamentale 440 Hz** e mediante la **distanza n in semitoni** da A_4 è possibile determinare, con la formula $freq[Hz] = 440 \cdot 2^{\frac{n}{12}}$, le frequenze di tutti i restanti elementi (ognuno costituito dalla terna semitono, ottava e se disponibile l'alterazione). Passando da un semitono ad una data ottava allo stesso semitono ma di ottava successiva vi è un raddoppio di frequenza mentre considerando il senso inverso la frequenza viene dimezzata.

Nella Tabella 3.1 sono mostrate le frequenze in Hz assegnate ai semitoni di ottave differenti (dalla -1 alla 10), definite a partire dalla frequenza fondamentale A_4.

	Ottava											
Semitono	−1	0	1	2	3	4	5	6	7	8	9	10
C	8.18	16.35	32.70	65.41	130.81	261.63	523.25	1046.50	2093.01	4186.01	8372.02	16744.04
C♯/D♭	8.66	17.32	34.65	69.30	138.59	277.18	554.37	1108.73	2217.46	4434.92	8869.84	17739.69
D	9.18	18.35	36.71	73.42	146.83	293.66	587.33	1174.66	2349.32	4698.64	9397.27	18794.55
D♯/E♭	9.72	19.45	38.89	77.78	155.56	311.13	622.25	1244.51	2489.02	4978.03	9956.06	19912.13
E	10.30	20.60	41.20	82.41	164.81	329.63	659.26	1318.51	2637.02	5274.04	10548.08	21096.16
F	10.91	21.83	43.65	87.31	174.61	349.23	698.46	1396.91	2793.83	5587.65	11175.30	22350.61
F♯/G♭	11.56	23.12	46.25	92.50	185.00	369.99	739.99	1479.98	2959.96	5919.91	11839.82	23679.64
G	12.25	24.50	49.00	98.00	196.00	392.00	783.99	1567.98	3135.96	6271.93	12543.85	25087.71
G♯/A♭	12.98	25.96	51.91	103.83	207.65	415.30	830.61	1661.22	3322.44	6644.88	13289.75	26579.50
A	13.75	27.50	55.00	110.00	220.00	**440.00**	880.00	1760.00	3520.00	7040.00	14080.00	28160.00
A♯/B♭	14.57	29.14	58.27	116.54	233.08	466.16	932.33	1864.66	3729.31	7458.62	14917.24	29834.48
B	15.43	30.87	61.74	123.47	246.94	493.88	987.77	1975.53	3951.07	7902.13	15804.27	31608.53

Tabella 3.1: Frequenze semitoni per ottava

3.5 Segnale sinusoidale a frequenza variabile (sweep sinusoidale)

Un **segnale sinusoidale** è un segnale descritto mediante la funzione seno. Tra i vari parametri necessari alla funzione vi è anche quello molto rilevante della frequenza. Solitamente viene costruito il segnale (mediante un oscillatore sinusoidale) utilizzando un valore di frequenza in Hz che resta sempre costante nel tempo. Quindi si potrebbe generare un segnale audio, associato ad una coppia semitono e ottava, descrivendolo come un segnale sinusoidale con frequenza pari al valore corrispondente della coppia scelta (o un altro valore di frequenza qualsiasi dalla Tabella 3.1).

Rispetto al caso standard però si vuol fare in modo che il segnale sinusoidale possa variare il valore di frequenza nel corso del tempo e questo cambiamento è legato al fatto che si sceglie una diversa coppia semitono e ottava. Un segnale sinusoidale la cui frequenza varia nel corso del tempo viene chiamato **sweep sinusoidale** (o sweep di frequenza o chirp).

Nel dettaglio l'implementazione della sweep sinusoidale viene fatta utilizzando un **accumulatore di fase** (partendo dal valore 0) a cui viene sommato nel corso del tempo un relativo delta di variazione di fase, aggiornato ad ogni cambio di frequenza (definito come

$incremento\_fase = \frac{2\pi \cdot nuova\_frequenza}{frequenza\_campionamento}$). Inoltre si controlla che l'accumulatore di fase sia sempre compreso tra 0 e 2π in quanto è l'intervallo richiesto per l'applicazione della funzione seno.

3.6 Spettrogramma audio

Lo **spettrogramma** di un segnale audio permette di analizzare come varia lo spettro di frequenze del segnale considerato nel corso del tempo. Per questo lo spettrogramma è costituito da 3 principali componenti, le quali possono essere utilizzate su un grafico solitamente bidimensionale (oppure tridimensionale) per la rappresentazione dei risultati, ovvero:

- **Tempo** (asse delle ascisse x in scala lineare);
- **Frequenza** (asse delle ordinate y in scala lineare o logaritmica);
- **Ampiezza** di una particolare frequenza ad un dato istante temporale (punto con data ascissa e ordinata) rappresentato come **valore di intensità di un pixel** (in scala di grigi o a colori). Solitamente vengono adoperati pixel a colori che utilizzano una mappa colore in cui si associano colori tendenti al blu scuro per bassi valori di ampiezza, tendenti al viola-rosso per medi valori di ampiezza e tendenti al giallo per alti valori di ampiezza (generalmente chiamata mappa colore magma). Se per i pixel viene utilizzata la scala di grigi si associa il nero per bassi valori fino ad arrivare al bianco per alti valori. Per l'ampiezza può essere utilizzata una scala logaritmica al fine di esprimere i decibels (dB).

A livello implementativo lo spettrogramma viene definito suddividendo l'intero asse dei tempi del segnale considerato in tanti intervalli uguali in durata (in ms, chiamate finestre temporali) ed applicando ad ogni intervallo la **trasformata di Fourier** al segnale in esso contenuto. In questo modo si ottengono i valori di ampiezza del segnale in funzione della frequenza per ognuna delle suddivisioni utilizzate. Lo spettrogramma finale si ottiene unendo assieme le trasformate di Fourier delle varie finestre temporali impiegate.

In generale la trasformata di Fourier permette, a partire da un segnale periodico (come un segnale sonoro), la sua decomposizione come somma di segnali sinusoidali (le varie componenti costituenti, utile ad esempio anche per il filtraggio di componenti indesiderate), ognuno con propria frequenza e ampiezza. Essendo la trasformata molto complessa computazionalmente, per il calcolo dello spettrogramma dell'audio viene utilizzata la sua versione efficiente chiamata **Fast Fourier Transform (FFT)** [31].

Capitolo 4

Gestione nativa dell'audio nei dispositivi Android

4.1 Libreria Oboe

Per le funzionalità sonore di input/output utilizzate nell'app Andoid ImageSound è necessario gestire l'audio con elevate prestazioni [32], soprattutto per quanto riguarda il **minimizzare il valore di latenza**. Nel dettaglio la latenza dell'audio di output rappresenta il tempo che intercorre nello scambio dei dati tra app e il componente fisico speaker mentre la latenza dell'audio di input riguarda il passaggio delle informazioni acquisite dal microfono verso l'applicazione.

Per soddisfare il requisito di gestione dell'audio con alte prestazioni il sistema Android mette a disposizione **2 diverse librerie, scritte entrambe in linguaggio nativo** C e C++ (per prestazioni migliori a priori rispetto al Java), ovvero:

- **OpenSL ES:** implementazione orientata ad Android dell'API OpenSL ES 1.0.1 [33] definita da Khronos Group con alcune necessarie variazioni legate all'implementazione tramite NDK. Le funzionalità audio vengono esposte con una struttura simile a quella utilizzata da relative **classi standard presenti nel Framework API Java** (*MediaPlayer* e *MediaRecorder*) al fine di rendere più immediato il loro utilizzo;

- **AAudio:** nasce come alternativa leggera e di più facile utilizzo rispetto la libreria OpenSL ES a partire da Android 8.1 (livello API 27) [34]. La libreria permette lo scambio di dati fra l'applicazione ed un dispositivo hardware audio di input (microfono) o di output (speaker) mediante la **lettura o scrittura di un buffer** di memoria contenuto in un **flusso audio (stream)**. La lettura o scrittura del buffer nel flusso può essere fatta mediante opportune funzioni bloccanti sincrone (si attendono i risultati per proseguire l'elaborazione) o non bloccanti asincrone nello stesso thread di esecuzione dell'app oppure da una callback (invocazione periodica asincrona) in un thread separato ad alta priorità. La **callback ad alta priorità** consente di soddisfare più facilmente il requisito di bassa latenza dell'audio. Si può agire ulteriormente sulle **prestazioni dello stream** scegliendo fra 3 modalità che

variano principalmente la dimensione del buffer ovvero bassa latenza (buffer piccolo), risparmio energetico (buffer grande) oppure un bilanciamento delle modalità precedenti. Il flusso audio viene definito principalmente da 3 elementi ovvero se il dispositivo audio è una sorgente o una destinazione (direzione del flusso), se il flusso ha un accesso esclusivo o condiviso al dispositivo audio e il **formato dei dati** utilizzati. Quest'ultimo riguarda la scelta del formato dei campioni audio tra PCM intero con segno a 16 bit o PCM float a 32 bit, se utilizzare audio mono a un canale o stereo e la frequenza di campionamento. L'implementazione dello stream audio viene definita mediante un'**architettura a stati** (come gli stati per aprire, avviare o fermare il flusso) con le opportune funzioni asincrone di transizione.

In questi ultimi anni però viene sconsigliato di utilizzare direttamente una delle due precedenti librerie in quanto Google ha rilasciato una nuova libreria C++ sostitutiva chiamata **Oboe** [35] [36]. L'obiettivo di Oboe è quello di ottenere le migliori performance audio in termini di latenza che si possono ottenere per il dispositivo fisico utilizzato. Oboe in realtà non è altro che un **wrapper** che utilizza la libreria OpenSL ES se il dispositivo Android presenta un livello API compreso tra 16 e 26, la libreria più accurata AAudio se maggiore o uguale a 27 (versione Oreo 8.1), fornendo così un'interfaccia unica che astrae dal dispositivo fisico sottostante (Figura 4.1). Astrazione rafforzata dal fatto che l'API utilizzata per Oboe presenta la **stessa struttura di quella di AAudio** quindi è conveniente implementare le applicazioni con l'API di AAudio per renderle automaticamente retrocompatibili anche con con quelle che utilizzavano OpenSL ES.

Quindi riassumendo la nuova libreria come AAudio utilizza i concetti principali di:

- Stream audio con associato il relativo buffer per la lettura/scrittura;
- Architettura a stati per l'implementazione dello stream e funzioni asincrone di transizione;
- Callback ad alta priorità.

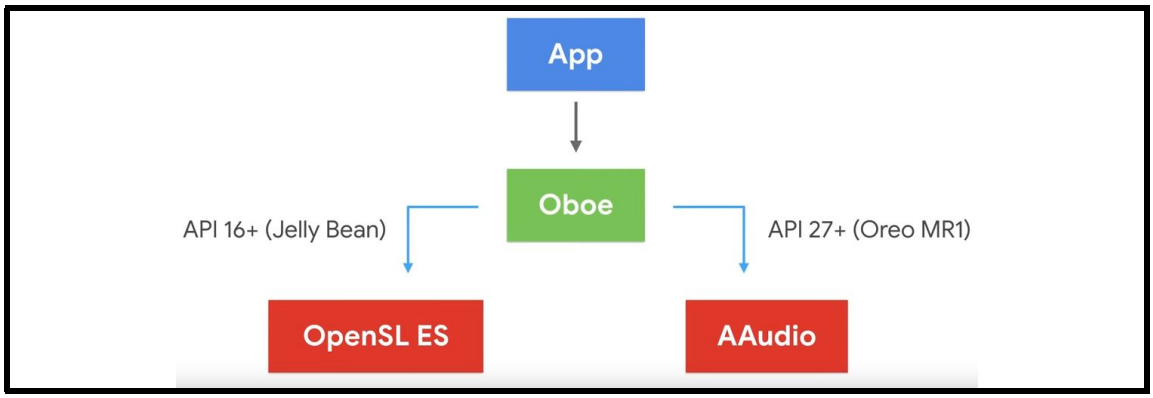

Figura 4.1: Libreria utilizzata da Oboe in base alla versione di Android

Come definito precedentemente vi è una **correlazione fra dimensione del buffer e bassa latenza** ottenuta nel flusso audio. In particolare si utilizza il termine burst per indicare i campioni audio o frame considerati dal dispositivo attuale (sempre in medesimo numero) durante ogni singola operazione di lettura/scrittura, come si può vedere nella Figura 4.2. Viene stabilito in Oboe che per ottenere la più bassa latenza possibile si deve fare in modo che la dimensione del buffer sia pari al **valore ottimale 2 burst** ed in questo caso viene chiamato double buffer (Figura 4.3).

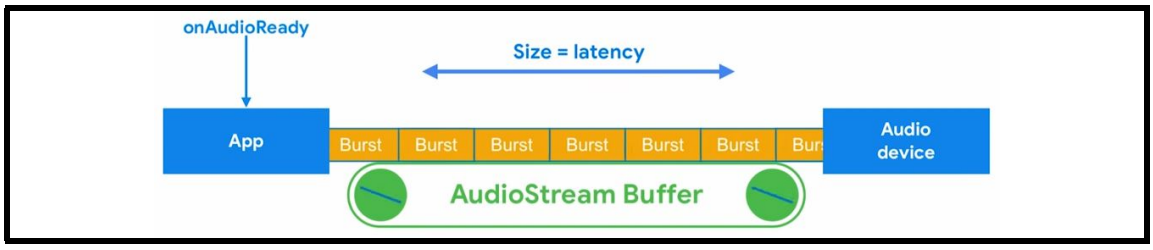

Figura 4.2: Buffer dello stream Oboe definito con burst

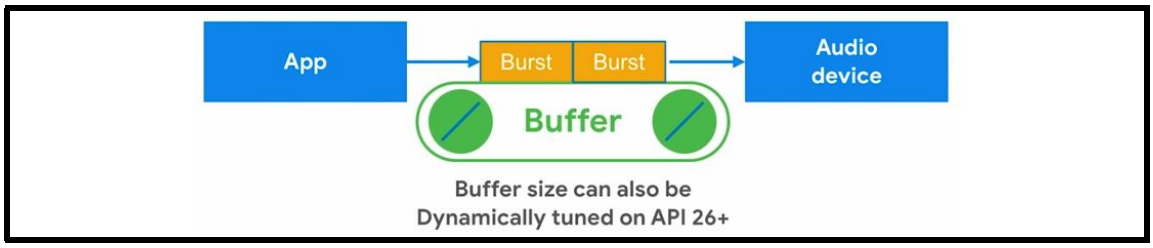

Figura 4.3: Buffer dello stream Oboe con latenza ottimale (2 burst)

4.2 Stream Oboe con direzione output: scrittura del buffer

L'effettiva creazione dello stream audio di Oboe viene fatta mediante un opportuno oggetto ***builder***, definito nella libreria stessa, nel quale devono essere **specificati i parametri audio** (gli stessi presenti anche in AAudio) che verranno utilizzati per la **costruzione del flusso**. Tra i vari parametri i principali sono:

- Dispositivo audio utilizzato (mediante un ID);
- Direzione del flusso (input per la registrazione di campioni tramite microfono o output per sentire il suono prodotto con lo speaker);
- Tipo di condivisione del dispositivo audio, settato a esclusivo;
- Frequenza di campionamento (48KHz);
- Numero di canali, settato a mono (per migliorare le prestazioni sono copiati gli stessi campioni nel canale sinistro e destro per produrre un audio stereo);

- Formato campioni audio (tra int e float);
- Prestazioni dello stream, settato a bassa latenza;
- Definizione della callback ad alta priorità (per la direzione output i campioni sono scritti nel buffer del flusso mentre per l'input sono letti dal buffer).

Dopo aver settato in modo opportuno i parametri del *builder* viene creato tramite questo il flusso audio. A questo punto si fa partire il flusso audio, per eseguire la callback ad alta priorità, mediante il metodo *start*. Periodicamente il buffer verrà scritto/letto fino a quando non si decide di fermare lo stream con il metodo *stop* e di chiuderlo definitivamente con il metodo *close*.

Considerando quindi un **flusso audio con direzione output** l'app dovrà **scrivere un burst nel buffer** dello stream (campioni audio prodotti con una determinata elaborazione) ad ogni chiamata della callback ad alta priorità per la riproduzione sonora da parte dello speaker.

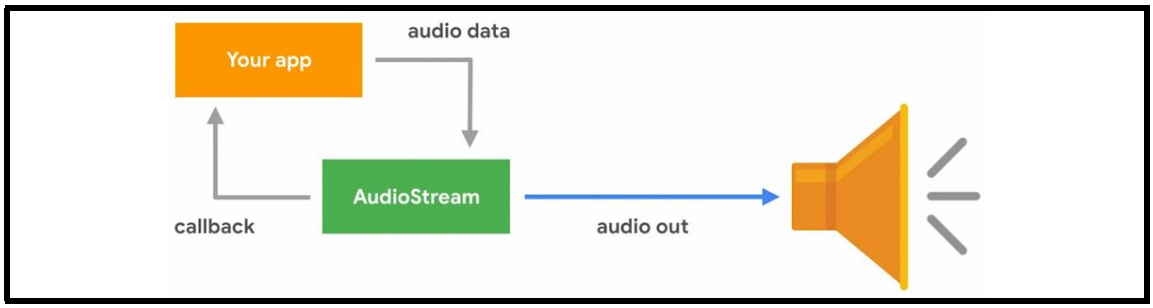

Figura 4.4: Stream Oboe in output

4.3 Stream Oboe con direzione input: lettura del buffer

Il **flusso audio con direzione input** agisce in modo inverso rispetto al caso precedente. Infatti il microfono acquisisce l'audio registrato, lo scrive a burst nel buffer per fare in modo che ad ogni chiamata della callback ad alta priorità l'app **legge dal buffer** il contenuto e lo salva in una propria area di memoria (per svolgerci un'elaborazione qualsiasi).

Figura 4.5: Stream Oboe in input

Capitolo 5

Gestione nativa della fotocamera nei dispositivi Android

5.1 Libreria OpenCV

Per la gestione in Android di tutte le immagini, acquisite come frame mediante la fotocamera del dispositivo oppure quelle prodotte utilizzando informazioni sonore, includendo anche le relative elaborazioni (come la conversione dal modello colore RGB a HSL), viene utilizzata la libreria **OpenCV** [37] (Open Source Computer Vision Library).

OpenCV è una libreria originariamente sviluppata dalla Intel che si occupa sia del **machine learning che della computer vision in tempo reale**, mettendo quindi in relazione (con modelli approssimati) immagini bidimensionali 2D con lo spazio reale tridimensionale al fine di estrapolare ed interpretare in modo automatico informazioni logiche opportune. La libreria è multipiattaforma (supporta Windows, Linux, Mac OS, iOS e Android) ed espone delle interfacce nei linguaggi di programmazione C++ (linguaggio principale con cui è stata scritta la libreria), C, Python, Java.

OpenCV contiene oltre 2500 algoritmi ottimizzati incorporati in una **struttura modulare** dove alcuni dei principali moduli presenti sono:

- ***core*:** componente principale di OpenCV che definisce le strutture dati e le funzionalità di base di ausilio a tutti gli altri moduli;
- ***imgproc*:** modulo per l'elaborazione delle immagini con funzionalità come l'applicazione di filtri, trasformazioni geometriche (ad esempio il ridimensionamento e la distorsione), conversione dello spazio colore, istogrammi, ecc. ;
- ***video*:** fornisce funzionalità per l'analisi dei video come la stima del moto, rimozione dello sfondo e tracciamento degli oggetti;
- ***calib3d*:** permette di mettere in relazione lo spazio 2D con quello 3D attraverso la calibrazione della telecamera e la ricostruzione dello spazio tridimensionale;

- ***features2d*:** framework per la gestione delle features 2D nell'immagine ovvero l'estrazione di caratteristiche rilevanti (come punti o strutture specifiche) dalle immagini al fine di svolgere operazioni di corrispondenza tra queste;
- ***objdetect*:** consente la rilevazione degli oggetti nell'immagine;
- ***android*:** contiene funzionalità di supporto esclusive per la piattaforma Android come la gestione della fotocamera del dispositivo in modo nativo.

In OpenCV tutte le **immagini raster** sono considerate matrici bidimensionali, appartenenti alla classe ***Mat*** (abbreviazione di matrice) del modulo *core* con associata la profondità in bit dell'immagine (numero di canali) ed in caso di immagini colorimetriche (non in scala di grigi) il relativo modello colore. La libreria nella **piattaforma Android** [38] fa uso del linguaggio **C++ per la parte nativa** mentre per accedere alle medesime funzioni nel linguaggio **Java utilizza JNI** (mediante il caricamento della libreria statica *opencv_java4.so*). In realtà nell'app viene utilizzato per i calcoli più complessi OpenCV solamente nel linguaggio C++, il Java per elementi di interfaccia grafica in modo da ottenere nelle elaborazioni più rilevanti le migliori prestazioni possibili, senza nessun overhead. Inoltre utilizzando il linguaggio C++ si facilità anche l'interazione con la libreria per la gestione del suono Oboe, anche essa definita in linguaggio nativo.

5.2 Elaborazione parallela delle immagini

Per fare in modo di impiegare, nelle elaborazioni di conversione fra immagine e suono, multipli thread (flussi di esecuzione) in parallelo è stato utilizzato il **costrutto *parallel_for_*** [39] messo a disposizione da OpenCV. Infatti questo permette di svolgere un'elaborazione, che coinvolge ogni pixel dell'immagine, in modo parallelo utilizzando multipli thread.

Il costrutto *parallel_for_* non fa altro che inizialmente richiedere o il **numero massimo di thread che può ottenere dal dispositivo** per l'elaborazione (comportamento di default nel caso il numero massimo di thread da utilizzare non è stato definito esplicitamente) o il **numero massimo di thread specificati**. A questo punto **suddivide il carico di pixel in modo equilibrato e distinto tra i thread** utilizzati e alla fine di tutte le elaborazioni **restituisce l'output finale unificato** (lo stesso che si sarebbe ottenuto con un'elaborazione sequenziale come può essere l'immagine elaborata finale). Internamente *parallel_for_* non fa altro che utilizzare pthread (thread dello standard POSIX) ottimizzato chiamato TBB se il dispositivo presenta un processore Intel altrimenti ripiega su pthread normale non ottimizzato.

Nell'applicativo è stata mantenuta sia la possibilità di eseguire l'elaborazione di conversione in modo sequenziale che in modo parallelo con *parallel_for_* (scelta automatica del numero massimo di thread) mediante un opportuno parametro che viene scelto dall'utente.

5.3 La camera preview di OpenCV

Per l'acquisizione dei frame immagine mediante la fotocamera del dispositivo **non vengono utilizzate metodologie standard** (per facilitare l'implementazione) di gestione della camera nel linguaggio Java messe a disposizione da Android stesso. Questo per **evitare l'elevato calo di prestazioni**, in termini di latenza, dovuto all'esecuzione della logica in Java e al continuo passaggio dal linguaggio Java a C++ mediante JNI durante l'interazione con le funzionalità sonore di Oboe. Il calo di prestazione potrebbe essere anche accettabile per quanto riguarda le elaborazioni che utilizzano un singolo frame immagine che costituisce una foto (conversione da foto a suono e viceversa) mentre risulta totalmente inaccettabile per quelle che utilizzano multipli frame immagine al secondo ovvero le elaborazioni che gestiscono video (conversione da video a suono continuo e viceversa). Invece viene **gestita la fotocamera del dispositivo con OpenCV**, tramite l'**estensione della classe Java *CameraGLSurfaceView*** presente nel modulo *android*. L'estensione della classe viene fatta per adattare la classe alle esigenze dell'app aggiungendo elementi quali attributi o metodi di supporto alle elaborazioni.

La *CameraGLSurfaceView* non è altro che una camera chiamata **preview** con implementazione in linguaggio nativo C++ e per accelerare ulteriormente il processo stesso fa uso delle **OpenGL 2.0** [40]. OpenGL non è altro che una specifica che definisce un insieme di funzioni con il loro comportamento da cui vengono definite implementazioni specifiche in base all'hardware utilizzato (soprattutto la GPU) mediante il rilascio di opportune librerie. Le OpenGL permettono di gestire più velocemente i frame immagine della fotocamera utilizzando la scheda grafica del dispositivo anziché la normale CPU.

L'intero **ciclo di esecuzione della camera preview** per un'elaborazione generale prevede i seguenti passi:

1. **avvio** del processo di acquisizione dei frame immagine della fotocamera;
2. ricezione continua, ad intervalli di tempo regolari e in linguaggio nativo dei frame immagine nel modello colore RGBA mediante una **lettura con le OpenGL 2.0**;
3. eventuale catena di conversione dei frame immagine dal modello colore RGBA a RGB a HSL mediante la libreria nativa OpenCV per svolgere un **mapping verso elementi sonori** della libreria nativa Oboe;
4. **scrittura con le OpenGL 2.0** dei frame immagine RGBA ricevuti ad intervalli di tempo regolari (effettuando una variazione o meno dei relativi pixel) al fine di essere visualizzati da parte dell'app;
5. conclusa l'intera elaborazione si **interrompe la camera preview** in modo da non ricevere più aggiornamenti ad intervalli regolari dei frame immagine.

Un accorgimento che viene fatto, sempre per migliorare le prestazioni in termini di latenza, consiste nel **limitare le dimensioni massime utilizzabili (larghezza e altezza) per i frame immagine** acquisiti dalla camera preview al valore standard 1280X920 se il dispositivo viene utilizzato con un'orientazione orizzontale (landscape) oppure a 920X1280

se il dispositivo viene utilizzato con un'orientazione verticale (portrait). Questo viene fatto in quanto, durante il processo di conversione da immagine a suono, deve essere effettuato comunque un **ridimensionamento dell'immagine** di input a dimensioni inferiori gestibili (massime dimensioni possibili di 512X512). Quindi acquisire immagini senza definire un limite nelle dimensioni sarebbe solamente uno spreco di risorse per quanto riguarda la memoria utilizzata e al maggior overhead richiesto nel ridimensionamento di una grande immagine (foto scattata utilizzando il sensore della fotocamera senza limitazioni) in una molto più piccola. Anche se il valore utilizzato per definire il limite è anch'esso superiore alle dimensioni necessarie risulta adeguato al fine di avere un **compromesso tra overhead richiesto e compatibilità generica delle dimensioni** con le fotocamere dei dispositivi Android (ottenuta mediante l'impiego di un valore standard disponibile in tutte le fotocamere). Viene utilizzato un **valore massimo per le dimensioni e non un valore specifico** in modo tale da **permettere ad OpenCV di scegliere, fra le varie dimensioni** compatibili con il dispositivo impiegato, quelle più opportune in grado di ottimizzare (in termini di prestazioni) la libreria stessa.

5.4 Problematiche riscontrate nella camera preview: orientazione e dimensioni ottimali

Utilizzando la **camera preview sono stati riscontrati 2 problemi** che si presentavano solamente durante l'utilizzo su **dispositivi reali Android**, funzionando invece in modo corretto nel test mediante emulatore virtuale su Android Studio.

Il primo errore riguarda l'**orientazione del dispositivo** ovvero la fotocamera appare con la giusta rotazione solamente se si utilizza il dispositivo in modalità landscape. In Android esistono **3 tipologie di orientazioni** del dispositivo ovvero la verticale (**portrait**), l'orizzontale senza rotazione (**landscape**, solitamente con i tasti fisici alla destra del dispositivo) e l'orizzontale con rotazione di 180 gradi (**landscape reverse**, rotazione di 180 gradi della landscape precedente). Quindi il **comportamento scorretto della preview si verifica nelle modalità portrait e landscape reverse** sia utilizzando la fotocamera anteriore che posteriore. La fotocamera appare con la giusta orientazione negli emulatori in quanto la webcam utilizzata per i test è quella integrata in un portatile e questa viene vista come se il dispositivo fosse disposto sempre orizzontalmente, anche nel caso verticale.

L'errore di rotazione è dovuto al fatto che la **camera preview di OpenCV è stata progettata solamente per essere utilizzata con un'orientazione orizzontale** del dispositivo (riducendo in questo caso l'accessibilità dell'app). Si potrebbe risolvere il problema dell'orientazione semplicemente andando a rovesciare in modo opportuno con il **metodo *flip* di OpenCV** i frame immagine in input per restituire in output quelli rovesciati. L'approccio però risulta inutilizzabile in termini di prestazioni per le elaborazioni che gestiscono video in quanto si deve **applicare la funzione a multipli frame al secondo**. Invece la corretta soluzione efficiente adottata è quella di agire sull'attributo di tipo array chiamato ***texCoord2D*** per la gestione delle coordinate bidimensionali (definito nella classe *CameraGLRendererBase* della libreria OpenCV, acceduto da *CameraGLSurfaceView*).

L'attributo *texCoord2D* è costituito da **8 elementi ognuno con valore numerico binario** (0 o 1) ed ognuna delle sequenze possibili corrisponde ad una determinata rotazione dei frame immagine. La grande importanza di questa metodologia (rispetto al *flip*) è che *texCoord2D* viene utilizzato solamente una volta per posizionare la fotocamera e dopodiché tutti i successivi frame che arrivano presentano la stessa rotazione stabilita, senza **nessun overhead prestazionale** richiesto. Quindi sono state definite le **6 sequenze necessarie di orientazione** (3 per la fotocamera posteriore e 3 per l'anteriore, determinate mediante tentativi come lo scambio di componenti o inversione dei valori binari del valore di default) ovvero:

1. portrait per la fotocamera posteriore;
2. landscape per la fotocamera posteriore (valore di default utilizzato da OpenCV);
3. landscape reverse per la fotocamera posteriore;
4. portrait per la fotocamera anteriore;
5. landscape per la fotocamera anteriore;
6. landscape reverse per la fotocamera anteriore;

Al fine di utilizzare le 6 sequenze determinate è stato **reso l'attributo *texCoord2D* della libreria OpenCV modificabile** (non più read-only, togliendo il modificatore *final*) e **aggiunto un metodo (*cambiaTexCoord2DRotazioneSchermo*)** per settare in modo opportuno la sequenza *texCoord2D* in base alla rotazione e al tipo di fotocamera utilizzata (anteriore o posteriore). Con tale modifica la fotocamera sul dispositivo reale appare sempre disposta correttamente mentre sull'emulatore appare rovesciata, come dovrebbe essere normalmente.

Il secondo errore riguardava il fatto che la **camera preview restava completamente nera**, non acquisendo frame, **solamente su alcuni dispositivi reali Android**, funzionando invece correttamente negli altri e sull'emulatore. Il problema non era correlato alla nuova modifica applicata all'orientazione in quanto anche escludendola la fotocamera restava comunque nera.

L'errore invece era **legato alla ricerca da parte di OpenCV delle dimensioni migliori**, fra quelle compatibili con il dispositivo, da utilizzare per i frame immagine della camera preview (**metodo *cacPreviewSize*** della classe *Camera2Renderer*, acceduto anche questo da *CameraGLSurfaceView*). Nel caso non riusciva a trovare le dimensioni ottimali anziché utilizzare una dimensione standard (come 640X480 o 480X640, molto utilizzata nella libreria stessa) andava ad utilizzare per la larghezza e altezza il valore 0 (immagine 0X0), producendo un'immagine nulla quindi la fotocamera restava nera. Inoltre l'**ottimizzazione era pensata solamente per l'orientazione orizzontale** per cui è stata creata la camera preview (larghezza > altezza), producendo risultati non ottimizzati ed ideali per quella verticale.

La problematica è stata risolta modificando il metodo *cacPreviewSize* della libreria OpenCV per fare in modo di determinare le **dimensioni ideali per il caso orizzontale**

(larghezza > altezza) **o il caso verticale** (altezza ≥ larghezza, invertendo la variabile che rappresenta la larghezza con quella dell'altezza nell'algoritmo originale definito nella libreria) e nel caso queste **non vengono trovate, viene usato il valore di default** 640X480 (caso orizzontale) o 480X640 (caso verticale). Con tale modifica l'applicazione funziona correttamente anche sui dispositivi reali in cui precedentemente la fotocamera restava nera.

Le modifiche apportate al codice della libreria OpenCV per la risoluzione delle 2 problematiche riscontrate vengono mostrate nel dettaglio nell'Appendice A.1 (orientazione) e nell'Appendice A.2 (dimensioni ottimali).

Capitolo 6

Conversione da immagine a suono: processo Image-Sound

6.1 La curva di Hilbert

Un frattale è un oggetto geometrico definito dalla ripetizione di un dato pattern dove ogni elemento costituente può variare in scala ma deve presentare sempre la medesima forma. La **curva di Hilbert** è una curva frattale che riempie il piano definito da Hilbert e solitamente viene utilizzata per effettuare una **corrispondenza tra spazio 2D e 1D (e viceversa)**, conservando nella trasformazione il concetto di **località** (vicinanza) tra gli elementi. Questo comporta che le **dimensioni del dato 2D devono essere sia quadrate** (stesso valore numerico per entrambe le componenti) che una **potenza di 2**.

Considerando come **dato 2D un'immagine**, al fine di rispettare entrambi i vincoli sulle dimensioni viene effettuato un **ridimensionamento con eventuale distorsione** (se la larghezza in pixel è diversa dall'altezza) delle relative componenti. Nello specifico nell'applicazione sono utilizzate **7 possibili dimensioni**, le quali possono essere scelte dall'utente tra 8X8, 16X16, 32X32, 64X64, 128X128, 256X256 (valore default) e 512X512. In realtà esistono anche versioni modificate di Hilbert applicabili ad input 2D con dimensioni arbitrarie però tali versioni sono sconsigliate perché aumentano in modo significativo la complessità dell'algoritmo e riducono la proprietà di località.

La **componente 1D invece definisce una distanza lungo la curva** e punti vicini nella curva corrispondono a pixel vicini nello spazio 2D. In particolare il primo elemento di distanza (inizio curva) è associato al pixel più in basso a sinistra dell'immagine mentre l'ultimo elemento di distanza corrisponde al pixel più in basso a destra (fine curva, definendo concettualmente un percorso di attraversamento).

In breve l'implementazione della curva di Hilbert consiste nel **suddividere ricorsivamente il piano in 4 aree al fine di calcolare le componenti trasformate** (mediante la somma di una quantità numerica, calcolata con operazioni binarie) e terminare il processo quando si raggiunge la definizione di tutte queste componenti (raggiunto il caso base dell'algoritmo). Alle suddivisioni viene associato un livello L e con un dato livello significa che in ogni regione vi sono L^2 elementi base (celle non ulteriormente suddivisibili). Tra

livelli contigui viene anche applicata una funzione di rotazione in modo da adeguare il sistema di coordinate al nuovo livello. Nel passaggio da spazio 2D a 1D l'algoritmo inizia dal livello più alto per andare a scendere (dal quadrato finale alle celle) mentre nel passaggio da spazio 1D a 2D si inizia dal livello più basso per andare a salire (dalle celle per arrivare al quadrato finale).

Nella Figura 6.1 viene mostrato un esempio di applicazione dell'algoritmo di Hilbert (stessa logica utilizzata nell'app) che a partire da un'immagine 2D quadrata applica la suddivisione ricorsiva per ottenere alla fine una linea 1D che rappresenta la scomposizione lineare dell'immagine con caratteristiche di località.

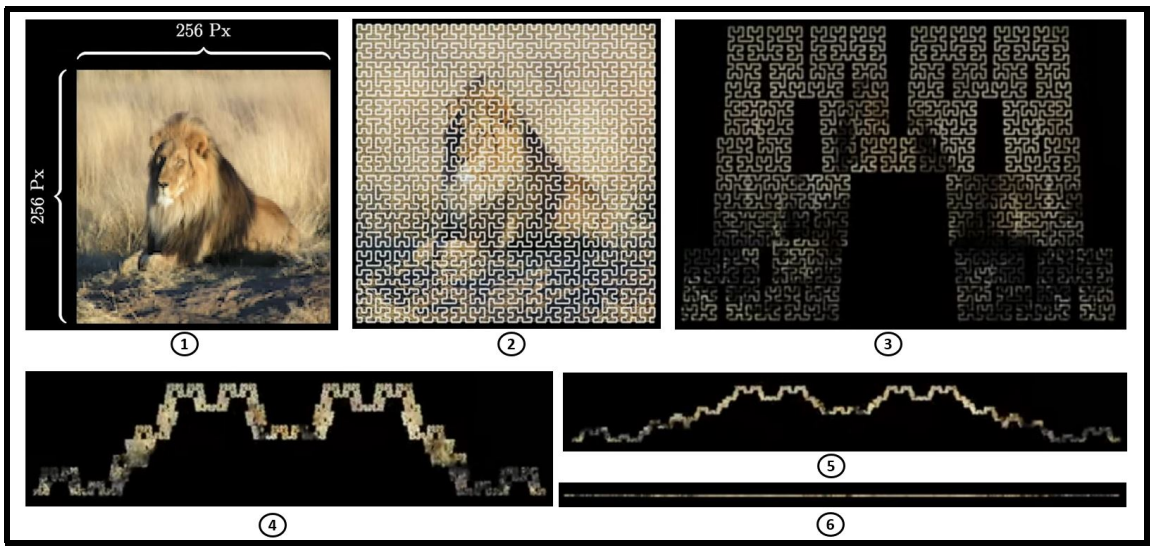

Figura 6.1: Applicazione dell'algoritmo di Hilbert ad un'immagine

6.2 Relazione di Newton tra note musicali e tonalità colore

La **relazione di Newton**, come mostrato nella Figura 6.2 e Figura 6.3, stabilisce l'**associazione tra i 7 colori dell'arcobaleno e le 7 classi di note musicali** ovvero:

- D (Re) $\Longrightarrow$ Rosso;
- E (Mi) $\Longrightarrow$ Arancione;
- F (Fa) $\Longrightarrow$ Giallo;
- G (Sol) $\Longrightarrow$ Verde;
- A (La) $\Longrightarrow$ Ciano;
- B (Si) $\Longrightarrow$ Blu;
- C (Do) $\Longrightarrow$ Viola.

Per il mapping secondo Newton, siccome il modello colore RGB non permette di separare le informazioni di colore da quelle di intensità, è stato **utilizzato il modello colore HSL** che mette a disposizione, tra le altre componenti, quella fondamentale che è la **tonalità H**. Solitamente la tonalità viene rappresentata come **angolo in gradi nel range 0-359°** mentre nella libreria **OpenCV la componente tonalità viene dimezzata** in modo che i valori siano nel **range 0-179°** e la componente viene rappresentata con un valore numerico intero.

L'implementazione della relazione di Newton viene fatta andando a **suddividere il range di valori di tonalità di OpenCV in 7 intervalli non uniformi**, corrispondenti ai 7 colori dell'arcobaleno. I 7 intervalli non sono uniformi in quanto si è considerato, per effettuare la suddivisione, il concetto di **angolo in cui i colori considerati risultano puri**. Sapendo quindi gli angoli in cui i 7 colori risultano puri è stata effettuata una suddivisione manuale in intervalli della rappresentazione grafica fornita da OpenCV per la componente tonalità. Nella Tabella 6.1 viene mostrata la suddivisone numerica della tonalità in intervalli utilizzata nell'app mentre nella Figura 6.4 viene mostrata la suddivisione manuale della rappresentazione grafica di OpenCV.

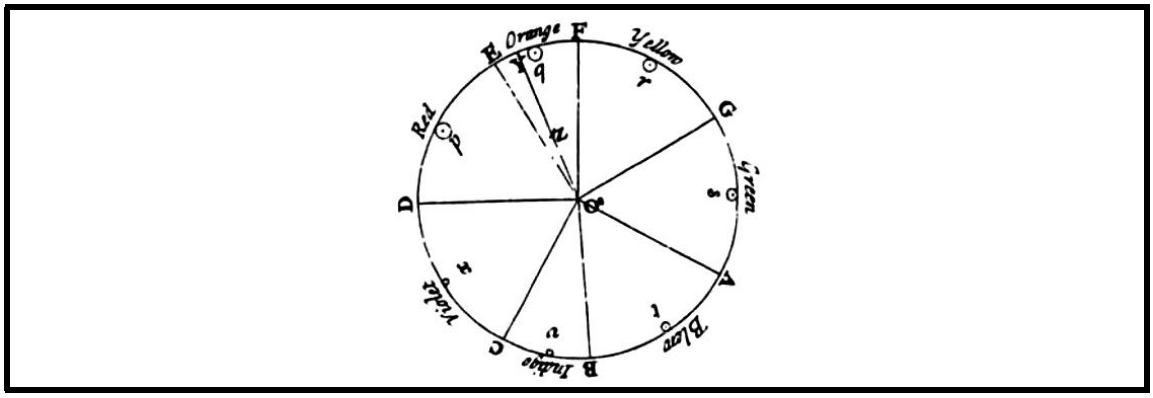

Figura 6.2: Ruota dei colori di Newton

Figura 6.3: Relazione di Newton tra le note musicali e i 7 colori dell'arcobaleno

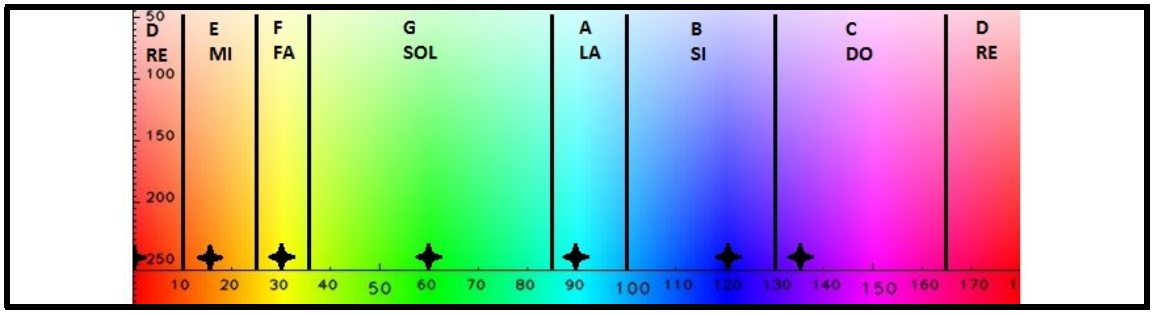

Figura 6.4: Suddivisione grafica della tonalità di OpenCV nei 7 colori dell'arcobaleno (le stelle indicano i colori puri)

Classe nota	**Colore**	**Angolo colore puro [°]**	**Angolo colore puro OpenCV [°]**	**Intervallo tonalità OpenCV (0-179°)**
D (Re)	**Rosso**	0	0	166-179 + 0-9
E (Mi)	**Arancione**	30	15	10-25
F (Fa)	**Giallo**	60	30	26-35
G (Sol)	**Verde**	120	60	36-85
A (La)	**Ciano**	180	90	86-100
B (Si)	**Blu**	240	120	101-130
C (Do)	**Viola**	270	135	131-165

Tabella 6.1: Suddivisone della tonalità OpenCV HSL in intervalli

6.3 Mapping da pixel HSL a frequenza audio

Nel mapping da pixel immagine a suono, per la componente **luminosità L** del modello colore HSL, viene utilizzata l'**associazione con l'ottava musicale**. In OpenCV il valore numerico di luminosità è di tipo intero e presenta il **range 0-255**. Siccome nell'app vengono **utilizzate 11 ottave** (dalla -1 alla 9) il range di luminosità viene quindi suddiviso in 11 intervalli circa uniformi, generalmente di 23 elementi (Tabella 6.2). Utilizzando 11 ottave ed essendoci 12 semitoni per ottava vi sono 132 possibili semitoni adoperati per la generazione del suono.

Le frequenze di suoni udibili dall'uomo si trovano nell'intervallo che va dai 20 Hz ai 20 Khz quindi solitamente, per rispettare questo vincolo, si utilizzano le ottave nel range 0-8 (come succede ad esempio in un pianoforte a 88 o 108 tasti). Nell'app viene utilizzato un **range di ottave diverso rispetto al caso standard** (come l'ottava -1 che risulta non udibile per l'uomo) in quanto le **frequenze non sono usate così come sono per la generazione del suono**, producendo quindi output non udibili. Invece le frequenze vengono **utilizzate dalla sweep sinusoidale**, definita mediante un accumulatore di

fase, la quale tiene in considerazione la variazione di frequenza tra elementi adiacenti e non il valore istantaneo. Mediante il relativo **controllo che la fase** sia sempre compresa tra 0 e 2π si fa in modo che eventuali elementi sonori normalmente non udibili saranno invece udibili, anche se con alcune approssimazioni.

Per l'ultima componente del modello colore HSL ovvero la **saturazione S** viene utilizzata l'**associazione con l'alterazione musicale**. Anche in questo caso la componente numerica saturazione di OpenCV viene definita con un valore intero nel **range 0-255**. Quindi viene effettuata una **suddivisione del range in 3 intervalli circa uniformi** (generalmente di 85 elementi) per rappresentare le alterazioni bemolle, diesis e naturale (nessuna alterazione). La suddivisione viene mostrata nella Tabella 6.3.

Mediante il **mapping HSL - Frequenza**, per ciascuno dei pixel dell'immagine nel modello colore HSL, viene quindi generata la **terna nota musicale, ottava e alterazione al fine di calcolare la relativa frequenza** (come distanza dalla principale A_4). Queste frequenze vengono poi utilizzate per la vera produzione del suono finale in output.

Per mantenere **memorizzate in ordine le frequenze prodotte** (a partire da un'immagine o dai campioni audio) è stata utilizzata una **coda circolare, messa a disposizione dalla libreria Oboe, a cui sono state aggiunte alcune funzionalità** necessarie per il funzionamento dell'applicativo (come il *clear* per la cancellazione). La coda è **thread-safe per un singolo produttore e un singolo consumatore** (utilizza internamente una funzione di *mask* tra indice lettura e scrittura), risultando quindi idonea in quanto solo dopo aver terminato l'eventuale parte di codice parallela (con *parallel_for_*) si procede, con un singolo thread, alla scrittura delle frequenze prodotte nella coda. Anche la lettura delle frequenze dalla coda soddisfa il vincolo in quanto solo dopo aver letto tutte le frequenze necessarie, con un singolo thread, viene avviata l'eventuale parte parallela.

Ottava	**Intervallo luminosità OpenCV (0-255)**
−1	0-22
0	23-45
1	46-68
2	69-91
3	92-114
4	115-137
5	138-160
6	161-183
7	184-206
8	207-229
9	230-255

Tabella 6.2: Suddivisone della luminosità OpenCV HSL in intervalli

Alterazione	**Intervallo saturazione OpenCV (0-255)**
Bemolle	0-84
Naturale	85-169
Diesis	170-255

Tabella 6.3: Suddivisone della saturazione OpenCV HSL in intervalli

6.4 Struttura complessiva processo Image-Sound

Il processo complessivo di conversione da immagine a suono chiamato **Image-Sound**, senza considerare le funzionalità aggiuntive, prevede generalmente la seguente struttura principale (vi sono alcune **minime differenze di implementazione tra il considerare foto o video** ovvero denominate di tipo **non live o live**, le cui variazioni sono spiegate successivamente durante la descrizione dettagliata delle 4 modalità di elaborazione):

1. **Acquisizione frame immagine RGBA**

   Acquisizione della foto scattata mediante la camera preview, la quale effettua una lettura con le OpenGL 2.0 (funzione C++ *glReadPixels*).
   ⇓

2. **Preprocessing immagine OpenCV**

   Conversione RGB $\Longrightarrow$ Ridimensionamento Hilbert $\Longrightarrow$ Conversione HSL

   Vengono effettuate in sequenza una serie di elaborazioni alla foto scattata RGBA, mediante la libreria OpenCV, al fine di ottenere una struttura adeguata delle informazioni. Infatti si converte l'immagine da RGBA a RGB (funzione C++ *cvtColor*), poi si ridimensiona con distorsione la foto per Hilbert, rendendola quadrata con lato L (funzione C++ *resize*) ed infine si effettua la conversione nel modello colore HSL.
   ⇓

3. **Applicazione curva di Hilbert**

   Si converte con Hilbert l'immagine bidimensionale HSL in lista monodimensionale (o matrice a singola riga) con L^2 elementi.
   ⇓

4. **Applicazione mapping HSL - Frequenza con Newton**

   Si converte la lista monodimensionale con terne HSL in lista di frequenze mediante il mapping HSL - Frequenza che utilizza la relazione di Newton.
   ⇓

5. **Preprocessing frequenze**

   Ridimensionamento elementi con interpolazione $\Longrightarrow$ Round frequenze

   Si ridimensiona la lista di frequenze in modo da avere un numero di elementi pari al sample rate (48 Khz, quindi 48.000 campioni al secondo) moltiplicato i secondi di audio che si desiderano produrre (tra 1 e 5 secondi, scelti dall'utente). Il ridimensionamento provoca un'interpolazione delle frequenze costituenti e quindi è necessario effettuare il round delle frequenze al semitono musicale più vicino. Nel round si controlla anche che le frequenze ottenute non siano minori o maggiori al semitono musicale permesso, settando nel caso i valori al minimo o massimo.

   ‖
   ‖
   ==========
   Inizia la parte bidirezionale della struttura in quanto si consumano porzioni di frequenze al fine di produrre porzioni di campioni audio in output per l'ascolto e ricominciare nuovamente il processo di selezione, fino al completamento.
   ==========
   ‖
   ⇓

6. **Generazione campioni audio sweep sinusoidale**

   Mediante la sweep sinusoidale si generano i campioni audio (formato float o intero, scelto dall'utente) consumando elementi dalla lista di frequenze.

   ⇓ ⇑

7. **Ascolto audio stereo in output con Oboe**

   Mediante Oboe si scrivono in output nel canale sinistro e destro (audio stereo) i campioni audio ottenuti al fine di ascoltare il suono finale prodotto.

Capitolo 7

Conversione da suono ad immagine: processo Sound-Image

7.1 Tipologie di campioni audio acquisiti tramite microfono

Come definito precedentemente per l'acquisizione dei campioni audio tramite microfono del dispositivo viene utilizzata la libreria Oboe. Questa mette a disposizione 2 possibili **formati audio da utilizzare per i campioni** (il tipo scelto viene impiegato dal *builder* per creare in modo opportuno lo stream) ovvero:

- ***I16*:** campioni audio PCM interi con segno a 16 bit (int), i cui valori numerici si trovano nell'intervallo da -32768 a $+32768$;
- ***Float*:** campioni audio PCM float a 32 bit, i cui valori numerici si trovano nell'intervallo da -1.0 a $+1.0$.

Oboe mette anche a disposizione funzioni opportune per effettuare la **conversione da un formato di campioni audio all'altro**. Infatti con la funzione *convertPcm16ToFloat* si passa dal tipo intero a quello float mentre la conversione inversa viene fatta con *convertFloatToPcm16*.

Solitamente si preferisce utilizzare, per ottenere una **maggiore precisione numerica, campioni audio in formato float**. Questo è dovuto all'impiego di valori con un maggior quantitativo di bit ovvero 32 rispetto a 16.

A **lato prestazionale risulta invece difficile stabilire il formato migliore** in quanto alcuni dispositivi Android ottengono performance migliori con il formato intero mentre altri con il formato float. Comunque nell'applicativo viene data la possibilità all'utente di scegliere il formato dei campioni audio più opportuno (di default viene utilizzato il formato float), tenendo in considerazione nella scelta anche le reali prestazioni ottenute con il proprio dispositivo.

7.2 Metodi per ottenere le frequenze dai campioni audio registrati

Per effettuare la conversione da suono ad immagine è necessaria una lista monodimensionale di frequenze in modo da ottenere, attraverso un opportuno mapping, la relativa immagine bidimensionale HSL associata (caso inverso rispetto a quello definito per la conversione da immagine a suono). Il problema che si presenta con i **campioni audio acquisiti mediante il microfono è che non sono espressi sotto forma di frequenza**, ma di un valore numerico int o float.

Il modo standard per ottenere i valori di frequenza, a partire dai campioni audio, consiste nell'utilizzare la **trasformata di Fourier su un insieme di campioni** al fine di estrarre, nelle multiple frequenze che costituiscono il segnale, quella principale. L'approccio però risulta **complesso dal punto di vista computazionale** (tenendo in considerazione i requisisti necessari di latenza), quindi si esclude l'impiego nell'app. Per ovviare al problema sono state definite **2 metodologie che utilizzano campioni audio interi, più approssimate ma con elevata efficienza** che sono:

- **Zero crossing:** tecnica **standard** [41] in cui si va a **contare il numero di passaggi da valore negativo a positivo o da positivo a negativo in campioni adiacenti** (attraversamenti dello 0), definendo il numero di cicli (metà degli attraversamenti dello 0) al fine di calcolare, assieme ai secondi temporali associati ai campioni considerati ($secondi\_temporali = \frac{numero\_campioni}{frequenza\_campionamento}$), il relativo valore di frequenza (che viene arrotondato al semitono musicale più vicino, $frequenza = \frac{numero\_cicli}{secondi\_temporali}$). Lo svantaggio della tecnica è che non può essere utilizzata su un singolo campione audio ma richiede sempre un insieme di elementi. Per questo ad un gruppo di campioni viene associata un'unica frequenza comune producendo quindi un **output finale con struttura a blocchi uniformi** (replicazione della stessa frequenza);

- **Divisione:** tecnica **non standard**, nominata in tale modo in quanto **suddivide il massimo valore** senza segno che può assumere un campione intero (32768) in **11 intervalli uniformi** (circa 2978) per rappresentare le **ottave** dalla -1 alla 9. Inoltre **ogni ottava viene suddivisa in 12 intervalli uniformi** (circa 248) per rappresentare i **12 semitoni musicali** che compongono un'ottava. Quello che si ottiene è un'associazione diretta tra campione audio intero e frequenza di una nota musicale quindi, rispetto la metodologia precedente, si ottiene un **output finale con struttura puntuale**. La problematica presente in questa metodologia è che solitamente, nelle **normali condizioni di registrazione** dei campioni audio, non si hanno valori numerici molto elevati (ambienti non eccessivamente rumorosi con valori fino a circa 12000, un terzo del range) quindi, le **frequenze ottenute con l'associazione, sono maggiormente concentrate nella parte iniziale dell'intero range** utilizzato (frequenze più basse). Invece si ottengono **risultati adeguati se la tecnica viene utilizzata in ambienti rumorosi** dove, con maggiore probabilità, ci si avvicina al valore massimo del range.

Nel caso vengono **utilizzati campioni float viene effettuata una conversione di questi da float a intero in un buffer temporaneo**, mediante l'opportuna funzione di Oboe. Entrambe le tecniche utilizzano solamente campioni interi a 16 bit e non float in quanto con valori a 32 bit i possibili elementi numerici utilizzabili, per la definizione delle metodologie, risultano eccessivi, non modellando in modo opportuno gli algoritmi impiegati come nel caso intero.

La metodologia da impiegare per ottenere le frequenze a partire dai campioni audio viene scelta dall'utente (di default viene utilizzata la divisione) a seconda se desidera ottenere output finali con una struttura a blocchi o puntuale.

7.3 Mapping da frequenza audio a pixel HSL

Dopo aver ottenuto la lista monodimensionale di frequenze, a partire dai campioni audio (con una delle 2 metodologie precedenti) è possibile determinare i relativi pixel HSL associati mediante il **mapping Frequenza - HSL** (utilizzando per le componenti gli stessi intervalli definiti nel caso inverso da immagine a suono). Come prima cosa a partire dal valore di frequenza si determina la terna nota musicale, ottava e alterazione mediante la distanza dalla frequenza principale A_4.

Per la **nota musicale** si utilizza l'associazione con la componente tonalità, impiegando solamente le **7 tonalità pure dei colori**.

Dopo aver stabilito quale dei 3 intervalli di saturazione utilizzare per l'**alterazione** considerata, viene adoperato un **valore di saturazione intermedio all'interno del range** ovvero con una distanza di circa 42 (Tabella 7.1).

Per l'ultima componente che è l'**ottava** si utilizza una strategia simile alla componente precedente ovvero dopo aver stabilito quale degli 11 intervalli di luminosità utilizzare, si adopera un **valore di luminosità intermedio all'interno del range**, con una distanza di circa 11 (Tabella 7.2).

L'assegnazione di un unico valore numerico possibile ai vari intervalli delle componenti HSL comporta una **riduzione del numero di possibili pixel HSL utilizzabili rispetto alla conversione inversa da immagine a suono**. Questa strategia però è necessaria per poter utilizzare, in entrambe le direzioni di conversione, i medesimi range definiti per le componenti, senza dover ricorrere ad algoritmi di implementazione diversi (**relazione bidirezionale**).

Alterazione	**Valore saturazione OpenCV (0-255)**
Bemolle	42
Naturale	127
Diesis	212

Tabella 7.1: Mapping da alterazione a saturazione OpenCV HSL

Ottava	Valore luminosità OpenCV (0-255)
−1	11
0	34
1	57
2	80
3	103
4	126
5	149
6	172
7	195
8	218
9	241

Tabella 7.2: Mapping da ottava a luminosità OpenCV HSL

7.4 Struttura complessiva processo Sound-Image

Il processo complessivo di conversione da suono ad immagine chiamato Sound-Image (inverso di Image-Sound), senza considerare le funzionalità aggiuntive, prevede generalmente la seguente struttura (ci sono minime differenze registrando audio con/senza durata finita):

1. **Registrazione campioni audio mono in input con Oboe**

   Mediante Oboe si registrano in input una quantità di campioni audio a singolo canale (mono) pari a sample rate (48 Khz, quindi 48.000 campioni al secondo) moltiplicato i secondi di audio che si desiderano acquisire (tra 1 e 5 secondi, scelti dall'utente).
   ⇓ ⇑

2. **Calcolo delle frequenze**

   Conversione campioni interi $\Longrightarrow$ **Zero crossing o Divisione**

   Se l'utente a scelto il formato float per i campioni audio viene fatta la conversione di questi a int. Si calcolano le frequenze a partire dai campioni audio interi utilizzando una struttura a blocchi (zero crossing) o puntuale (divisione).
   ‖
   ==========
   Termina la parte bidirezionale della struttura in cui si registrano gruppi di campioni audio (burst) al fine di determinare le relative frequenze e ricominciare nuovamente il processo di acquisizione, fino ad ottenere tutti gli elementi necessari.

$\Vert$
$\Downarrow$

3. **Preprocessing frequenze**

   Ridimensionamento elementi per Hilbert $\Longrightarrow$ Round frequenze

   Si ridimensiona la lista monodimensionale delle frequenze in modo da avere un numero di elementi compatibile con Hilbert (L^2). Il ridimensionamento provoca un'interpolazione delle frequenze costituenti e quindi è necessario effettuare il round delle frequenze alla nota musicale più vicina. Nel round si controlla anche che le frequenze ottenute non siano minori o maggiori al semitono musicale permesso, settando nel caso i valori al minimo o massimo.

   $\Downarrow$

4. **Applicazione curva di Hilbert**

   Si converte con Hilbert la lista monodimensionale quadrata delle frequenze in immagine bidimensionale.

   $\Downarrow$

5. **Applicazione mapping Frequenza - HSL con Newton**

   Si convertono le frequenze dell'immagine bidimensionale in terne HSL mediante il mapping Frequenza - HSL che utilizza la relazione di Newton.

   $\Downarrow$

6. **Generazione immagine preview OpenCV**

   Conversione RGB $\Longrightarrow$ Ridimensionamento preview $\Longrightarrow$ Conversione RGBA

   Vengono effettuate in sequenza una serie di elaborazioni all'immagine generata HSL, mediante la libreria OpenCV, al fine di ottenere l'immagine finale da visualizzare nella camera preview. Infatti si converte l'immagine da HSL a RGB (funzione C++ cvtColor), poi si effettua un ridimensionamento di questa alle dimensioni della preview ed infine si converte l'immagine ridimensionata da RGB a RGBA per essere visualizzata sulla camera preview, mediante una scrittura con le OpenGL 2.0.

Capitolo 8

Le 4 modalità di elaborazione nell'app ImageSound

8.1 Architettura software dell'applicazione

Per definire gli **elementi di interfaccia grafica** chiamati layout nell'app Android ImageSound viene utilizzato il linguaggio di markup **XML** (Extensible Markup Language), il quale consente di costituire documenti mediante l'impiego di un set di tag predefiniti o personalizzati (estensione delle etichette). I tag non sono altro che delimitatori di informazioni unitarie, contraddistinti dalla medesima parola univoca (in parentesi angolari <>) sia in apertura che in chiusura (con una barra / rispetto l'apertura), al fine di assegnare una semantica nota all'informazione considerata.

Per quanto riguarda il **codice scritto in linguaggio C++** viene effettuata la suddivisione concettuale, utilizzando file con lo stesso nome ma formato diverso, tra **definizione delle funzioni** (estensione **.h**) e relativa **implementazione** (estensione **.cpp**). Considerazioni simili (sulla struttura e relativi nomi dei file similari) vengono fatte anche per quanto riguarda la definizione delle funzioni Java, la cui vera implementazione viene fatta in C++ (chiamate con JNI).

Quindi a **livello software l'app risulta suddivisa logicamente in 7 parti** ovvero:

1. **Home dell'app:** pagina principale dell'applicazione da cui è possibile cambiare tra le 4 modalità di elaborazione, avviare o interrompere un'elaborazione e aprire la pagina delle impostazioni. La logica viene definita nel file *ImageSoundActivity.java* mentre il suo file di layout portrait e landscape è *image_sound_layout.xml*. Vi è anche una schermata di dialogo ausiliaria per confermare l'elaborazione nella modalità Sound-Image non live chiamata *layout alert_dialog_layout.xml*;

2. **Impostazioni:** pagina di impostazioni da cui l'utente può scegliere i parametri principali da impiegare negli algoritmi delle 4 modalità di elaborazione. La logica viene definita nel file *SettingActivity.java* mentre il suo file di layout portrait e landscape è *setting_image_sound_layout.xml*;

3. **Camera preview di OpenCV:** gestione della camera preview nelle 4 modalità di elaborazione per acquisire frame video o visualizzare immagini prodotte nella preview. Nel file *CameraGLSurfaceViewEstesa.java* vi è la definizione delle funzioni necessarie mentre la relativa implementazione avviene nel file *native-lib.cpp* per le chiamate JNI;

4. **Audio di output Oboe:** gestione audio di output (speaker) di Oboe per lo start e lo stop dello stream nelle 2 modalità Image-Sound. Nel file *EngineOutputImageSound.java* vi è la definizione delle funzioni necessarie mentre la relativa implementazione avviene utilizzando un insieme di file per la gestione di Oboe (*AudioEngineImageSound.h* e *AudioEngineImageSound.cpp*), generazione dell'oscillazione sinusoidale sweep (*Oscillator.h* e *Oscillator.cpp*) e *native-lib.cpp* per le chiamate JNI;

5. **Audio di input Oboe:** gestione audio di input (microfono) di Oboe per lo start e lo stop dello stream nelle 2 modalità Sound-Image. Nel file *EngineInputSoundImage.java* vi è la definizione delle funzioni necessarie mentre la relativa implementazione avviene utilizzando un insieme di file per la gestione di Oboe (*AudioEngineSoundImage.h* e *AudioEngineSoundImage.cpp*) e *native-lib.cpp* per le chiamate JNI;

6. **Utility:** elementi di supporto generali necessari all'intera applicazione come costanti, strutture dati e funzioni, distribuite su multipli file. I file *imageSoundUtility.h* e *imageSoundUtility.cpp* contengono principalmente le funzioni di conversione come quella della curva di Hilbert, conversione da pixel HSL a frequenza e viceversa, processo completo di conversione da immagine a suono e viceversa, restituzione della frequenza dal campione audio registrato (zero crossing e divisione), ecc. . Il *fileLockFreeQueue.h* definisce la coda circolare thread-safe di Oboe per la scrittura o lettura delle frequenze usando un singolo produttore e un singolo consumatore. Il file *native-lib.cpp* contiene tutte le implementazioni C++ delle funzioni definite in Java in modo da effettuare le chiamate tramite JNI;

7. **Funzionalità aggiuntive:** gestione delle funzionalità aggiuntive che possono essere utilizzate nelle 4 modalità di elaborazione. Le funzionalità che possono essere utilizzate nelle 2 modalità Image-Sound riguardano il riconoscimento dello stile o elemento architetturale presente nell'immagine oppure la possibilità di comporre musica. Invece nelle 2 modalità Sound-Image può essere svolto il riconoscimento, nell'audio registrato tramite microfono, dell'evento audio che accade o del genere musicale. Ulteriori informazioni e i relativi file utilizzati vengono descritti nel dettaglio nei successivi capitoli dedicati.

Nella Figura 8.1 viene quindi mostrato un riepilogo grafico complessivo dei file di codice utilizzati nell'app ImageSound, escludendo le funzionalità aggiuntive ma riportando le relazioni che vi sono tra le componenti.

Le **modalità di elaborazione** dell'app appartenenti alla tipologia **non live** hanno la caratteristica che per l'intero processo, sia i dati forniti in input che quelli restituiti in

output presentano una **durata limitata**, essendo utilizzata una sola porzione di informazione a partire da dati in tempo reale **non limitati in durata** (**live**). Invece nelle modalità live viene svolto, multiple volte in sequenza, lo stesso processo di elaborazione (similare alle corrispettive non live) utilizzando dati aggiornati in tempo reale, fintanto non si decide di interrompere l'esecuzione con un apposito pulsante *stop* (limitando la durata). Inoltre nelle modalità live viene fornita anche una **qualità percentuale dell'elaborazione** per rappresentare la sincronizzazione temporale presente tra i dati acquisiti o prodotti nell'elaborazione e l'esecuzione del processo stesso di conversione.

8.2 Modalità Image-Sound non live

In breve la **modalità Image-Sound non live** consente, dopo aver avviato l'elaborazione con il pulsante *start*, di premere sulla preview della fotocamera del dispositivo (dopo l'apposito messaggio esplicativo) per scattare la relativa foto. La **foto scattata viene utilizzata per generare, con il processo Image-Sound, un suono di durata finita in secondi** (tra 1 e 5, scelto dall'utente nelle impostazioni), riprodotto in output con lo speaker del dispositivo. Il suono così prodotto può anche essere riascoltato premendo l'apposito pulsante dedicato.

Scendendo invece a livello implementativo dell'intero processo di elaborazione, dopo aver avviato la camera preview si attende che l'utente scatti la foto premendo sulla preview (si visualizza l'immagine della fotocamera aggiornata in tempo reale, aspetto rappresentato con il booleano *false*). Lo scatto della foto viene simulato mandando inizialmente in output sulla preview sempre il frame acquisito in input inalterato e all'evento di pressione sulla preview, si memorizza il frame di input corrente mandando successivamente sempre questo in output (non prendendo più in considerazioni quelli nuovi di input con il booleano a *true*). Allo scatto della foto si inizia la conversione del frame immagine (dimensione massima di 1280X920) in suono. Il **frame della foto viene acquisito nel modello colore RGBA** tramite una lettura con le OpenGL 2.0.

A questo punto si crea un nuovo frame convertendo con OpenCV il frame della **foto dal modello colore RGBA a RGB** 24 bit. Poi si **ridimensiona il frame RGB a quadrato** in modo da avere le dimensioni necessarie ad Hilbert (come 256X256, con lato L). Il **frame quadrato RGB viene convertito nel modello colore HSL** e a questo punto si **applica Hilbert** in modo da ottenere una matrice a singola riga con le frequenze (**conversione da pixel HSL a frequenza semitoni musicali**) con un numero di elementi pari a L^2. La conversione mediante Hilbert può essere svolta sia in modo sequenziale con un unico thread o in modo parallelo con multipli thread (tipologia di thread da utilizzare per l'esecuzione dell'app che viene scelta dall'utente nelle impostazioni, tramite un parametro booleano).

Si **ridimensiona la matrice a singola riga di Hilbert** in modo da avere sample rate (48000) moltiplicato i secondi di audio che si vogliono ottenere in output. Il ridimensionamento provoca un'**interpolazione delle frequenze** costituenti e quindi è necessario effettuare il **round di queste al semitono musicale più vicino**, controllando anche che la frequenza ottenuta non sia minore o maggiore al range permesso (8.1758 Hz $\leq$ Freq

$\leq$ 15805.3667 Hz oppure $C_{-1} \leq Sem \leq B_9$), settando nel caso il valore al minimo o massimo. Il round delle frequenze può essere fatto sia in modo sequenziale che parallelo. Si **scrivono attraverso un unico thread tutte le frequenze così ottenute nella coda thread-safe** e a questo punto si fa partire la gestione dell'audio di output con Oboe.

Oboe non fa altro che creare un **oscillatore che preleva una alla volta le frequenze dalla coda**, ripetendo una serie di passi. Calcola l'incremento di fase corrente dalla frequenza ottenuta, lo somma all'accumulatore di fase (inizialmente a 0), se l'accumulatore di fase è maggiore di 2π sottrae 2π e a questo punto determina il campione audio float (range da -1 a $+1$) applicando la funzione seno all'accumulatore di fase moltiplicato l'ampiezza (volume dell'audio, scelto dall'utente nell'intervallo da 0.1 a 1.0). Il **campione audio così ottenuto viene scritto sia per il canale sinistro che per il destro (audio stereo)**. Se l'utente a richiesto un audio di output di tipo intero viene effettuata una conversione dei campioni float a int (range da -32768 a $+32768$). Quando si **raggiunge la fine della coda delle frequenze** si scrive in output, per la callback corrente, degli 0 ovvero assenza di audio e si chiede di interrompere la richiesta di ulteriori callback di scrittura audio.

Siccome sia il thread della camera preview che il thread audio di Oboe non possono essere interrotti direttamente dai relativi thread (si provoca un'eccezione ovvero un'alterazione a run-time del normale flusso di esecuzione dell'applicativo), allo scatto della foto viene fatto partire un'ulteriore **thread di sincronizzazione** che dopo 2 secondi aggiuntivi di margine, rispetto ai secondi di audio da produrre in output, interrompe entrambi i thread. Inoltre viene data anche la possibilità di **riascoltare l'audio prodotto** dalla foto scattata semplicemente memorizzando la prima volta l'indice di partenza della coda delle frequenze e riproducendole nuovamente in output con l'oscillatore.

Nella Figura 8.2 viene riportato in modo grafico l'intero processo di esecuzione della modalità Image-Sound non live.

8.3 Modalità Sound-Image non live

In breve la **modalità Sound-Image non live** consente, dopo aver avviato l'elaborazione con il pulsante *start*, di confermare (in un apposita finestra di dialogo) di voler registrare audio con una durata finita (tra 1 e 5 secondi) tramite il microfono del dispositivo. **L'audio registrato con durata finita viene utilizzato per generare, con il processo Sound-Image, un'immagine** che viene visualizzata sulla preview. Il suono registrato può anche essere riascoltato premendo l'apposito pulsante dedicato.

A livello implementativo l'intero processo di elaborazione inizia facendo partire sia la gestione dell'audio di input con Oboe, per la **registrazione dei campioni audio a singolo canale (mono)**, sia la camera preview. Oboe ad ogni callback non fa altro che determinare il numero di campioni audio rimanenti da acquisire, per la callback corrente, per raggiungere il totale sample rate moltiplicato i secondi audio da registrare. Nel caso con l'insieme di campioni correnti non si supera il totale richiesto, vengono acquisiti tutti altrimenti solo quelli necessari. Se l'utente ha richiesto la registrazione di audio di tipo float viene svolta una conversione dei campioni da float a int.

A questo punto se è stato richiesto di **calcolare le frequenze dai campioni audio** con il **zero crossing**, si determina un'unica frequenza per l'insieme dei campioni da acquisire altrimenti con la **divisione** si calcola una frequenza diversa per ciascuno dei campioni. Si **aggiungono tante frequenze nella coda thread-safe** quanti sono i campioni da acquisire (replicazione della stessa con zero crossing o ciascuna diversa con la divisione). Se la callback corrente termina le acquisizioni dei campioni totali richiesti si chiede di interrompere la richiesta di ulteriori callback di lettura audio e viene settato un booleano a *true* per indicare la terminazione dell'intero processo di registrazione audio.

La camera preview fatta partire inizialmente non fa altro che visualizzare sempre un frame video tutto nero (assenza di tutti i campioni audio necessari con il booleano di visualizzazione a *true*) fintanto il booleano per indicare la terminazione dell'intero processo di registrazione audio è *false*. Completata la registrazione audio invece si incomincia a trasformare l'audio registrato in immagine andando a prelevare le frequenze dalla coda thread-safe. Per questo si **leggono le frequenze dalla coda con un singolo thread per costituire una matrice a singola riga** con un numero di elementi pari a sample rate moltiplicato i secondi audio richiesti.

Si **ridimensiona la matrice a singola riga in modo da avere un numero di elementi compatibile con Hilbert** (come 256X256). Si **convertono le frequenze con dimensione quadrata (corrispondenti ai semitoni musicali) in immagine HSL**, controllando prima che le frequenze ottenute non siano minori o maggiori ai semitoni musicali permessi, svolgendo anche un'eventuale **round delle frequenze al semitono musicale più vicino** (a causa del ridimensionamento precedente che ha provocato un'interpolazione). La conversione ad immagine può essere svolta sia in modo sequenziale con un unico thread o in modo parallelo con multipli thread.

Si effettua la **conversione dell'immagine HSL nel modello colore RGB** e si svolge un **ridimensionamento alle dimensioni della camera preview**. A questo punto si **converte l'immagine da RGB a RGBA** 32 bit (con la trasparenza) per la **visualizzazione sulla preview** mediante una scrittura con le OpenGL 2.0. L'immagine RGBA creata viene memorizzata, in modo da visualizzare successivamente in output nella preview sempre la stessa immagine. Infine dal thread della camera preview si interrompe il thread audio di Oboe e si lancia un nuovo thread che dopo 0.5 secondi interrompe anche la camera preview. Inoltre viene data anche la possibilità di **riascoltare l'audio registrato** semplicemente memorizzando i campioni audio acquisiti precedentemente e riproducendoli nuovamente in output.

Nella Figura 8.3 viene riportato in modo grafico l'intero processo di esecuzione della modalità Sound-Image non live.

8.4 Modalità Image-Sound live

In breve la **modalità Image-Sound live** consente, dopo aver avviato l'elaborazione con il pulsante *start*, di utilizzare continuamente un determinato numero di frame immagine al secondo (scelto dall'utente nelle impostazioni), che costituisce un **video, al fine di generare con il processo Image-Sound una porzione di audio**, contenuta in un suono

finale di durata non definita, il quale viene riprodotto in output con lo speaker del dispositivo. Quando l'utente preme l'apposito pulsante *stop* per interrompere l'elaborazione anche il suono viene interrotto (non presenta più una durata non definita).

A livello implementativo l'intero processo di elaborazione inizia facendo partire la camera preview che visualizza l'immagine della fotocamera aggiornata in tempo reale (aspetto rappresentato con il booleano dei frame video sempre a *false*). Quindi si inizia la conversione del frame video corrente (dimensione massima di 1280X920) in suono come nella modalità Image-Sound non live. Come prima cosa si **acquisisce il frame video nel modello colore RGBA** mediante una lettura con le OpenGL 2.0. A questo punto si crea un nuovo frame **convertendo il frame immagine dal modello colore RGBA a RGB** 24 bit. Poi si **ridimensiona il frame RGB a quadrato** in modo da avere le dimensioni necessarie ad Hilbert (come 256X256, con lato L). Il **frame quadrato RGB viene convertito nel modello colore HSL** e a questo punto si **applica Hilbert** in modo da ottenere una matrice a singola riga con le frequenze (**conversione da pixel HSL a frequenza semitoni musicali**) con un numero di elementi pari a L^2. La conversione mediante Hilbert può essere svolta sia in modo sequenziale con un unico thread o in modo parallelo con multipli thread.

A questo punto si **ridimensiona la matrice a singola riga di Hilbert** in modo da avere un numero di elementi pari a sample rate (48000) diviso le immagini da elaborare al secondo (rispetto sample rate moltiplicato i secondi di audio, utilizzati della modalità non live). Il ridimensionamento provoca un'**interpolazione delle frequenze** costituenti e quindi è necessario effettuare il **round delle frequenze al semitono musicale più vicino** controllando anche che la frequenza ottenuta non sia minore o maggiore ai semitoni musicali permessi ($8.1758\ \text{Hz} \leq \text{Freq} \leq 15805.3667\ \text{Hz}$ oppure $C_{-1} \leq \text{Sem} \leq B_9$), settando nel caso il valore al minimo o massimo. Il round delle frequenze può essere fatto sia in modo sequenziale che parallelo. Si **scrivono attraverso un unico thread tutte le frequenze così ottenute nella coda thread-safe**.

Adesso se la conversione da frame video a suono ha impiegato meno di 1 diviso le immagini da elaborare al secondo secondi si attende (non viene svolta nessuna conversione da immagine a suono) il successivo frame video necessario da elaborare. Altrimenti passati i secondi necessari, solo la prima volta si fa partire la gestione dell'audio di output con Oboe (per avere un **gap iniziale di differenza tra camera preview e Oboe**) in quanto ora si hanno a disposizione le informazioni necessarie per far partire l'esecuzione (i primi sample rate diviso le immagini da elaborare al secondo campioni audio). Poi si incrementa di 1 il contatore delle immagini elaborate al secondo (inizialmente a 0) e nel caso sono state elaborate tutte le immagini richieste in un secondo, si resetta a 0 il contatore delle immagini elaborate al secondo e si fa partire un **thread per l'aggiornamento della qualità percentuale dell'elaborazione** mostrata all'utente. Qualità che viene determinata tenendo in considerazione in sample rate campioni (corrispondenti a 1 secondo temporale) quanti campioni sono stati mancati (Oboe necessita di frequenze da leggere dalla coda ma questa è vuota e quindi produce in output lo 0 ovvero assenza di suono) e quanti campioni sono stati prelevati correttamente.

Da qui in avanti la camera preview ripete sempre ciclicamente i medesimi passaggi precedenti: conversione da immagine a suono, attesa se l'elaborazione termina prima di

immagini da elaborare al secondo secondi, dopo 1 diviso immagini da elaborare al secondo secondi incrementa di 1 il contatore e se si sono elaborate tutte le immagini richieste in un secondo si imposta a 0 il contatore e si aggiorna la qualità.

Alla partenza Oboe non fa altro che creare un **oscillatore che preleva una alla volta le frequenze dalla coda**, ripetendo una serie di passi. Calcola l'incremento di fase corrente dalla frequenza ottenuta, lo somma all'accumulatore di fase (inizialmente a 0), se l'accumulatore di fase è maggiore di 2π sottrae 2π e a questo punto determina il campione audio float (range da -1 a $+1$) applicando la funzione seno all'accumulatore di fase moltiplicato l'ampiezza (volume dell'audio, scelto dall'utente nell'intervallo da 0.1 a 1.0). Il **campione audio così ottenuto viene scritto sia per il canale sinistro che per il destro (audio stereo)**. Se l'utente a richiesto un audio di output di tipo intero viene effettuata una conversione dei campioni float a int (range da -32768 a $+32768$).

Se al prelevamento della frequenza dalla coda questa è vuota si utilizza uno 0 ovvero assenza di suono e si incrementa di 1 il contatore dei campioni mancati. Se invece la frequenza viene prelevata correttamente si incrementa di 1 il contatore dei campioni corretti. Dopo sample rate campioni si **aggiorna il valore della qualità dell'elaborazione**. Infine quando l'**utente decide di interrompere l'elaborazione** vengono fermati sia il thread della camera preview che il thread audio di Oboe.

Nella Figura 8.4 viene riportato in modo grafico l'intero processo di esecuzione della modalità Image-Sound live.

8.5 Modalità Sound-Image live

In breve la **modalità Sound-Image live** consente, dopo aver avviato l'elaborazione con il pulsante *start*, di registrare continuamente sample rate campioni audio al secondo tramite il microfono del dispositivo. Il totale dei campioni audio al secondo viene suddiviso in un numero di gruppi pari al quantitativo di frame immagine che l'utente desidera produrre in un secondo. Ogni **gruppo di campioni audio viene utilizzato per generare, con il processo Sound-Image**, un'immagine che viene visualizzata sulla preview quindi sono mostrate **multiple immagini prodotte al secondo** (che costituisce un **video**). Quando l'utente preme l'apposito pulsante *stop* per interrompere l'elaborazione anche la visualizzazione sulla preview viene interrotta (il video non presenta più una durata non definita).

A livello implementativo l'intero processo di elaborazione inizia facendo partire sia la gestione dell'audio di input con Oboe, per la **registrazione dei campioni audio a singolo canale (mono)**, sia la camera preview. Oboe ad ogni callback, se l'utente ha richiesto la registrazione di audio di tipo float viene svolta una conversione dei campioni da float a int.

A questo punto se è stato richiesto di **calcolare le frequenze dai campioni audio** con il **zero crossing**, si determina un'unica frequenza per l'insieme dei campioni da acquisire altrimenti con la **divisione** si calcola una frequenza diversa per ciascuno dei campioni. Si **aggiungono tante frequenze nella coda thread-safe** quanti sono i

campioni da acquisire (replicazione della stessa zero crossing o ciascuna diversa con la divisione).

Se sono state effettuate tutte le callback di registrazione iniziale (per avere un **gap iniziale di differenza tra Oboe e camera preview**) si procede con l'**aggiornamento delle statistiche sulla qualità dell'elaborazione**. Infatti se all'inserimento di ogni frequenza nella coda, questa è piena, non viene inserita perdendone il valore e si incrementa di 1 il contatore dei campioni mancati. Se invece la frequenza viene inserita correttamente si incrementa di 1 il contatore dei campioni corretti. Dopo sample rate campioni si aggiorna il valore della qualità percentuale dell'elaborazione. Il **numero di callback di registrazione iniziale di gap** viene calcolato facendo il round della divisione tra [1 diviso immagini da elaborare al secondo] e [campioni audio ad ogni callback diviso il sample rate], dove ad ogni terminazione di callback audio viene decrementato di 1 il valore. Al raggiungimento dello 0 non si decrementa più ulteriormente il valore e viene settato solamente la prima volta un booleano a *true* per indicare alla camera preview che ora si hanno tutte le informazioni necessarie per far partire l'esecuzione (i campioni audio necessari per creare il primo frame video).

La camera preview fatta partire inizialmente non fa altro che visualizzare sempre un frame video tutto nero (assenza di tutti i campioni audio necessari con il booleano di visualizzazione a *true*) fintanto il booleano per indicare la presenza di tutte le informazioni necessarie è *false*. Quando il booleano per le informazioni necessarie è impostato a *true*, si incomincia a trasformare l'audio registrato in immagine, andando a prelevare le frequenze dalla coda come nella modalità Sound-Image non live. Per questo si **leggono le frequenze dalla coda con un singolo thread per costituire una matrice a singola riga** con sample rate diviso immagini da elaborare al secondo elementi (rispetto sample rate moltiplicato i secondi audio richiesti della modalità non live). Nel caso la frequenza non è disponibile (coda vuota) si sostituisce con la frequenza del semitono musicale permesso più basso (8.1758 Hz).

Si **ridimensiona la matrice a singola riga in modo da avere un numero di elementi compatibile con Hilbert** (come 256X256). Si **convertono le frequenze con dimensione quadrata (corrispondenti ai semitoni musicali) in immagine HSL**, controllando prima che le frequenze ottenute non siano minori o maggiori ai semitoni musicali permessi, svolgendo anche un'eventuale **round delle frequenze al semitono musicale più vicino** (a causa del ridimensionamento precedente che ha provocato un'interpolazione). La conversione ad immagine può essere svolta sia in modo sequenziale con un unico thread o in modo parallelo con multipli thread.

Si effettua la **conversione dell'immagine HSL nel modello colore RGB** e si effettua un **ridimensionamento alle dimensioni della camera preview**. A questo punto si effettua una **conversione da immagine RGB a RGBA** 32 bit (con la trasparenza) per la **visualizzazione sulla preview** mediante una scrittura con le OpenGL 2.0. Adesso se la conversione da suono a frame video ha impiegato meno di 1 diviso immagini da elaborare al secondo secondi si attende (non viene svolta nessuna conversione da suono a immagine, restituendo nella visualizzazione di output il frame video creato precedentemente) il successivo frame video richiesto da creare. Altrimenti passati i secondi necessari si incrementa di 1 il contatore delle immagini elaborate al secondo (inizialmente a 0) e nel

caso sono state elaborate tutte le immagini richieste in un secondo, si resetta a 0 il contatore delle immagini elaborate al secondo e si fa partire un **thread per l'aggiornamento della qualità percentuale dell'elaborazione** mostrata all'utente.

Da qui in avanti la camera preview ripete sempre ciclicamente i medesimi passaggi precedenti: conversione da suono a immagine, attesa se l'elaborazione termina prima di immagini da elaborare al secondo secondi, dopo 1 diviso immagini da elaborare al secondo secondi incrementa di 1 il contatore e se si sono elaborate tutte le immagini richieste in un secondo si imposta a 0 il contatore e si aggiorna la qualità. Infine quando l'**utente decide di interrompere l'elaborazione** vengono fermati sia il thread della camera preview che il thread audio di Oboe.

Nella Figura 8.5 viene riportato in modo grafico l'intero processo di esecuzione della modalità Sound-Image live.

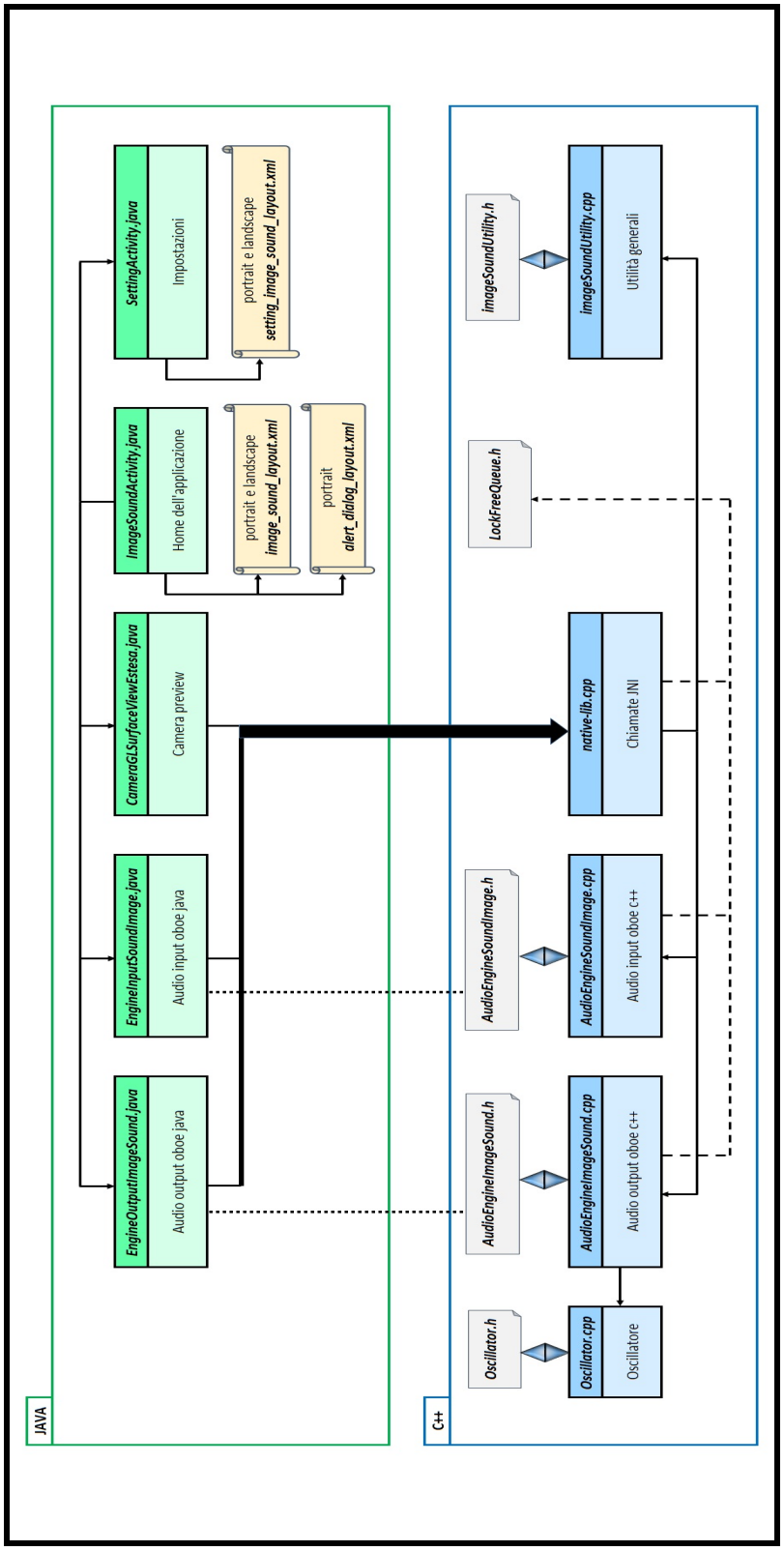

Figura 8.1: Architettura software ImageSound (C++ e Java) senza funzionalità aggiuntive

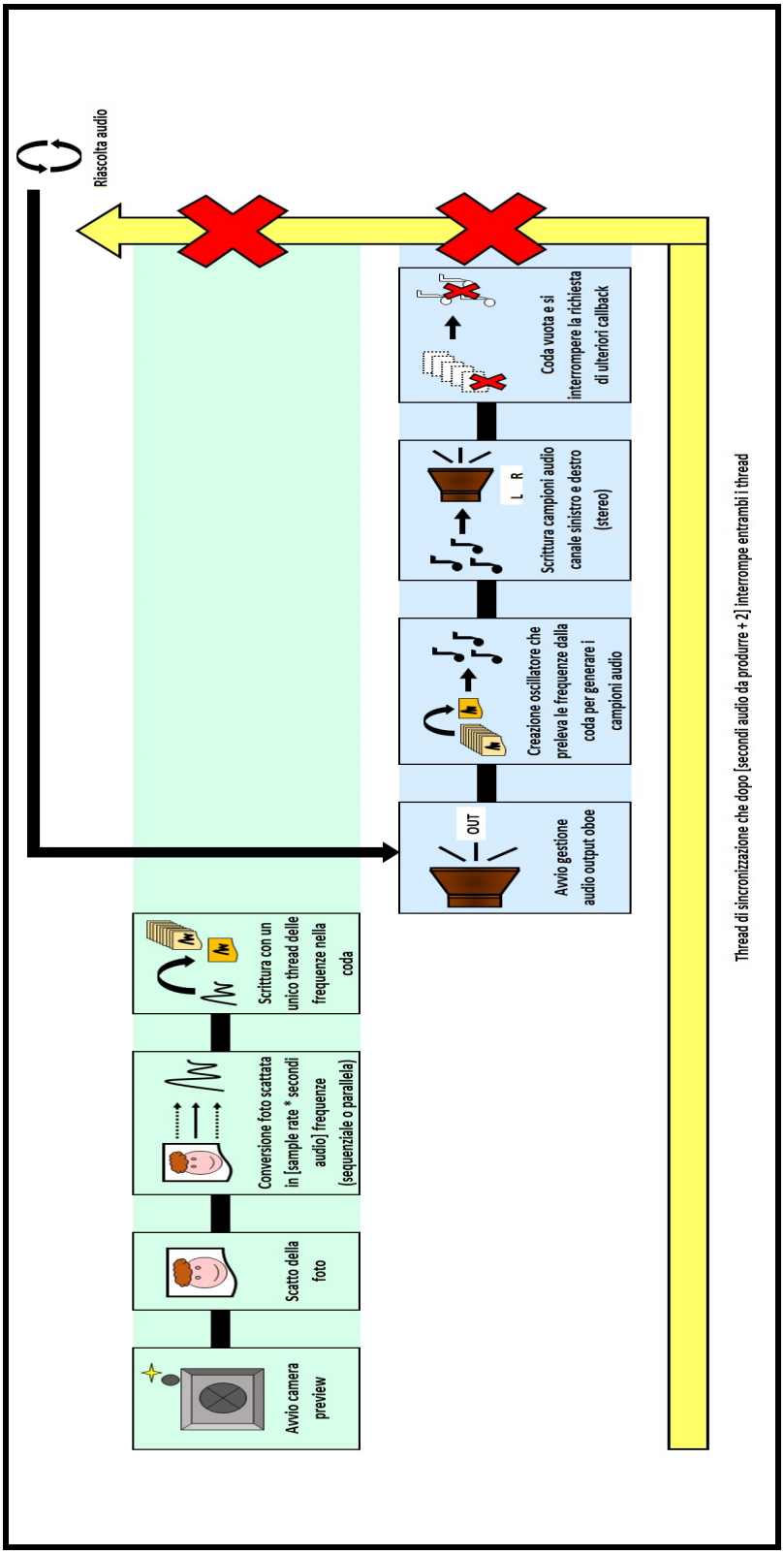

Figura 8.2: Schema di esecuzione modalità Image-Sound non live

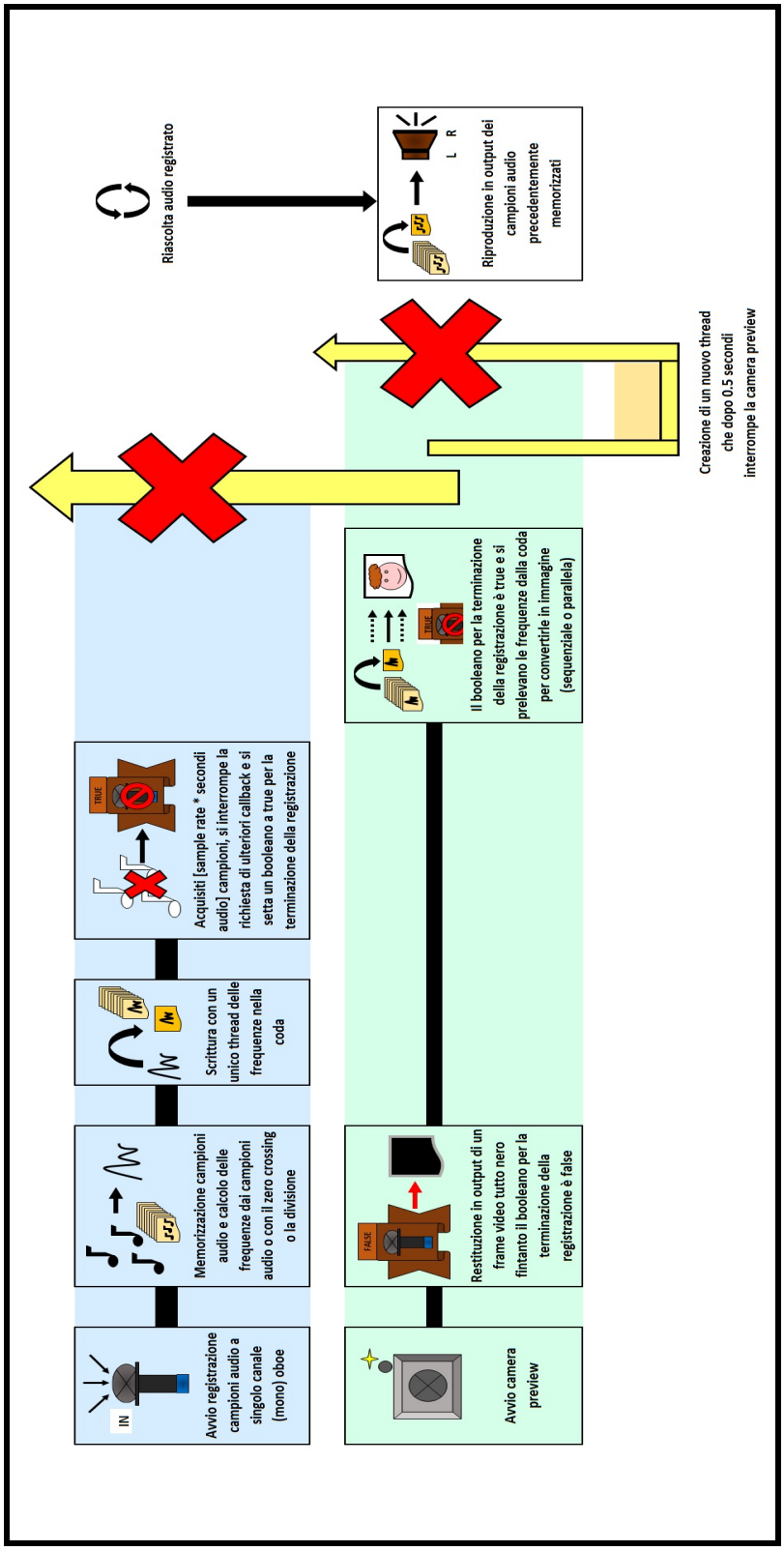

Figura 8.3: Schema di esecuzione modalità Sound-Image non live

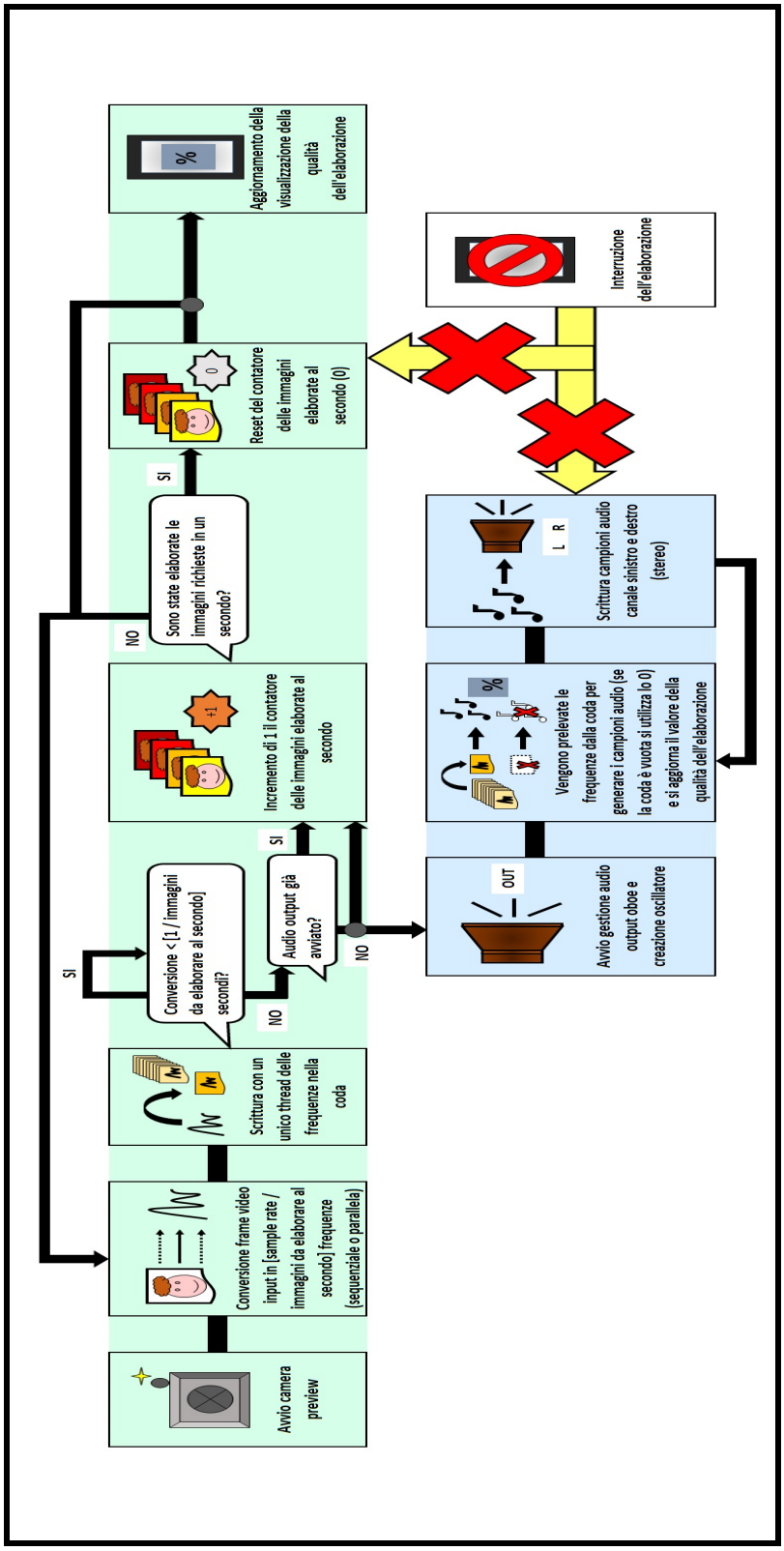

Figura 8.4: Schema di esecuzione modalità Image-Sound live

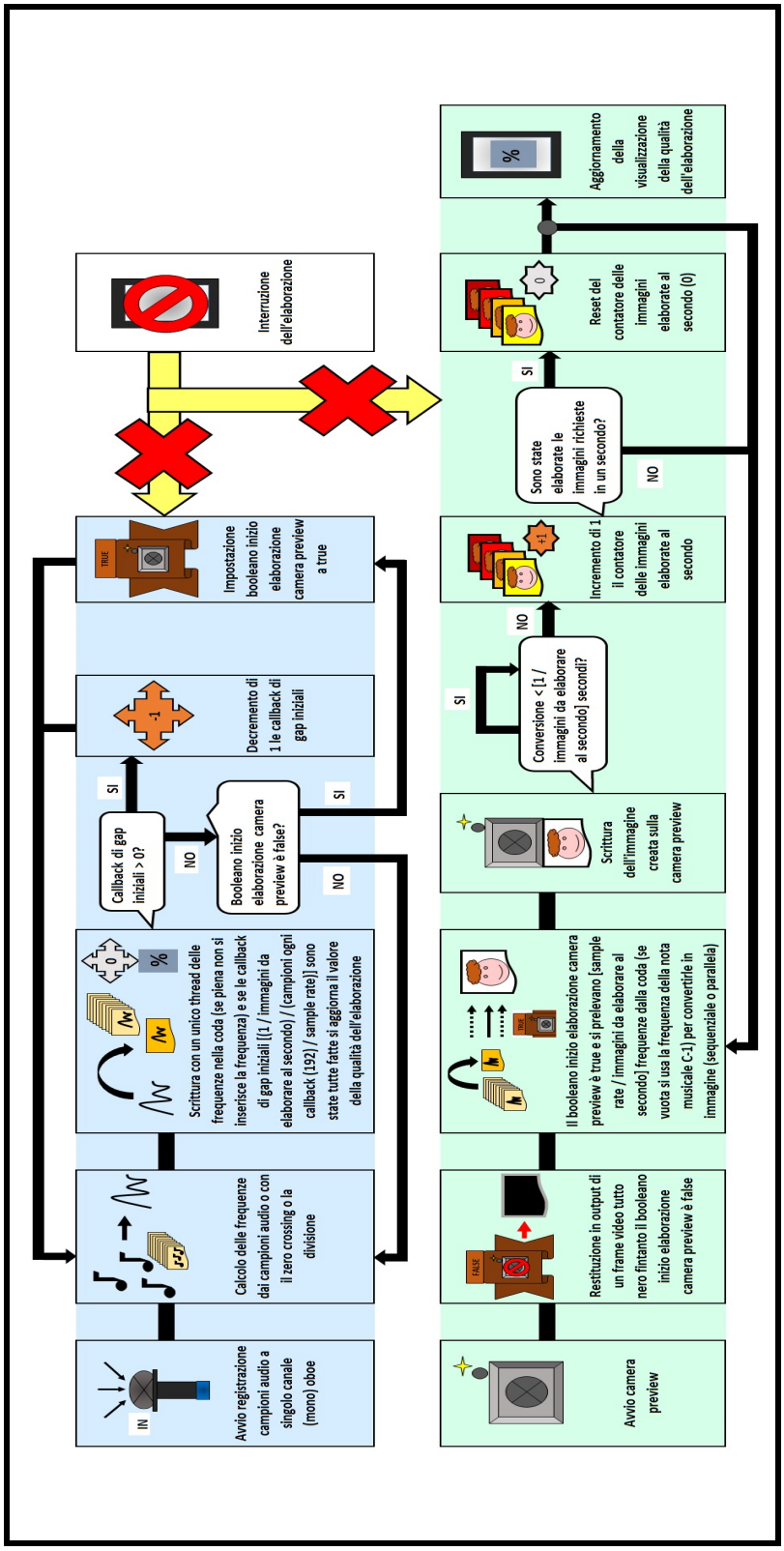

Figura 8.5: Schema di esecuzione modalità Sound-Image live

Capitolo 9

Funzionalità aggiuntive nelle 2 modalità Image-Sound

9.1 Processo Image-Sound con classificazione architetturale

Le funzionalità aggiuntive per le 2 modalità Image-Sound che riguardano la classificazione delle immagini non utilizzano principalmente, nel mapping da immagine a suono, le informazioni colorimetriche dei pixel dell'immagine ma considerano la label assegnata dal classificatore al frame immagine (le label sono state tradotte dall'inglese all'italiano). Infatti queste funzionalità si basano sul concetto di corrispondenza cross modale tra suoni, oggetti nello spazio e colori. La **corrispondenza cross modale mette in relazione elementi sensoriali diversi** ed in questo caso sono stati adoperati l'udito (suono prodotto), la vista (luminosità o colori presenti) e il tatto (dato dalla forma degli oggetti).

Siccome viene svolta la classificazione delle immagini in base allo stile o elemento architetturale presente in realtà gli oggetti nello spazio o forme 3D si trovano sul piano bidimensionale dell'immagine. Quindi l'**analisi delle forme 3D viene semplificata** mediante una rappresentazione su un grafico, dove nell'asse X delle ascisse viene riportata la complessità della forma (Complexity) mentre nell'asse Y delle ordinate viene riportata la spigolosità della forma (Angularity).

La **Complexity** può avere **3 possibili valori ovvero Bassa, Media o Alta** e per poterne stabilire il valore specifico viene utilizzato il concetto di **complessità visuale di immagini**. Infatti per poter stabilire la complessità di un quadro artistico vengono utilizzate le 7 dimensioni di Roberts, dove ad ogni dimensione può essere assegnato un punteggio di 1, 2 o 3. La complessità finale dell'opera è data dalla **media aritmetica delle complessità assegnate alle 7 dimensioni**. Nello specifico le **7 dimensioni di Roberts** sono:

1. **Incomprensibilità degli elementi:** quanto è difficile identificare gli elementi;
2. **Disorganizzazione:** com'è difficile organizzare gli elementi in una scena coerente;
3. **Quantità di elementi:** numerosità degli elementi;

4. **Varietà di elementi:** eterogeneità degli elementi;
5. **Asimmetria:** quanto è sbilanciata l'immagine;
6. **Varietà di colori:** eterogeneità dei colori;
7. **Aspetto tridimensionale:** quanto è tridimensionale l'immagine.

Le 7 dimensioni di Roberts inoltre sono impiegate, organizzate diversamente e rappresentate con termini similari, per valutare la **complessità visuale di immagini architettoniche**. Infatti da tali dimensioni sono stati definiti i seguenti **5 attributi architetturali**:

1. **Simmetria:** simmetria degli elementi, in relazione alla dimensione 5 di Roberts;
2. **Complessità:** numero di elementi presenti, in relazione alle dimensioni 1 e 3 di Roberts;
3. **Ritmo:** ripetizione degli elementi architetturali, in relazione alle dimensioni 4 e 6 di Roberts;
4. **Pattern:** strutture formali organizzate, in relazione alla dimensione 2 di Roberts;
5. **Stress:** direzione degli elementi, in relazione alla dimensione 7 di Roberts.

Essendo i 5 attributi architetturali legati a Roberts, viene utilizzata la media aritmetica delle 7 dimensioni di Roberts al fine di stabilire la complessità visuale di un'immagine. In particolare viene definita una **Complexity generale da assegnare ad una determinata classe di stile o elemento architetturale**, il cui valore generale viene stabilito osservando la complessità più frequente nella classe considerata.

Per l'**Angularity** sono utilizzati **3 possibili valori ovvero Rotondo, Rotondo-Spigoloso e Spigoloso** (con punte) e anche in questo caso viene assegnato un valore di **Angularity generale, da assegnare ad una determinata classe di stile o elemento architetturale**, a seconda della forma più frequente.

Generalmente si ha la concezione (da relativi studi al riguardo) che oggetti rotondi (basso valore Angularity) e meno complessi (basso valore Complexity) tendono ad essere associati ad un'elevata luminosità (brightness), al colore blu e ad emozioni piacevoli, di calma e di controllo. Viceversa oggetti spigolosi (alto valore Angularity) e molto complessi (alto valore Complexity) tendono ad essere associati ad una bassa luminosità, al colore rosso e da emozioni sgradevoli, emozionanti e dominanti. A livelli intermedi di complessità sono associati il colore arancione e il verde. Inoltre viene anche stabilito (secondo proprietà fisiche) che ad oggetti grandi solitamente sono associate pitch musicali basse (risonanza a basse frequenze) mentre ad oggetti piccoli solitamente sono associate pitch musicali alte (risonanza ad alte frequenze).

Tenendo in considerazione queste ultime considerazioni e dai 3 possibili valori sia per la componente Complexity che per l'Angularity, sono state definite **9 possibili configurazioni architetturali** (rappresentate graficamente nella Tabella 9.1) che possono essere associate ad una classe di stile o elemento architetturale (che rappresenta una forma 3D) ovvero:

1. **[Rotondo, Bassa complessità]:** si forza la tonalità generale a **blu** e si forza la luminosità di ogni pixel verso valori **alti di luminosità**;
2. **[Rotondo, Media complessità]:** si forza la tonalità generale a **viola** e si forza la luminosità di ogni pixel verso valori **medi di luminosità**;
3. **[Rotondo, Alta complessità]:** non si forza la tonalità generale verso un colore (si lascia **inalterata**) ma si forza la luminosità di ogni pixel verso valori **bassi di luminosità**;
4. **[Rotondo-Spigoloso, Bassa complessità]:** si forza la tonalità generale a **ciano** e si forza la luminosità di ogni pixel verso valori **alti di luminosità**;
5. **[Rotondo-Spigoloso, Media complessità]:** si forza la tonalità generale ad **arancione** e si forza la luminosità di ogni pixel verso valori **medi di luminosità**;
6. **[Rotondo-Spigoloso, Alta complessità]:** si forza la tonalità generale a **giallo** e si forza la luminosità di ogni pixel verso valori **bassi di luminosità**;
7. **[Spigoloso, Bassa complessità]:** non si forza la tonalità generale verso un colore (si lascia **inalterata**) ma si forza la luminosità di ogni pixel verso valori **alti di luminosità**;
8. **[Spigoloso, Media complessità]:** si forza la tonalità generale a **verde** e si forza la luminosità di ogni pixel verso valori **medi di luminosità**;
9. **[Spigoloso, Alta complessità]:** si forza la tonalità generale a **rosso** e si forza la luminosità di ogni pixel verso valori **bassi di luminosità**.

	Angularity		
Complexity	Rotondo	Rotondo - Spigoloso	Spigoloso
Alta	◎ - $L\Downarrow$	△ - $L\Downarrow$	✳ - $L\Downarrow$
Media	⊙ - $L\|\|$	△ - $L\|\|$	★ - $L\|\|$
Bassa	◯ - $L\Uparrow$	Λ - $L\Uparrow$	⋆ - $L\Uparrow$

Tabella 9.1: 9 configurazioni architetturali con componenti Complexity e Angularity

La forzatura della luminosità (associata all'ottava) di ogni pixel verso valori alti/medi/bassi di luminosità viene fatta mediante una media pesata, tra il valore corrente di

luminosità del pixel dell'immagine ed un valore alto/medio/basso di luminosità (ottava 8/6/4), associando un elevato peso al valore di forzatura della luminosità (circa 70%). Considerazioni simili vengono applicate anche per quanto riguarda la tonalità (associata ad una nota musicale), utilizzando però come valori di forzatura quelli delle 7 tonalità pure dei colori. Viene utilizzata una **forzatura dei valori e non un'assegnazione diretta** in modo tale che l'**output finale ottenuto dipende, in minima parte, anche dai pixel dell'immagine** favorendo una variabilità dei risultati tra immagini appartenenti alla stessa classe.

Quindi con le funzionalità aggiuntive che utilizzano la **classificazione architetturale delle immagini, si varia il mapping HSL - Frequenza** nel processo Image-Sound al fine di seguire i seguenti passi:

1. classificazione dell'immagine RGB in base allo stile o elemento architetturale;
2. dalla label determinata si stabilisce l'associazione con la configurazione architetturale da utilizzare (tra le 9 possibili);
3. con la configurazione architetturale si stabilisce la tonalità e la luminosità da utilizzare nella forzatura dei valori;
4. si svolge la media pesata tra i pixel HSL dell'immagine ed i valori di forzatura opportuni (con proporzione di circa il 70% per la forzatura);
5. si applica il classico mapping HSL - Frequenza ai pixel HSL con applicata la media pesata per andare a stabilire le relative frequenze.

9.2 Classificazione delle immagini in base allo stile architetturale

Per svolgere la classificazione di un'immagine in base allo **stile architetturale** sono stati utilizzati **2 dataset** ovvero:

- **ArchitecturalStyle Recognition:** dataset sbilanciato (diverso numero di elementi per le classi) composto da circa 10000 immagini, suddivise tra **25 stili architetturali** (tra cui Art Deco, Barocco, Gotico, etc.) [42]. Nello specifico il dataset è composto da **2 sotto dataset** con 25 label ciascuno ovvero il primo con circa 5000 immagini provenienti dal sito Wikimedia Commons mentre nel secondo sono state utilizzate immagini provenienti da Google. Le **immagini Wikimedia presentano un'elevata qualità e ottime condizioni** sia per quanto riguarda la luce che la posizione di scatto della foto. Questo fa si che il dataset risulta poco adeguato per l'utilizzo nel caso reale, dove difficilmente si raggiungono le condizioni ottimali. Per ovviare a questo è stato aggiunto il secondo dataset di **immagini Google con condizioni totalmente opposte**. Infatti le immagini presentano scarse condizioni di luce, scritte testuali sovrapposte all'immagine, rotazioni o distorsioni delle foto, acquisizione solo di alcune porzioni dell'oggetto completo, al fine di modellare il caso

reale. L'unione dei 2 sotto dataset fa si che il **dataset finale sia un bilanciamento tra condizioni ideali e reali**, risultando idoneo per l'utilizzo in dispositivi mobili. Il problema del dataset è che **non contiene uno stile architetturale molto importante ovvero il Rinascimento**;

- **MonuMAI:** dataset sbilanciato composto da circa 1500 immagini, suddivise tra **4 stili architetturali** [43]. In realtà dell'intero dataset viene **utilizzata solamente una classe che è quella del Rinascimento** (circa 300 immagini). Le restanti 3 classi non sono state prese in considerazione in quanto già presenti nel dataset ArchitecturalStyle Recognition. Le immagini di MonuMAI sono state **pensate per l'utilizzo su dispositivi mobili**, risultando quindi compatibili con il bilanciamento tra condizioni ideali e reali del dataset con 25 stili architetturali.

Quindi il dataset finale utilizzato per effettuare la **classificazione delle immagini in base allo stile architetturale è costituito da 26 classi**, dove 25 label appartengono a ArchitecturalStyle Recognition mentre l'ultima label Rinascimento appartiene a MonuMAI. Ad un architettura si può assegnare una tra le 26 possibili label, suddivise tra le 9 configurazioni architetturali definite, che vengono mostrate nella Tabella 9.2 (viene indicato anche l'anno di inizio e di fine dello stile).

	Angularity		
Complexity	Rotondo	Rotondo - Spigoloso	Spigoloso
Alta	Art Nouveau (1895, 1915) Bizantina (600, 800)	Postmoderna (1970, 2000) Rinascimentale (1420, 1525)	Egizia (−2580, −2400) Decostruttivismo (1980, 2000) Gotica (1200, 1600)
Media	Art Deco (1930, 2000) Palladiana (1500, 1900)	Achemenide (−550, −220) Beaux-Arts (1671, 1968) Novelty (1900, 2000)	Barocca (1600, 1700) Edoardiana (1901, 1914) Romanica (800, 1200) Neorussa (1825, 1917)
Bassa	Bauhaus (1919, 1933) Scuola di Chicago (1900, 1970) Stile Internazionale (1920, 1930)	Foursquare Americano (1890, 1930) Craftsman Americano (1860, 1930) Georgiana (1720, 1840) Neogreca (1800, 1900)	Coloniale (1600, 1900) Stile Queen Anne (1880, 1910) Tudor Revival (1900, 1920)

Tabella 9.2: 26 label per lo stile di una architettura

Inoltre nell'app, per completezza, sono state definite **2 versioni utilizzabili per la classificazione dello stile architetturale**:

- **Condizioni ideali:** per le 25 label di ArchitecturalStyle Recognition vengono **utilizzate solamente le immagine di Wikimedia, escludendo quelle di Google**. Questo porta ad una maggiore accuratezza di previsione (circa 54%) ma è utilizzabile prevalentemente per immagini in condizioni ottimali (di luce e di posizione adeguata di scatto);

- **Condizioni generiche:** viene utilizzato il dataset ArchitecturalStyle Recognition nella sua interezza con **sia le immagine di Wikimedia che quelle di Google**. Questo porta ad una minore accuratezza di previsione (circa 48%) ma è utilizzabile per immagini in condizioni arbitrarie.

9.3 Classificazione delle immagini in base all'elemento architetturale

Anche per svolgere la classificazione di un'immagine in base l'**elemento architetturale** sono stati utilizzati **2 dataset** ovvero:

- **Architectural Heritage Elements (AHE):** dataset sbilanciato composto da circa 10000 immagini con dimensione 128X128, suddivise tra **10 elementi architetturali** (tra cui Altare, Campanile, Gargoyle, etc.) [44];
- **Cultural Heritage Orthodox Churches:** dataset bilanciato composto da 800 immagini con dimensione 128X128, suddivise tra **4 elementi architetturali** presenti in chiese ortodosse [45]. Del dataset non viene considerata la classe Cupola (vista dall'esterno) in quanto già presente nel dataset AHE mentre sono **considerate le restanti 3 classi** ovvero Lampadario, Affresco e Lunetta.

Entrambi i dataset risultano compatibili tra di loro in quanto presentano immagini con le stesse dimensioni (128X128) e stesse condizioni di acquisizione ovvero le **immagini provengono dagli stessi siti che sono Flickr e Wikimedia Commons**. L'integrazione delle 3 classi del secondo dataset nel primo consente di descrivere in modo più completo l'argomento elementi architetturali mediante 13 classi. Quindi ad un **elemento architetturale si può assegnare una tra le 13 possibili label**, suddivise tra le 9 configurazioni architetturali definite, che vengono mostrate nella Tabella 9.3.

L'accuratezza di previsione, rispetto al dataset per lo stile architetturale, è di molto superiore ovvero circa il 93%. Questo dipende sia dal fatto che sono state utilizzate meno label (la metà ovvero 13 rispetto 26) sia dalla minore complessità intrinseca del problema analizzato. Infatti la **rilevazione dello stile architetturale è più complessa** in quanto può essere vista come l'identificazione di multipli elementi contemporaneamente che sono in relazione tra di loro.

	Angularity		
Complexity	Rotondo	Rotondo - Spigoloso	Spigoloso
Alta	Altare	Cupola (esterna)	Campanile
Media	Cupola (interna) Volta	Abside Arco rampante	Gargoyle
Bassa	Affresco Lunetta	Vetrata	Lampadario Colonna

Tabella 9.3: 13 label per l'elemento di una architettura

9.4 Creazione dei modelli addestrati per la classificazione architetturale

A livello pratico, al fine di arrivare a svolgere la classificazione architetturale, è necessario come prima cosa di andare a **produrre un modello addestrato di machine learning**. L'addestramento del modello viene fatto utilizzando la **libreria Keras** (di interfacciamento a TensorFlow) in un codice scritto in linguaggio Python (con l'ausilio dell'IDE PyScripter).

Per l'addestramento sono stati considerati i dataset finali definiti precedentemente (sia per lo stile che per l'elemento architetturale) con una suddivisione 70% di training e 30% di validation (per la valutazione delle prestazioni del modello). Siccome entrambi i **dataset utilizzati sono sbilanciati**, questo potrebbe portare ad eventuali problematiche sia nella valutazione dell'accuracy che nella dominanza delle predizioni verso una singola classe. Per questo in fase di addestramento, viene utilizzata una funzione che determina un **peso di importanza da assegnare ad ogni classe in base al numero di immagini utilizzate**. Infatti viene assegnato un maggiore peso a classi che presentano poche immagini e un peso minore a classi che presentano molte immagini.

Per la valutazione dell'**accuracy finale del modello**, utilizzando l'accuracy del validation, è stato osservato il **valore numerico più basso del parametro *loss* del validation** (per una minore distanza tra predizioni e label vere delle classi), stabilendo così l'**epoca migliore delle 10 di addestramento**. Infatti le 10 epoche di addestramento definiscono 10 diversi insiemi di pesi che possono essere utilizzati per il modello (pesi utilizzati nella rete) e tra questi viene salvato, assieme al modello, solamente l'insieme di pesi dell'epoca migliore. Per il **salvataggio del modello con i relativi pesi viene utilizzato il formato SaveModel** di Keras [46], il quale crea una directory contenente

tutte le informazioni necessarie per il ripristino del modello con i relativi pesi. Il modello con i relativi pesi potevano anche essere salvati su un unico file (rispetto la directory con multipli file) con il formato hdf5 (file con estensione h5) però tale formato presenta problemi durante il ripristino di modelli che utilizzano una architettura composta non predefinita (custom, con relativo warning *CustomMaskWarning*).

Tra i vari layer, che compongono l'architettura utilizzata per il modello Keras di tipo custom, il più rilevante è quello di ***Mobilenet-v2***. *Mobilenet-v2* è una architettura, solitamente utilizzata per la classificazione delle immagini, che è stata **ottimizzata per l'uso su dispositivi mobili**. *Mobilenet-v2* consente anche di utilizzare il modello già addestrato con ImageNet (caricamento dei relativi pesi) ovvero un dataset composto da oltre 14 milioni di immagini, con più di 20000 label riguardanti il riconoscimento di oggetti. Mediante l'utilizzo di ***Mobilenet-v2*** **con i pesi di ImageNet** è stato possibile **raggiungere adeguati valori di accuracy nelle 2 tipologie di classificazioni architetturali** (che altrimenti sarebbero stati molto bassi). Questo perché il modello inizia il nuovo addestramento di tipo architetturale già disponendo di una base di conoscenza (distinzione fra una grande quantità di oggetti), molto utile soprattutto per affrontare il problema più complesso della classificazione dello stile architetturale.

Il modello TensorFlow addestrato non può essere utilizzato direttamente in Android ma richiede una **conversione preventiva di questo in modello TFLite**, il quale utilizza una rappresentazione a grafo. Il modello TFLite così ottenuto viene **esportato su un file .tflite** ed assieme ad un altro **file testuale (formato txt) con indicati i nomi delle label** (uno per riga), costituiscono gli elementi necessari per poter svolgere le predizioni di stile o elemento architetturale, con TensorFlow Lite, anche in Android.

Riassumendo l'insieme di passi necessari per arrivare ad ottenere i file richiesti per svolgere le predizioni architetturali anche in Android sono:

1. addestramento del modello (che include il layer *Mobilenet-v2* con i pesi di ImageNet) per 10 epoche con le immagini dei dataset architetturali (70% train e 30% validation) e relativo salvataggio su disco, con formato SaveModel, del modello con i migliori pesi;

2. ripristino del modello con i migliori pesi dalla directory SaveModel;

3. conversione del modello in TFLite;

4. esportazione del modello TFLite su file .tflite;

5. esportazione delle label del modello su file .txt.

9.5 Classificazione architetturale con i modelli addestrati in Android

Per la **gestione del processo di classificazione architetturale in Android (inferenza)**, a partire dai file creati sia dei modelli addestrati che delle label (file .tflite e .txt), è stata creata un'opportuna classe Java chiamata ***ArchitectureClassifierTFLite.java***. Nella classe sono stati definiti un insieme di metodi come la restituzione di un'istanza del classificatore TFLite a partire dal file .tflite, il caricamento delle label dal file .txt, la classificazione architettatele (stile o elemento) dell'immagine per restituire la relativa label e il rilascio di tutte le risorse allocate dal classificatore. Il file *native-lib.cpp* viene utilizzato per svolgere le chiamate JNI.

Nello specifico l'insieme di file utilizzati (in **coppie modello addestrato e label**) per svolgere le classificazioni architetturali in Android, suddivisi nelle 3 tipologie analizzate, sono:

- **Stile condizioni ideali:** *Stile_Arch_Wikim_Sbil.tflite* e *Label_Stile_Arch_Wikim_Sbil.txt*;
- **Stile condizioni generiche:** *Stile_Arch_WikimEGoogle_Sbil.tflite* e *Label_Stile_Arch_WikimEGoogle_Sbil.txt*;
- **Elementi:** *Elementi_Arch_Cult_Sbil.tflite* e *Label_Elementi_Arch_Cult_Sbil.txt*.

Android richiede che le immagini fornite per svolgere le inferenze siano oggetti *Bitmap* Java mentre i frame immagine sono oggetti C++ della classe *Mat* di OpenCV. Quindi è richiesto per le immagini di passare dalla *Mat* C++ a *Mat* Java e convertire la *Mat* Java in *Bitmap*, mediante il metodo *matToBitmap* definito da OpenCV. Inoltre la *Bitmap* viene ridimensionata alla dimensione necessaria in ingresso al classificatore (128X128) e l'immagine viene scritta in un buffer di memoria acceduto dal classificatore stesso (*ByteBuffer* per migliore le prestazioni delle inferenze).

Nelle impostazioni dell'app l'utente può scegliere se attivare (selezionando uno fra i 3 possibili tipi) o meno la funzionalità aggiuntiva per svolgere la classificazione architetturale nelle modalità Image-Sound. La label testuale determinata dalla classificazione architetturale viene mostrata graficamente nell'app mediante una casella di testo. Inoltre nell'app non viene implementata nessuna preventiva esclusione dell'immagine in caso questa non contenga nessun aspetto architettonico tra quelli possibili. Questo per non incidere eccessivamente sulle prestazioni complessive dell'applicazione mediante multipli classificatori in cascata, che svolgono quindi un'operazione di filtraggio dell'immagine.

9.6 Funzionalità componi musica

Per la **funzionalità componi musica**, che consente di generare un suono in modo visivo mediante i colori, viene utilizzato un principio simile a quello impiegato per la classificazione architetturale. Infatti anziché andare ad impiegare la label associata all'immagine

per stabilire quale delle 9 configurazioni utilizzare (per sapere come variare la nota/tonalità e l'ottava/luminosità), consente invece all'utente di definire una propria configurazione personalizzata attraverso la **scelta diretta della nota musicale e/o l'ottava preferita**. Questo comporta che la funzionalità componi musica **può essere attivata nelle impostazioni solamente se la classificazione architetturale è disattivata** (scelta manuale della configurazione rispetto ad una scelta automatica). Viene data anche la possibilità di decidere se utilizzare effettivamente il valore di nota musicale e/o ottava scelta oppure se applicare una forzatura degli elementi a tali valori (media pesata, con il maggior peso di circa il 70% per le componenti scelte).

Capitolo 10

Funzionalità aggiuntive nelle 2 modalità Sound-Image

10.1 Classificazione degli eventi audio con YAMNet

Le funzionalità aggiuntive per le 2 modalità Sound-Image svolgono delle classificazioni dell'audio registrato tramite microfono al fine di determinare l'evento audio presente e il relativo genere musicale. L'**evento audio** non è altro che un avvenimento che accade all'interno del suono tra un insieme di possibili elementi (come Discorso, Risata, Bussare, Tuono, Tosse, etc.), secondo la codifica delle label utilizzata dal classificatore YAMNet [47] [48].

YAMNet infatti è una rete neurale la cui architettura contiene il layer principale *Mobilenet-v1*, il cui modello è stato addestrato mediante il dataset AudioSet, composto da **521 diversi eventi audio**. Per la piattaforma Android viene rilasciato sia il file del modello YAMNet addestrato e convertito in formato tflite (*lite_model_yamnet_tflite_1.tflite*) sia il file testuale in formato txt con le relative 521 label (*label_lite_model_yamnet_tflite_1.txt*). Per facilitare la comprensione degli eventi audio è stata svolta una localizzazione da inglese ad italiano delle 521 label originali mediante una traduzione manuale (Tabella 10.1).

Il modello richiede come **input un array monodimensionale di campioni audio mono nel formato float** (range di valori da -1 a $+1$). L'input viene elaborato con una finestra di 0.96 secondi (frame, dove la frequenza di campionamento è 16 KHz), che viene fatta scorrere ogni 0.48 secondi, fintanto non si raggiunge la fine dell'array mediante multipli frame. Il modello restituisce in **output 3 tipi di informazioni** ovvero:

- ***Scores*:** predizione di ognuno dei frame rispetto le 521 label. La label finale associata ad ognuno dei frame viene ottenuta con una funzione di aggregazione di tipo max, andando ad osservare il nome della label, tra le 521, il cui valore numerico è più grande. Come label complessiva tra i multipli frame viene utilizzata quella più frequente;
- ***Embeddings*:** insieme di parametri numerici interni della rete, per ognuno dei frame, forniti all'ultimo layer dell'architettura per effettuare la classificazione finale.

Eventi audio collegati al genere musicale (153 label)						
Cantando	Coro	Jodel	Canto	Mantra	Bambino che canta	Canto sintetico
Rap	Canticchiare	Musica	Strumento musicale	Strumento a pizzico	Chitarra	Chitarra elettrica
Basso	Chitarra acustica	Chitarra d'acciaio	Tapping chitarra	Strimpellare	Banjo	Sitar
Mandolino	Cetra	Ukulele	Tastiera musicale	Pianoforte	Piano elettrico	Organo
Organo elettronico	Organo Hammond	Sintetizzatore	Campionatore	Clavicembalo	Percussioni	Batteria
Drum machine	Tamburo	Tamburo rullante	Rimshot	Rullo di tamburi	Grancassa	Timpano
Tabla	Piatto	Hi-hat	Wood block	Tamburello	Sonaglio (strumento)	Maracas
Gong	Campane tubolari	Percussioni di maglio	Xilofono	Glockenspiel	Vibrafono	Steelpan
Orchestra	Ottone	Corno francese	Tromba	Trombone	Strumento ad arco	Sezione corde
Violino	Pizzicato	Violoncello	Contrabbasso	Strumento a fiato	Flauto	Sassofono
Clarinetto	Arpa	Campana	Armonica (strumento)	Fisarmonica	Cornamuse	Didgeridoo
Shofar	Theremin	Singing bowl	Scratching (tecn. esecuzione)	Musica pop	Musica hip hop	Beatbox
Musica rock	Heavy metal	Punk rock	Grunge	Rock progressivo	Rock and roll	Rock psichedelico
Rhythm and blues	Musica soul	Reggae	Country	Musica swing	Bluegrass	Funk
Musica folk	Musica mediorientale	Jazz	Disco	Musica classica	Musica lirica	Musica elettronica
Musica house	Tecno	Dubstep	Drum and bass	Elettronica	Musica da ballo elettronica	Musica d'ambiente
Musica trance	Musica dell'America Latina	Salsa	Flamenco	Blues	Musica per bambini	Musica new age
Musica vocale	A capella	Musica d'Africa	Afrobeat	Musica cristiana	Musica gospel	Musica dell'Asia
Musica carnatica	Musica di Bollywood	Ska	Musica tradizionale	Musica indipendente	Canzone	Musica di sottofondo
Sigla musicale	Jingle (musica)	Colonna sonora	Ninna nanna	Musica per videogiochi	Musica di Natale	Musica dance
Musica per matrimoni	Musica felice	Musica triste	Musica tenera	Musica emozionante	Musica arrabbiata	Musica spaventosa
Suoneria	Unità di effetti	Effetto coro	Effetto sonoro	Televisione	Radio	

Eventi audio non collegati al genere musicale (368 label)						
Discorso	Bambino che parla	Conversazione	Narrazione	Balbettio	Sintetizzatore vocale	Gridare
Strillare	Verso	Urlo	Bambino che grida	Urlando	Sussurrando	Risata
Risata di bambino	Ridacchiare	Risatina	Ridere di pancia	Ridere sotto i baffi	Piangere singhiozzando	Bambino che piange
Piagnucolare	Piangere con lamento	Sospiro	Gemito	Grugnito	Fischiettio	Respirazione
Respirazione asmatica	Russare	Sussulto	Ansimare	Sniffa	Tosse	Schiarimento gola
Starnuto	Annusare	Correre	Mescola	Passi	Masticazione	Mordere
Gargarismi	Rimbombo di stomaco	Ruttare	Singhiozzo	Scoreggia	Mani	Schiocco delle dita
Battere le mani	Battito cardiaco	Soffio cardiaco	Tifo	Applausi	Chiacchiere	Folla
Rumore del parlato	Bambini che giocano	Animale	Animali domestici	Cane	Abbaiare	Guaire
Ululare	Bau bau	Ringhiando	Piagnucolare (cane)	Gatto	Fusa	Miagolio
Soffiare (gatto)	Miagolio (gatto amore)	Animali da fattoria	Cavallo	Zoccolio	Nitrire	Bovini
Muggire	Campanaccio	Maiale	Grugnire oink	Capra	Belare	Pecora
Pollame	Gallo	Chiocciare	Canto del gallo	Tacchino	Glu glu tacchino	Anatra
Qua qua anatra	Oca	Starnazzare	Animali selvaggi	Grandi felini ruggenti	Ruggito	Uccello
Canto degli uccelli	Cinguettare	Stridere uccello	Piccione	Tubare	Corvo	Gracchiare
Gufo	Bubolare	Sbattere le ali	Canidi	Roditori	Topo	Picchiettio
Insetto	Grillo	Zanzara	Mosca comune	Ronzare	Imenotteri (ape, vespa)	Rana
Gracidare	Serpente	Sonaglio	Vocalizzazione balena	Vento	Foglie frusciante	Rumore vento (microfono)
Temporale	Tuono	Acqua	Piovere	Goccia di pioggia	Pioggia in superficie	Ruscello
Cascata	Oceano	Onde surf	Vapore	Gorgoglio	Fuoco	Scoppiettare
Veicolo	Barca	Barca a vela	Barca a remi	Motoscafo	Nave	Veicolo a motore (stradale)
Macchina	Clacson	Strombazzare clacson	Allarme dell'auto	Alzacristalli elettrici	Slittamento	Stridore di pneumatici
Macchina di passaggio	Auto da corsa	Camion	Freno ad aria compressa	Clacson per camion	Segnale acustico inversione	Furgone dei gelati
Autobus	Veicolo di emergenza	Auto della polizia (sirena)	Ambulanza (sirena)	Camion dei pompieri (sirena)	Motociclo	Rumore del traffico
Trasporto ferroviario	Treno	Fischiare del treno	Clacson del treno	Vagone del treno	Ruote del treno stridono	Metropolitana
Aereo	Motore aeronautico	Motore a reazione	Elica	Elicottero	Aeroplano	Bicicletta
Skateboard	Motore	Motore leggero (alta freq.)	Trapano odontoiatrico	Taglia erba	Motosega	Motore medio (media freq.)
Motore pesante (bassa freq.)	Battito motore	Avviamento motore	Motore al minimo	Motore su di giri	Porta	Campanello di casa
Din Don	Porta scorrevole	Sbattere	Bussare	Picchiettare	Cigolare	Armadio aperto o chiuso
Cassetto aperto o chiuso	Piatti o pentole o padelle	Posate	Tritare (cibo)	Frittura (cibo)	Forno a microonde	Miscelatore
Rubinetto dell'acqua	Lavello (riemp. o lav.)	Vasca da bagno (riemp. o lav.)	Asciugacapelli	Sciacquone	Spazzolino	Spazzolino elettrico
Aspirapolvere	Cerniera (abbigliamento)	Chiavi che tintinnano	Moneta (che cade)	Forbici	Rasoio elettrico	Mischiare le carte
Digitando	Macchina da scrivere	Tastiera del computer	Scrivere	Allarme	Telefono	Campanello telefono suona
Selezione telefonica	Segnale di linea	Segnale di occupato	Sveglia	Sirena	Sirena protezione civile	Cicalino
Rilevatore di fumo	Allarme antincendio	Sirena da nebbia	Fischio	Fischietto a vapore	Meccanismi	Cricchetto
Orologio	Tic	Tic tac	Ingranaggi	Pulegge	Macchina da cucire	Ventilatore meccanico
Aria condizionata	Registratore di cassa	Stampante	Telecamera	Fotocamera reflex	Strumenti	Martello
Martello pneumatico	Segare	Limatura (raspa)	Levigatura	Utensile elettrico	Trapano	Esplosione
Sparo	Mitragliatrice	Scarica di fucileria	Fuoco di artiglieria	Pistola a cappuccio	Fuochi d'artificio	Petardo
Scoppiare	Eruzione	Botto	Legna	Taglio	Scheggia	Crepa
Vetro	Tintinnio	Frantumare	Liquido	Spruzzi	Sciabordare	Spremere
Gocciolare	Versare	Tracimare	Sgorgare	Riempire (con liquido)	Spray	Pompa (liquido)
Agitare	Bollire	Sonar	Freccia	Sfrecciare	Tonfo	Tonfo scontro
Accordatore elettronico	Rimbalzo del basket	Picchiare	Schiaffo	Colpire	Distruggere schianto	Rottura
Rimbalzare	Frusta	Lembo	Graffiare	Raschiare	Strofinare	Rotolare
Schiacciare	Spiegazzare	Strappare	Bip	Din	Ding	Fragore
Strillo	Scricchiolare	Fruscio	Ronzio	Sferragliare	Sfrigolare	Cliccare
Clicchettio	Rombo	Lasciar cadere	Scampanellare	Mormorare	Sibilo	Rimbalzo
Sgranocchiare	Silenzio	Onda simusoidale	Armonica	Tono cinguettio	Impulso	Dentro piccola stanza
Dentro ampia stanza o ingr.	Dentro spazio pubblico	Fuori urbano o artificiale	Fuori rurale o naturale	Riverbero	Eco	Rumore
Rumore ambientale	Interferenza	Ronzio di rete	Distorsione	Tono laterale	Cacofonia	Rumore bianco
Rumore rosa	Rumore intermittente	Vibrazione	Registrazione sul campo			

Tabella 10.1: 521 label degli eventi audio YAMNet

Svolgendo solamente inferenze con YAMNet, gli *embeddings* non vengono considerati in quanto utili solamente nel caso il modello viene addestrato nuovamente, per scopi specifici, a partire dalla relativa base di conoscenza di AudioSet;

- **Spettrogramma log mel:** spettrogramma dell'intero segnale audio, espresso come array di float, le cui frequenze sono convertite in scala mel (nome per esprimere il concetto di melodia) ovvero una scala logaritmica maggiormente adatta per la distinzione delle variazioni tra le pitch da parte dell'ascoltatore. In questo caso la

scala mel è stata suddivisa in 64 possibili intervalli (bins). Per l'ampiezza dello spettrogramma sono utilizzati i dB (log dello spettrogramma mel) e lo spettrogramma viene calcolato mediante una trasformata di Fourier.

10.2 Classificazione del genere musicale con GTZAN

GTZAN [49] è un dataset bilanciato costituito da 1000 traccie audio wav mono a 16 bit, ognuna con durata 30 secondi e frequenza di campionamento di 22050 Hz. Le 1000 traccie audio appartengono a **10 generi musicali** (Tabella 10.2) con 100 traccie audio per genere. Il dataset include anche dei file in formato csv contenenti un insieme di caratteristiche musicali (features) estratte sia per le 1000 traccie audio wav con durata 30 secondi, sia per 10000 traccie audio wav con durata 3 secondi, ottenute dalla suddivisione delle 1000 in 10 parti.

Genere musicale	
Blues	Classica
Country	Disco
Hip hop	Jazz
Metal	Pop
Reggae	Rock

Tabella 10.2: 10 label dei generi musicali GTZAN

Il dataset GTZAN è stato utilizzato durante lo svolgimento di un tirocinio, riguardante la valutazione della similarità tra brani musicali (in linguaggio Python), per addestrare opportuni modelli CNN con Keras (rete neurale che usa l'operazione matematica di convoluzione). Infatti esiste una **correlazione tra genere musicale di un brano musicale ed il relativo spettrogramma**. Per questo sono stati addestrati multipli modelli con Keras, utilizzando come input della rete lo spettrogramma log mel calcolato dai brani GTZAN, salvando i modelli con i migliori pesi su disco al fine di poter definire dei classificatori di genere musicale (Figura 10.1 e Figura 10.2). La creazione di multipli modelli è dovuto alla variazione di **formato in cui viene fornito lo spettrogramma log mel** (come array di valori numerici float in Figura 10.3 oppure tramite rappresentazione ad immagine bidimensionale a colori con mappa magma in Figura 10.4) oppure all'insieme di brani GTZAN utilizzati.

Oltre ai brani GTZAN con durata 30 o 3 secondi forniti dal dataset stesso, viene svolto un **augmentation** dei brani con durata 30 secondi per ottenere 5000 brani musicali. L'incremento della varietà dei dati disponibili consente di migliorare l'accuracy finale del modello in quanto l'analisi del dominio del problema risulta più completa e generica. L'augmentation non viene applicato alla durata 3 secondi in quanto con 50000 elementi vi è un eccessivo incremento dei tempi di addestramento del modello, diventando quindi

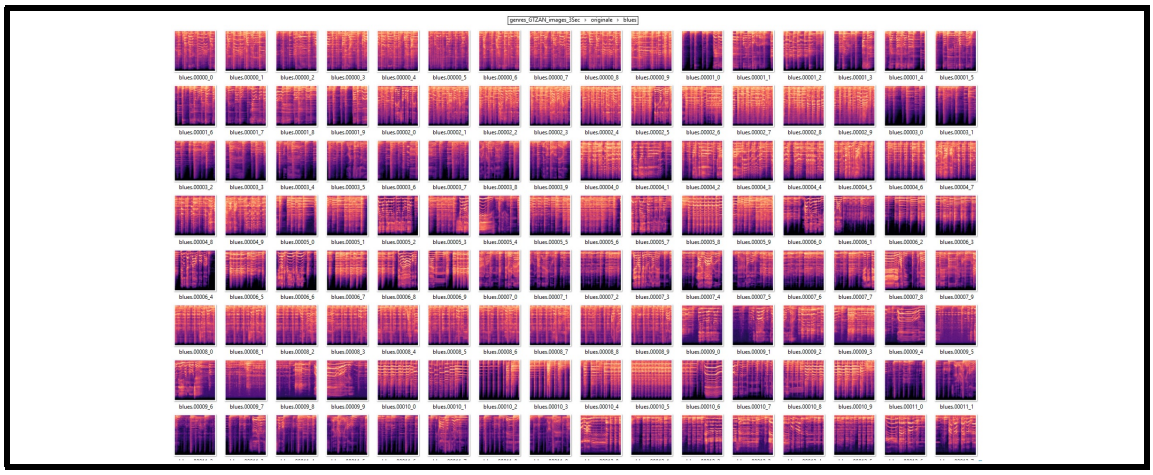

Figura 10.1: Esempi di spettrogramma immagine GTZAN

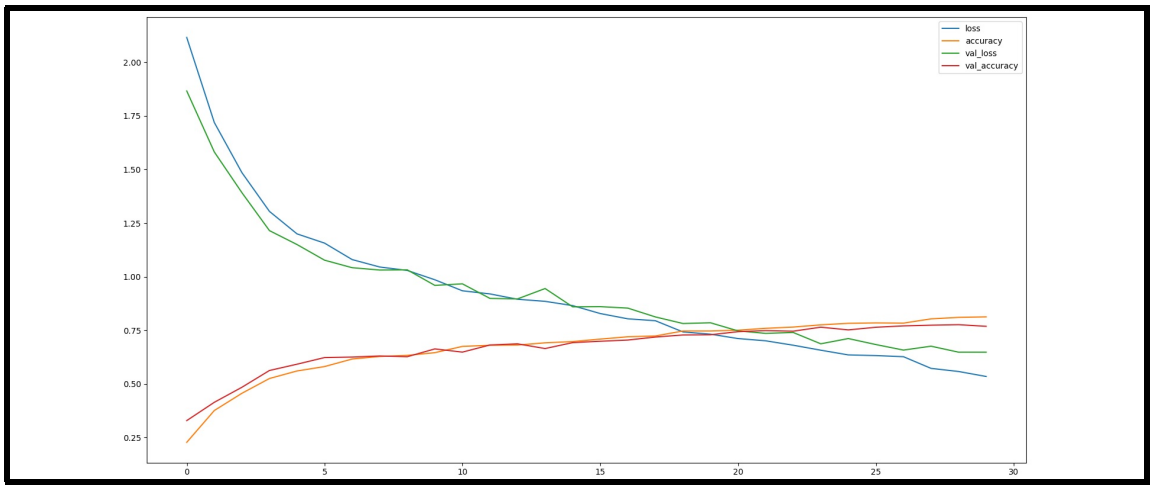

Figura 10.2: Grafico di addestramento spettrogramma immagine GTZAN 3 secondi

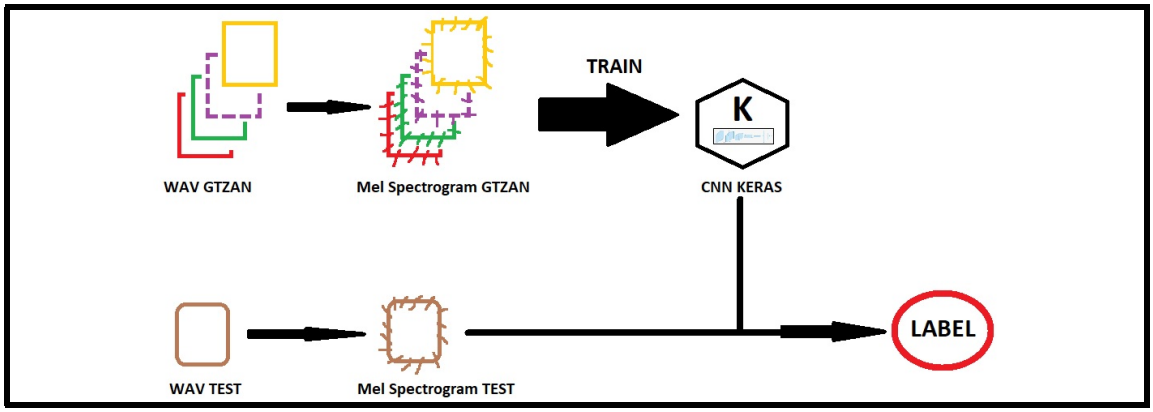

Figura 10.3: Schema di addestramento spettrogramma numerico GTZAN

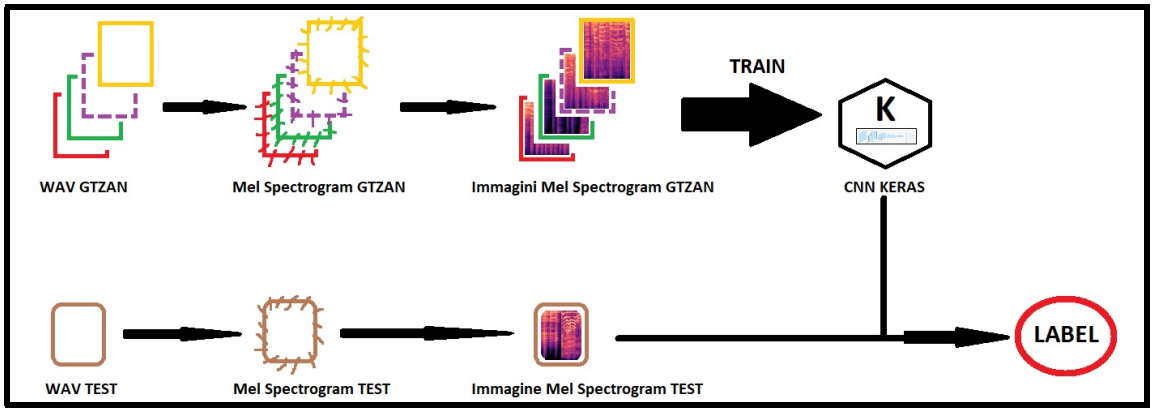

Figura 10.4: Schema di addestramento spettrogramma immagine GTZAN

difficilmente utilizzabile a lato pratico. Per l'**aumento dei brani musicali sono state applicate 4 diverse operazioni** (generazione di 4000 brani da sommare ai 1000 iniziali, utilizzando nelle funzioni opportuni parametri di input per non fare variare il genere musicale) ovvero l'aggiunta di rumore (audio di disturbo), la variazione delle pitch (mantenendo le caratteristiche strutturali), la dilatazione temporale (variazione della velocità) e slittamento temporale orizzontale dei campioni audio (Figura 10.5).

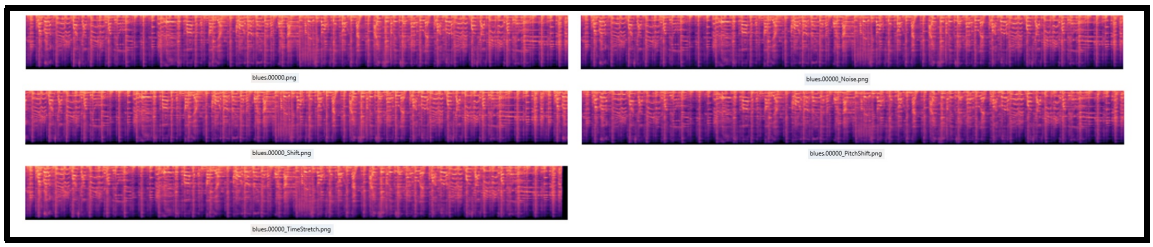

Figura 10.5: Esempio di augmentation dello spettrogramma immagine GTZAN

Viene utilizzata la seguente convenzione di denominazione per indicare le varie **tipologie di GTZAN** ovvero **durata breve** per la durata 3 secondi, **durata lunga** per la durata 30 secondi e **durata lunga accurata** per la durata 30 secondi con augmentation. La durata breve è preferibile sceglierla in caso di audio registrato con bassa durata (come 1 o 2 secondi) mentre si utilizza la durata lunga e la durata lunga accurata in caso di 3 o 4 o 5 secondi di audio (maggiore accuratezza di classificazione per il caso con augmentation). Si può comunque scegliere la durata breve anche per l'audio con durata di almeno 3 secondi o le 2 durate lunghe per l'audio con durata massima di 2 secondi, sapendo però che i risultati di classificazione ottenuti non sono ottimali.

Anche in questo caso per l'utilizzo dei modelli addestrati GTZAN in Android è necessario svolgere una conversione di questi in modello TFLite e disporre dei file testuali delle label. In particolare tra tutti i modelli addestrati nel tirocinio ne vengono selezionati 5 per l'utilizzo in Android, i cui file file utilizzati in **coppie modello TFLite e label** sono:

- **Spettrogramma immagine GTZAN 3 secondi (durata breve):** *Mel_Spectr_Imm_GTZAN_3sec_NoBatchNorm.tflite* e *Label_Mel_Spectr_Imm_GTZAN_3sec_NoBatchNorm.txt*, l'accuracy del validation è circa il 77%;
- **Spettrogramma immagine GTZAN 30 secondi (durata lunga):** *Mel_Spectr_Imm_GTZAN_30sec_NoBatchNorm.tflite* e *Label_Mel_Spectr_Imm_GTZAN_30sec_NoBatchNorm.txt*, l'accuracy del validation è circa il 60%;
- **Spettrogramma immagine GTZAN 30 secondi con augmentation (durata lunga accurata):** *Mel_Spectr_Imm_GTZAN_30sec_augment_NoBatchNorm.tflite* e *Label_Mel_Spectr_Imm_GTZAN_30sec_augment_NoBatchNorm.txt*, l'accuracy del validation è circa il 87%;
- **Spettrogramma numerico GTZAN 3 secondi (durata breve):** *Mel_Spectr_Audio_GTZAN_3sec.tflite* e *Label_Mel_Spectr_Audio_GTZAN_3sec.txt*, l'accuracy del validation è circa il 69%;
- **Spettrogramma numerico GTZAN 30 secondi (durata lunga):** *Mel_Spectr_Audio_GTZAN_30sec_poolSizeCresc.tflite* e *Label_Mel_Spectr_Audio_GTZAN_30sec_poolSizeCresc.txt*, l'accuracy del validation è circa il 41%.

10.3 Wrapper di interfacciamento tra spettrogramma YAMNet e GTZAN

Disponendo sia del modello YAMNet per ottenere lo spettrogramma log mel, a partire dall'audio registrato tramite microfono, sia dei modelli GTZAN per ottenere il genere musicale a partire dallo spettrogramma log mel, quello che serve è un wrapper di interfacciamento tra questi per i dati comuni elaborati. Infatti il **wrapper di interfacciamento** permette di definire un metodo di **conversione della dimensionalità o formato dello spettrogramma log mel di YAMNet** in base alla struttura dei **dati di ingresso richiesti da parte del modello addestrato GTZAN** selezionato. Per questo vengono definiti **2 diversi approcci** a seconda se viene utilizzato un modello GTZAN che richiede lo spettrogramma log mel di tipo numerico oppure rappresentato come immagine.

Nell'approccio che impiega lo **spettrogramma log mel di tipo numerico** viene svolta una conversione delle dimensionalità dei dati principalmente mediante un **sottocampionamento (downsampling)** delle informazioni disponibili nello spettrogramma di YAMNet con eventuale replicazione (per il raggiungimento degli elementi necessari), dove il numero di elementi selezionati dipende dalla durata GTZAN scelta (3 o 30 secondi). Le informazioni disponibili nello spettrogramma di YAMNet sono maggiori all'aumentare dei secondi di registrazione dell'audio scelti e la selezione di questi elementi viene fatta in modo casuale (random con valore seed stabilito, per consentire la replicazione dei risultati ottenuti). Il sottocampionamento consente di **gestire, le diverse frequenze di campionamento con cui sono stati acquisiti i dati, mediante un'unica frequenza** che è quella necessaria a GTZAN.

Per l'approccio che impiega lo **spettrogramma log mel rappresentato come immagine** viene svolta una conversione di formato degli elementi float dello spettrogramma per ottenere dei pixel colore RGB. Come prima cosa viene fatta una normalizzazione nel range 0-255 dei valori float (discretizzazione dei valori). Il range 0-255 viene suddiviso in **5/10/20 intervalli uniformi (livelli)**, andando ad associare ad **ogni intervallo un valore RGB**. I valori RGB impiegati sono stati stabiliti manualmente andando a considerare la **mappa colore magma**, la stessa utilizzata in fase di addestramento per le immagini dello spettrogramma GTZAN. Questa mappa colore presenta sfumature di colore blu-viola per bassi valori della scala, rosso-arancione per medi valori e giallo per alti valori (Figura 10.6). Variando gli intervalli utilizzati per la mappa colore tra 5/10/20 (Tabella 10.3) si varia la risoluzione della mappa magma originale tra 0.2/0.1/0.05, dove l'impiego di più intervalli comporta l'utilizzo di un maggior numero colori quindi migliore precisione di rappresentazione dell'immagine finale. Inoltre l'immagine dello spettrogramma log mel così ottenuta viene ridimensionata con le dimensioni richieste in input dal modello GTZAN (150X150 per il GTZAN 3 secondi oppure 370X50 per il GTZAN di 30 secondi con/senza augmentation).

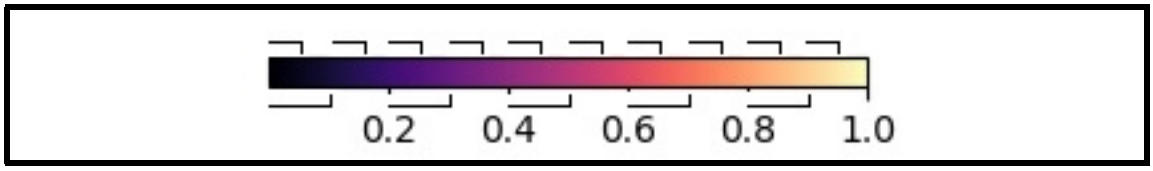

Figura 10.6: Codifica dei colori della mappa magma

Range numerico intervallo	Codifica 5 livelli (Colore, RGB, Num. Intervallo)	Codifica 10 livelli (Colore, RGB, Num. Intervallo)	Codifica 20 livelli (Colore, RGB, Num. Intervallo)
0-11	, (R:61, G:15, B:113), 0	, (R:24, G:15, B:61), 0	, (R:8, G:7, B:30), 0
12-24	, (R:61, G:15, B:113), 0	, (R:24, G:15, B:61), 0	, (R:24, G:15, B:61), 1
25-37	, (R:61, G:15, B:113), 0	, (R:61, G:15, B:113), 1	, (R:44, G:17, B:95), 2
38-50	, (R:61, G:15, B:113), 0	, (R:61, G:15, B:113), 1	, (R:61, G:15, B:113), 3
51-62	, (R:134, G:39, B:121), 1	, (R:104, G:28, B:129), 2	, (R:82, G:19, B:124), 4
63-75	, (R:134, G:39, B:121), 1	, (R:104, G:28, B:129), 2	, (R:104, G:28, B:129), 5
76-88	, (R:134, G:39, B:121), 1	, (R:134, G:39, B:121), 3	, (R:123, G:35, B:130), 6
89-101	, (R:134, G:39, B:121), 1	, (R:134, G:39, B:121), 3	, (R:134, G:39, B:121), 7
102-113	, (R:224, G:76, B:103), 2	, (R:186, G:56, B:120), 4	, (R:163, G:48, B:126), 8
114-126	, (R:224, G:76, B:103), 2	, (R:186, G:56, B:120), 4	, (R:186, G:56, B:120), 9
127-139	, (R:224, G:76, B:103), 2	, (R:224, G:76, B:103), 5	, (R:207, G:64, B:112), 10
140-152	, (R:224, G:76, B:103), 2	, (R:224, G:76, B:103), 5	, (R:224, G:76, B:103), 11
153-164	, (R:254, G:161, B:110), 3	, (R:247, G:114, B:92), 6	, (R:239, G:93, B:94), 12
165-177	, (R:254, G:161, B:110), 3	, (R:247, G:114, B:92), 6	, (R:247, G:114, B:92), 13
178-190	, (R:254, G:161, B:110), 3	, (R:254, G:161, B:110), 7	, (R:252, G:140, B:99), 14
191-203	, (R:254, G:161, B:110), 3	, (R:254, G:161, B:110), 7	, (R:254, G:161, B:110), 15
204-215	, (R:252, G:251, B:189), 4	, (R:254, G:209, B:148), 8	, (R:254, G:187, B:129), 16
216-228	, (R:252, G:251, B:189), 4	, (R:254, G:209, B:148), 8	, (R:254, G:209, B:148), 17
229-241	, (R:252, G:251, B:189), 4	, (R:252, G:251, B:189), 9	, (R:253, G:233, B:170), 18
242-255	, (R:252, G:251, B:189), 4	, (R:252, G:251, B:189), 9	, (R:252, G:251, B:189), 19

Tabella 10.3: Suddivisione della mappa magma in intervalli

10.4 Processo Sound-Image con classificazione audio

Anche in questo caso, come nella classificazione architetturale, per la gestione del processo di classificazione dell'audio in Android è stata creata un'opportuna classe Java chiamata ***SoundClassifierTFLite.java***. Nella classe sono stati definiti un insieme di metodi come la restituzione di un'istanza per i classificatori evento/genere di YAMNet/GTZAN a partire dai relativi file .tflite, il caricamento delle label dai file .txt, applicazione del wrapper di interfacciamento di tipo numerico o immagine, la classificazione dell'evento audio YAMNet, la classificazione del genere musicale GTZAN e il rilascio di tutte le risorse allocate dai classificatori.

Riassumendo le funzionalità aggiuntive per le 2 modalità Sound-Image richiedono la seguente sequenza:

1. classificazione dell'audio registrato tramite microfono mediante YAMNet al fine di determinare l'evento audio presente (tra i 521) e lo spettrogramma log mel (non si considerano gli *embeddings*);

2. applicazione del wrapper di interfacciamento di tipo numerico o immagine (a seconda del tipo GTZAN scelto) allo spettrogramma log mel;

3. classificazione dello spettrogramma con wrapper mediante GTZAN al fine di determinare il genere musicale presente (tra i 10).

Viene data anche la possibilità all'utente di scegliere nelle impostazioni se **non svolgere la classificazione di genere musicale**, determinando solamente l'evento audio oppure se svolgere la classificazione di genere solamente per **eventi opportuni di YAMNet**. Gli eventi opportuni di YAMNet sono quegli **eventi in cui risulta logico determinare il genere musicale** (come Canto, Chitarra, Pianoforte rispetto ad Abbaiare, Russare, Piovere) e sono stati selezionati **153 eventi su 521**. Se l'utente decide di svolgere la classificazione di genere musicale può scegliere anche l'algoritmo da utilizzare tra i 5 possibili (collegati ai 5 modelli addestrati GTZAN, con 3 per lo spettrogramma di tipo immagine e 2 per il tipo numerico). Sia la label della classificazione dell'evento sia quella del genere determinate vengono mostrate nell'app, assieme allo spettrogramma rappresentato come immagine (scegliendo se rappresentarlo con 5, 10 o 20 livelli).

Capitolo 11

Esecuzione dell'app ImageSound

11.1 Installazione dell'app

Per l'**installazione dell'applicazione** Android ImageSound sono necessari i seguenti passi:

1. abilitare in *Impostazioni-Sicurezza* del dispositivo Andoid la **possibilità di installare applicazioni da fonti sconosciute** (il percorso definito può anche variare a seconda del dispositivo specifico utilizzato, come ad esempio *Impostazioni avanzate-Sicurezza*);
2. mediante un qualsiasi file manager accedere al file dell'app con estensione .apk ed aprirlo;
3. accettare alla domanda di conferma per l'installazione dell'applicativo al fine di completare l'istallazione del file .apk (in modo facoltativo si può disabilitare la possibilità di installare applicazioni da fonti sconosciute, abilitata inizialmente nelle impostazioni);
4. premere sull'icona dell'app per aprire l'applicazione (Figura 11.1);
5. confermare, se richiesto, i **permessi per utilizzare la fotocamera e il microfono** del dispositivo (da Android 6.0 in su).

Figura 11.1: Icona dell'app ImageSound

11.2 Hompepage dell'app

Dalla **pagina principale dell'app** (Figura 11.2 e Figura 11.3) vi sono principalmente 6 elementi grafici (alcuni dei quali possono essere premuti al fine di svolgere opportune azioni) ovvero:

- ***Impostazioni* (pulsante):** apertura della schermata delle impostazioni per la regolazione dei parametri di elaborazione;
- **Icona modalità elaborazione (immagine):** variazione tra le 4 modalità di elaborazione attraverso la scelta in un relativo menu (Figura 11.4);
- ***Start* (pulsante):** avvia l'elaborazione della modalità specificata;
- ***Stop* (pulsante):** interrompe l'elaborazione della modalità specificata;
- ***Senti audio* (pulsante):** disponibile solamente nelle 2 modalità non live per risentire l'audio prodotto/acquisito con l'elaborazione;
- **Informazioni elaborazione (casella di testo):** vengono mostrate le informazioni aggiuntive prodotte con l'elaborazione come le label di classificazione, la qualità percentuale di elaborazione nelle 2 modalità live e la visualizzazione dello spettrogramma immagine (come sfondo della casella di testo).

L'applicazione può essere utilizzata sia con un'**orientazione verticale o orizzontale del dispositivo** mediante l'impiego di 2 layout diversi (disposizione degli elementi grafici), il cui tipo specifico viene scelto in base all'orientazione corrente del dispositivo.

Figura 11.2: Hompepage delle 2 modalità Image-Sound

Figura 11.3: Hompepage delle 2 modalità Sound-Image

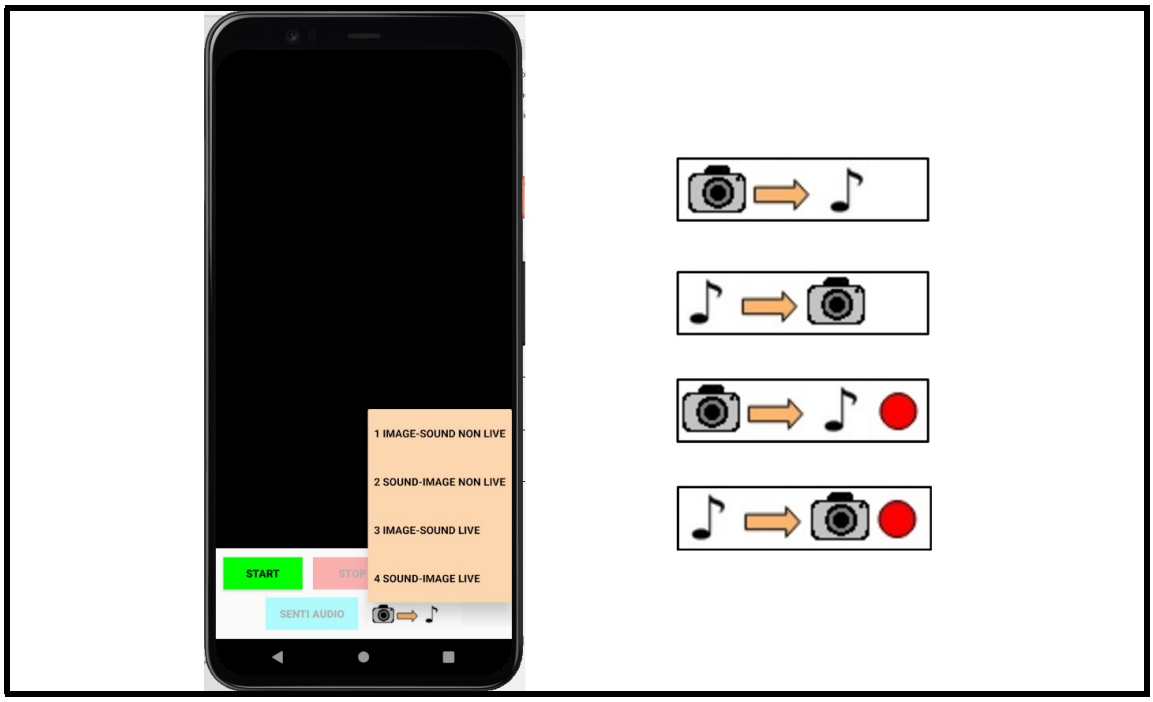

Figura 11.4: Scelta fra le 4 modalità nella hompepage

11.3 Impostazioni dell'app

La schermata delle **impostazioni dell'app** è stata suddivisa concettualmente in 5 parti, ognuna contraddistinta con un diverso colore ed una descrizione iniziale (per facilitare la scelta dei parametri all'utente) ovvero:

- **Info ImageSound:** unica parte senza scelta di parametri, per la descrizione in breve dell'intero funzionamento dell'app (Figura 11.5);
- **Impostazioni generali:** parametri che ricorrono in tutte 4 le modalità di Image-Sound, come la scelta della fotocamera posteriore o anteriore del dispositivo (Figura 11.6);
- **Classificazione architetturale:** parametri riguardanti la classificazione architetturale di stile o elemento che vengono utilizzati solamente nelle 2 modalità Image-Sound (Figura 11.7);
- **Componi musica:** parametri per la scelta della nota e/o ottava tramite assegnazione/media pesata con cui andare a produrre l'audio nelle 2 modalità Image-Sound (Figura 11.8);
- **Classificazione eventi audio:** parametri riguardanti la classificazione degli eventi audio e del relativo genere musicale nelle 2 modalità Sound-Image (Figura 11.9).

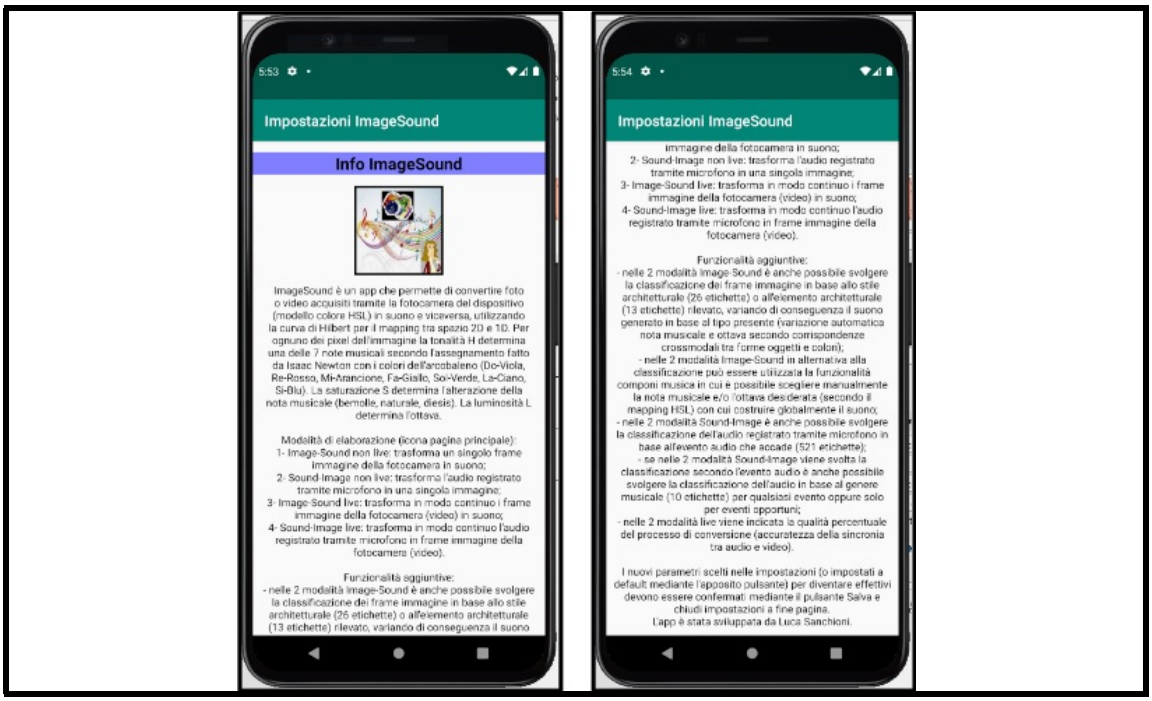

Figura 11.5: Impostazioni dell'app: info ImageSound

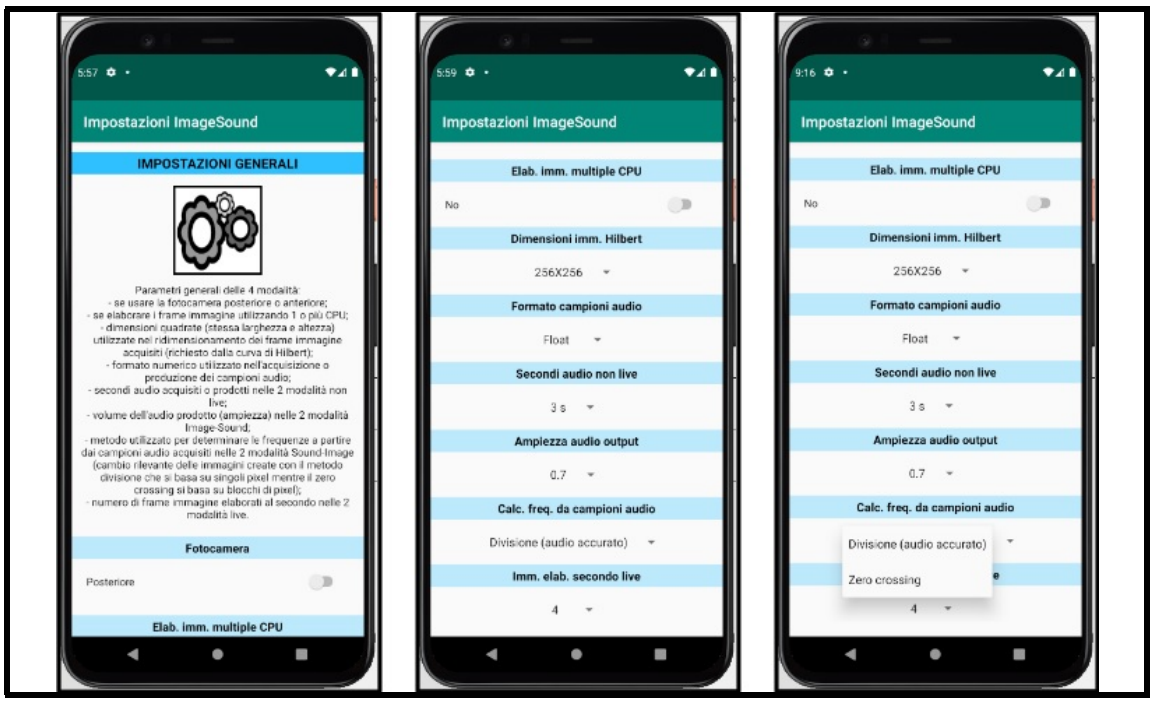

Figura 11.6: Impostazioni dell'app: impostazioni generali

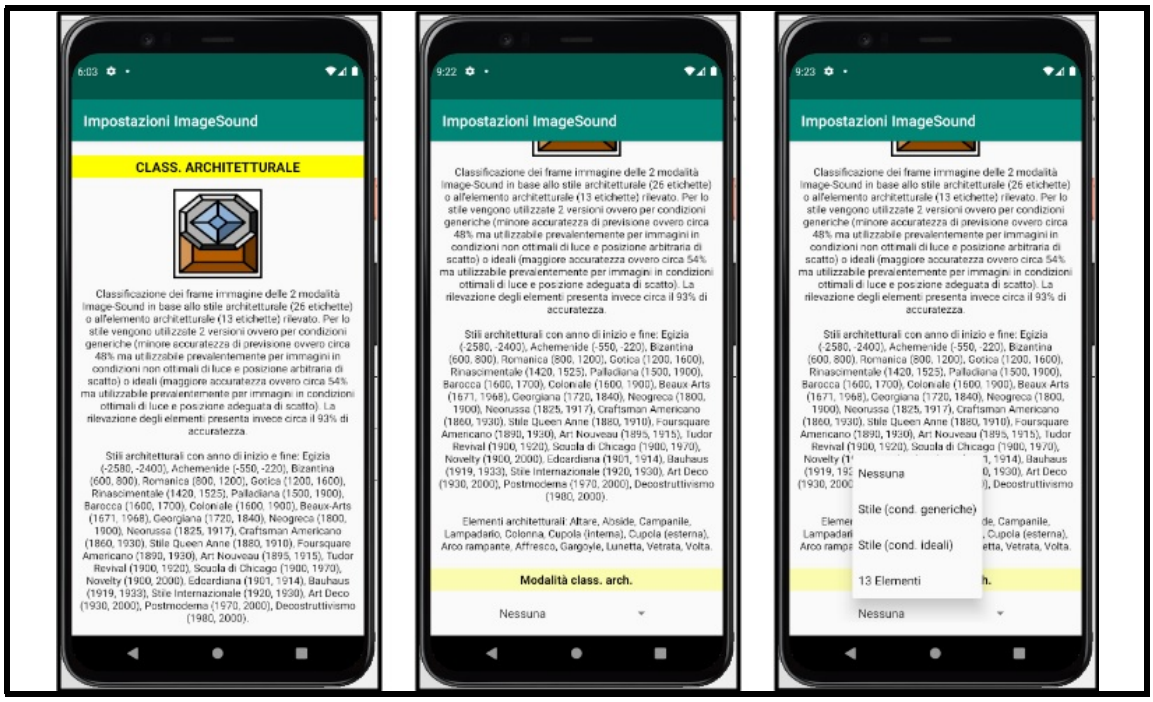

Figura 11.7: Impostazioni dell'app: classificazione architetturale

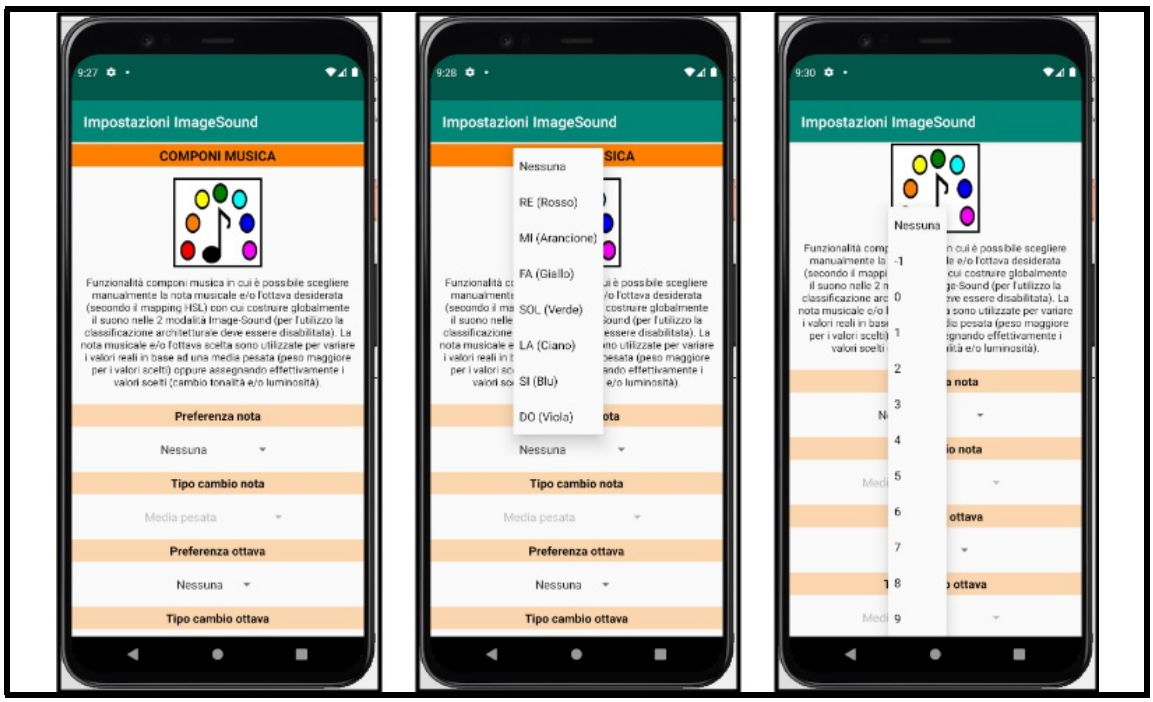

Figura 11.8: Impostazioni dell'app: componi musica

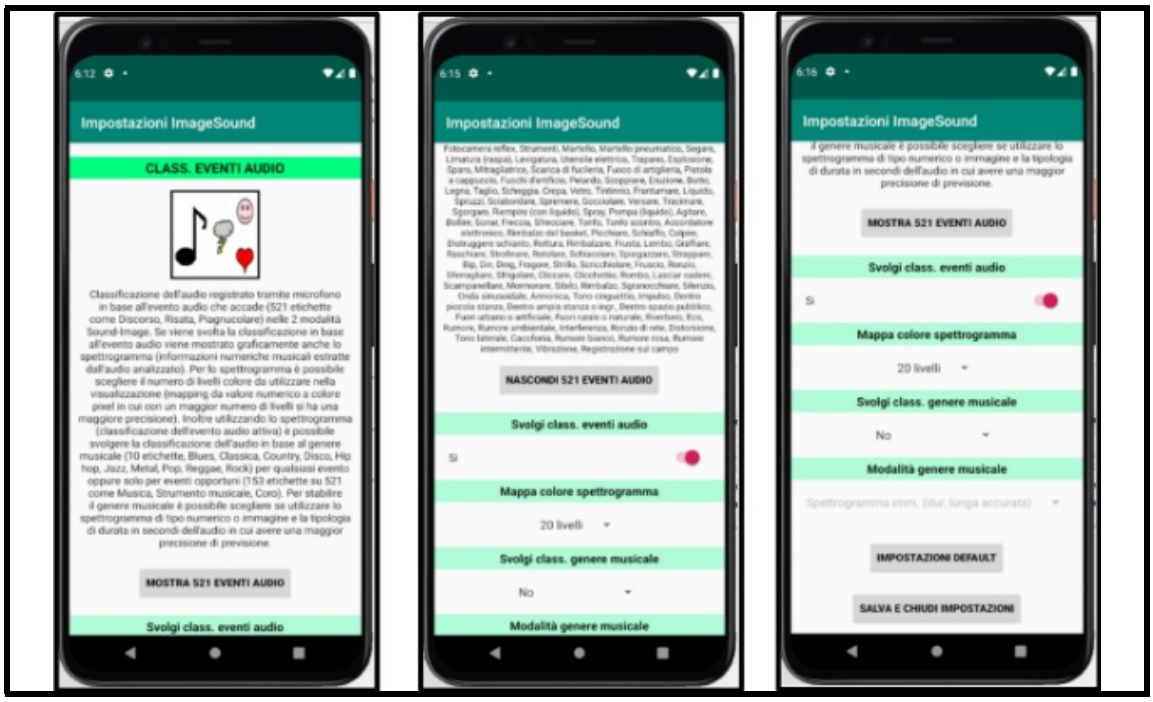

Figura 11.9: Impostazioni dell'app: classificazione eventi audio

Nelle impostazioni, dopo aver scelto i nuovi parametri, questi devono essere confermati mediante il pulsante ***Salva e chiudi impostazioni*** a fine pagina. In questo modo i parametri salvati vengono mantenuti anche in riaperture successive dell'app. L'utente dalle impostazioni può ritornare, mediante il tasto back del dispositivo, alla pagina principale dell'applicazione. Premendo il pulsante ***Impostazioni default*** si possono riportare globalmente tutti i parametri di elaborazione ai valori predefiniti di default.

11.4 Esecuzione della modalità Image-Sound non live

Dopo aver scattato la foto è possibile ascoltare il relativo audio prodotto (Figura 11.10 e Figura 11.11). Si può far dipendere il suono prodotto dalla classificazione dello stile o elemento architetturale (visualizzazione testuale della label determinata) oppure in modo creativo mediante la composizione della musica (scelta della nota e/o ottava).

11.5 Esecuzione della modalità Sound-Image non live

L'audio registrato tramite microfono, con durata limitata in secondi, viene trasformato in immagine (Figura 11.12 e Figura 11.13). Si può anche stabilire l'evento audio presente nell'audio registrato ed il relativo genere musicale (visualizzazione testuale delle label e dello spettrogramma come immagine).

11.6 Esecuzione della modalità Image-Sound live

Il video (multipli frame al secondo) viene convertito in suono fintanto non si decide di interrompe l'elaborazione (Figura 11.14 e Figura 11.15). Si può far dipendere il suono prodotto dalla classificazione dello stile o elemento architetturale oppure in modo totalmente creativo mediante la composizione della musica (scelta della nota e/o ottava). Viene anche mostrata la qualità percentuale dell'elaborazione.

11.7 Esecuzione della modalità Sound-Image live

L'audio registrato viene convertito in video (multipli frame al secondo) fintanto non si decide di interrompe l'elaborazione (Figura 11.16 e Figura 11.17). Si può anche stabilire l'evento audio presente nell'audio registrato ed il relativo genere musicale. Viene anche mostrata la qualità percentuale dell'elaborazione.

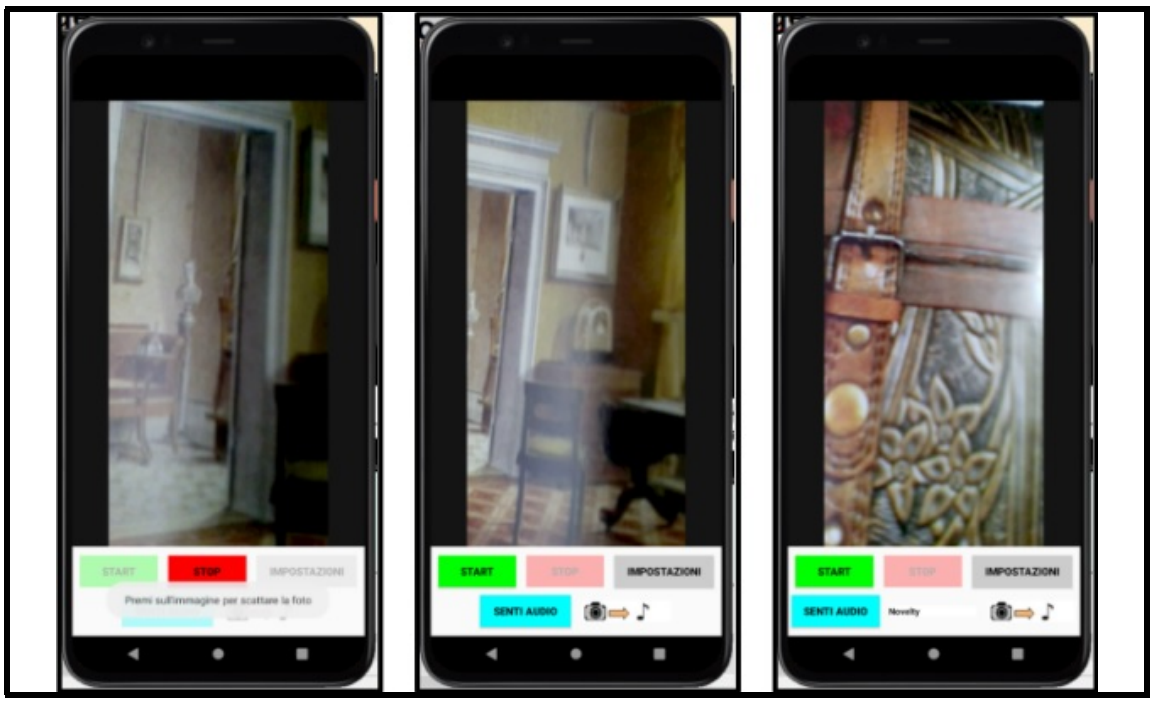

Figura 11.10: Esempio 1 di output della modalità Image-Sound non live

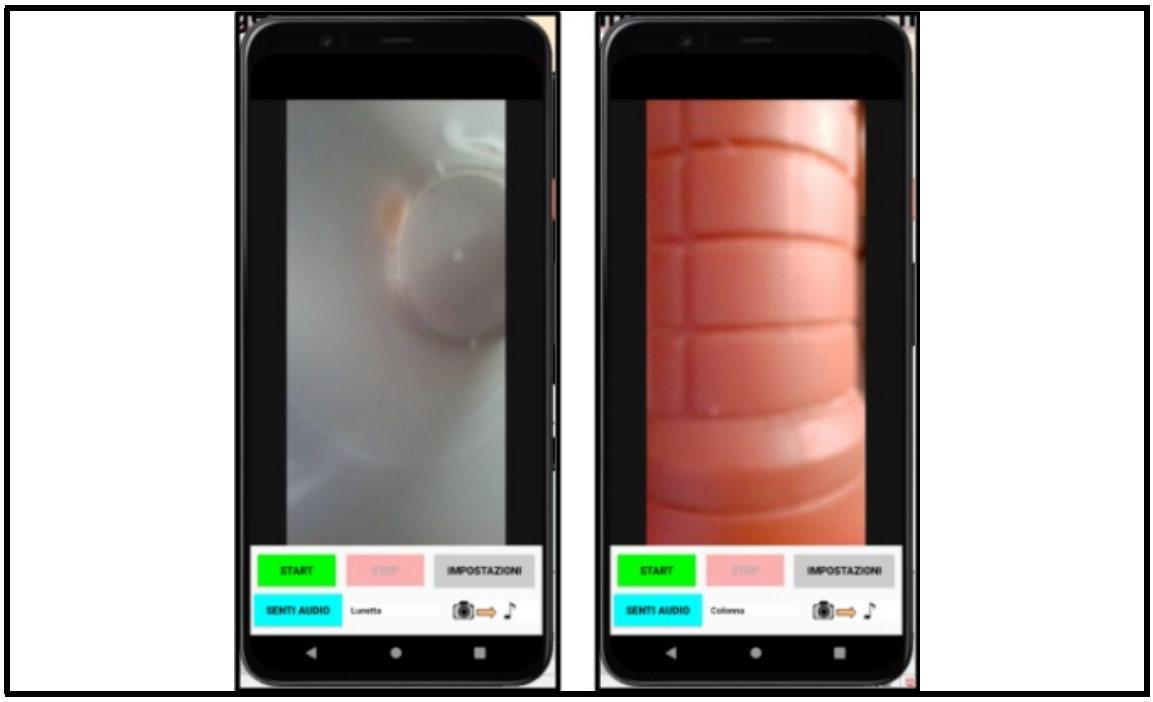

Figura 11.11: Esempio 2 di output della modalità Image-Sound non live

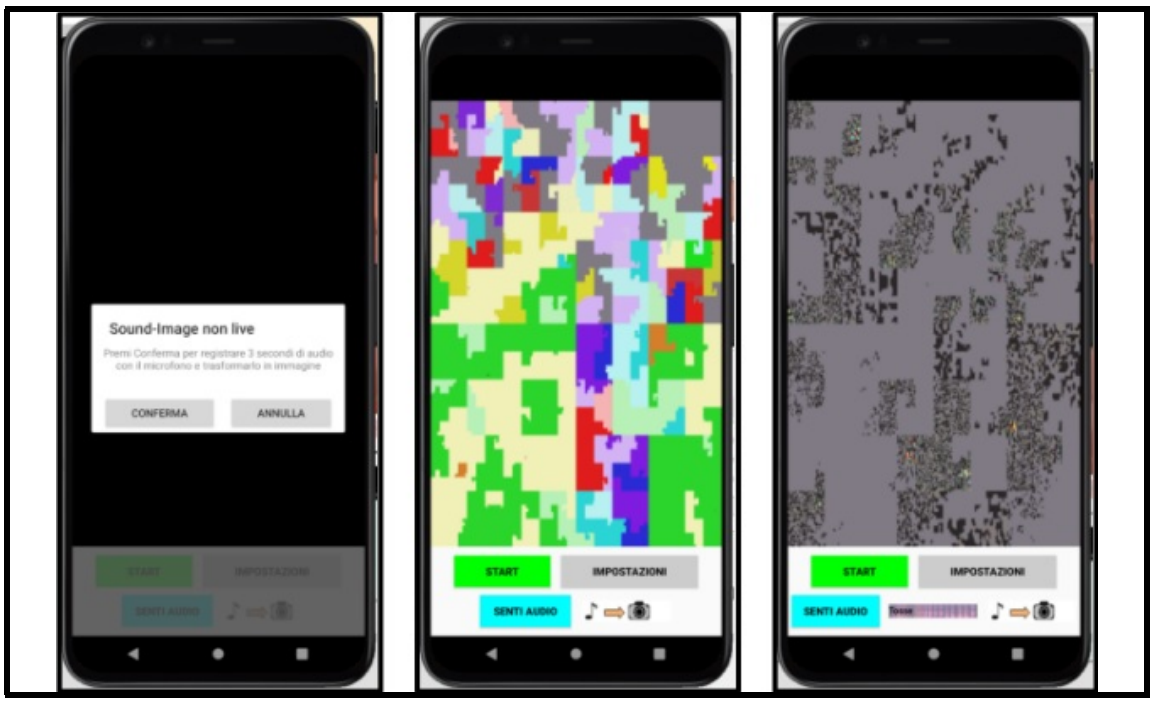

Figura 11.12: Esempio 1 di output della modalità Sound-Image non live

Figura 11.13: Esempio 2 di output della modalità Sound-Image non live

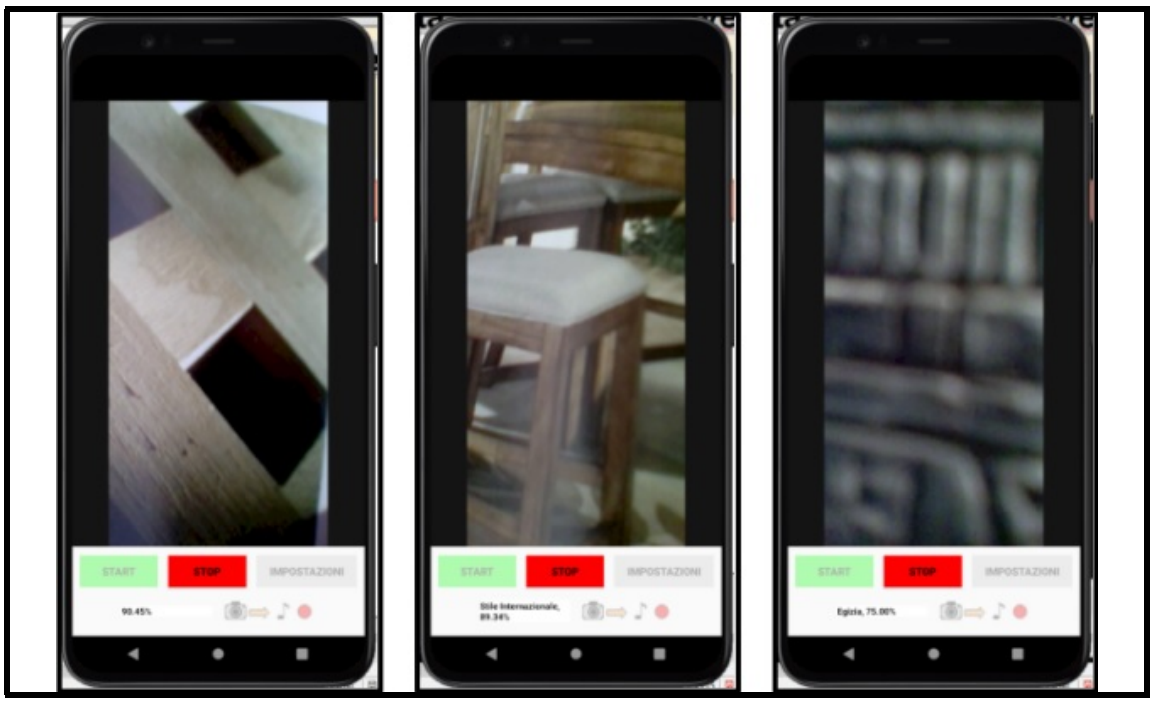

Figura 11.14: Esempio 1 di output della modalità Image-Sound live

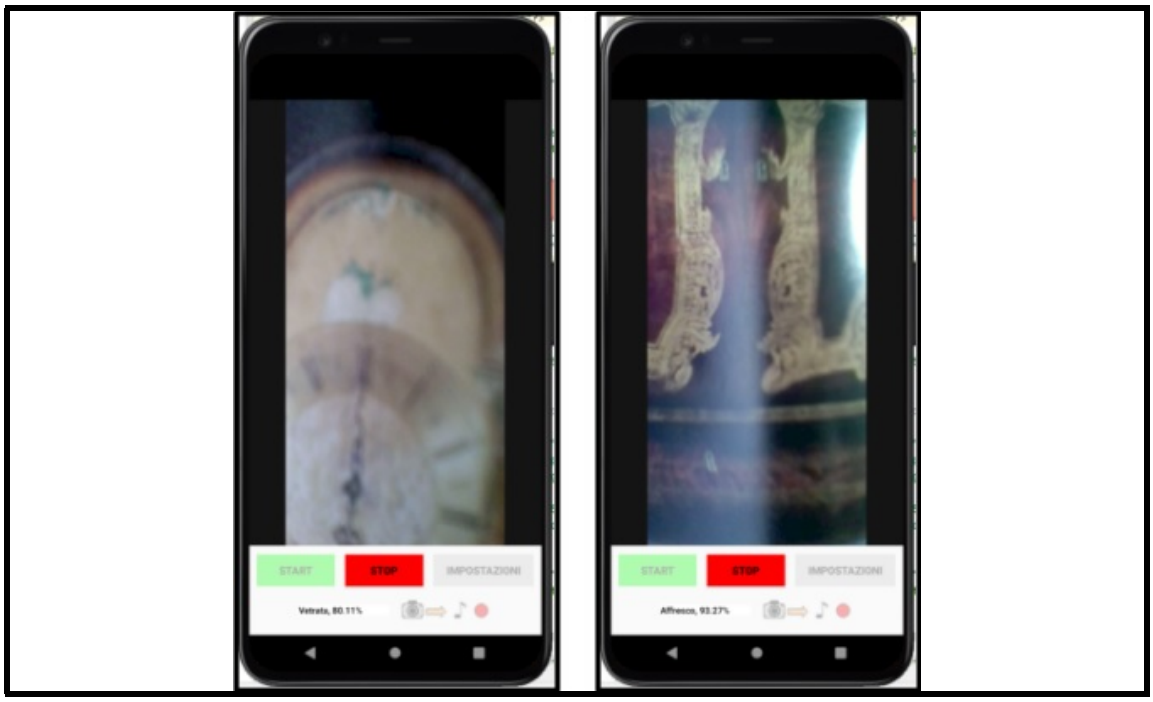

Figura 11.15: Esempio 2 di output della modalità Image-Sound live

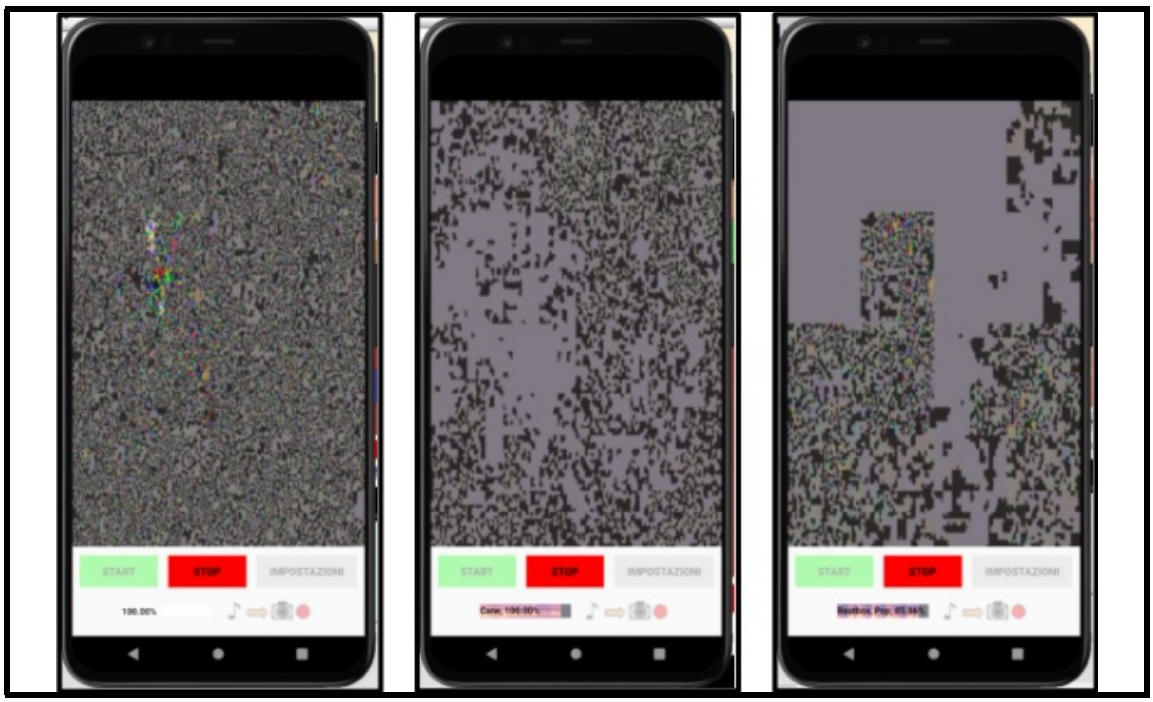

Figura 11.16: Esempio 1 di output della modalità Sound-Image live

Figura 11.17: Esempio 2 di output della modalità Sound-Image live

Capitolo 12

Considerazioni finali

12.1 Conclusioni

Per la generazione di informazioni uditive a partire da quelle visive o viceversa è necessario stabilire una relazione bidirezionale diretta opportuna (**mapping**) fra queste due tipologie eterogenee di dati. L'uso simultaneo di informazioni eterogenee, associate ad un medesimo concetto, consente di osservare un elemento comune sotto un altro punto di vista, permettendo di valorizzare maggiormente il momento temporale considerato.

L'applicazione Android ImageSound definisce uno degli infiniti mapping che potrebbero essere utilizzati al fine di unire il mondo delle immagini (foto scattata) con i suoni (audio registrato). Il mapping viene definito a partire dal concetto chiave della **relazione di Newton**, il quale stabilisce l'associazione tra i 7 colori dell'arcobaleno e le 7 classi di note musicali. Mediante la relazione di Newton viene definita la corrispondenza bidirezionale tra immagini nel modello colore HSL (modello adeguato per le tonalità) e le frequenze delle note musicali. Per il passaggio da spazio 2D dell'immagine a 1D del suono e viceversa viene utilizzata la **curva di Hilbert**.

Questi ultimi elementi portano a definire il **processo Image-Sound** per il passaggio da immagine a suono e il **processo Sound-Image** per l'operazione inversa (da suono ad immagine). Il processo Sound-Image anche se utilizza, in modo inverso, gli stessi passi concettuali di quello Image-Sound, presenta però un componente aggiuntivo per la **conversione da campioni audio registrati a frequenze** delle note musicali. Sia per il processo Image-Sound sia per quello Sound-Image vengono definite 2 metrologie chiamate **non live o live** al fine di considerare nella conversione foto o video (multiple immagini al secondo). Nell'applicativo vengono quindi utilizzate **4 modalità di elaborazione**, le quali possono essere scelte da parte dell'utente.

Per l'implementazione software nell'app dell'acquisizione dei frame immagine, tramite **fotocamera del dispositivo**, viene utilizzata la preview della libreria C++ **OpenCV**. Alla preview sono state apportate modifiche rispetto alla sua originale definizione al fine di migliorare l'accessibilità (orientazione) e la compatibilità con i dispositivi Android (dimensioni ottimali). Invece per la **gestione dell'audio**, registrato in input o riprodotto in output, viene utilizzata la libreria C++ **Oboe**.

Vengono anche definite **funzionalità aggiuntive** che permettono, nel processo Image-Sound, di svolgere la **classificazione dell'immagine in base allo stile o elemento architetturale** e utilizzare questa label nella generazione del suono. Invece le funzionalità aggiuntive nel processo Sound-Image consentono di svolgere una **classificazione del suono registrato in base all'evento audio presente e al relativo genere musicale**.

Nelle **impostazioni dell'app** l'utente può scegliere i principali parametri utilizzati nelle 4 modalità di elaborazione. Per facilitare la scelta dei parametri, le impostazioni sono state suddivise concettualmente in sezioni, ognuna accompagnata da una relativa descrizione.

Con la realizzazione dell'app Android ImageSound purtroppo si è messo in evidenza che lo studio per la definizione di **algoritmi, in grado di mettere in relazione informazioni uditive e visive, risulta un argomento poco approfondito** rispetto alla grande importanza che possiede. Questo è dovuto al fatto che momentaneamente tale settore appare come astratto e poco definito a causa degli infiniti mapping che possono essere utilizzati. Nel futuro, attraverso la consolidazione formale e pratica di alcuni concetti principali guida, la valutazione sul settore cambierà totalmente andando ad assumere una rilevante importanza e permettendo inoltre la nascita, in multipli contesti, di nuove tecniche totalmente innovative (rispetto ai giorni nostri).

12.2 Sviluppi futuri

L'applicazione Android ImageSound potrebbe essere migliorata nei seguenti elementi:

- utilizzo per le immagini dei **modelli colore HSV o HSI** al posto di quello HSL, al fine di valutare eventuali differenze nel mapping da immagine a suono e viceversa;
- impiego di **ulteriori tecniche per il calcolo delle frequenze** a partire dai campioni audio acquisiti tramite microfono;
- nel processo Sound-Image utilizzare lo **spettrogramma immagine**, dell'audio registrato, al posto delle frequenze per la generazione dell'immagine finale;
- utilizzo di **multipli segnali sinusoidali** per la generazione del suono di output rispetto al singolo attuale. Si ottiene un suono più uniforme ma si incrementa la complessità computazionale dell'applicazione;
- regolazione in **automatico, per le modalità live, delle immagini da elaborare al secondo** in base al valore della qualità percentuale dell'elaborazione. Se la qualità si trova al di sopra di una soglia superiore per almeno n secondi, si aumenta il numero di immagini da elaborare mentre se scende al di sotto di una soglia inferiore, si decrementa il numero di immagini;
- mostrare nell'applicazione un grafico con i campioni audio prodotti nell'arco del tempo e dare la possibilità all'utente di **scegliere un intervallo di campioni per mostrare a quali pixel dell'immagine appartengono**.

Appendice A

Modifiche apportate al codice di OpenCV per la risoluzione delle problematiche della camera preview

A.1 Problematica orientazione

Porzione modificata/aggiunta nella classe Java *CameraGLRendererBase*

```
// MODIFICATA

// attributo originale definito libreria (con il final)
/*private final float texCoord2D[] = {
   0,  0,
   0,  1,
   1,  0,
   1,  1 };*/

// tolto il final per rendere l'attributo modificabile
private float texCoord2D[] = {
   0,  0,
   0,  1,
   1,  0,
   1,  1 };


// AGGIUNTA

// metodo aggiunto che cambia i valori di texCoord2D in base alla rotazione
    dello schermo e al tipo di fotocamera (anteriore o posteriore)
public void cambiaTexCoord2DRotazioneSchermo(int rotazione, boolean cameraFront){
```

```
// valore default Surface.ROTATION_180 o non alterazione
boolean texCoord2DUguale = true;

// camera posteriore
if (!cameraFront){

   // ActivityInfo.SCREEN_ORIENTATION_PORTRAIT
   if (rotazione == Surface.ROTATION_0){

      texCoord2DUguale = Arrays.equals(texCoord2D, new float[] { 1, 0, 0, 0,
          1, 1, 0, 1 });

      if (!texCoord2DUguale){

         texCoord2D = new float[] {
            1,  0,
            0,  0,
            1,  1,
            0,  1 };
      }

   }
   // ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE
   else if (rotazione == Surface.ROTATION_90){

      texCoord2DUguale = Arrays.equals(texCoord2D, new float[] { 0, 0, 0, 1,
          1, 0, 1, 1 });

      if (!texCoord2DUguale){

         texCoord2D = new float[] {
            0,  0,
            0,  1,
            1,  0,
            1,  1 };

      }

   }
   // Surface.ROTATION_270, ActivityInfo.SCREEN_ORIENTATION_REVERSE_LANDSCAPE
   else if (rotazione == Surface.ROTATION_270){

      texCoord2DUguale = Arrays.equals(texCoord2D, new float[] { 1, 1, 1, 0,
          0, 1, 0, 0 });

      if (!texCoord2DUguale){


         texCoord2D = new float[] {
```

```
            1,  1,
            1,  0,
            0,  1,
            0,  0 };
      }

   }
}
// camera frontale
else{

   // ActivityInfo.SCREEN_ORIENTATION_PORTRAIT
   if (rotazione == Surface.ROTATION_0){

      texCoord2DUguale = Arrays.equals(texCoord2D, new float[] { 0, 0, 1, 0,
          0, 1, 1, 1 });

      if (!texCoord2DUguale){

         texCoord2D = new float[] {
            0,  0,
            1,  0,
            0,  1,
            1,  1 };
      }

   }
   // ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE
   else if (rotazione == Surface.ROTATION_90){

      texCoord2DUguale = Arrays.equals(texCoord2D, new float[] { 1, 0, 1, 1,
          0, 0, 0, 1 });

      if (!texCoord2DUguale){

         texCoord2D = new float[] {
            1,  0,
            1,  1,
            0,  0,
            0,  1 };

      }

   }
   // Surface.ROTATION_270, ActivityInfo.SCREEN_ORIENTATION_REVERSE_LANDSCAPE
   else if (rotazione == Surface.ROTATION_270){

      texCoord2DUguale = Arrays.equals(texCoord2D, new float[] { 0, 1, 0, 0,
          1, 1, 1, 0 });
```

```
        if (!texCoord2DUguale){

           texCoord2D = new float[] {
              0,  1,
              0,  0,
              1,  1,
              1,  0 };
        }

     }
  }


  // diverso dal verticale rovesciato in cui l'activity non subisce variazioni
      o rotazione inalterata
  // rotazione != Surface.ROTATION_180
  if (!texCoord2DUguale)
      tex2D.put(texCoord2D).position(0);

}
```

A.2 Problematica dimensioni ottimali

Porzione modificata/aggiunta nella classe Java *Camera2Renderer*

```
// metodo di OpenCV che determina le dimensioni migliori da utilizzare per i
    frame immagine della camera preview
boolean cacPreviewSize(final int width, final int height) {

  Log.i(LOGTAG, "cacPreviewSize: "+width+"x"+height);

  if(mCameraID == null) {
     Log.e(LOGTAG, "Camera isn't initialized!");
     return false;
  }

  CameraManager manager = (CameraManager)
      mView.getContext().getSystemService(Context.CAMERA_SERVICE);

  try {

     CameraCharacteristics characteristics =
         manager.getCameraCharacteristics(mCameraID);

     StreamConfigurationMap map = characteristics
         .get(CameraCharacteristics.SCALER_STREAM_CONFIGURATION_MAP);
```

```
int bestWidth = 0, bestHeight = 0;


// INIZIO PORZIONE MODIFICATA/AGGIUNTA

// MODIFICATA

// codice originale libreria che considera solo una larghezza maggiore
    dell'altezza (solo landscape, senza if e else)
/*float aspect = (float)width / height;
for (Size psize : map.getOutputSizes(SurfaceTexture.class)) {
   int w = psize.getWidth(), h = psize.getHeight();
   Log.d(LOGTAG, "trying size: "+w+"x"+h);
   if ( width >= w && height >= h &&
   bestWidth <= w && bestHeight <= h &&
   Math.abs(aspect - (float)w/h) < 0.2 ) {
      bestWidth = w;
      bestHeight = h;
   }
}*/


// AGGIUNTA

// il codice originale viene inserito nel ramo if (larghezza deve essere
    maggiore dell'altezza, landscape)
if (width > height){

   float aspect = (float)width / height;
   for (Size psize : map.getOutputSizes(SurfaceTexture.class)) {
      int w = psize.getWidth(), h = psize.getHeight();
      Log.d(LOGTAG, "trying size: "+w+"x"+h);
      if ( width >= w && height >= h &&
      bestWidth <= w && bestHeight <= h &&
      Math.abs(aspect - (float)w/h) < 0.2 ) {
         bestWidth = w;
         bestHeight = h;
      }
   }

}
// aggiunto il ramo else per considerare il caso verticale (altezza deve
    essere maggiore o uguale della larghezza, portrait)
else{

   float aspect = (float)height / width;
   for (Size psize : map.getOutputSizes(SurfaceTexture.class)) {
      // inversione delle componenti
      int w = psize.getHeight(), h = psize.getWidth();
```

```
            Log.d(LOGTAG, "trying size: "+w+"x"+h);
            if ( width >= w && height >= h &&
            bestWidth <= w && bestHeight <= h &&
            Math.abs(aspect - (float)h/w) < 0.2 ) {
               bestWidth = w;
               bestHeight = h;
            }
         }
      }

      // non trovando le migliori dimensioni sono utilizzate dimensioni standard
      if (bestWidth == 0 || bestHeight == 0){

         // landscape
         if (width > height){
            bestWidth = 640;
            bestHeight = 480;
         }
         // portrait
         else{
            bestWidth = 480;
            bestHeight = 640;
         }
      }

      // FINE PORZIONE MODIFICATA/AGGIUNTA


      Log.i(LOGTAG, "best size: "+bestWidth+"x"+bestHeight);

      if( bestWidth == 0 || bestHeight == 0 ||
      mPreviewSize.getWidth() == bestWidth &&
      mPreviewSize.getHeight() == bestHeight )
         return false;
      else {
         mPreviewSize = new Size(bestWidth, bestHeight);
         return true;
      }

   } catch (CameraAccessException e) {
      Log.e(LOGTAG, "cacPreviewSize - Camera Access Exception");
   } catch (IllegalArgumentException e) {
      Log.e(LOGTAG, "cacPreviewSize - Illegal Argument Exception");
   } catch (SecurityException e) {
      Log.e(LOGTAG, "cacPreviewSize - Security Exception");
   }
   return false;
}
```

Bibliografia

[1] Wang, B. T., Liu, Y. H., & Li, X. M. (2017, December). A summary of blind guiding methods converting images to sounds. In *Proceedings of APSIPA Annual Summit and Conference* (Vol. 2017, pp. 12-15).

[2] Brown, D., Macpherson, T., & Ward, J. (2011). Seeing with sound? Exploring different characteristics of a visual-to-auditory sensory substitution device. *Perception*, 40(9), 1120-1135.

[3] Spyridis, H. C., & Moustakas, A. K. (2007, July). Image to Sound and Sound to Image Transform. In *Proceedings of the 4th Sound and Music Computing Conference (SMC07)* (pp. 11-13).

[4] Zhang, X., Wang, J. M., Duan, X. J., & Sun, Y. K. (2014). An Efficient Method of Image-Sound Conversion Based on IFFT for Vision Aid for the Blind. *Lecture Notes on Software Engineering*, 2(1), 54-57.

[5] Yeo, W. S., & Berger, J. (2006, September). Raster scanning: a new approach to image sonification, sound visualization, sound analysis, and synthesis. In *ICMC*.

[6] Mengucci, M., Henriques, J. T., Cavaco, S., Correia, N., & Medeiros, F. (2012). From color to sound: Assessing the surrounding environment. In *Proceedings of the Conference on Digital Arts and New Media (ARTECH)* (pp. 345-348).

[7] Cavaco, S., Mengucci, M., Henriques, J. T., Correia, N., & Medeiros, F. (2013, May). From pixels to pitches: unveiling the world of color for the blind. In *2013 IEEE 2nd International Conference on Serious Games and Applications for Health (SeGAH)* (pp. 1-8). IEEE.

[8] Cavaco, S., Henriques, J. T., Mengucci, M., Correia, N., & Medeiros, F. (2013). Color sonification for the visually impaired. *Procedia Technology*, 9, 1048-1057.

[9] Bologna, G., Deville, B., & Pun, T. (2009, June). Blind navigation along a sinuous path by means of the See ColOr interface. In *International Work-Conference on the Interplay Between Natural and Artificial Computation* (pp. 235-243). Springer, Berlin, Heidelberg.

[10] Politis, D., Margounakis, D., & Karatsoris, M. (2008, February). Image to sound transforms and sound to image visualizations based on the chromaticism of music. In *7th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Databases* (pp. 309-317).

[11] Politis, D., & Margounakis, D. (2008). A Synopsis of Sound-Image Transforms based on the Chromaticism of Music. *WSEAS Transactions on Computers*, 7(8), 1113-1127.

[12] Kawamura, A. (2017). On sound signal processing in image to sound mapping technique. *Applied Acoustics*, 117, 1-11.

[13] PrachiPhursutkar, A. B., Veer, P., Tai Chormale, P., & Bhandari, G. M. A Secured Application for Generating Acoustic Signal for Blind People.

[14] Verma, J., Desai, K., & Guptas, B. (2013). Image to Sound Conversion. *International Journal of Advance Research in Computer Science and Management Studies (IJARCSMS)*, 1(6).

[15] Fanzeres, L. A., & Nadeu, C. (2021). Sound-to-Imagination: Unsupervised Crossmodal Translation Using Deep Dense Network Architecture. *arXiv preprint arXiv:2106.01266*.

[16] Caivano, J. L. (1994). Color and sound: Physical and psychophysical relations. *Color Research & Application*, 19(2), 126-133.

[17] Grand, F. (2007). Organized Data for Organized Sound Space Fitting Curves in Sonification. *Georgia Institute of Technology*.

[18] Jablonska, J., Trocka-Leszczynska, E., & Tarczewski, R. (2015). Sound and architecture-mutual influence. *Energy Procedia*, 78, 31-36.

[19] Lin, A., Scheller, M., Feng, F., Proulx, M. J., & Metatla, O. (2021, May). Feeling colours: Crossmodal correspondences between tangible 3d objects, colours and emotions. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (pp. 1-12).

[20] Kocaoğlu, R., & Olguntürk, N. (2018). Color and visual complexity in abstract images. *Color Research & Application*, 43(6), 952-957.

[21] Nadal Roberts, M. (2007). Complexity and aesthetic preference for diverse visual stimuli. (*Doctoral dissertation, Universitat de les Illes Balears*).

[22] Tayyebi, S. F., & Demir, Y. (2020). Musical Preferences Correlate Architectural Tastes: An Initial Investigation of the Correlations Between the Preferred Attributes. *Advanced Journal of Social Science*, 7(1), 96-108.

[23] *Platform Architecture* [Online] (30 Agosto 2021). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://developer.android.com/guide/platform

[24] *Get started with the NDK* [Online] (12 Ottobre 2022). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://developer.android.com/ndk/guides

[25] *Meet Android Studio* [Online] (24 Ottobre 2022). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://developer.android.com/studio/intro

[26] *SDK Platform release notes* [Online] (24 Ottobre 2022). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://developer.android.com/studio/releases/platforms

[27] *TensorFlow* [Online] (19 Agosto 2022). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://github.com/tensorflow/tensorflow/blob/master/README.md

[28] *TensorFlow Lite per Android* [Online] (7 Settembre 2022). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://www.tensorflow.org/lite/android

[29] *Keras: Deep Learning for humans* [Online] (22 Luglio 2022). [Ultimo Accesso: 26 Ottobre 2022].
Disponibile: https://github.com/keras-team/keras/blob/master/README.md

[30] *Color Models* [Online] (11 Aprile 2022). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://www.intel.com/content/www/us/en/develop/documentation/ipp-dev-reference/top/volume-2-image-processing/image-color-conversion/color-models.html

[31] Schatzman, J. C. (1996). Accuracy of the discrete Fourier transform and the fast Fourier transform. *SIAM Journal on Scientific Computing*, 17(5), 1150-1166.

[32] *High-performance audio* [Online] (26 Ottobre 2021). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://developer.android.com/ndk/guides/audio

[33] *OpenSL ES* [Online] (27 Dicembre 2019). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://developer.android.com/ndk/guides/audio/opensl

[34] *AAudio* [Online] (13 Ottobre 2021). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://developer.android.com/ndk/guides/audio/aaudio/aaudio

[35] *Oboe audio library* [Online] (15 Marzo 2022). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://developer.android.com/games/sdk/oboe

[36] *Full Guide To Oboe* [Online] (17 Gennaio 2022). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://github.com/google/oboe/blob/main/docs/FullGuide.md

[37] *OpenCV: Introduction* [Online] (26 Ottobre 2022). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://docs.opencv.org/4.x/d1/dfb/intro.html

[38] *OpenCV: OpenCV4Android SDK* [Online] (26 Ottobre 2022). [Ultimo Accesso: 26 Ottobre 2022].
Disponibile: https://docs.opencv.org/4.x/da/d2a/tutorial_O4A_SDK.html

[39] *How to use the OpenCV parallel_for_ to parallelize your code* [Online] (26 Ottobre 2022). [Ultimo Accesso: 26 Ottobre 2022].
Disponibile: https://docs.opencv.org/4.x/dc/ddf/tutorial_how_to_use_OpenCV_parallel_for_new.html

[40] *Displaying graphics with OpenGL ES* [Online] (25 Agosto 2022). [Ultimo Accesso: 26 Ottobre 2022].
Disponibile: https://developer.android.com/develop/ui/views/graphics/opengl

[41] Toledo-Perez, D. C., Rodríguez-Reséndiz, J., & Gómez-Loenzo, R. A. (2020). A study of computing zero crossing methods and an improved proposal for EMG signals. *IEEE Access*, 8, 8783-8790.

[42] *ArchitecturalStyle Recognition* [Online] (31 Agosto 2020). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://github.com/dumitrux/architectural-style-recognition/blob/master/README.md

[43] *MonuMAI dataset* [Online] (21 Luglio 2020). [Ultimo Accesso: 26 Ottobre 2022].
Disponibile:
https://github.com/ari-dasci/OD-MonuMAI/blob/master/README.md

[44] *Architectural-Heritage-Elements-Prediction* [Online] (6 Agosto 2020). [Ultimo Accesso: 26 Ottobre 2022].
Disponibile: https://github.com/newbieeashish/Architectural-Heritage-Elements-Prediction/blob/master/README.md

[45] *Cultural Heritage Dataset - Orthodox Churches* [Online] (12 Maggio 2020). [Ultimo Accesso: 26 Ottobre 2022].
Disponibile: https://www.kaggle.com/datasets/rjankovic/cultural-heritage-orthodox-churches

[46] *Salva e carica i modelli Keras* [Online] (10 Gennaio 2022). [Ultimo Accesso: 26 Ottobre 2022].
Disponibile: https://www.tensorflow.org/guide/keras/save_and_serialize

[47] *Classificazione del suono con YAMNet* [Online] (6 Gennaio 2022). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://www.tensorflow.org/hub/tutorials/yamnet

[48] *Trasferisci l'apprendimento con YAMNet per la classificazione del suono ambientale* [Online] (26 Gennaio 2022). [Ultimo Accesso: 26 Ottobre 2022]. Disponibile: https://www.tensorflow.org/tutorials/audio/transfer_learning_audio

[49] *GTZAN Dataset - Music Genre Classification* [Online] (24 Marzo 2020). [Ultimo Accesso: 26 Ottobre 2022].
Disponibile: https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification

Ringraziamenti

Vorrei svolgere i ringraziamenti secondo lo stile di ImageSound. Infatti come i pixel che costituiscono un'immagine o le note che compongono una canzone, ognuno degli elementi risulta fondamentale al fine della realizzazione del prodotto finale. Ciascuno degli elementi fornisce un contributo con i propri colori o la propria musicalità come le persone mediante consigli, sostegni, momenti di svago, periodi gioiosi o di tristezza e normali azioni abituali della vita quotidiana. Anche se tutti questi aspetti sono indipendenti tra loro, in realtà supportano involontariamente l'obiettivo comune di non abbandonare mai i propri progetti e di continuare fino al loro completamento, superando qualsiasi ostacolo incontrato. Per questo ringrazio indistintamente, a prescindere dal contesto, qualsiasi persona incontrata lungo il cammino in questi ultimi anni e con cui ho scambiato opinioni o pensieri perché hanno permesso la realizzazione dell'app ImageSound, un mio momentaneo piccolo sogno.