



# UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in  
Informatica

ELABORATO FINALE

## REALIZZAZIONE DI UN SISTEMA BASATO SU ANDROID PER L'ACQUISIZIONE MULTIMEDIALE DI LEZIONI UNIVERSITARIE

Supervisore

Marco Ronchetti

Laureando

Matteo Contrini

Anno accademico 2018/2019

# Ringraziamenti

*La presente tesi è stata redatta nell'ambito di un tirocinio formativo svolto presso l'azienda AXIA Studio. Grazie a Tiziano, Chloe e al prof. Ronchetti per avermi seguito nel percorso.*

# Indice

<b>Sommario</b>	<b>2</b>
<b>1 Introduzione</b>	<b>4</b>
1.1 LODE: introduzione e motivazioni . . . . .	4
1.2 La soluzione LodeBox . . . . .	4
1.3 Hardware alternativo . . . . .	5
<b>2 Acquisizione video HDMI</b>	<b>5</b>
2.1 Le API android.hardware.Camera . . . . .	5
2.2 Le API android.hardware.camera2.* . . . . .	6
2.3 SDK alternativi . . . . .	7
<b>3 Acquisizione video RTSP</b>	<b>8</b>
3.1 Il protocollo RTSP . . . . .	8
3.2 Registrazione con ffmpeg . . . . .	8
<b>4 Realizzazione della modalità “kiosk”</b>	<b>9</b>
4.1 La modalità a schermo intero . . . . .	9
4.2 L' <i>overscan</i> di sistema . . . . .	10
4.3 La modalità “lock task” . . . . .	11
4.4 L'avvio automatico dell'applicazione . . . . .	12
4.5 L'animazione di avvio . . . . .	12
<b>5 Sospensione della registrazione</b>	<b>13</b>
<b>6 Sincronizzazione tra video e audio</b>	<b>16</b>
6.1 Definizione del problema . . . . .	16
6.2 Proposta di soluzione . . . . .	16
6.3 Implementazione e sperimentazione . . . . .	17
<b>7 Rilevamento delle differenze tra fotogrammi</b>	<b>20</b>
7.1 Il formato YUV420SP . . . . .	21
7.2 Confronto pixel per pixel . . . . .	22
7.3 Confronto parziale . . . . .	23
7.4 Estrazione slide in post-elaborazione . . . . .	25
<b>8 Acquisizione audio</b>	<b>27</b>
8.1 Registrazione con MediaRecorder . . . . .	27
8.2 Acquisizione raw con AudioRecord . . . . .	28
<b>9 Conclusione</b>	<b>28</b>
<b>Bibliografia</b>	<b>30</b>
<b>A La libreria MobileFFmpeg</b>	<b>31</b>
<b>B Acquisizione audio PCM</b>	<b>32</b>

# Sommario

L'oggetto di questa tesi è lo sviluppo di un sistema di cattura e registrazione multimediale di lezioni universitarie, conferenze e seminari. Il sistema prodotto si basa su un dispositivo fisico dotato di apposito hardware per la cattura video, sul quale viene eseguito un apposito software su sistema operativo Android.

Il sistema è stato realizzato nell'ambito del progetto LODE (Lectures On DEMand) del professor Marco Ronchetti (Università di Trento), ed è stato elaborato durante un tirocinio formativo presso l'azienda AXIA Studio, con l'obiettivo finale di affinare la soluzione e di immetterla sul mercato delle soluzioni IoT accademiche.

Il progetto LODE, nel cui contesto questo lavoro si inserisce, è un progetto con l'obiettivo di fornire una soluzione low cost per l'acquisizione video delle lezioni universitarie. L'ultima versione del sistema prevede la possibilità di registrare flussi multimediali multipli, in particolare il video in uscita dal computer del docente, il video acquisito da una videocamera IP che riprende il docente e/o la lavagna, e l'audio di un microfono indossato dal docente. I contenuti sono poi elaborati e pubblicati su una pagina web accessibile dagli studenti del corso.

Una funzionalità del sistema prevede anche che durante le lezioni lo studente possa catturare degli screenshot di ciò che viene proiettato in quel momento, e inserire delle annotazioni testuali o disegnare sulle catture.

Nella sua ultima versione, LODE prevede l'uso di un dispositivo fisico, soprannominato LodeBox, che incorpora un "mini-computer" Raspberry Pi. Tramite apposite estensioni hardware, in particolare un modulo "HDMI to CSI-2", il dispositivo è in grado di acquisire un input HDMI come se si trattasse del video di una videocamera, e di sfruttare funzionalità come l'anteprima su schermo e la registrazione H.264 tramite encoder hardware. Il dispositivo prevede anche la possibilità di collegare un proiettore tramite uscita HDMI, un microfono tramite dongle USB, e una videocamera IP/RTSP (Real Time Streaming Protocol) tramite la rete locale.

Questa soluzione si scontra però con delle difficoltà che rendono difficile la realizzazione di un sistema affidabile e distribuibile su larga scala. Tra le problematiche principali si evidenziano la difficoltà nel gestire situazioni "plug and play", come la disconnessione e la riconnessione del cavo HDMI, la mancanza di storage integrato duraturo e ad alte prestazioni, le scarse garanzie sulla disponibilità del modulo hardware di conversione HDMI.

Ci si è orientati quindi verso la ricerca di SBC (Single-Board Computer) alternative più adatte per la realizzazione di applicazioni multimediali. Tra le soluzioni più accessibili dal punto di vista economico spiccano molte board basate sul sistema operativo Android, il quale offre il beneficio di avere una piattaforma ben nota e documentata. Si ha inoltre la possibilità di sostituire il dispositivo hardware riutilizzando gran parte del codice scritto.

Gran parte del mio lavoro si è quindi concentrato sullo sviluppo di alcune applicazioni Java/Kotlin, con lo scopo di verificare la fattibilità di una versione di LodeBox con piattaforma Android.

La peculiarità fondamentale di molte board Android con input HDMI è che permettono di sfruttare direttamente le API di Android per l'acquisizione di video e immagini, consentendo di conseguenza di scrivere codice che non sia strettamente legato all'hardware. La piattaforma Android prevede due versioni delle API per l'accesso alla fotocamera, in particolare la classe `android.hardware.Camera` e le classi contenute in `android.hardware.camera2.*`. A partire da Android 5.0 (livello API 21), la classe `Camera` è deprecata ed è invece consigliato l'uso delle API `camera2`, più avanzate ma anche più complesse da usare. Ciò nonostante, non è garantito che ogni dispositivo con Android 5 o superiore supporti a pieno le API `camera2`.

È stato quindi necessario approfondire il funzionamento di entrambe le versioni delle API, per determinare se entrambe permettono di soddisfare i requisiti di LODE e quale versione conviene usare a seconda dell'hardware che si ha a disposizione.

Oltre all'input HDMI, si ha la necessità di registrare anche una videocamera esterna, che per contenere i costi è una videocamera IP raggiungibile nella rete locale tramite il protocollo di streaming RTSP. Di conseguenza, il flusso può essere registrato direttamente sulla board, utilizzando una versione della libreria `ffmpeg` compilata per funzionare su Android.

Un altro punto importante dello sviluppo di un'applicazione embedded per Android è la realizzazione di una modalità *kiosk*, cioè una situazione in cui l'utente può utilizzare solo ed esclusivamente una singola applicazione, senza poter accedere alle altre parti del sistema operativo o ad altre funzioni. Questo ha portato a sperimentare diverse soluzioni per assicurarsi che l'applicazione resti sempre a schermo intero, tra cui la modalità "lock task" di Android e l'avvio automatico come applicazione launcher.

Il risultato di questa fase è stato un insieme di applicazioni-prototipo che sono state sperimentate sul campo durante alcune lezioni presso l'Università di Trento, permettendo di raccogliere importanti riscontri sul funzionamento del sistema.

A questo punto la fattibilità delle funzionalità di base che il sistema deve fornire è stata verificata, e i passi successivi hanno riguardato altri aspetti per migliorare il funzionamento del software.

Uno di questi è la possibilità di mettere in pausa e riprendere la registrazione della lezione. L'idea considerata è stata quella di non sospendere la registrazione ma di escludere in post-produzione i segmenti di video corrispondenti alle pause. Questo è agevolmente ottenibile grazie a `ffmpeg` e con l'ausilio di uno script che generi il comando (potenzialmente lungo) per ritagliare i segmenti. Nel caso in cui non fosse possibile avere una registrazione unica, una tecnica simile può essere adottata per unire più segmenti video in fase di post-elaborazione.

Un'altra questione approfondita riguarda la sincronizzazione dei flussi, con lo scopo principale di evitare che l'audio risulti troppo sfasato rispetto al video, ed evitare che sia visibile lo sfasamento tra voce registrata e labiale del relatore. Il problema sorge principalmente per via della presenza del video RTSP, la cui latenza è difficilmente stimabile. La soluzione sperimentale proposta prevede di incorporare l'audio nella registrazione video HDMI (in modo da ottenere una naturale sincronizzazione tra i due flussi acquisiti via cavo), e di occuparsi invece di sincronizzare video HDMI e RTSP. L'implementazione sperimentale è stata ottenuta mostrando su schermo un marcitore visivo che permetta di individuare con precisione un istante comune tra i due flussi video, e di conseguenza sincronizzarli.

Come accennato, un'altra funzione del sistema LODE prevede che lo studente possa prendere appunti in tempo reale durante lo svolgimento delle lezioni, catturando degli screenshot di ciò che vede in quel momento. Per evitare di inviare inutilmente screenshot al server nel caso in cui non ci sia stato nessun cambiamento percepibile nell'immagine, un sistema intelligente può rilevare le differenze tra i fotogrammi per determinare se l'immagine è nuova o invariata. Dopo aver appurato empiricamente che il confronto di tutti i pixel di due fotogrammi risulta computazionalmente dispendioso, un'alternativa considerata è quella di scegliere con un fattore di casualità un numero limitato di pixel "salienti". A livello intuitivo, il metodo sviluppato realizza una griglia con un numero oscillante di righe e colonne, le cui intersezioni determinano una quantità limitata di pixel che possono essere utilizzati per rilevare rapidamente le differenze tra i fotogrammi.

I capitoli di questa tesi approfondiscono più dettagliatamente le problematiche e le soluzioni che sono state sintetizzate in questo sommario.

# 1 Introduzione

Questo capitolo introduce il progetto LODE sviluppato presso l'Università di Trento, nel cui contesto questa tesi si inserisce. Sono approfonditi motivazioni e vantaggi, ed è introdotta l'ultima versione del sistema, cioè un dispositivo hardware dal nome LodeBox. Sono infine presentate le principali problematiche legate a LodeBox e l'hardware alternativo considerato per l'evoluzione del progetto.

## 1.1 LODE: introduzione e motivazioni

LODE (Lectures On DEmand) è un progetto realizzato dal professor Marco Ronchetti e collaboratori presso l'Università di Trento. Si presenta come una soluzione per l'acquisizione in formato video e audio di lezioni universitarie, con la particolarità di essere una soluzione a basso costo e facilmente manovrabile [8].

Le lezioni registrate possono poi essere consultate tramite una pagina web appositamente generata, che combina il video del computer del docente, il video ripreso da una videocamera posizionata in aula e l'audio catturato da un microfono.

I vantaggi che questo sistema offre sono molteplici, tra cui: la possibilità per gli studenti-lavoratori di seguire le lezioni in remoto a qualsiasi orario; la possibilità di recuperare le lezioni in caso di assenze non volontarie (es. malattia); supporto per gli studenti che non comprendono bene la lingua del corso; la possibilità di rivedere porzioni specifiche di qualsiasi lezione in qualsiasi momento, e di verificare quindi la qualità dei propri appunti e il livello di comprensione.

Le versioni più recenti di LODE prevedono anche un'interfaccia web utilizzabile dagli studenti durante lo svolgimento della lezione. Questo strumento permette di catturare degli screenshot di quanto proiettato dal docente in quell'istante, e di scrivere o disegnare annotazioni sulle catture. In questo modo gli studenti sono coinvolti in modo meno passivo e seguono la lezione con più attenzione.

## 1.2 La soluzione LodeBox

L'ultima versione di LODE prevede l'utilizzo di una board Raspberry Pi per eseguire il software di acquisizione. Il dispositivo è inserito in una piccola scatola che espone dei connettori verso l'esterno. Tra questi sono presenti un ingresso HDMI per collegare un computer e un'uscita HDMI per collegare un proiettore o uno schermo. Le porte USB permettono di collegare un telecomando e un "dongle" per l'acquisizione dell'audio di un microfono con jack 3,5 mm.

La scatola include un modulo "HDMI to MIPI CSI-2", che permette di convertire il segnale HDMI nel formato seriale della fotocamera. Il video HDMI è infatti acquisito utilizzando la libreria `PiCamera` per Python, che permette di acquisire il segnale come se si trattasse del video di una fotocamera. La libreria semplifica di molto lo sviluppo, perché espone delle funzioni di alto livello per svolgere operazioni come abilitare l'anteprima del video a schermo intero, la registrazione con un encoder H.264<sup>1</sup> con accelerazione hardware e la cattura di screenshot.

Sempre in ottica di riduzione dei costi, LodeBox prevede la ripresa del docente tramite una qualsiasi videocamera IP (wireless o cablata) che supporti il protocollo RTSP (Real Time Streaming

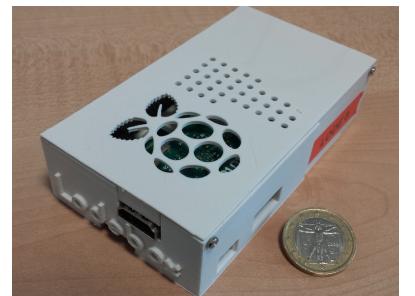


Figura 1.1: La scatola di LodeBox.

<sup>1</sup>H.264, conosciuto anche come AVC o MPEG-4 Part-10, è il più popolare codec di compressione video. Nato nel 2003, è ancora oggi lo standard de facto in numerosi ambiti, tra cui lo streaming video.

Protocol). La registrazione avviene sul dispositivo utilizzando `ffmpeg/avconv`<sup>2</sup> in modalità copia (senza ricodifica), richiedendo quindi un uso molto basso di risorse.

Raspberry Pi non prevede spazio di archiviazione interno, e richiede quindi di utilizzare una scheda microSD o una chiavetta USB per memorizzare le registrazioni. Le memorie di tipo flash hanno però vita limitata, a seconda di quanto intensivamente sono utilizzate, e questo riduce di conseguenza l'affidabilità a lungo termine del sistema. Inoltre, la soluzione di usare la porta USB non risulta percorribile, per via delle prestazioni molto scarse che impediscono di registrare tre flussi in contemporanea.

Un altro svantaggio di questa soluzione è il livello di “fault tolerance” in caso di eventi come la disconnessione (volontaria o meno) del cavo HDMI in ingresso. Dato che l'input HDMI è mappato sull'interfaccia della fotocamera, non è previsto che la fotocamera possa essere improvvisamente scollegata. Secondo le prove effettuate, risulta molto difficile rilevare in modo affidabile la disconnessione del cavo HDMI. Nei casi in cui è possibile, non risulta invece fattibile il recupero dell'applicazione, perché le operazioni sulla `PiCamera` sollevano eccezioni o bloccano indefinitamente l'esecuzione del codice.

Infine, un altro problema è posto dalla presenza del modulo di conversione HDMI-CSI, la cui disponibilità e compatibilità a lungo termine non sono garantite.

### 1.3 Hardware alternativo

Per poter evolvere LodeBox in una soluzione più affidabile e adatta alla produzione e distribuzione, si sono quindi cercate SBC (Single-Board Computer) alternative, preferibilmente pensate per la realizzazione di applicazioni multimediali.

Dal punto di vista tecnico, i principali vincoli per la scelta di una nuova scheda erano la presenza dell'input HDMI e di un encoder H.264 hardware, la possibilità di collegare un microfono e il costo accessibile.

Le soluzioni considerate erano inoltre di tipo pre-industriale, quindi non necessariamente già presenti sul mercato ma personalizzabili per soddisfare i requisiti del prodotto, e offrivano anche determinate garanzie di disponibilità e supporto per un periodo di tempo prolungato.

Molte board con sistema operativo Android soddisfano queste caratteristiche, e consentono in aggiunta di avere a disposizione una piattaforma nota, documentata, di facile sviluppo e per cui sono disponibili molte risorse. In molti casi l'input HDMI è reso disponibile tramite l'interfaccia della fotocamera CSI, da cui deriva la possibilità di acquisire l'input tramite le API di Android per l'accesso alla fotocamera.

## 2 Acquisizione video HDMI

Nelle sezioni che seguono vengono introdotti i principali metodi tipicamente disponibili su board Android per l'accesso all'ingresso HDMI. Si tratta in particolare delle due API per l'accesso alla fotocamera, a cui ci si riferisce spesso come “Camera1” e “Camera2”, e talvolta di SDK forniti dal produttore.

### 2.1 Le API `android.hardware.Camera`

La classe `Camera` è stata introdotta nella prima versione di Android ed è il metodo più semplice per accedere alla fotocamera. Va notato però che la classe è stata messa in stato di deprecazione a partire da Android 5.0 (livello API 21), e Google consiglia di utilizzare invece le API `Camera2` per realizzare nuove applicazioni.<sup>3</sup> Tuttavia, non tutti i dispositivi con Android 5.0 o superiore supportano

---

<sup>2</sup>`ffmpeg` è uno strumento estremamente popolare per l'elaborazione di video e audio tramite linea di comando. Supporta numerosi formati, codec e protocolli, e permette di effettuare operazioni quali il muxing, transmuxing, transcoding e altre funzioni più avanzate. `avconv` è un fork di `ffmpeg` nato per via di divergenze sui metodi di sviluppo, ed è stato incluso per un breve periodo in alcune distribuzioni Linux in sostituzione di `ffmpeg`.

nativamente le API Camera2. Esiste infatti una modalità LEGACY che dà la possibilità di usare le API Camera2 nonostante l'implementazione sia in realtà la stessa delle API Camera1.

Entrando nel merito, per mostrare il video acquisito sullo schermo è prima di tutto necessario aggiungere un nuovo controllo nella **Activity**<sup>4</sup> dell'applicazione Android. Si utilizza per questo un elemento chiamato **SurfaceView**, che permette di disegnare contenuti su schermo in modo ottimizzato, fluido e senza consumare le risorse del thread della grafica. Quando una **SurfaceView** diventa visibile su schermo, una callback informa del fatto che la sottostante **Surface** è stata creata. È possibile a questo punto accedere, tramite il metodo d'istanza **getHolder()**, a un'implementazione dell'interfaccia **SurfaceHolder**, che permette di configurare e controllare la superficie di disegno.

Ottenuto il **SurfaceHolder**, si può procedere con la creazione di un'istanza di **Camera** tramite il metodo statico **Camera.open()**. Si collega quindi la superficie alla **Camera** tramite il metodo **setPreviewDisplay(SurfaceHolder)**. Infine, una chiamata a **startPreview()** permette di avviare l'anteprima su schermo.

A questo punto, creando un'istanza della classe **android.media.MediaRecorder** è possibile registrare il video acquisito, mostrando allo stesso tempo l'anteprima su schermo. Più in dettaglio, si tratta di seguire una rigida sequenza di passaggi<sup>5</sup> per configurare correttamente il **MediaRecorder** e collegare l'input **Camera** al registratore, tramite il metodo **setCamera(Camera)**. La registrazione può infine essere avviata con il metodo **start()**.

La cattura di un singolo screenshot si può invece ottenere tramite il metodo **takePicture(...)**, esposto dalla classe **Camera**. Questo metodo permette di ricevere tramite delle callback i dati del fotogramma catturato, sia in formato non compresso che in JPEG.

Questo metodo presenta però un possibile effetto collaterale, riscontrato durante i test, ossia la possibilità che alla cattura di un fotogramma l'anteprima del video su schermo venga interrotta. Durante lo sviluppo si è quindi optato per una modalità alternativa per l'acquisizione degli screenshot: è possibile infatti aggiungere una callback alla **Camera** tramite il metodo **setPreviewCallback**, e ricevere così a flusso continuo una copia di ogni fotogramma in formato non compresso<sup>6</sup>.

Sorge a questo punto un altro problema, cioè la necessità di convertire l'immagine in formato JPEG, per permettere l'upload dello screenshot sul server. La piattaforma Android offre una classe **YuvImage** con un metodo **compressToJpeg(...)**, che soffre però di un memory leak che porta dopo qualche minuto al crash dell'applicazione [4], come verificato nei test svolti. Il bug è stato risolto solo in Android 9.0<sup>7</sup>, rendendo quindi necessaria un'alternativa. Si è optato per una soluzione che prevede la conversione in modo ottimizzato dal formato “non compresso” (NV21) a **Bitmap**<sup>8</sup>, e successivamente in JPEG tramite il metodo **compress(...)** offerto da **Bitmap** (che non soffre di memory leak). Nonostante il numero di passaggi, il risultato si è rivelato soddisfacente dal punto di vista delle prestazioni, con una media di tempo di compressione tra i 450 e i 500 millisecondi.

Il seguente estratto di codice mostra la conversione da NV21 a JPEG. La variabile **context** è l'**Activity** dell'applicazione, mentre **data** contiene i dati dell'immagine sorgente. Alla riga 4 è possibile configurare la qualità dell'immagine JPEG prodotta (in questo caso 80 su 100).

```
1 RenderScript rs = RenderScript.create(context);
2 Bitmap bitmap = Nv21Image.nv21ToBitmap(rs, data, width, height);
3 ByteArrayOutputStream stream = new ByteArrayOutputStream();
4 boolean ok = bitmap.compress(Bitmap.CompressFormat.JPEG, 80, stream);
```

## 2.2 Le API `android.hardware.camera2.*`

A partire da Android 5.0, un nuovo insieme di classi permette una gestione più avanzata della fotocamera. L'equivalente della classe **Camera** è **CameraDevice**, che però non espone più i metodi visti nella

<sup>3</sup><https://developer.android.com/reference/android/hardware/Camera>

<sup>4</sup>Una **Activity** rappresenta una pagina dell'applicazione.

<sup>5</sup>Documentati qua: <https://developer.android.com/guide/topics/media/camera#capture-video>

<sup>6</sup>Si tratta in realtà del formato NV21, cioè una variante di Android di Y'CbCr 4:2:0, una codifica del colore lossy anche se con una perdita di dettaglio solitamente impercettibile. Il formato è approfondito nella sezione 7.1.

<sup>7</sup>Il commit del fix è disponibile qua: <https://android.googlesource.com/platform/frameworks/base/+/1c3ded3>

<sup>8</sup>Libreria easyRS: <https://github.com/silvaren/easyrs>

sezione precedente.

Viene introdotto invece il concetto di pipeline, cioè flussi paralleli forniti dal sistema operativo per diversi scopi. Ad esempio, la pipeline di “preview” fornisce un flusso con latenza molto bassa ma trascurando leggermente la qualità dell’immagine. La pipeline per le foto fornisce un’immagine ad alta qualità, mentre la pipeline per la registrazione video offre qualità dell’immagine e scansione costante dei fotogrammi in uscita [6].

Ottenuto il `CameraDevice`, la prima operazione da eseguire è creare una sessione di cattura, alla quale vanno collegati i *target* di acquisizione (metodo `createCaptureSession(...)`). Android utilizza questi target per configurare le pipeline interne e allocare i buffer necessari per la produzione dei fotogrammi [9]. I target possono essere `SurfaceView`, oppure istanze di `ImageReader`, che ricevono i fotogrammi singoli. La sessione creata è immutabile, e cioè non è possibile modificarne i target se non creandone una nuova.

Una volta configurata la sessione, il metodo `createCaptureRequest(int)` esposto dalla classe `CameraDevice` permette di inviare una richiesta di cattura alla fotocamera. Questo meccanismo permette di catturare in qualsiasi istante un fotogramma, che avrà caratteristiche diverse a seconda della pipeline scelta come parametro (per esempio `TEMPLATE_PREVIEW`, `TEMPLATE_STILL_CAPTURE`, `TEMPLATE_RECORD`, ecc.).

Per mostrare l’anteprima su schermo si può utilizzare il metodo `setRepeatingRequest(...)`, che si occupa di inviare richieste di cattura il più frequentemente possibile, con l’effetto di avere il video mostrato in tempo reale su schermo.

Quando si vuole iniziare la registrazione del video, è necessario interrompere la cattura di sessione e crearne una nuova, aggiungendo tra i *target* una `Surface` legata a un’istanza di `MediaRecorder` (il metodo `getSurface()` permette di ottenere l’oggetto). In questo modo la richiesta di cattura invierà i fotogrammi al `MediaRecorder`, che si occuperà di passarli all’encoder e di scriverli su file.

## 2.3 SDK alternativi

Alcuni produttori forniscono degli SDK (Software Development Kit) specifici per utilizzare l’ingresso HDMI su un determinato hardware. In questi casi, le API “camera” di Android potrebbero non essere disponibili.

Inoltre, è spesso possibile sfruttare i `BroadcastReceiver` di Android per ricevere dal sistema eventi come la disconnessione e la riconnessione del cavo HDMI, e gestire quindi la sospensione e la ripresa dell’anteprima su schermo ed eventualmente della registrazione del video. Il seguente blocco di codice mostra come registrare un `BroadcastReceiver` per ricevere determinati eventi. Il `receiver` deve poi essere de-registrato alla sospensione dell’applicazione.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    BroadcastReceiver receiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            // Lettura stato di connessione. Varia in base all'hardware
            boolean isConnected = intent.getBooleanExtra("state", false);
            // Gestire qua connessione/disconnessione
        }
    };

    IntentFilter filter = new IntentFilter("android.intent.action.HDMI_PLUGGED");
    super.registerReceiver(receiver, filter);
}
```

# 3 Acquisizione video RTSP

Come anticipato, il sistema LODE prevede la possibilità di registrare con una videocamera ciò che avviene in aula, in modo da rendere più coinvolgente la fruizione delle lezioni. Per limitare i costi, la videocamera è di tipo IP (Internet Protocol), e permette lo streaming locale in tempo reale tramite il protocollo applicativo RTSP.

## 3.1 Il protocollo RTSP

RTSP (Real Time Streaming Protocol) è un protocollo di rete per il controllo di flussi multimediali. Si dice che è *di controllo* perché non è usato per lo scambio di dati multimediali, ma di messaggi con lo scopo di richiedere determinate operazioni al server. Per fare qualche esempio, è possibile ottenere informazioni sui flussi disponibili, riprodurre un flusso specifico o metterlo in pausa [3].

Il trasferimento vero e proprio dei dati multimediali avviene invece tramite RTP (Real-time Transport Protocol), un protocollo progettato per consentire il trasporto in tempo reale di contenuti video e audio. Le implementazioni di RTP sono tipicamente basate su UDP<sup>9</sup>, protocollo di livello trasporto non connesso e non affidabile particolarmente adatto per situazioni in cui la latenza è più importante dell'affidabilità, ma spesso offrono compatibilità anche con TCP<sup>10</sup>.

## 3.2 Registrazione con ffmpeg

Un flusso video RTSP può essere facilmente registrato tramite lo strumento **ffmpeg**, un progetto open source che incorpora il supporto a numerosi protocolli, formati e codec per l'elaborazione del video e dell'audio. Il video acquisito può essere salvato anche in modalità copia, e cioè salvando il *bitstream* ricevuto (es. formato H.264) senza nessuna elaborazione. Questo permette di evitare la ricodifica del video e di risparmiare risorse hardware.

Può essere inoltre scelto un qualsiasi formato contenitore (es. MP4) compatibile con il codec del video. L'operazione di inserire un *bitstream* in un contenitore si chiama *muxing*, ed è molto leggera a livello di CPU. Una scelta conveniente per il contenitore può essere MPEG-2 Transport Stream, un formato pensato per sistemi di distribuzione che soffrono di perdita di dati (es. la televisione digitale terrestre) e tollerante a interruzioni forzate del muxing, come nel caso di mancanza di corrente.

Il comando seguente acquisisce il video da una videocamera IP e lo salva in un file **out.ts**:

```
> ffmpeg -i rtsp://admin:admin@192.168.178.30:88/videoMain  
        -c:v copy -an out.ts -y
```

In alternativa a MPEG-2 TS si può scegliere anche MPEG-2 PS (Program Stream), che ha un overhead di muxing inferiore per via dell'assenza di controllo degli errori a livello di contenitore:

```
> ffmpeg -i rtsp://admin:admin@192.168.178.30:88/videoMain  
        -c:v copy -an -f vob out.mpg -y
```

In una applicazione Android, è possibile utilizzare dei wrapper appositamente realizzati per sfruttare **ffmpeg** sulle architetture tipiche di Android (tra cui ARM). Per le sperimentazioni è stata presa in considerazione la libreria open source **MobileFFmpeg**<sup>11</sup>, anche per via dell'ottimo stato di mantenimento e aggiornamento del progetto.

**MobileFFmpeg** è fornita in diverse varianti, a seconda delle versioni di Android che si desidera supportare e dei moduli di **ffmpeg** di cui si necessita. Per fare un esempio, il pacchetto identificato come

<sup>9</sup>User Datagram Protocol

<sup>10</sup>Transmission Control Protocol

<sup>11</sup><https://github.com/tanersener/mobile-ffmpeg>

`com.arthenica:mobile-ffmpeg-min:4.2.LTS` è una versione base (`min`) che non include il supporto a nessuna libreria esterna (nessun encoder/decoder), ma che comprende comunque il supporto a RTSP e al muxing. LTS sta a indicare che la versione minima di Android supportata è 4.1, a differenza della versione non LTS che richiede Android 7.0 o superiore.

La libreria è molto semplice da usare e non presenta particolari criticità. Una breve panoramica delle principali funzionalità è inserita nell'allegato A.

## 4 Realizzazione della modalità “kiosk”

Nell'ambito dei sistemi *embedded* si utilizza spesso la locuzione *modalità kiosk* per indicare tutte le situazioni in cui il sistema deve comportarsi come un “chiosco digitale” e limitare l'utilizzo a specifiche funzioni. Su Android si possono adottare diverse tecniche per ottenere una modalità simile, nascondendo quindi la presenza del sistema operativo Android. In questo capitolo sono presentate le tecniche sperimentate, tra cui la modalità a schermo intero, la modifica dell'*overscan* di sistema, la modalità *lock task*, l'avvio automatico dell'applicazione e la modifica dell'animazione di avvio del sistema.

### 4.1 La modalità a schermo intero

A partire da Android 4.1, è stata introdotta una gestione granulare della modalità a schermo intero. Si tratta di base della possibilità di nascondere, utilizzando appositi *flag*, la barra di stato e la barra di navigazione di Android, presenti rispettivamente nella parte superiore e inferiore dello schermo.

```
private void goFullScreen() {
    getWindow().getDecorView().setSystemUiVisibility(
        View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
        | View.SYSTEM_UI_FLAG_FULLSCREEN);
}
```

Questo metodo va chiamato sia all'avvio dell'applicazione che nei casi in cui la modalità a schermo intero potrebbe essere automaticamente disabilitata, ovvero quando l'applicazione perde e poi riacquisisce il “focus”.

```
@Override
public void onResume(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    goFullScreen();
}

@Override
public void onWindowFocusChanged(boolean hasFocus) {
    super.onWindowFocusChanged(hasFocus);
    if (hasFocus) {
        goFullScreen();
    }
}
```

Questa soluzione ha l'effetto di nascondere le due barre di sistema, ma solo fintantoché l'utente non preme un qualsiasi punto dello schermo. Per questo Google ha introdotto anche una “modalità immersiva”, che permette di mantenere l'applicazione a schermo intero fino a quando l'utente non scorre dai bordi dello schermo, come mostrato in seguito.<sup>12</sup>

<sup>12</sup><https://developer.android.com/training/system-ui/immersive>

```

private void goFullScreen() {
    getWindow().getDecorView().setSystemUiVisibility(
        // Nasconde barra di navigazione e di stato
        View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
        | View.SYSTEM_UI_FLAG_FULLSCREEN
        // Modalità immersiva
        View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY
        // Evita lo spostamento del layout della pagina
        | View.SYSTEM_UI_FLAG_LAYOUT_STABLE
        | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
        | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN);
}

```

## 4.2 L'*overscan* di sistema

Come accennato, la modalità a schermo intero di Android permette comunque all'utente di usare le barre di sistema scorrendo dai lati dello schermo, e quindi potenzialmente di uscire dall'applicazione. In un sistema embedded questo non è desiderabile, ed esiste quindi un metodo alternativo per impedire che le barre di sistema vengano mostrate.

La soluzione prevede l'utilizzo della funzione *overscan* del servizio di sistema `WindowManager`, il quale si occupa di gestire la visualizzazione delle finestre, la rotazione dello schermo, le animazioni, le transizioni, ecc.

L'*overscan* di sistema permette di modificare l'area di disegno delle finestre, o in altre parole i margini dello schermo. Impostando un margine negativo l'area di disegno dell'interfaccia di Android sarà estesa oltre l'area visibile dello schermo, con l'effetto di tagliare porzioni dell'interfaccia, come mostrato nella figura 4.1.

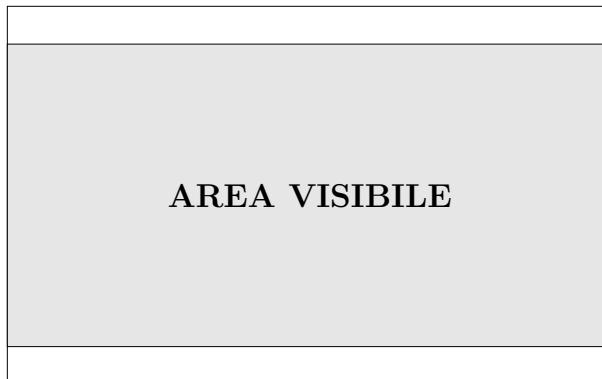


Figura 4.1: Rappresentazione dell'*overscan* di sistema. La finestra sconfina l'area visibile.

Nell'esempio seguente viene utilizzato il comando `wm` (`WindowManager`) della shell di Android per impostare un margine negativo al lato superiore e inferiore dello schermo, in modo da nascondere barra di stato e di navigazione, alte in questo caso 25dp e 48dp<sup>13</sup> (i valori possono essere diversi a seconda del dispositivo).

```

> adb shell wm
usage: wm [subcommand] [options]
wm size [reset|WxH|WdpXHdp]
wm density [reset|DENSITY]
wm overscan [reset|LEFT, TOP, RIGHT, BOTTOM]

> adb shell wm overscan 0,-25,0,-48

```

<sup>13</sup>Density-independent Pixels, un'unità di misura che tiene in considerazione la densità dello schermo.

Una volta configurato l'overscan, l'impostazione dovrebbe essere memorizzata in modo permanente e sopravvivere al riavvio del sistema. Tuttavia, durante le sperimentazioni si è verificato almeno una volta che l'overscan non fosse ripristinato in automatico. Potrebbe quindi essere opportuno eseguire il comando ad ogni avvio del sistema, anche avviando un processo dall'applicazione stessa.

Per completezza, l'overscan può essere disabilitato con questo comando:

```
> adb shell wm overscan reset
```

### 4.3 La modalità “lock task”

In aggiunta ai metodi illustrati nei capitoli precedenti, Android fornisce a partire dalla versione 5.0 una modalità chiamata *lock task*, che fa parte di un insieme più ampio di API per la realizzazione di dispositivi dedicati (in passato chiamati COSU, Corporate-Owned Single-Use). Queste API fanno a loro volta parte di Android Enterprise, che propone soluzioni specifiche per casi d'uso aziendali.

Quando la modalità lock task viene abilitata, il sistema operativo entra in una modalità rigida che permette di usare solo un insieme di applicazioni definite manualmente tramite una *whitelist*. Inoltre, la barra di stato viene disabilitata, le notifiche sono soppresse, e l'utilizzatore non può navigare nel sistema al di fuori delle app inserite in whitelist.<sup>14</sup>

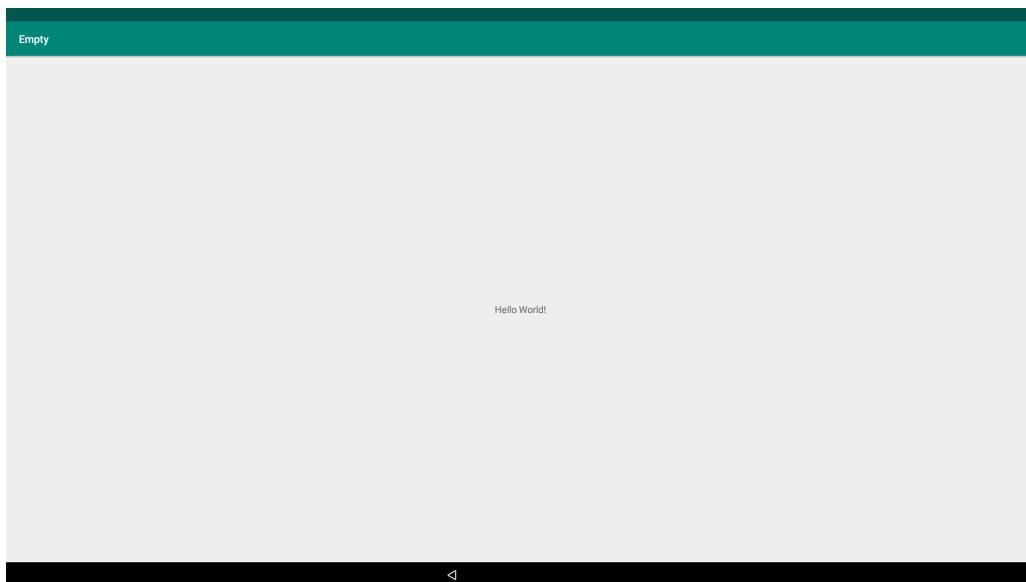


Figura 4.2: In modalità lock task la barra di stato e di navigazione sono disabilitate.

Per implementare la modalità lock task sono necessari due componenti: un Device Policy Controller (DPC) e una **Activity** da lanciare. Il DPC deve essere un'applicazione “proprietaria del dispositivo”, e cioè deve avere dei privilegi speciali per modificare alcune funzioni di sistema. Ha infatti il compito di configurare la modalità lock task elencando i nomi dei *package* da inserire in whitelist.

La procedura per la realizzazione di un DPC è articolata ed è dettagliata nella documentazione di Android<sup>15</sup>, ma è sufficiente sapere che viene fornito un metodo per scegliere quali applicazioni possono essere eseguite in modalità lock task:

```
DevicePolicyManager dpm =  
    (DevicePolicyManager) getSystemService(Context.DEVICE_POLICY_SERVICE);  
ComponentName adminName = getComponentName(this);  
dpm.setLockTaskPackages(adminName, new String[] { "it.unitn.lode" });
```

Google fornisce inoltre un'applicazione chiamata “Test DPC” che implementa numerose funzionalità legate alle policy aziendali, tra cui la configurazione della whitelist. L'applicazione è particolarmente utile per testare le impostazioni.

<sup>14</sup><https://developer.android.com/work/dpc/dedicated-devices/lock-task-mode>

<sup>15</sup><https://developer.android.com/guide/topics/admin/device-admin.html#developing>

te utile durante lo sviluppo, perché permette di effettuare test rapidamente senza sviluppare un DPC. L'installazione è semplice e prevede di impostare "Test DPC" come proprietario del dispositivo:<sup>16</sup>

```
> adb install TestDPC.apk  
> adb shell dpm set-device-owner com.afwsamples.testdpc/.DeviceAdminReceiver
```

A questo punto, la Activity che vuole entrare in modalità lock task può semplicemente chiamare il metodo `startLockTask()`:

```
@Override  
public void onResume() {  
    super.onResume();  
  
    DevicePolicyManager dpm =  
        (DevicePolicyManager) getSystemService(Context.DEVICE_POLICY_SERVICE);  
  
    if (dpm.isLockTaskPermitted(getApplicationContext())) {  
        startLockTask();  
    }  
}
```

Se ben configurata, questa soluzione, combinata con le tecniche mostrate nei capitoli precedenti, permette di avere un'applicazione a schermo intero da cui è impossibile uscire.

## 4.4 L'avvio automatico dell'applicazione

Una Activity può essere configurata per sostituire la schermata "home" di Android, ed essere quindi la prima ad essere mostrata all'avvio del sistema.

Il blocco di codice seguente mostra una porzione del file `AndroidManifest.xml`, tramite il quale è stato forzato il fatto che possa esistere una sola istanza alla volta di `MainActivity`, e che questa debba agire come attività "home" predefinita:

```
<activity android:name=".MainActivity"  
         android:launchMode="singleTask">  
    <intent-filter>  
        <action android:name="android.intent.action.MAIN"/>  
        <category android:name="android.intent.category.LAUNCHER"/>  
        <category android:name="android.intent.category.DEFAULT" />  
        <category android:name="android.intent.category.HOME" />  
    </intent-filter>  
</activity>
```

## 4.5 L'animazione di avvio

Come ultimo aspetto, un requisito di un sistema embedded potrebbe essere quello di non mostrare i marchi di Android o del produttore dell'hardware. Fortunatamente, Android permette in modo abbastanza facile di sostituire l'animazione di avvio del sistema con una a piacere.<sup>17</sup>

In fase di avvio, il sistema legge il file `/system/media/bootanimation.zip` ed estrae una descrizione dell'animazione (`desc.txt`) e un insieme di file PNG rappresentanti i fotogrammi da mostrare in sequenza su schermo.

Per fare un esempio, si prenda come riferimento questo file di descrizione:

```
400 200 10  
c 0 0 part0 #fffff
```

<sup>16</sup><https://codelabs.developers.google.com/codelabs/cosu/index.html#6>

<sup>17</sup><https://android.googlesource.com/platform/frameworks/base/+/master/cmdu/bootanimation/FORMAT.md>

La prima riga indica parametri generali dell'animazione, in particolare la larghezza, l'altezza e il numero di fotogrammi al secondo da renderizzare.

La seconda riga indica invece un blocco di fotogrammi che devono essere eseguiti secondo precise regole. In particolare:

- la lettera **c** indica che l'animazione verrà eseguita fino al suo completamento, anche nel caso in cui il sistema sia pronto prima;
- la prima occorrenza del numero 0 indica quante volte l'animazione deve essere ripetuta, in questo caso infinite volte, mentre la seconda il numero di fotogrammi di attesa prima della riproduzione del blocco successivo;
- **part0** è il nome della cartella in cui trovare la lista di file che compongono l'animazione;
- infine, l'ultimo parametro determina il colore di sfondo nel caso in cui l'animazione sia trasparente o non copra l'intero schermo.

I file ottenuti possono quindi essere inseriti in un archivio ZIP senza compressione, con un comando simile a questo:

```
zip -0qry -i \*.txt \*.png @ ../bootanimation.zip *.txt part*
```

Infine, il file **bootanimation.zip** va caricato sul dispositivo Android tramite **adb** e la modalità debug con accesso root:

```
adb root  
adb remount  
adb push bootanimation.zip /system/media  
adb reboot
```

## 5 Sospensione della registrazione

Durante l'acquisizione di una lezione potrebbero esserci dei momenti di pausa in cui la registrazione video e audio deve essere sospesa. A seconda dei requisiti, la pausa della registrazione può essere richiesta sia durante lo svolgimento della lezione, ad esempio tramite un telecomando, oppure in seguito alla conclusione della lezione in una fase di post-elaborazione.

Ricordando quanto mostrato nei capitoli 2 e 3, prendiamo come riferimento la registrazione effettuata con la classe **MediaRecorder** di Android per l'ingresso HDMI e con **ffmpeg** per quanto riguarda la videocamera RTSP.

A partire da Android 7.0, **MediaRecorder** espone i metodi **pause()** e **resume()** per mettere in pausa e riprendere la registrazione, producendo in ogni caso un solo file video in uscita. Tuttavia, molte board multimediali non supportano Android 7.0. Si aggiunge inoltre il problema di sospendere anche la cattura **ffmpeg**.

Una alternativa è quindi quella di interrompere la registrazione in corrispondenza della pausa, per poi avviare una nuova. Per quanto riguarda il **MediaRecorder** questo si traduce nei metodi **stop()** e **start()**, mentre per **ffmpeg** nella chiusura del processo e l'avvio di uno nuovo.

Alla fine si avranno diversi file corrispondenti agli spezzoni registrati, che possono essere rapidamente uniti ad esempio utilizzando **ffmpeg**. Nel caso in cui i video siano file MP4, l'unione dei file si può ottenere utilizzando il demuxer chiamato **concat** [2], che permette di simulare un input di **ffmpeg** come se si trattasse di un file unico concatenato. Si deve innanzitutto creare un file **list.txt** contenente i percorsi ai segmenti da unire:

```
> cat list.txt
file 'seg1.mp4'
file 'seg2.mp4'
file 'seg3.mp4'
```

Quindi eseguire il seguente comando, dove `-f concat` indica il formato dell'input, in questo caso un input “virtuale” che è ottenuto dalla concatenazione dei file indicati in `list.txt`:

```
> ffmpeg -f concat -i list.txt -c copy merged.mp4
```

Alcuni formati, tra cui MPEG-2 TS, sono progettati per permettere l'unione di più segmenti semplicemente concatenandoli a livello di file. In pratica, in ambiente Linux questo si traduce in:

```
> cat seg*.ts > merged.ts
```

Equivalentemente, con `ffmpeg`:

```
> ffmpeg -i "concat:seg1.ts|seg2.ts|seg3.ts" -c copy merged.ts
```

Un metodo alternativo per ottenere la funzionalità di pausa è quello di registrare la lezione intera e di occuparsi delle pause in una fase di post-elaborazione. Al termine della registrazione si otterrebbe quindi un file video unico, da cui verrebbero rimossi i segmenti indesiderati. Questo consente sia di avere una funzione di pausa/ripresa durante la lezione, sia di poter rimuovere a posteriori sezioni della lezione che non si vuole vengano pubblicati.

La procedura di “ritaglio” del video si può ottenere con diversi strumenti, ma qui viene proposta una procedura che utilizza `ffmpeg`.

Si supponga di avere un video di 30 secondi e che si voglia rimuovere la porzione di video dal secondo 10 al secondo 20. Questo è equivalente a dire che si vuole ottenere un video in cui vengono conservati i segmenti da 0 a 10 secondi e da 20 a 30 secondi del video originale.

Il comando seguente sfrutta le catene di filtri di `ffmpeg` per ottenere il risultato appena detto (la traccia audio viene tralasciata per semplicità).

```
1 > ffmpeg -i input.mp4 \
2     -filter_complex \
3         "[0:v]trim=start=0:end=10,setpts=PTS-STARTPTS[0v]; \
4          [0:v]trim=start=20,setpts=PTS-STARTPTS[1v]; \
5          [0v][1v]concat=n=2:v=1:a=0[ov]" \
6     -map [ov] \
7     -preset veryfast -r 25 -g 250 -crf 25 -refs 1 \
8     -threads 0 out.mp4 -y
```

Ciascuna catena di filtri, separata da un punto e virgola, prende in input una traccia, la elabora con una serie di filtri, e produce un output. Nello specifico, alla riga 3 viene presa come sorgente la traccia video del primo input (`[0:v]`), vengono applicati i filtri `trim` (per ritagliare una porzione del video) e `setpts` (per sistemare i timestamp<sup>18</sup>), e infine il risultato viene etichettato come `[0v]`. La stessa sequenza di operazioni viene ripetuta una seconda volta per ritagliare il secondo segmento di video ed etichettarlo con `[1v]`.

La catena successiva (riga 5) considera i due segmenti etichettati `[0v]` e `[1v]` e applica il filtro `concat`, che concatena i due segmenti specificando anche numero di tracce video e audio risultanti. La traccia ottenuta viene quindi selezionata come traccia del file di output (riga 6), e seguono infine alcuni parametri di codifica per l'encoder `x264` [7].

---

<sup>18</sup>La sigla PTS sta per *presentation timestamp* e indica l'istante temporale in cui un fotogramma deve essere mostrato. Il filtro `trim` non modifica il timestamp dei fotogrammi tagliati, per cui senza il filtro `setpts` l'output non sarebbe quello desiderato.

Con l'aumentare del numero dei segmenti da ritagliare e con l'aggiunta di una traccia audio, il comando `ffmpeg` può diventare complesso da scrivere, leggere e comprendere. La composizione del comando è quindi automatizzabile con uno script che prenda in input gli intervalli da rimuovere e produca in output il testo del comando da eseguire. Il codice che segue è una implementazione di esempio in JavaScript, che è facilmente adattabile per includere anche una traccia audio.

```

1  let segmentsToCut = [ [20, 30] ];
2  let segmentsToKeep = [ [0, segmentsToCut[0][0]] ];
3
4  // Convert [segments to cut] to [segments to keep]
5  for (let i = 0; i < segmentsToCut.length; i++) {
6      let cur = segmentsToCut[i];
7      let next = segmentsToCut[i+1] || [-1, -1];
8
9      segmentsToKeep.push([cur[1], next[0]]);
10 }
11
12 let cmd = 'ffmpeg -i input.mp4 -filter_complex ''';
13
14 // Add segments to trim (keep) to the command
15 segmentsToKeep.forEach((trim, i) => {
16     cmd += ` [0:v]trim=start=${trim[0]} `;
17
18     if (trim[1] != -1) {
19         cmd += ` :end=${trim[1]} `;
20     }
21
22     cmd += `,setpts=PTS-STARTPTS[${i}v];`;
23 });
24
25 // List segments to concatenate
26 for (let i in segmentsToKeep) {
27     cmd += `[${i}v]`;
28 }
29
30 cmd += `concat=n=${segmentsToKeep.length}:v=1:a=0[ov]"`;
31 cmd += ' -map [ov]';
32 cmd += ' -preset veryfast -r 25 -g 250 -crf 25 -refs 1';
33 cmd += ' -threads 0 output.mp4 -y';

```

# 6 Sincronizzazione tra video e audio

Nelle sezioni che seguono viene presentato il problema della sincronizzazione dei flussi video e audio, il cui effetto più evidente è lo sfasamento tra le parole e i movimenti del docente/relatore. I flussi sono acquisiti in modi diversi, per cui la sincronizzazione dell'inizio delle registrazioni non è banale. Viene qui introdotta una soluzione che sfrutta un marcitore visivo per individuare un punto preciso e condiviso tra le registrazioni, in modo da poterle allineare.

## 6.1 Definizione del problema

Come accennato, il problema sorge perché i tre flussi provengono da sorgenti e strumenti di acquisizione diversi. Si tratta in particolare di:

- il video dell'ingresso HDMI, registrato con la classe `MediaRecorder` di Android o eventualmente con SDK dedicati. Nel primo caso si ha un metodo `start()` che si può prendere come istante approssimativo di inizio della registrazione, ma non è possibile ottenere il *timestamp* preciso di inizio effettivo della registrazione (cioè quello del primo fotogramma);
- il video della videocamera IP, registrato con `ffmpeg` tramite il protocollo RTSP. La latenza di inizio registrazione è difficilmente stimabile, e cioè, in altre parole, dopo l'avvio del comando `ffmpeg` non è possibile sapere con precisione l'istante a cui corrisponde il primo fotogramma acquisito. La latenza è infatti influenzata da diversi fattori, tra cui la rete, i tempi di codifica, i buffer di trasmissione e di ricezione. Non è quindi facile trovare un intervallo di tempo abbastanza preciso per sincronizzare il video con gli altri flussi;
- l'audio di un microfono, collegato via cavo direttamente alla board Android. Questo può essere registrato con la stessa istanza di `MediaRecorder` del primo punto, ottenendo quindi una sincronizzazione naturale, almeno a livello teorico.

A questo punto si ha che le due sorgenti acquisite via cavo (HDMI e audio) sono tra loro sincronizzate, mentre resta il problema di allineare il video RTSP con la traccia audio. Uno sfasamento di anche solo 200-300 millisecondi risulta infatti percepibile e può rendere fastidiosa la fruizione del video.

## 6.2 Proposta di soluzione

Con le premesse della sezione precedente, una soluzione percorribile è la sincronizzazione tra il video HDMI e il video RTSP, in modo da ottenere transitivamente una sincronizzazione anche con l'audio.

Questo obiettivo è raggiungibile con una ragionevole precisione utilizzando un riferimento visivo che possa essere proiettato su schermo e quindi catturato dalla videocamera. Questo riferimento dovrebbe essere rilevabile in modo automatizzato, in modo da svolgere la funzione di “ciak”. Nella pratica potrebbe trattarsi di una schermata verde mostrata per un secondo dall'applicazione, in modo simile alle schermate monocolori utilizzate spesso dai proiettori in caso di assenza di segnale.

La figura 6.1 mostra su una linea del tempo gli intervalli delle registrazioni RTSP e HDMI. Supponiamo in questo caso che la registrazione della videocamera parta per prima, e che la linea verticale indichi l'istante temporale in cui il marcitore visivo viene catturato dalla videocamera. Potrebbe quindi trattarsi del momento in cui la schermata verde compare oppure scompare dalla proiezione.

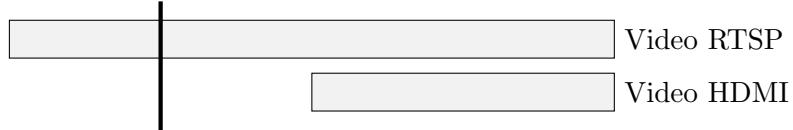


Figura 6.1

Quello che ci interessa ottenere ora è che le due registrazioni video inizino nello stesso istante della linea del tempo. Dobbiamo quindi tagliare un pezzo del video RTSP in modo che il primo fotogramma RTSP corrisponda al primo fotogramma HDMI. Per farlo ci servono due intervalli di tempo, indicati come A e B nella figura 6.2.

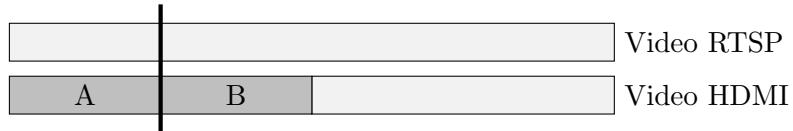


Figura 6.2

Ottenerne l'intervallo B è abbastanza facile, trattandosi di una sottrazione tra *timestamp*. Siamo infatti a conoscenza sia del *timestamp* di avvio della registrazione HDMI sia di quello del riferimento visivo (la linea verticale). L'intervallo B è quindi ottenibile con relativa precisione calcolando la sottrazione tra questi due valori.

L'intervallo A è invece equivalente al tempo trascorso tra l'inizio della registrazione RTSP e l'istante in cui il riferimento visivo compare su schermo. Questo deve essere ricavato analizzando il video tramite strumenti di *computer vision*, come mostrato nella sezione 6.3.

Ottenuti gli intervalli A e B, la loro somma determina lo sfasamento tra video RTSP e HDMI. La rimozione di ( $A + B$ ) secondi dall'inizio del video RTSP permette quindi di sincronizzare i due flussi video e di conseguenza anche l'audio.

### 6.3 Implementazione e sperimentazione

Il primo passaggio per implementare la soluzione è predisporre la visualizzazione del “ciak”. Come accennato può trattarsi di una schermata monocromatica a schermo intero, ottenibile su Android tramite una *View* con uno sfondo colorato:

```
<?xml version="1.0" encoding="utf-8" ?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <View android:id="@+id/rect"
        android:visibility="GONE"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#00ff00" />

</FrameLayout>
```

La *View* è inizialmente nascosta (*GONE*) e può essere resa visibile via codice così:

```
findViewById(R.id.rect).setVisibility(View.VISIBLE);
```

A questo punto è necessario fare uso di timer e ritardi appositamente studiati per ottenere la situazione dello schema della figura 6.1, cioè fare in modo che la videocamera riprenda il momento in cui la schermata verde viene abilitata.

Passando alla fase di post-elaborazione del materiale acquisito, l'obiettivo principale è riuscire a ricavare dalla registrazione RTSP il tempo in cui compare il marcitore visivo, cioè l'intervallo A della figura 6.2. Per fare ciò è possibile usare OpenCV, una nota libreria di visione artificiale.

Più in dettaglio, la tecnica sperimentata consiste nel calcolare la media delle tre componenti di colore RGB per ciascun fotogramma, e di provare poi a rilevare le variazioni significative di verde. Concettualmente, il parametro da considerare per individuare le variazioni è la “distanza” tra il colore verde e le altre componenti di colore, e cioè, in formula,  $G - \frac{R+B}{2}$ . Il blocco di codice che segue calcola questa distanza per ogni fotogramma del video.

```

1  #!/usr/bin/python3
2  import cv2 # pip install opencv-python==4.1.0.25
3
4  vid = cv2.VideoCapture("rtsp.mp4")
5
6  while vid.isOpened():
7      ret, img = vid.read()
8
9      if not ret:
10         break
11
12      average = img.mean(axis=0).mean(axis=0)
13
14      b, g, r = average
15      diff = g - ((r + b) / 2)
16
17      millis = vid.get(cv2.CAP_PROP_POS_MSEC) / 1000
18      print(str(millis) + " => " + str(diff))

```

Per comprendere meglio cosa sta succedendo, conviene usare una rappresentazione grafica dei dati raccolti. La figura 6.3 mostra la variazione nel tempo delle tre coordinate RGB di un video in cui al secondo 6 gran parte dello schermo ripreso diventa verde (figura 6.4). Dal grafico è certamente evidente che al secondo 6 avviene un cambiamento significativo e che il valore del verde aumenta leggermente, ma se si osserva meglio non risulta in realtà così facile individuare con precisione e in modo automatizzato l'istante di inizio e fine della schermata verde.

La figura 6.5 mostra invece la distanza tra la componente verde e la media delle altre componenti, come calcolato alla riga 15 dello script Python. Qua il salto è molto più facile da individuare, anche in modo automatizzato, perché si tratta di un delta di quasi 20 che avviene in modo repentino dopo un intervallo di stabilità.

Per la precisione, la variazione in questo caso avviene leggermente prima del secondo 6, come si evince con evidenza dall'output dello script:

```
[...]
5.747986463620982 => -7.589908854166694
5.81405527354766 => -7.586672634548577
5.880124083474336 => -7.579766167534601
5.946192893401014 => 11.272211914062495
6.012261703327693 => 12.108198242187711
6.078330513254371 => 12.092488064236306
6.144399323181049 => 12.112571614583288
6.210468133107726 => 12.133828125000036
[...]
```

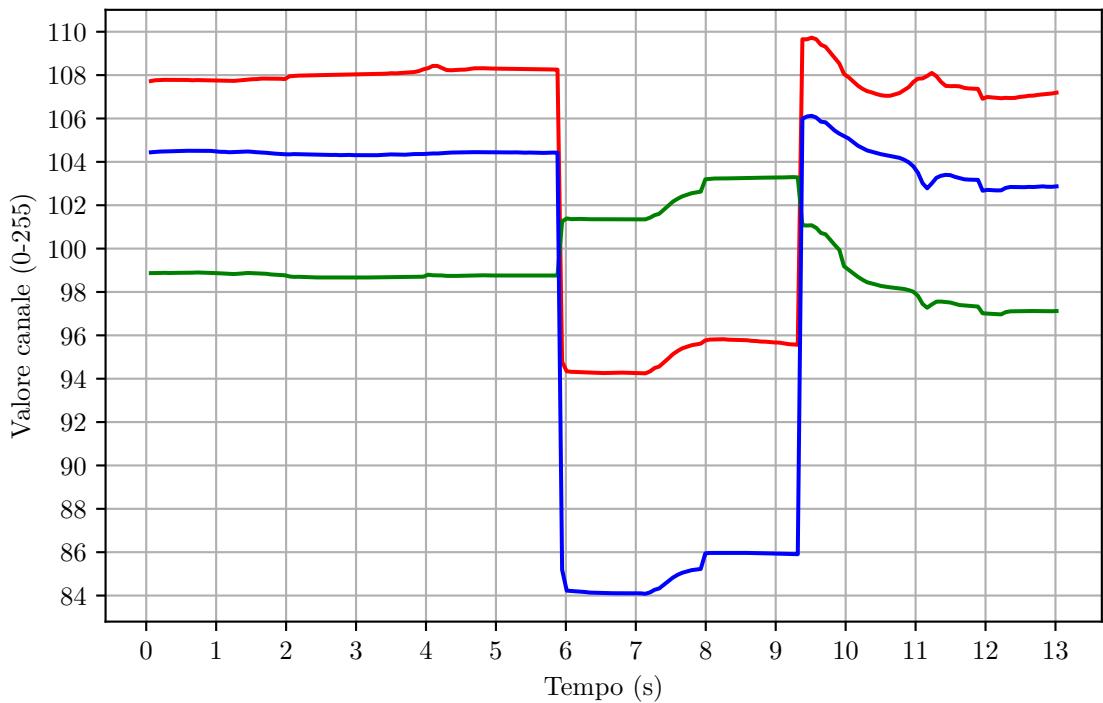


Figura 6.3: Variazione delle componenti colore RGB in relazione al tempo.

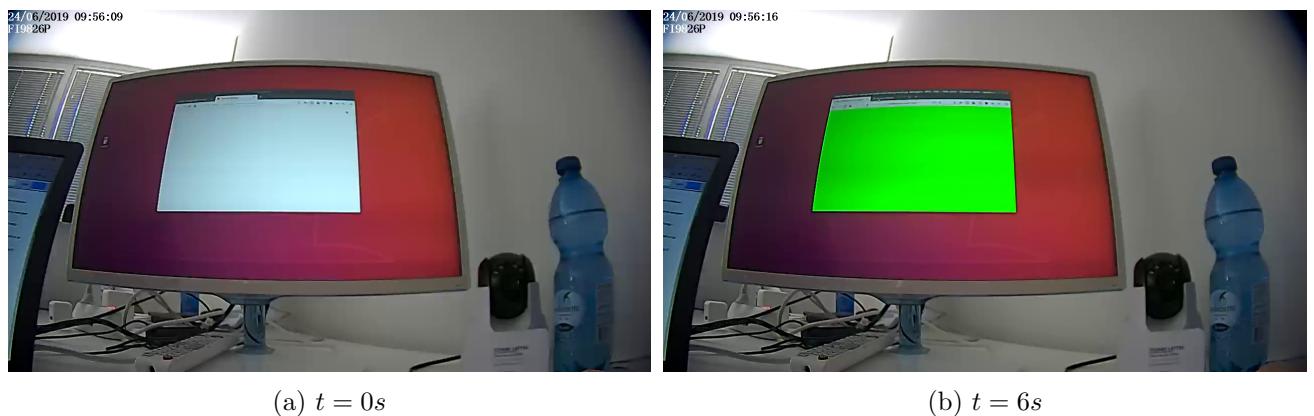


Figura 6.4: Due fotogrammi del video analizzato, rispettivamente il primo fotogramma e un fotogramma di poco successivo al secondo 6.

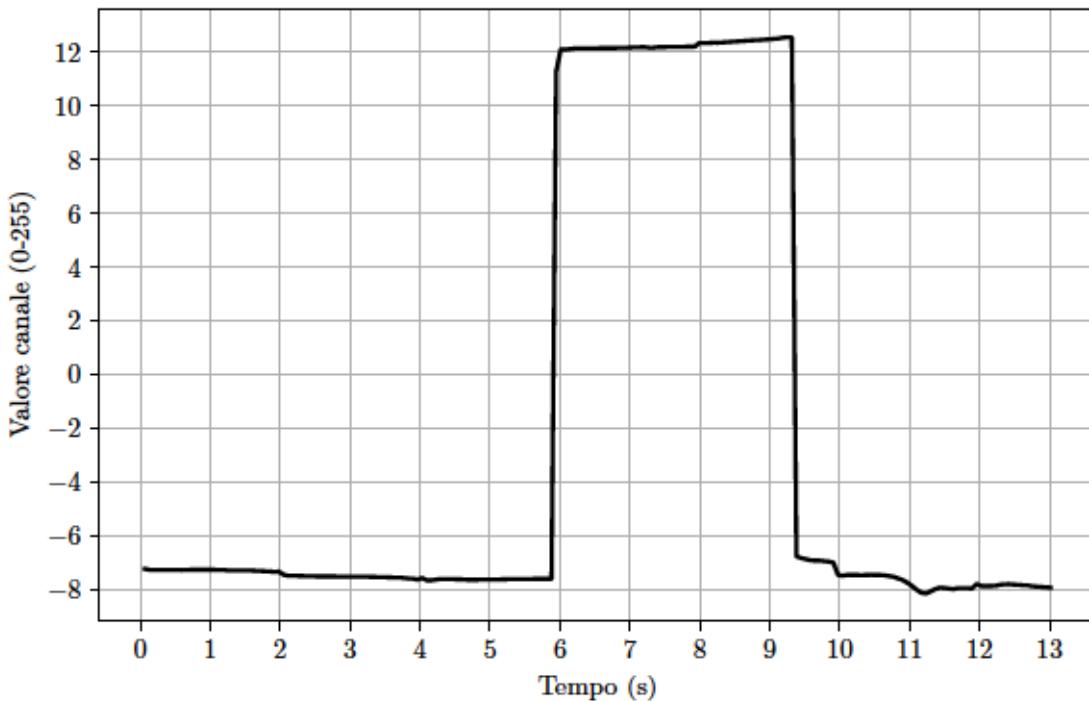


Figura 6.5: Variazione della variabile `diff`, cioè la distanza tra componente verde e la media delle altre componenti, in relazione al tempo.

È quindi 5,94 secondi il valore corrispondente all'intervallo A dello schema 6.2, che può essere sommato a B per ottenere l'intervallo di tempo da tagliare all'inizio del video RTSP.

Supponendo che  $A + B$  dia come risultato 8, possiamo ora ritagliare il video RTSP e ottenere un ragionevole sincronismo tra i due flussi video. Il blocco che segue è una versione minima di un comando che usa `ffmpeg` per rimuovere i primi 8 secondi del file `rtsp.mp4`.

```
> ffmpeg -ss 8 -i rtsp.mp4 -r 25 rtsp-synced.mp4 -y
```

I risultati ottenuti con questa soluzione sono soddisfacenti, con un disallineamento di circa 70-80 millisecondi a favore del video RTSP (il video RTSP risulta più avanti del dovuto). Questo sfasamento pare però essere consistente, ed è probabilmente dovuto al fatto che il `timestamp` memorizzato alla chiamata del metodo `start()` per l'avvio della registrazione HDMI non è preciso. Questo dipende dal fatto che il metodo `start()` ritorna istantaneamente, ma la registrazione effettiva inizia in realtà qualche decina di millisecondi dopo. Considerato che questo ritardo è consistente e l'hardware è fisso, una possibilità potrebbe essere di compensarlo manualmente nel calcolo dell'intervallo  $A + B$ .

## 7 Rilevamento delle differenze tra fotogrammi

Una funzionalità innovativa introdotta dal sistema LODE è la possibilità per gli studenti di acquisire in tempo reale durante le lezioni degli screenshot di quanto viene proiettato, utilizzando un'apposita applicazione web. Dal punto di vista tecnico questo significa che il “box” deve poter acquisire singolarmente dei fotogrammi dall'ingresso HDMI e inviarli al server.

Per evitare che richieste di screenshot molto vicine provochino inutili upload di screenshot, è possibile implementare un sistema per rilevare se uno screenshot è effettivamente nuovo, e in caso negativo riusare quello precedente.

## 7.1 Il formato YUV420SP

Prima di iniziare a lavorare sul confronto delle immagini, è opportuno familiarizzare con il formato YUV420SP, che si incontra frequentemente nello sviluppo Android con il nome di NV12 o NV21.

Prima di tutto, YUV420 è un termine approssimativo per riferirsi a Y'CbCr 4:2:0 [5], un formato di pixel appartenente alla famiglia Y'CbCr, in cui il colore di un pixel è scomposto nelle componenti Y' (luminanza, scala di grigi<sup>19</sup>), Cb/U (proiezione del blu) e Cr/V (proiezione del rosso).

Questa divisione permette di applicare una tecnica chiamata sottocampionamento della crominanza, che riduce la risoluzione delle due componenti della crominanza (Cb e Cr), lasciando a piena risoluzione la luminanza. Questa tecnica è fondata sul fatto che l'occhio umano è molto più sensibile all'intensità di luce che al colore (figura 7.1), per cui anche se riduciamo i "bit" per la rappresentazione della crominanza la differenza è nella gran parte dei casi quasi impercettibile [10]. La grandissima maggioranza dei video normalmente fruibili tramite Internet o la televisione digitale sono codificati con un formato dei pixel Y'CbCr 4:2:0, che è il più diffuso in ambiti non professionali.

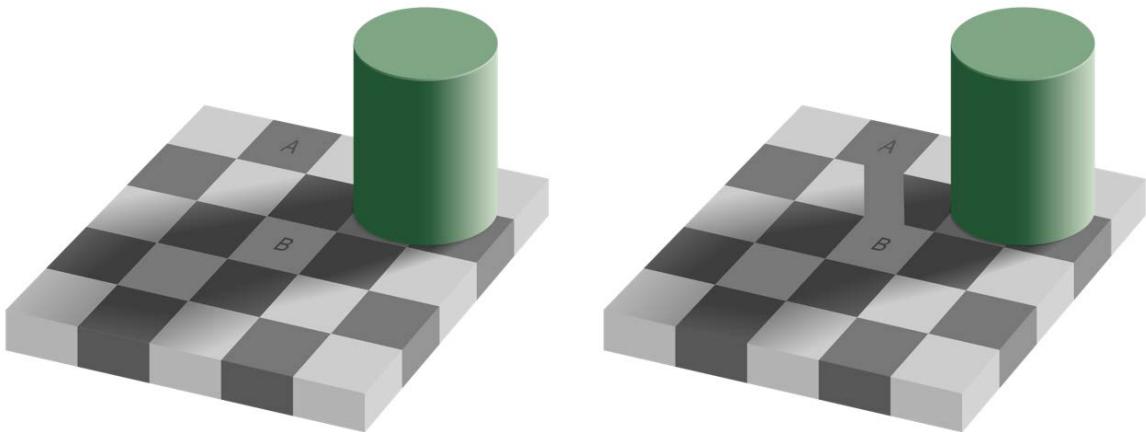


Figura 7.1: Rappresentazione grafica realizzata dal MIT<sup>20</sup> per mostrare come l'occhio umano è molto sensibile all'intensità della luce (luminanza). L'illusione ottica fa credere che i due riquadri A e B siano di sfumature di grigio diverse, quando in realtà sono identici.

La figura 7.2 mostra il sottocampionamento della crominanza 4:2:0 applicato a una griglia di dimensione 4 x 2 pixel. La componente Y, cioè la luminanza, viene campionata a piena risoluzione, e cioè per ogni pixel vengono catturate informazioni piene. L'informazione sulla crominanza, composta dalla componente blu e rossa<sup>21</sup>, viene invece catturata a 1/4 della risoluzione, e cioè è condivisa tra 4 pixel. In confronto a RGB24, questo formato richiede in media 12 bit per pixel anziché 24 (da qui il nome "NV12"), con un risparmio di dati del 50% senza sacrificare visibilmente la qualità complessiva.

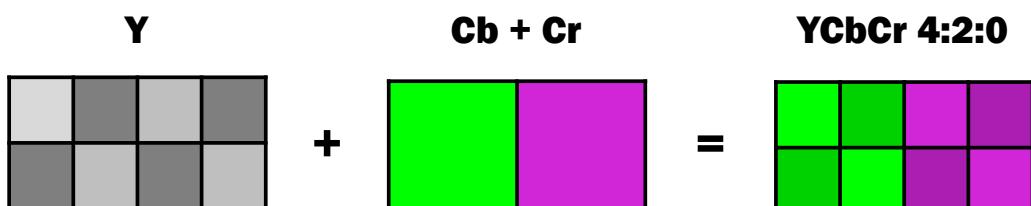


Figura 7.2: Sottocampionamento della crominanza 4:2:0. Il 4 indica la larghezza della griglia, che ha altezza 2 (fissa). Il 2 indica che la risoluzione orizzontale è dimezzata (2 campioni), mentre lo zero che non ci sono campioni diversi tra la prima e la seconda riga.

<sup>19</sup>La suddivisione delle componenti in Y, U e V deriva dal mondo analogico, quando c'era la necessità di supportare sia tv in bianco e nero che a colori. Avere una componente in scala di grigi isolata permette infatti di mantenere la retrocompatibilità pur introducendo il colore. In ambito digitale, è più corretto usare il termine Y'CbCr anziché YUV.

<sup>20</sup><http://persci.mit.edu/gallery/checkershadow>

<sup>21</sup>L'informazione sul verde non è esplicitamente memorizzata ma rappresenta circa il 60% della componente Y: <https://news.ycombinator.com/item?id=1892248>

Nel momento in cui YUV420 deve essere rappresentato sotto forma di bit, i tre piani (Y, U e V) vengono isolati. Ad esempio, un pixel verrebbe rappresentato così (ogni lettera è un bit) [11]:

Mentre in RGB ciascun pixel richiederebbe 8 bit per canale, in YUV420 i due piani della crominanza (blu e rosso) ne richiedono solo 1/4, cioè 2.

Di conseguenza, 2 pixel sarebbero rappresentati così, dove ciascuna coppia di U/V è relativa a un pixel:

Il formato YUV420SP è invece una variante di YUV420 *Semi Planar*, che significa che i piani U e V sono intrecciati in un unico piano. Un pixel YUV420SP (NV12) verrebbe quindi rappresentato così:

E 2 pixel così:

Come accennato, esiste anche il formato NV21, supportato da Android, che è uguale a NV12 ma con la differenza che le componenti U e V all'interno del secondo piano sono scambiate, come mostrato nell'esempio:

Come si nota dagli schemi, in NV12/NV21 le informazioni sulla luminanza sono raggruppate all'inizio e utilizzano 1 byte per pixel. Questa caratteristica tornerà utile nelle prossime sezioni, in combinazione al fatto che Android fornisce le immagini acquisite dall'ingresso HDMI anche in formato NV21.

## 7.2 Confronto pixel per pixel

La prima possibilità esplorata è la più immediata: catturare i fotogrammi in formato “non compresso” (NV21) e confrontarli byte per byte. La prima coppia di byte che ha valori diversi determina l'esistenza di una differenza tra i due fotogrammi.

Ricordando la classe `Camera` di Android introdotta nella sezione 2.1, si può sfruttare il metodo `setPreviewCallback(...)` per ricevere i dati “raw” che vengono mostrati su schermo.<sup>22</sup>

```
camera.setPreviewCallback(new PreviewCallback() {
    @Override
    public void onPreviewFrame(byte[] data, Camera camera) {
        // `data` contiene i dati in formato NV21
    }
});
```

Supponendo ora di avere due array di byte contenenti due fotogrammi catturati in momenti diversi, si può concludere che se le due immagini sono identiche anche i valori dei pixel e quindi i relativi “byte” saranno identici. Di conseguenza, per confrontare i due array si può usare il metodo

---

<sup>22</sup><https://developer.android.com/reference/android/hardware/Camera.PreviewCallback.html>

`Arrays.equals(arr1, arr2)` contenuto in `java.util`, che è implementato con un ciclo che confronta gli array byte per byte e si ferma eventualmente alla prima differenza.<sup>23</sup>

Questo metodo di rilevamento delle differenze è stato verificato e funziona in modo affidabile, ma ha lo svantaggio non trascurabile di non essere molto performante. Su uno degli hardware testati, il confronto di due fotogrammi con risoluzione 1920 x 1080 (e quindi di 3 110 400 byte<sup>24</sup>) richiedeva in media tra i 400 e i 450 millisecondi. Su un'altra board simile il tempo medio risultava di circa 100 millisecondi.

Come accennato nella sezione 7.1, l'occhio umano è più sensibile alla luminanza che alla crominanza. Si può quindi tentare di ridurre la quantità di byte da confrontare selezionando soltanto il piano Y, che rappresenta la luminanza.

In questo caso il confronto va implementato manualmente per considerare soltanto una parte dell'array, quella corrispondente al primo piano, che qua si suppone utilizzi un byte per pixel come nel caso di NV12/NV21:

```
public boolean isLuminanceEqual(byte[] arr1, byte[] arr2, int width, int height) {  
    int maxLength = width * height;  
  
    for (int i = 0; i < maxLength; i++) {  
        if (arr1[i] != arr2[i]) {  
            return false;  
        }  
    }  
  
    return true;  
}
```

Assumendo una risoluzione di 1920 x 1080 pixel, il numero di byte da verificare si riduce a 2 073 600, cioè 2/3 del totale. La riduzione è interessante ma non risolutiva in termini di prestazioni.

### 7.3 Confronto parziale

Una soluzione alternativa a confrontare l'intera immagine è individuare un sottoinsieme di pixel, distribuito in modo da rendere sufficientemente probabile il rilevamento delle differenze.

Si parte dal presupposto che durante una lezione vengano proiettati contenuti le cui variazioni sono abbastanza evidenti. Si pensi ad esempio a una presentazione di slide dove con il passaggio da una pagina all'altra varia l'intero contenuto proiettato o viene aggiunta una riga di testo, oppure a una finestra del browser dove lo scorrimento della pagina provoca lo spostamento di tutto il contenuto.

Si può quindi costruire una griglia virtuale i cui punti di intersezione rappresentano i pixel da confrontare, come mostrato in figura 7.3. L'approccio funziona bene in molti casi: prendendo come riferimento la figura si immagini come l'aggiunta di una riga verrebbe facilmente rilevata dai punti stabiliti (che sono fissi).

Ci sono tuttavia altri casi in cui i cambiamenti potrebbero riguardare delle aree non rilevate dai punti selezionati, come si deduce dalla figura 7.4.

Si possono quindi considerare delle strategie alternative che permettono di aggirare questo problema. Ad esempio:

- il numero di colonne può essere aumentato in modo da rendere la griglia più “fitta” e rilevare non solo aggiunte di frasi ma anche brevi parole o simboli;
- il numero di righe e colonne della griglia può essere ricalcolato ad ogni controllo aggiungendo un fattore di casualità. Se un controllo non riesce a rilevare differenze, la volta successiva potrebbe invece riuscirci, dato che i pixel campione saranno diversi (con ragionevole probabilità);

<sup>23</sup><https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/Arrays.java#l12668>

<sup>24</sup>(1920 · 1080)/2



Figura 7.3: La griglia è composta da 9 righe orizzontali e 49 verticali, per un totale di 441 punti di intersezione.

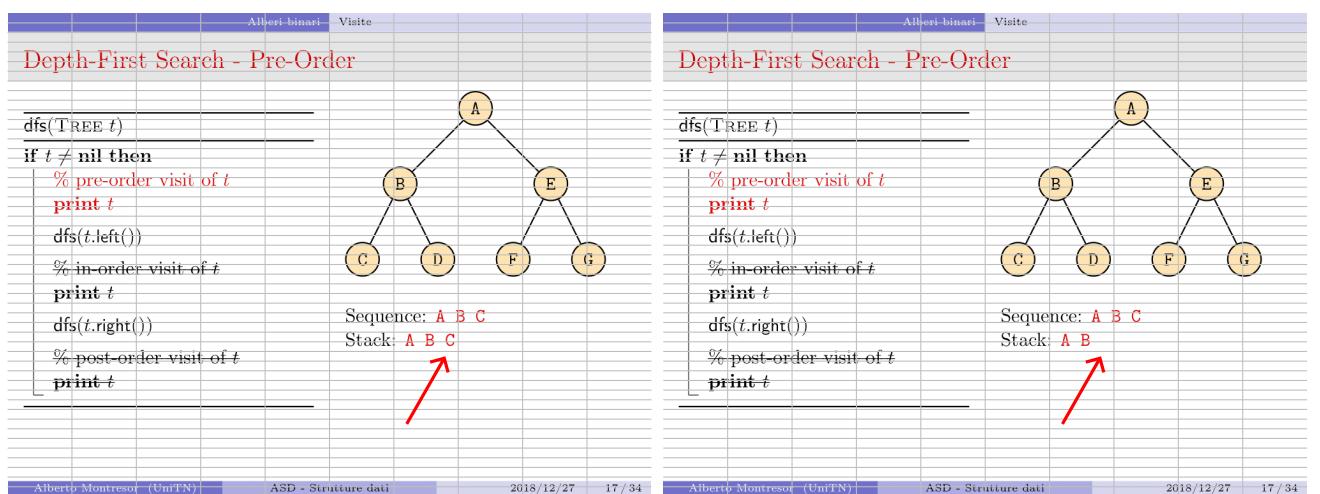


Figura 7.4: Anche in questo caso i punti di intersezione sono 441. Si noti però che l'area puntata dalla freccia presenta delle differenze tra i due fotogrammi che non vengono rilevate. Le immagini sono state gentilmente offerte dal professor Alberto Montresor.

- la distanza orizzontale e verticale tra i punti può essere mantenuta costante, ma si può invece aggiungere uno scorrimento graduale verso destra e verso il basso dei punti, in modo da coprire aree diverse ad ogni controllo;
- la strategia di scegliere un numero casuale di righe e colonne può essere resa più intelligente, per evitare che la stessa coordinata orizzontale e/o verticale venga considerata in diverse combinazioni della griglia.

La strategia del secondo punto è stata sperimentata su diversi contenuti e ha dato ottimi risultati. Gran parte dei cambiamenti venivano rilevati al primo controllo, e in ogni caso entro 3-4 controlli. Nei test l'intervallo di verifica delle differenze era 500 millisecondi, per cui il ritardo massimo di rilevamento risultava di circa 2 secondi.

Nei test effettuati il numero di colonne e righe della griglia veniva scelto in modo casuale ad ogni controllo, scegliendo rispettivamente dall'intervallo [10, 20] e [60, 80]. Con questi dati, nel peggiore dei casi il numero di pixel/byte da verificare è 1 600, un valore estremamente limitato se confrontato con i 3 milioni di byte della sezione 7.2. Le prestazioni di questa soluzione sono risultate di conseguenza ottime, con un tempo di confronto solitamente non superiore a 1-2 millisecondi.

## 7.4 Estrazione slide in post-elaborazione

Un'altra possibile applicazione del rilevamento delle differenze è l'estrazione di “istantanee” dal video registrato, in fase di post-elaborazione. I fotogrammi estratti potrebbe poi essere utilizzati come indici legati a punti specifici del video, consentendo di costruire funzionalità come il salto preciso a una slide, oppure in generale come anteprime rappresentative del video.

Per estrarre tutti i fotogrammi “unici” da un video ci viene ancora una volta in aiuto `ffmpeg`, in particolare con un filtro chiamato `freezedetect` introdotto nella versione 4.2 (agosto 2019).

Il filtro `freezedetect` prevede due opzioni [2]:

- **noise**: indica la soglia di rumore sopra la quale due fotogrammi vengono considerati diversi. Questa soglia può essere specificata in decibel (aggiungendo dB al valore), oppure con un numero decimale nell'intervallo [0, 1]. Il valore predefinito è  $-60dB$ , equivalente a 0,001;
- **duration**: l'intervallo di tempo oltre il quale un fotogramma viene considerato in stato di “freeze”, e quindi invariato secondo il parametro **noise**. Il valore predefinito è 2 secondi.

Il comando che segue configura il filtro `freezedetect` con una soglia di rumore pari a 0,01 e una durata minima di 5 secondi. Questi parametri sono stati ricavati empiricamente utilizzando come input dei video catturati durante reali lezioni universitarie. L'intervallo di 5 secondi è in particolare pensato per evitare che cambi momentanei della schermata vengano considerati rilevanti.

```
> ffmpeg -i input.mp4 \
          -vf "freezedetect=noise=0.01:duration=5,metadata=print:file=log.txt" \
          -an -f null -
```

Nel comando si può notare che oltre a `freezedetect` viene applicato un secondo filtro chiamato `metadata`. Questo filtro si occupa di leggere i valori prodotti dal filtro `freezedetect` e di scriverli in un file chiamato `log.txt`.

I metadati prodotti sono identificati da tre chiavi, in particolare:

- `lavfi.freezedetect.freeze_start`: indica il *timestamp* del primo fotogramma in cui è stato individuato il freeze (inclusi i primi secondi di **duration**);
- `lavfi.freezedetect.freeze_duration`: indica in secondi l'intervallo totale durante il quale il fotogramma è rimasto invariato;
- `lavfi.freezedetect.freeze_end`: indica il *timestamp* dell'ultimo fotogramma dell'intervallo.

Dopo aver eseguito il comando, il file `log.txt` conterrà i dati raccolti. Nell'esempio che segue sono state rilevate 3 istantanee, ai secondi 0, 132 e 200.

```
frame:192 pts:451200 pts_time:5.01333
lavfi.freezedetect.freeze_start=0
frame:5416 pts:11594764 pts_time:128.831
lavfi.freezedetect.freeze_duration=128.831
lavfi.freezedetect.freeze_end=128.831
frame:5764 pts:12334264 pts_time:137.047
lavfi.freezedetect.freeze_start=132.031
frame:8439 pts:18041731 pts_time:200.464
lavfi.freezedetect.freeze_duration=68.433
lavfi.freezedetect.freeze_end=200.464
frame:8648 pts:18493226 pts_time:205.48
lavfi.freezedetect.freeze_start=200.464
```

Figura 7.5: Contenuto del file di output `log.txt`.

Se si vogliono ora estrarre i fotogrammi individuati, ci sono due strade. La prima è quella di ritagliare più volte i primi N secondi del video (con `-ss N`) e poi estrarre il primo fotogramma immediatamente successivo (`-frames:v 1`):

```
> ffmpeg -ss 1 -i input.mp4 \
         -ss 133 -i input.mp4 \
         -ss 201 -i input.mp4 \
         -map 0:v -frames:v 1 out1.png \
         -map 1:v -frames:v 1 out2.png \
         -map 2:v -frames:v 1 out3.png -y
```

Si osservi che l'offset in secondi è stato incrementato di uno e arrotondato per difetto, in modo da evitare eventuali artefatti di codifica dovuti al cambio di scena.

La soluzione precedente ha lo svantaggio di caricare più volte il file in parallelo, causando un uso di memoria RAM molto elevato nel caso in cui i fotogrammi da estrarre siano tanti (più di qualche decina).

L'alternativa è quindi usare il filtro `select`, che permette di selezionare solo determinati fotogrammi dall'input e di passarli all'output. Questo metodo fa un uso molto limitato di memoria ma è anche molto più lento, perché non sfrutta la funzionalità di seeking rapido offerta dall'opzione `-ss`.

Sono inoltre richiesti gli indici dei fotogrammi, anziché i relativi *timestamp*, che vanno quindi estratti dall'output mostrato nella figura 7.5. In compenso non è però necessario aggiungere un offset ai valori, perché gli indici dei fotogrammi sono già relativi a un momento successivo di 5 secondi rispetto all'inizio del freeze.

```
> ffmpeg -i input.mp4 \
         -vf "select='eq(n,192)+eq(n,5764)+eq(n,8648)'" \
         -vsync 0 %d.png
```

In questo caso c'è da notare l'aggiunta dell'opzione `-vsync 0`, che ha lo scopo di disabilitare la duplicazione dei fotogrammi che viene normalmente applicata per simulare un framerate costante.

A questo punto un possibile uso dei file generati è la generazione di un mosaico, con la possibilità di configurare la dimensione delle singole immagini e il numero di colonne e righe della griglia:

```
> ffmpeg -i %d.png -vf "scale=200:-1,tile=5x3" \
         -frames:v 1 tiles.png -y
```

# 8 Acquisizione audio

La funzione di registrazione dell'audio è stata affrontata solo in modo marginale nei capitoli precedenti, ma è in realtà di primaria importanza. La registrazione deve essere infatti affidabile e tollerante a eventuali perdite di segnale in input, e idealmente dovrebbe anche consentire di segnalare in tempo reale il livello di volume, l'assenza di segnale o la presenza di picchi.

Si assume in questo capitolo che la board Android sia dotata di un ingresso audio (es. jack 3,5mm), o in alternativa che permetta di collegare una scheda audio tramite cavo USB. In entrambi i casi l'ingresso dovrebbe essere reso disponibile da Android in modo prioritario e automatico tramite le classi di registrazione di Android.

## 8.1 Registrazione con MediaRecorder

Il `MediaRecorder` è stato accennato nel capitolo 2 nell'ambito della registrazione dell'input video, ma è adatto anche per l'acquisizione e compressione di audio. Il suo uso è molto semplice ed è illustrato nel seguente blocco di codice:

```
final MediaRecorder recorder = new MediaRecorder();
recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
recorder.setOutputFormat(MediaRecorder.OutputFormat.AAC_ADTS);
recorder.setOutputFile("/sdcard/audio/test.aac");
recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
recorder.setAudioChannels(1);
recorder.setAudioSamplingRate(44_100);
recorder.setAudioEncodingBitRate(128_000);

try {
    recorder.prepare();
} catch (IOException e) {
    Log.e(TAG, "MediaRecorder preparation error", e);
    return;
}

recorder.start();
//recorder.stop();
```

In questo esempio l'audio viene acquisito dalla sorgente identificata come `MIC`, e compresso con il codec `AAC-LC`<sup>25</sup>. Tenendo presente che nell'ambito di LODE l'audio registrato è solitamente la voce di una persona che parla, una scelta può essere quella di registrare un solo canale audio con un bitrate abbastanza basso.

Il `MediaRecorder` espone inoltre un metodo `getMaxAmplitude()`, che secondo la documentazione ritorna la *massima ampiezza assoluta misurata dall'ultima chiamata al metodo, oppure 0 alla prima chiamata*<sup>26</sup>. Sfortunatamente, non sono disponibili altri dettagli ufficiali se non supposizioni da verificare empiricamente.

Il metodo ritorna infatti un valore intero a 32 bit con segno, ma a quanto pare contiene in realtà una rappresentazione a 16 bit senza segno (`ushort`) dell'ampiezza del segnale.<sup>27</sup> Questo valore può

<sup>25</sup> Advanced Audio Codec - Low Complexity. È un codec molto diffuso e ampiamente supportato, ed è considerato il successore di MP3. A parità di bitrate, AAC offre una qualità generalmente superiore rispetto a MP3.

<sup>26</sup>[https://developer.android.com/reference/android/media/MediaRecorder.html#getMaxAmplitude\(\)](https://developer.android.com/reference/android/media/MediaRecorder.html#getMaxAmplitude())

<sup>27</sup><https://stackoverflow.com/q/10655703/1633924>

quindi essere la base per un approfondimento, ad esempio per provare a calcolare il livello di pressione sonora in decibel.

## 8.2 Acquisizione *raw* con `AudioRecord`

Nel caso se ne presentasse la necessità, Android offre anche una classe `AudioRecord` che fornisce un accesso più di basso livello all'ingresso audio/microfono. Permette infatti di leggere il segnale PCM (Pulse Code Modulation) non compresso a 16bit, con una frequenza di campionamento a piacere.

Un esempio di codice che mostra alcuni particolari della registrazione PCM con `AudioRecord` è inserito nell'allegato B.

# 9 Conclusione

Nei capitoli di questa tesi sono stati approfonditi i principali aspetti legati allo sviluppo di LodeBox su sistema operativo Android. Il lavoro svolto ha avuto come esito la produzione di diversi “moduli” che risolvono problematiche specifiche, confermando infine la fattibilità di un LodeBox basato su Android.

La prima e principale criticità ha riguardato l’acquisizione del video HDMI del computer del docente (capitolo 2), che deve essere affidabile e allo stesso tempo evitare di far lievitare il costo dell’hardware. Fortunatamente questo punto è stato risolto in modo soddisfacente, grazie all’individuazione di una categoria di dispositivi appositamente studiati per la realizzazione di applicazioni multimediali.

Allo stesso tempo va però notato che ogni *board* si comporta in modo differente, offrendo un diverso supporto alle API `Camera` o a SDK dedicati. La presenza o assenza di questi strumenti può fare la differenza per quanto riguarda la qualità della soluzione sviluppata, ed è quindi consigliabile ottenere dal produttore del dispositivo la documentazione necessaria per assicurarsi dei requisiti dell’ingresso HDMI. Alcune cose da considerare sono ad esempio la versione HDMI supportata, l’eventuale possibilità di acquisire l’audio, il supporto a HDCP<sup>28</sup>, la risoluzione massima, ecc.

Per quanto riguarda l’acquisizione di una videocamera IP tramite protocollo RTSP (capitolo 3), il metodo più facile e affidabile è risultato appoggiarsi a `ffmpeg`, strumento di elaborazione audio-video molto diffuso e ben mantenuto. Non sono risultate criticità nell’adozione di questa soluzione, al di là dei problemi di sincronizzazione intrinseci dello streaming, affrontati nel capitolo 6.

Come terzo componente si ha la funzione di registrazione dell’audio di un microfono (capitolo 8). Anche in questo caso la conclusione è sintetica, perché in presenza di predisposizione hardware la registrazione può essere effettuata senza particolari problemi.

I tre aspetti appena esposti costituiscono il “nocciolo” del sistema LODE. Sono stati però approfonditi altri punti che possono andare a migliorare la qualità complessiva del sistema.

Si tratta innanzitutto di tentare di mascherare la presenza del sistema operativo Android, integrando diverse tecniche per ottenere una modalità *kiosk*. Questa parte introduce una certa complessità nel mantenimento dell’applicazione, ad esempio per la necessità di avere un DPC (Device Policy Controller) per la gestione della modalità lock task.

L’unione di tutti i metodi esposti nel capitolo 4 consente comunque di ottenere un ottimo risultato, in modo che la presenza di un sistema operativo specifico sia quasi completamente nascosta. Fa eccezione la fase di avvio del sistema, durante la quale qualche dettaglio potrebbe svelare la presenza di Android. Per concludere con una sintesi, le tecniche analizzate sono state cinque, e cioè: la modifica dell’animazione di avvio, l’apertura automatica dell’applicazione come schermata home, la modalità lock task (per disabilitare i controlli di sistema), la modalità a schermo intero e l’impostazione dell’overscan di sistema (come alternativa alla modalità lock task).

Il punto successivo riguarda la funzione di pausa della registrazione (capitolo 5), che può essere ottenuta principalmente con due approcci. Il primo consiste nel registrare diversi spezzoni di video

---

<sup>28</sup>High-Bandwidth Digital Content Protection

da unire in post-produzione, mentre il secondo nel registrare un video unico da cui poi rimuovere i segmenti indesiderati.

Entrambi gli approcci hanno i loro vantaggi e svantaggi. Ad esempio, avere una registrazione continua semplifica l'implementazione, ma non è necessariamente applicabile. Durante una lezione potrebbero infatti presentarsi eventi che costringono a interrompere la registrazione, come lo scollegamento del cavo HDMI o il cambio della risoluzione.

D'altra parte, la presenza di segmenti multipli da unire in post-elaborazione rende più difficile l'integrazione con un sistema di sincronizzazione.

Il capitolo 6 ha infatti approfondito un sistema sperimentale per la sincronizzazione di due flussi video (HDMI e RTSP), basandosi sulla presenza di un marcitore visivo per allineare l'inizio delle due registrazioni. Questo metodo utilizza una libreria di *computer vision* come `OpenCV` per estrarre il *timestamp* di questo riferimento, permettendo di sincronizzare i due flussi con un po' di aritmetica e l'aiuto di `ffmpeg`.

Questa tecnica può a prima vista sembrare fragile, e richiede sicuramente sperimentazioni sul campo, ma nei primi test si è rilevata una soluzione sufficientemente precisa per risolvere il problema della sincronizzazione.

Infine, la funzione di cattura e invio di screenshot ad un server in tempo reale ha richiesto di ideare una strategia intelligente per evitare inutili upload di immagini. La soluzione sperimentata prevede un sistema di rilevamento delle differenze tra fotogrammi ed è risultata molto soddisfacente, sia dal punto di vista delle prestazioni che della precisione dei risultati. Come visto nel capitolo 7, la tecnica si basa sulla scelta della strategia migliore per la selezione di alcuni pixel “importanti” da mettere a confronto. L'obiettivo è duplice: ridurre il più possibile il tempo di confronto mantenendo allo stesso tempo una probabilità sufficientemente alta di rilevare la presenza di differenze.

Complessivamente mi ritengo molto soddisfatto del lavoro svolto, che mi ha consentito da un lato di approfondire un ambito particolare come lo sviluppo di sistemi embedded/IoT, e dall'altro di espandere e consolidare le mie conoscenze in termini di acquisizione e elaborazione del video digitale.

# Bibliografia

- [1] Documentazione Android. <https://developer.android.com>. Ultimo accesso 25/08/2019.
- [2] Documentazione ffmpeg. <https://ffmpeg.org/ffmpeg-all.html>. Ultimo accesso 20/08/2019.
- [3] Real Time Streaming Protocol (RTSP). RFC 2326, RFC Editor, aprile 1998.
- [4] Jones Adam. Android system API YuvImage. compressToJpeg has a memory leak at the native level. <https://codar.club/blogs/android-system-api-yuvimage-compressstojpeg-has-a-memory-leak-at-the-native-level.html>, gennaio 2019. Ultimo accesso 14/05/2019.
- [5] Poynton Charles. YUV and luminance considered harmful. [https://poynton.ca/PDFs/YUV\\_and\\_luminance\\_harmful.pdf](https://poynton.ca/PDFs/YUV_and_luminance_harmful.pdf), 2008. Ultimo accesso 24/08/2019.
- [6] Google. Build a universal camera app (Google I/O '18). <https://youtu.be/d1gLZCSLmaA>, maggio 2018.
- [7] Ozer Jan. *Video Encoding by the Numbers. Eliminate the Guesswork from your Streaming Video*. Doceo Publishing, dicembre 2016.
- [8] Ronchetti Marco. Strumenti per favorire l'apprendimento nei primi anni di università. Parte 2: Il sistema LodeBox. In Raffone Alessandra, editor, *La città educante. Metodologie e tecnologie innovative a servizio delle smart communities*, pages 137–146. Liguori, 2018.
- [9] Wahltinez Oscar. Understanding Android camera capture sessions and requests. <https://medium.com/androiddevelopers/understanding-android-camera-capture-sessions-and-requests-4e54d9150295>, settembre 2018. Ultimo accesso 25/06/2019.
- [10] Bradley Stevens. Why luminance is the key component of color. <http://vanseodesign.com/web-design/color-luminance/>, dicembre 2014. Ultimo accesso 24/08/2019.
- [11] VideoLAN. YUV - VideoLAN wiki. <https://wiki.videolan.org/YUV>. Ultimo accesso 24/08/2019.

# Allegato A La libreria MobileFFmpeg

La libreria MobileFFmpeg permette di eseguire istanze di `ffmpeg` su dispositivi Android, rendendo trasparente la gestione del loro ciclo di vita.

Come dettagliatamente indicato dalla documentazione<sup>29</sup>, esistono 8 varianti della libreria, a seconda delle librerie incluse (`min`, `min-gpl`, `https`, `https-gpl`, `audio`, `video`, `full`, `full-gpl`). Essendo interessati soltanto all'acquisizione di un flusso tramite il protocollo RTSP, la versione `min` è sufficiente.

Il pacchetto può essere configurato per l'installazione modificando il file `build.gradle` del modulo dell'applicazione, aggiungendo una nuova riga all'interno del blocco `dependencies`.

```
dependencies {
    implementation com.arthenica:mobile-ffmpeg-min:4.2.LTS
}
```

In questo caso è stata scelta la versione LTS, in modo da supportare versioni di Android inferiori a 7.0.<sup>30</sup>

Il comando della sezione 3.2 può essere a questo punto acquisito come segue:

```
String command = "-i rtsp://admin:admin@192.168.178.30:88/videoMain " +
    "-c:v copy -an /sdcard/out.ts -y";

FFmpeg.execute(command);
```

Il metodo `execute` è bloccante, e ritorna solo quando il relativo processo `ffmpeg` termina, per cui deve essere eseguito all'interno di un thread. L'acquisizione può essere poi manualmente terminata in modo “gentile” chiamando `FFmpeg.cancel()`.

Per monitorare l'esecuzione di `ffmpeg`, la libreria mette a disposizione la possibilità di registrare due callback, una per ricevere l'output del processo e una per le statistiche “parsate”:

```
Config.enableLogCallback(new LogCallback() {
    public void apply(LogMessage message) {
        String text = message.getText();
    }
});

Config.enableStatisticsCallback(new StatisticsCallback() {
    public void apply(Statistics stats) {
        // getVideoFrameNumber(), getVideoFps(), getSize(),
        // getTime(), getBitrate(), getSpeed()
    }
});
```

Un punto da precisare è che questa libreria non consente l'esecuzione di istanze multiple di `ffmpeg`, ma esistono delle alternative<sup>31</sup> (non sperimentate) che permettono di farlo usando un approccio implementativo diverso.

<sup>29</sup><https://github.com/tanersener/mobile-ffmpeg/wiki/Packages>

<sup>30</sup><https://github.com/tanersener/mobile-ffmpeg/wiki/LTS-Releases>

<sup>31</sup><https://github.com;bravobit/FFmpeg-Android>

# Allegato B Acquisizione audio PCM

Il blocco di codice che segue configura e avvia l'acquisizione della sorgente audio di default del dispositivo, impostando come formato PCM 16bit a un canale con frequenza di campionamento 44,1kHz.

```
1 final int SAMPLING_RATE_IN_HZ = 44100;
2 final int CHANNEL_CONFIG = AudioFormat.CHANNEL_IN_MONO;
3 final int AUDIO_FORMAT = AudioFormat.ENCODING_PCM_16BIT;
4
5 final int BUFFER_SIZE_FACTOR = 2;
6 final int BUFFER_SIZE = BUFFER_SIZE_FACTOR *
7     AudioRecord.getMinBufferSize(SAMPLING_RATE_IN_HZ, CHANNEL_CONFIG, AUDIO_FORMAT);
8
9 AudioRecord recorder = new AudioRecord(
10     MediaRecorder.AudioSource.DEFAULT,
11     SAMPLING_RATE_IN_HZ,
12     CHANNEL_CONFIG,
13     AUDIO_FORMAT,
14     BUFFER_SIZE);
15
16 recorder.startRecording();
17 isRecording = true;
```

L'acquisizione vera e propria dei campioni audio avviene però in un thread separato, in cui vengono caricati i dati audio in un buffer, a ciclo continuo. Un particolare da notare è che il tipo del buffer è `short[]`, perché deve contenere i valori dell'ampiezza del segnale a 16 bit.

Questi valori possono poi essere elaborati per ricavare una media e calcolare misure come il livello di pressione sonora o la “loudness” in LUFS, secondo la raccomandazione EBU R128<sup>32</sup>.

Nel momento in cui il buffer viene scritto su file (metodo `writeShortArrayToFile`), deve essere però effettuata la conversione in byte, assicurandosi di usare l'ordine dei byte “little endian”, il più comune nell'ambito dell'audio PCM.

```
1 Thread recordingThread = new Thread(new RecordingRunnable(), "RecordingThread");
2 recordingThread.start();

1 class RecordingRunnable implements Runnable {
2     @Override
3     public void run() {
4         final File file = new File("/sdcard/audio/test.pcm");
5
6         short[] buffer = new short[BUFFER_SIZE];
7
8         try (final FileOutputStream outStream = new FileOutputStream(file)) {
9             while (isRecording) {
```

<sup>32</sup><https://tech.ebu.ch/docs/r/r128.pdf>

```

10             int readSize = recorder.read(buffer, 0, buffer.length);
11
12         if (readSize < 0) {
13             throw new RuntimeException("Reading of audio buffer failed");
14         }
15
16         writeShortArrayToFile(buffer, outStream, readSize);
17     }
18
19     outStream.close();
20 }
21
22 private void writeShortArrayToFile(short[] buffer,
23                                     FileOutputStream outStream,
24                                     int readSize) {
25     try {
26         ByteBuffer bb = ByteBuffer.allocate(Short.SIZE / Byte.SIZE * readSize);
27         bb.order(ByteOrder.LITTLE_ENDIAN);
28         ShortBuffer ss = bb.asShortBuffer();
29         ss.put(buffer, 0, readSize);
30         outStream.write(bb.array(), 0, bb.limit());
31     } catch (IOException e) {
32         throw new RuntimeException(e);
33     }
34 }
35 }
36 }
```