Georg Vassilev

Developing Digital Audio Workstation for Android

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

25 April 2023

Abstract

Author:	Georg Vassilev
Title:	Developing Digital Audio Workstation for Android
Number of Pages:	36 pages
Date:	25 April 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Applications
Supervisors:	Hannu Markkanen, Main Evaluator
	Anne Pajala, Language Advisor

The objective of this thesis was to analyse the current condition of hardware audio processing technologies and audio software development opportunities for Android devices. This study was carried out for a mobile audio workstation application development company. The starting point of the research was to investigate the basic low-level mechanics of the devices used for audio capturing and subsequent processing, and to explore the middleware and high-level solutions that are readily available for developers in the Android Studio.

This thesis is based on a thorough exploration of literature related to audio processing technologies and Android app development. In addition, various tools and programs such as MediaRecorder, AudioTrack, and MediaPlayer were used to carry out simulations and tests.

The outcome of this study is a mobile audio workstation application that is capable of recording audio samples with the built-in microphone and organising them into virtual channels for easy post-processing. The application focuses on fast and straightforward mixing and mastering and allows users to record sounds, apply effects and modulations, and combine them into songs. In addition, users can record audio over externally imported audio tracks, making the application useful for karaoke-style recordings.

Keywords: Audio Applications, Audio Processing, Android, Kotlin

List of Abbreviations

DAW: Digital Audio Workstation

PCM: Pulse-code modulation

EQ: Equalization

ORM: Object-relational mapping

WAV: Waveform Audio File Format

MVVM: Model-View-ViewModel

MVC: Model-View-Controller

DAO: Data Access Object

API: Application Programming Interface

UI: User Interface

FFT: Fast Fourier Transform

Contents

1 Introduction 1
2 Quality of the Sound 3
2.1 Sampling Properties 3
2.2 Bit Depth 5
3 Audio Reproduction and Common Audio Processing Tools 7
3.1 Monophonic and Stereophonic Audio Reproduction 7
3.2 Effects 7
3.2.1 Equaliser 8
3.2.2 Compressor 8
3.2.3 Reverb 9
4 Audio Processing in Android 11
4.1 MediaPlayer 11
4.2 MediaRecorder 11
4.3 AudioTrack 12
4.4 AudioRecord 13
5 Creating Audio-Processing Application 14
5.1 Application requirements 14
5.2 Application architecture 15
5.2.1 MVVM Pattern 15
5.2.2 MVC Pattern 15
5.2.3 Application structure 16
5.2.3 Database 18
5.2.4 ViewModel 20
5.3 Audio processing overview 21
5.4 Effects implementation 24
5.4.1 Equaliser 25
5.4.2 Compressor 26
5.4.3 Reverberator 26
5.5 Mixing 27
5.6 Application User Interface 28
5.6.1 LibraryFragment 28
5.6.2 TrackListFragment 30
5.6.3 EffectFragment 33
6 Results 34
6.1 Testing and application problems 34
6.2 Future of the application and upcoming features 35
Conclusion 36
References 37

1 Introduction

Mobile devices have evolved from simple communication tools to powerful computers capable of complex operations and data processing. The use of sensors and interactive interfaces allows mobile devices to collect various types of data from the environment and provide endless opportunities for creating digital content. [1 pp. 56-64]

The purpose of this thesis is to explore the current capabilities of mobile devices for audio production and to develop an application that leverages basic Android Studio libraries and Kotlin to provide an easy-to-use and efficient tool for mobile audio recording. This project was carried out for the teacher giving private singing lessons who complained about the lack of tools for quick and easy multi-channel audio recording and processing for his lessons.

This thesis aims to answer the question: How can mobile devices be used for audio production, and what is the potential of Android devices and Kotlin programming language for developing an efficient and user-friendly audio recording application. The intended audience for the audio recording application includes individuals or organisations seeking a solution to produce mixed audio recordings for a range of purposes, such as entertainment, education, or professional use.

Audio production encompasses a wide range of activities, including recording, editing, mixing, and mastering audio content. While there are some audio recording and editing applications available on Android, they are often limited in their functionality and are not suited for professional use. Additionally, the lack of dedicated hardware for audio production on Android devices further limits the capabilities of these applications.

Despite these limitations, there are still some options available for users who want to produce audio content on their Android devices. Some applications offer basic recording and editing functionality, while others provide more advanced

features such as multitrack mixing, effects processing, and MIDI support. Some Android devices also have built-in microphones that are capable of recording high-quality audio, making them suitable for on-the-go recording [2 pp. 447-458].

After this introduction, the thesis will provide an overview of the current state of mobile audio production and the limitations of existing solutions. The potential of Android devices and Kotlin for audio production will be discussed, and the development process of our audio recording application will be presented. Finally, the performance of the application will be evaluated, and potential improvements and future work will be discussed.

2 Quality of the Sound

Digital audio recording involves converting analog sound waves into digital signals using analog-to-digital converters. This process has revolutionised audio capture and manipulation, enabling higher fidelity, accuracy, and flexibility. Digital audio can be easily edited and processed using software tools, allowing for greater creativity and experimentation.

This section discusses two important aspects of digital audio quality: sampling properties and bit depth. Sampling refers to the process of converting continuous analog sound waves into discrete digital signals, while bit depth determines the resolution and dynamic range of the digital signal. Understanding these concepts is crucial for achieving high-quality digital audio recordings.

2.1 Sampling Properties

The main goal of digital audio is to reproduce the qualities of the analog sound wave in a way that our perception is not able to recognize the difference. Human ear is capable of hearing the audio frequencies lying in a range of 20 Hz to 20 000 Hz. This brings us to the concept of sampling.

A sample is a value of a signal at a specific point in time and space. To capture samples from a continuous signal, a sampler is utilised, which discretizes the signal at specified intervals. Ideally, a perfect sampler would create samples at precise points that match the instantaneous value of the continuous signal. By combining a sequence of these samples, the original signal can be reconstructed up to a specific frequency limit known as the Nyquist limit. To achieve this, the sequence of samples is passed through a low-pass filter to reconstruct the continuous signal with the highest possible fidelity [3 pp.482].

Sampling in signal processing refers to the process of converting a continuous-time signal into a discrete-time signal, such as converting a sound

wave into a series of "samples". This usage of the term differs from its usage in statistics. When a system performs multiple measurements, snapshots can be taken to accurately rebuild the resolution and complexity of an analog wave, provided that a sufficient number of measurements with an adequate range of amplitude values is taken into account. The influence of the sampling rate on the process of converting analog signal to digital is depicted in figure 1.

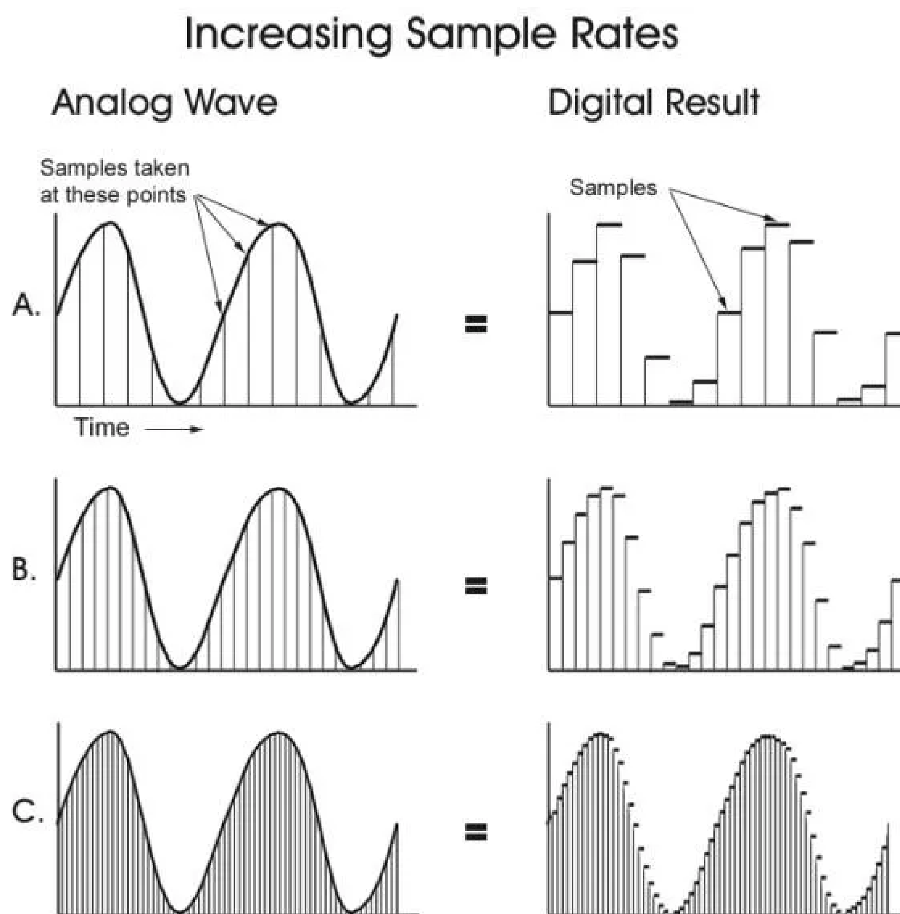

Figure 1: Sampling rate visualisation.[3]

Figure 1 provides a visualisation of the sampling process and demonstrates how analog audio signals can be accurately represented using a high enough sample rate.

2.2 Bit Depth

Digital audio recording involves the conversion of continuous analog sound waves into discrete digital signals. One of the most important factors that determine the quality of digital audio is the bit depth. Bit depth refers to the number of bits of information that are used to represent the amplitude of each sample of the audio signal. In other words, it determines the resolution of the digital signal.

The number of possible amplitude levels that an analog audio wave can have is infinite. However, digital audio samples must be specified with a finite value for the amplitude. The bit depth determines the number of possible amplitude values that can be recorded for each sample. The most popular bit depths for digital audio are 16-bit, 24-bit, and 32-bit [3 pp. 490].

Higher bit depth means that more possible values can be expressed, resulting in a wider dynamic range of the signal. For instance, 16-bit audio can record 65,536 amplitude values per sample, while 24-bit audio can record 16,777,216 values per sample, and 32-bit audio can record 4,294,967,296 values per sample. The wider the dynamic range, the greater the range of volume levels that can be recorded, resulting in higher quality audio recordings.

The dynamic range is a crucial factor in determining the quality of digital audio recordings. The maximum dynamic range for 16-bit digital audio is 96dB, while the maximum dynamic range for 24-bit audio is 144dB. A bit depth of 16-bit for a sampling rate of 44.1kHz is sufficient to reproduce the dynamic range and audible frequency for the average listener. This is why it has become the standard CD format [3 pp. 112-113].

In summary, the bit depth is a critical factor in digital audio recording that determines the resolution of the digital signal and the dynamic range of the audio. The higher the bit depth, the wider the range of possible amplitude values that can be recorded, resulting in higher quality audio recordings.

3 Audio Reproduction and Common Audio Processing Tools

In this chapter, the topics of audio reproduction and common audio processing tools will be covered. Monophonic and stereophonic audio reproduction will be discussed, with a focus on the importance of spatial fidelity in perceived sound quality. The chapter will also cover three commonly used audio processing tools: equalisers, compressors, and reverberators.

3.1 Monophonic and Stereophonic Audio Reproduction

Neither directional nor spatial information is provided in monophonic audio reproduction. Spatial effects can be produced by including a second channel in the sound system. Rumsey discovered that 30% of the perceived sound quality is due to spatial fidelity [4 p. 12].

First off, despite the fact that there is only one microphone, Android appears to just accept the app's request to record in stereo. The audio signal will be precisely duplicated, resulting in identical data on the left and right channels. The resulting file will have the exact identical waveform for both channels and will still be in stereo.

Second, although some devices include many microphones, they are not designed for stereo recording. They're there to help with noise reduction. By comparing the differences in the audio signals coming from the two microphones, they can extract just the difference and deliver it to the apps. Most of the new Android devices support stereo recording [5].

3.2 Effects

This section explores three commonly used audio processing tools: equalisers, compressors, and reverberators. Each tool has a unique function and can significantly affect the overall sound of an audio signal.

3.2.1 Equaliser

An equaliser (EQ) is a digital signal processing tool that allows for the adjustment of the frequency response of an audio signal. It enables the user to selectively boost or cut specific frequency ranges, thus shaping the overall tonality of the sound. The equaliser achieves this by splitting the audio spectrum into several frequency bands, each of which can be independently boosted or attenuated. The specific range of frequencies that can be adjusted depends on the number of bands available in the equaliser.

Equalisers are commonly used in music production, sound reinforcement, and home audio systems. They can be used to correct or enhance the tonal balance of an audio system, compensate for the acoustic properties of a room, or creatively manipulate the sound to achieve a desired effect. Different equaliser settings can also be used to tailor the sound to different genres of music or to suit personal preferences [6 pp. 173-175].

3.2.2 Compressor

A compressor is an audio processing tool used to reduce the dynamic range of an audio signal by attenuating the level of the louder portions of the signal while leaving the quieter portions relatively unchanged. This results in a more consistent overall loudness, making the audio easier to listen to and reducing the risk of distortion or clipping in audio playback systems. Compressors are commonly used in music production, broadcasting, and live sound reinforcement.

According to a study by Murphy [7 p. 2249], compressors are particularly effective in controlling the level of vocals in music production, allowing for a more consistent and balanced sound. Compressors can also be used to increase the perceived loudness of a signal by raising the level of quieter portions of the signal while attenuating the louder portions, a process known as "upward compression".

Compressors typically have several parameters that can be adjusted, including the threshold, ratio, attack time, and release time. The threshold determines the level at which compression begins to occur, while the ratio determines the amount of attenuation applied to the signal above the threshold. The attack time determines how quickly the compressor reacts to changes in the signal level, while the release time determines how quickly the compressor returns to normal operation after the signal falls below the threshold.

Overall, compressors are a powerful tool for controlling the dynamic range of audio signals, allowing for a more consistent and pleasant listening experience.

3.2.3 Reverb

A reverberator is an audio effect that simulates the sound of a physical space, such as a concert hall, by adding a series of reflections to an audio signal. This creates a sense of space and depth in the sound, as well as blending the various elements of the mix together.

According to Smith [8], a reverberator works by processing the audio signal through a series of delay lines and feedback loops. Each delay line simulates a reflection of sound off a surface in the simulated space, and the feedback loops simulate the decay of the sound over time as it bounces around the space. By adjusting the parameters of the delay lines and feedback loops, the user can control the character and size of the simulated space.

Reverberators can be used in a variety of applications, from music production to sound design for film and television. They can be used to create a sense of space and ambience in a mix, to simulate specific environments, or to add a creative effect to a sound.

There are many different types of reverberators, including algorithmic reverbs, convolution reverbs, and plate reverbs [9 p. 86]. Each type has its own unique character and set of parameters for controlling the sound.

Overall, reverberators are a powerful tool for adding depth and space to audio signals, and are an essential part of the audio engineer's toolkit.

4 Audio Processing in Android

Android has an extensive audio framework that includes various Application Programming Interfaces (API) for tasks such as audio capture, playback, effects, and routing. While the basic functionalities are already supported by Android's framework, for more advanced audio manipulation, custom solutions or third-party libraries are necessary. This chapter provides an overview of the audio processing frameworks available in Android.

4.1 MediaPlayer

MediaPlayer is a built-in Android class that provides a convenient way to play audio and video files within an Android application. It is a high-level API that simplifies the process of playing media files and provides various features for controlling playback.

MediaPlayer supports a wide range of media formats. It also supports streaming media from the internet or other sources.

In addition to basic playback controls, MediaPlayer provides features such as volume control, playback speed control, and looping. It also supports audio focus, which allows the app to handle audio playback when other apps or system sounds are playing.

Overall, MediaPlayer is a powerful and versatile class that simplifies the process of playing media files within an Android application [10 p.125].

4.2 MediaRecorder

MediaRecorder is another built-in Android class that provides a convenient way to record audio and video files within an Android application. It is a high-level

API that simplifies the process of recording media files and provides various features for controlling the recording process.

In addition to basic recording controls, MediaRecorder provides features such as audio and video quality settings, bitrate control, and file size limit. It also supports audio source selection, which gives an opportunity to choose the microphone or other audio sources for recording.

In summary, while the Android MediaRecorder provides a simple and convenient way to record audio and video on Android devices, it is limited in its sound processing capabilities. To apply additional sound processing to audio data, developers must use the lower-level API and have access to specific data buffers before the audio files are saved or played.

4.3 AudioTrack

PCM stands for Pulse Code Modulation, which is a method used to digitally represent analog signals such as sound. The Android AudioTrack class is used to play and control a single PCM audio resource. If more than one AudioTrack object is used, PCM audio buffers can be fed to Android's digital audio memory pool for layered playing.

AudioTrack has two operating modes: static and streaming. In the streaming mode, a continuous stream of data is written to an AudioTrack object. This mode is suitable for dealing with longer sounds or when data is being streamed from a network source. However, for brief sounds that can be stored in memory and require low latency to be played, the static mode is preferred. This mode is ideal for user interface feedback or game audio that is frequently activated by the end user.[11]

4.4 AudioRecord

For Java programs to record audio from the platform's audio input devices, the AudioRecord class maintains the audio resources. By "drawing" (reading) the data from the AudioRecord object, this is accomplished. An AudioRecord object initialises the audio buffer that it will use to store the new audio data when it is created. How long an AudioRecord may record before "overrunning" data that hasn't been read yet depends on the size of this buffer, which is defined during construction. Data from the audio hardware should be read in chunks less than the size of the entire recording buffer [12 pp. 58-64].

The AudioRecord class provides developers with a range of useful methods to record audio, such as startRecording() and stop(). The startRecording() method initialises the audio recording process, while stop() stops the recording. Additionally, the getState() method can be used to determine the current state of the AudioRecord object. Other methods, such as read() and read(short[], int, int), can be used to read audio data from the buffer. Developers can also set the sample rate, audio format, and number of channels using the setRecordParams() method. By using the AudioRecord class in Java, developers can easily record audio from the platform's audio input devices and perform a range of operations on the recorded audio data.

5 Creating Audio-Processing Application

This chapter covers the development of a mobile application that functions as a digital audio workstation (DAW) and allows users to record audio, organise them into virtual channels, and perform post-processing on the audio signals. The chapter outlines the minimum features required for the application, including recording and storing audio samples, multichannel stereo playback, and applying simple effects such as EQ, compression, reverb, and delay to separate audio tracks. It also describes the application's architecture, which is based on the Model-View-ViewModel (MVVM) pattern with a Controller class for the audio player logic. Additionally, the chapter explains the implementation of the Room Database as the Object Relational Mapping (ORM) for the project and provides details about the Song and Track classes used to store persistent data.

5.1 Application requirements

The objective was to develop a mobile application that functions as a DAW by allowing users to record audio, using their device's built-in or external microphone, organise them into virtual channels, and perform post-processing on the audio signals. The application prioritises quick and easy mixing for users.

The minimum features for the application include recording and storing audio samples using the microphone, multichannel stereo playback of the recorded tracks, and applying simple effects such as EQ, compression, reverb, and delay to separate audio tracks. Additionally, the app allows users to export mixed audio samples in mp3 or wav formats, and apply simple mastering presets to enhance the exported audio.

The application is designed to be scalable, which means it can handle growth and increased usage without sacrificing performance or stability. As a result, the application can easily accommodate the addition of new features such as

external audio input capability, external audio import, network connection for collaborative recordings, and audio synthesis capabilities in the future, while maintaining its efficiency and reliability.

5.2 Application architecture

5.2.1 MVVM Pattern

The MVVM pattern is a widely used software architecture pattern in modern Android app development. It helps to separate the application's UI, or View, from its data and business logic, which are represented by the Model. The ViewModel acts as a mediator between the View and the Model, providing data to the View and handling user input. This separation of concerns allows for easier management and testing of each component independently.

Specifically, the Model represents the data and business logic of the application, while the View is responsible for rendering the UI. The ViewModel is responsible for coordinating between the Model and View, and for handling user input. By separating these components, the MVVM pattern enables a clear separation of concerns, which leads to improved maintainability, testing, and code reusability [13 pp. 703-706].

5.2.2 MVC Pattern

The Model-View-Controller (MVC) pattern is a widely used software architecture pattern that separates an application into three distinct components. The Model represents the data and business logic of the application, while the View handles the user interface and how the user interacts with the application. The Controller acts as an intermediary between the Model and View, handling user input and updating both components as needed.

The MVC pattern offers several benefits for software development, including improved modularity, maintainability, and code reusability. By separating the application into these three distinct components, it becomes easier to make changes to one part of the application without affecting the others. This separation also makes it easier to debug and test the application and scale it as needed. In addition, the MVC pattern makes it easier to maintain and update the application over time, as developers can focus on one component at a time instead of making changes across the entire application [14 pp. 707-712].

5.2.3 Application structure

It was decided to use the MVVM as a base pattern for the application, but also to implement a Controller class for the audio player logic. *Figure 2* provides the visual representation of the application structure and its components.

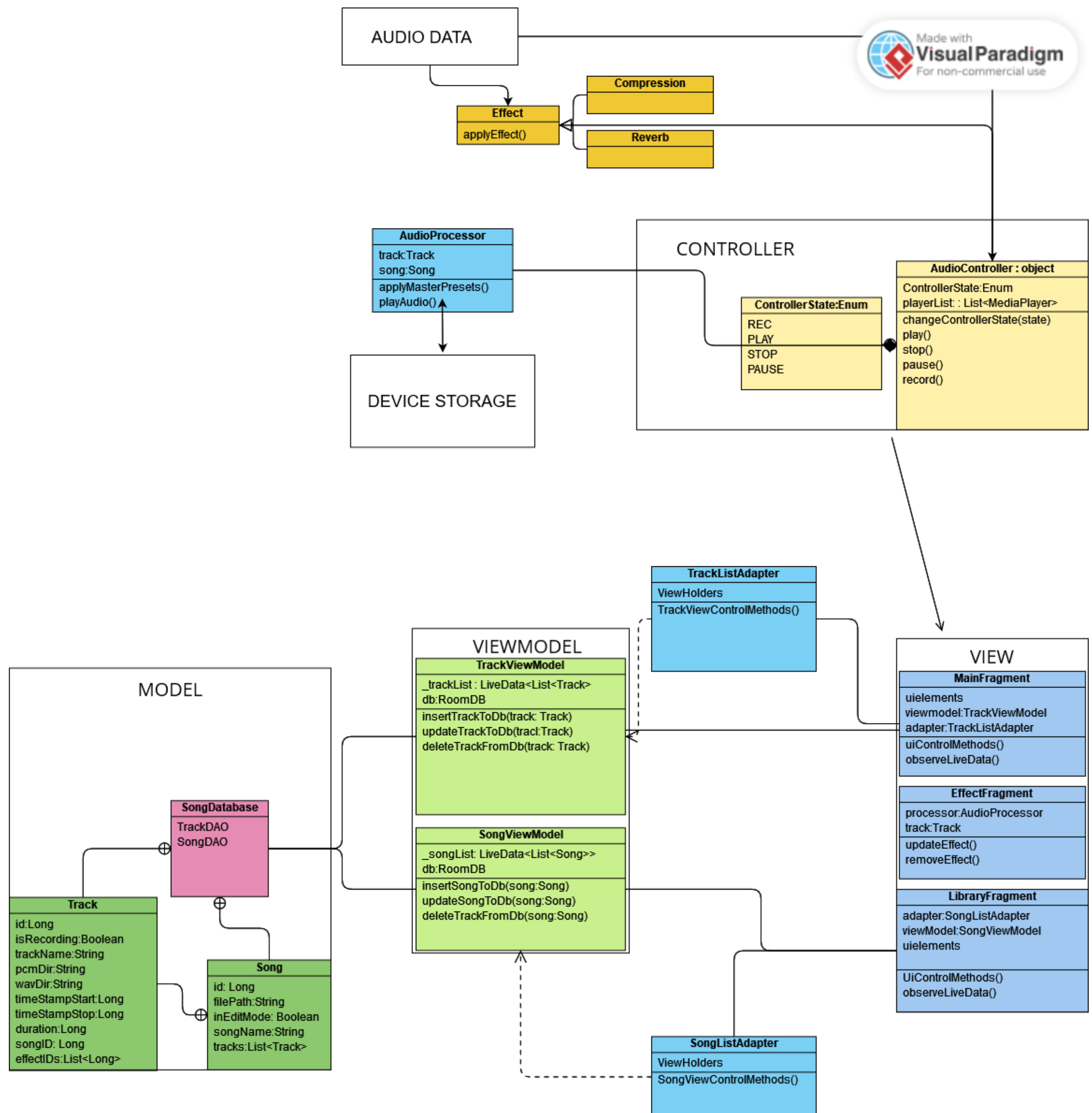

Figure 2. UML Class Diagram of the application.

As illustrated in Figure 2 the application is using several components. Each component performs its own function in the architecture.

The ViewModel in the application is used to provide data stored in the Room database to the View.

The View consists of three UI Fragments. LibraryFragment stores the instances of the Song class created by the user and shows the relevant information about them. Each Song instance creates its own TrackListFragment on click, where the user is capable of recording the tracks, play them back separately or

altogether and set the volume of the audio playback. The EffectFragment is responsible for applying the effects to the tracks and to test them in real time.

All three views are connected to the singleton controller class which provides the control to the audio playback and processing. AudioController is a state machine that acts as a layer between the UI of the application, and the AudioProcessor instance that is responsible for all the realtime audio processing and effects.

5.2.3 Database

It was decided to use the Room Database as the Object Relational Mapping (ORM) for the project. Room is mapping the database objects to Kotlin objects. It offers a layer of abstraction over SQLite that enables fluid database access while utilising all of SQLite's capabilities [15]. In addition to providing an abstraction layer over SQLite, Room offers a number of other advantages that make it a popular choice for Android developers. One of these advantages is that Room comes with built-in support for LiveData and RxJava, making it easy to build reactive and responsive user interfaces. Additionally, Room simplifies the process of creating and managing database schemas, with features such as database migrations, which enable developers to update their database schemas without losing data. Room also offers support for Kotlin coroutines, which makes it easy to perform database operations asynchronously, without blocking the main thread. Overall, Room is a powerful and flexible ORM that offers developers a simple, yet powerful, way to work with databases on the Android platform.

The application is using data classes to store the objects into the database and to provide the persistent data. The *listing 1* is showing the Song class which is the main class for representing the mixed audio.

```
@Parcelize
@Entity(
    tableName = "songs"
)
data class Song(
    @PrimaryKey(autoGenerate = true)
    val id: Long,
    val wavFilePath: String?,
    var inEditMode: Boolean,
    var songName:String?
    ):Parcelable {
}
```

Listing 1. Song data class is annotated as an @Entity for database persistence using Room. The @Entity annotation is used to indicate that the Song class is a database entity, and should be persisted in the database using Room.

The Song class represents a song instance and this class is a reference object for the tracks that are produced by the user. The referencing is done by providing the id as a foreign key parent column for the Track. Although it stores a file path for the mixed file when it is created.

The *Listing 2* is presenting the Track class which represents an audio recording. It provides metadata to Room to generate the necessary database tables and columns for the entity, based on the class properties.

```
@Entity(
    tableName = "tracks",
    foreignKeys = [ForeignKey(
        entity = Song::class,
        parentColumns = ["id"],
        childColumns = ["songID"]
    )]
)
data class Track(
    @PrimaryKey(autoGenerate = true)
    val id:Long,
```

```
    var isRecording:Boolean?,
    var trackName:String,
    var volume:Float,
    val wavDir:String,
    val timeStampStart:Long,
    var timeStampStop:Long?,
    var duration:Long?,
    val songID: Long,
    @TypeConverters(TypeConverter::class)
    var equalizer: String?,
    var compressor: String?,
    var reverb: String?
    ):Parcelable
```

Listing 2. Track data class

As could be seen from Listing 2, The properties represent metadata for the track, such as its name, volume, and effects applied. The class also defines a foreign key relationship with the Song class using the songID property.

The database class defines two abstract methods that return Data Access Object (Dao) interfaces for two database entities, Song and Track. The companion object inside the SongDB class provides a singleton instance of the SongDB database, which is synchronised to avoid multiple instances creation.

There are also Dao interfaces for Song and Track entities, which contain methods to insert, delete and retrieve data from the database. These methods are annotated with specific queries that are used by Room for generating SQL code.

Notably, the ViewModel is using LiveData, a reactive and lifecycle-aware data holder, which allows users to easily observe changes in data and update the user interface accordingly. The ViewModel class acts as a persistent layer between the LiveData and the Observer class in the fragment.

5.2.4 ViewModel

ViewModel is a key component of the Android Architecture Components that helps manage the UI-related data in an Android app [9]. Its purpose is to store and manage data that must be retained across configuration changes, such as screen rotations or changes in device orientation.

The ViewModel object is designed to survive configuration changes by keeping the ViewModel instance in memory as long as the associated activity is alive [9]. This means that it can hold the state of the UI-related data, even when the fragment is destroyed and recreated. This ensures that the UI state is maintained across configuration changes and that the data is not lost.

ViewModel is often used in conjunction with LiveData, which is a data holder class that can be observed for changes [10]. The ViewModel updates the LiveData object, and the UI components that are observing the LiveData get notified of the changes, allowing them to update the UI in real-time. By using ViewModel, developers can ensure that the UI remains responsive and user-friendly, even in the face of configuration changes.

One of the main benefits of using ViewModel is that it helps to separate the concerns of the presentation layer and the business logic layer [11]. This makes the code more modular and testable, allowing developers to easily modify and extend the app's functionality without affecting the UI. Overall, ViewModel is a crucial component of the Android Architecture Components, providing a robust and reliable way to manage the UI-related data and ensuring that the app remains responsive and user-friendly.

5.3 Audio processing overview

After the audio is recorded to the file it is stored as PCM data which is playable by the AudioTrack API that is provided by the Android. Each audio track acts like a separate channel using its own AudioProcessor class instance. AudioProcessor is utilising the CoroutineScope and ExecutorService interfaces.

In Android, CoroutineScope is a context where coroutines can run. It manages the lifecycle of coroutines and provides a structured concurrency mechanism for managing asynchronous operations. CoroutineScope is typically used in conjunction with coroutines to launch and cancel asynchronous operations in a structured way [10].

ExecutorService is the interface that provides an easy way to execute tasks in a separate thread in order to improve the performance of audio processing. By using ExecutorService, developers can manage threads more efficiently and avoid blocking the UI thread, which can cause the application to become unresponsive.

ExecutorService can be used to handle complex audio processing tasks such as encoding, decoding, and filtering audio streams. It allows for parallel processing of audio data, which can help to reduce latency and improve overall performance [12].

All the audio processing is done inside of the AudioProcessor instance. The required data is injected to this object when it is created. AudioController is keeping track of the playlist by utilising a MutableList of Pairs as shown in the *Listing 4*.

```
val trackList: MutableList<Pair<Track, AudioProcessor>> =
mutableListOf()
```

Listing 4. Tracklist of the AudioController singleton class.

In this way multiple AudioProcessor instances could be handling the audio processing for the corresponding tracks. *Listing 5* shows the method that AudioProcessor is utilising for the processed audio playback.

```
private fun playWithProcessing() {
   executor.execute() {
       createAudioTrack(PLAYBACK_BUFFER_SIZE,
FLOAT_AUDIO_FORMAT, CHANNELS_STEREO)
       var inputStream =
BufferedInputStream(file.inputStream())
```

```
        val buffer = ByteArray(PLAYBACK_BUFFER_SIZE)
        audioTrack?.play()
        isPlaying = true
        while (controllerState ==
AudioController.ControllerState.PLAY
            || controllerState ==
AudioController.ControllerState.PLAY_REC
        ) {
            val floats = toFloatArrayMono(buffer)
            val processed = effectChain(floats)
            if (inputStream.read(buffer) <= 0) {
                if (looping) {
                    audioTrack?.flush()
                    inputStream.close()
                    inputStream =
BufferedInputStream(file.inputStream())
                } else {
                    break
                }
            }
            audioTrack?.write(processed, 0, processed.size,
AudioTrack.WRITE_BLOCKING)
        }
        stopAudioTrack()
        inputStream.close()
    }
}
```

Listing 5. AudioTrack playback through the effects chain.

This method from Listing 5 is responsible for playing back an audio file while applying audio processing effects. It utilises an ExecutorService to offload the audio processing and playback to a separate thread, preventing it from blocking the UI thread.

The method first creates an audio track with the specified playback buffer size, audio format, and number of channels. It then reads the audio data from a file

into a BufferedInputStream, which is used to feed the audio data to the audio track.

During playback, the method continuously reads audio data from the input stream, converts it to a FloatArray, applies a series of audio processing effects to the audio data using the effectChain() method, and writes the resulting audio data to the audio track.

The playback loop runs as long as the controllerState is either PLAY or PLAY_REC, indicating that the audio should continue playing. If the end of the file is reached, the method will either restart playback from the beginning (if looping is enabled) or stop playback altogether.

Finally, the audio track is stopped, and the input stream is closed to release system resources.

5.4 Effects implementation

It was decided to create an abstract class Effect and use it as a template for creating different types of audio processing effects that can be applied to either an array of bytes or an array of floats.

By extending the Effect class and implementing its abstract methods, developers can create custom audio effects that can be used in the application. This class provides a useful abstraction for handling audio processing logic and can help to reduce code duplication and improve code organisation. *Listing 6* is showing the abstract class Effect, the purpose of this class is to provide a template for defining different effects.

```
abstract class Effect {
   abstract fun apply(byteArray: ByteArray):ByteArray
   abstract fun apply(floatArray: FloatArray):FloatArray
}
```

Listing 6. Effect class.

The apply methods from Listing 6 take in an array of either bytes or floats as input and apply the effect to the data in that array, returning a modified array as output.

Subclasses of the Effect class can be created to implement specific types of effects, such as equalisation (EQ), compression, or reverb. By implementing the apply methods, these subclasses can customise the specific effects they apply to the input data, and return the modified data as output.

5.4.1 Equaliser

It was decided to use the parametric equaliser for the application, which allows for more precise control over each band's frequency range, width, and gain (Appendix 1). There are also other types of equalisers, such as the shelving equaliser and the notch filter.

The equaliser applies the equalisation to specific frequency bands. Each band is defined by a range of frequencies and a gain value. The gain value determines how much the amplitude of the signal in that frequency range will be boosted or attenuated. The gain is calculated using the formula gain = 10^(dB/20), where dB is the gain value in decibels.

For each frequency band, the function computes the start and end bins in the Fast Fourier transform (FFT) output that correspond to the frequency range of the band. It then applies the gain to the signal in each of these bins, by scaling the magnitude of the complex number representing the bin by the gain, while preserving the phase of the signal.

After all frequency bands have been equalised, the inverse FFT is computed using the complexInverse method of the FFT object. This transforms the signal back from the frequency domain to the time domain. Finally, the resulting audio samples are returned as a FloatArray.

5.4.2 Compressor

In the application the compressor performs audio compression on an input signal represented as a FloatArray (Appendix 2). The input signal is processed sample by sample and the output is stored in a new FloatArray.

The function applies a variable gain to the input signal based on a set of parameters including the threshold level, ratio, attack and release times, and knee. The gain is adjusted in a way that reduces the dynamic range of the signal, resulting in a more even output level. The function also includes a limiter function that prevents the output from exceeding a maximum level.

The attack and release times are used to smooth the gain adjustments, avoiding abrupt changes that can introduce artefacts. The makeupGain parameter is used to apply additional gain to the output signal after compression, if desired.

5.4.3 Reverberator

In the application code defines a function processReverb which takes in an array of floating-point audio samples and returns a modified version of the same array with a reverb effect applied to it (Appendix 3).

The function first checks if the reverb percentage is zero and returns the input array if so. If not, it calls mixCombsWithDry function to mix the input audio with comb filters, which are defined by the Comb inner class. The Comb class defines a feedback delay network, which processes the audio and produces a reverb effect by creating multiple copies of the original signal with varying delays and decay factors.

The mixCombsWithDry function then applies a mix of the original dry signal and the processed reverb signal based on the reverb percentage. Finally, the function applies two all-pass filters defined by the allPassFilter function. The

purpose of the all-pass filters is to modify the phase of the processed signal without affecting its amplitude.

5.5 Mixing

All the effects in the application are applied in real time to the raw PCM data while playing the file. Afterall when the user is ready to mix the tracks into a song the AudioProcessor enters a loop that repeats for the number of times needed to process the entire audio file. *Listing 7* demonstrates part of the mixing algorithm.

```
for (track in tracks) {
    setTheTrack(track)
    withContext(Dispatchers.Main) {
        callback.onProcessingProgress(track.trackName)
    }
    val inputFile = File(track.wavDir)
    val bytesBuffer = ByteBuffer.allocateDirect(bufferSize)
    withContext(Dispatchers.IO) {
        val stream = inputFile.inputStream()
        stream.skip(read.toLong())
        read = stream.channel.read(bytesBuffer, offset)
        stream.close()
    }
    val bytes = bytes Buffer.array()
    val toFloat =
toFloatArrayMono(bytes.take(bufferSize).toByteArray())
    val tmpBuffer = effectChain(toFloat)
    val audioFloats = FloatArray(bufferSize / 2)
    for (i in tmpBuffer.indices) {
        audioFloats[i] = tmpBuffer[i]

        if (track == tracks[0]) {
            mixed[i] = audioFloats[i] * track.volume/100
        } else {
            mixed[i] = mixed[i] + audioFloats[i]
            mixed[i] = mixed[i] * track.volume/100
```

```
        }
        if (mixed[i] > 1.0f) mixed[i] = 1.0f
        if (mixed[i] < -1.0f) mixed[i] = -1.0f

        mixedShort[i] = (mixed[i] * 32768.0f).toInt().toShort()
    }
}
```

Listing 7. Mixing the tracks to a Song.

For each iteration of the loop, the algorithm reads a chunk of audio data from each track's input file, applies an effect chain to the data, and mixes the resulting audio data with the data from the other tracks. The mixed audio data is then converted to a byte array and written to the output file.

When the output file is ready, the program writes the WAV header to the beginning of the file, this allows the file to be played by most audio systems.

5.6 Application User Interface

5.6.1 LibraryFragment

LibraryFragment provides the playlist of the mixed songs for the user. The *Figure 3* shows a playlist of 2 songs that are available for the user interaction.

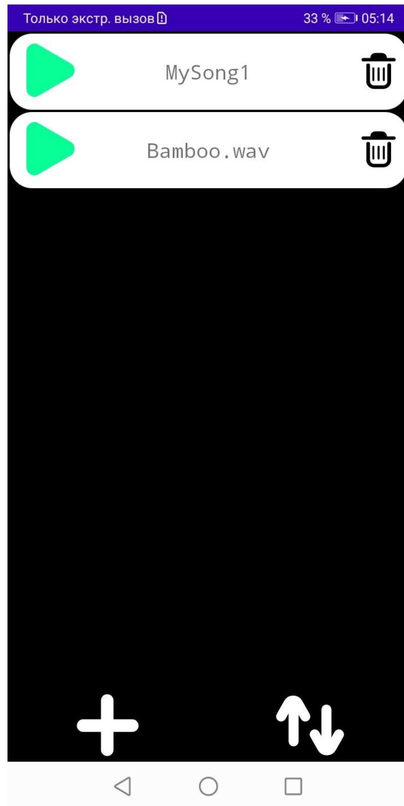

Figure 3. LibraryFragment

As illustrated in Figure 3, Users are able to create a new song by clicking the plus sign or import the existing audiofile. When the new Song is being created, the fragment calls the ViewModel, which is creating the Song instance dummy and saving it to the Room database.

By clicking the play button the fragment is calling the AudioController which is changing its state and setting the song's mixed file to the playback. If the mixed file is not existent, the play button is not showing up.

The recycler view in the fragment is controlled by the viewmodel that contains a LiveData object that is observed in the fragment. *Listing 3* shows the LiveData object that contains the list of the existing songs.

```
private val _songList: LiveData<List<Song>> =
db.songDao().getAllSongs()
val songList: LiveData<List<Song>>
   get() = _songList
```

Listing 8. songList LiveData object in the viewmodel

Listing 8 illustrates the LiveData property that receives the new value each time when it is saved to the database.

When the new instance of the Song is stored to the database observer notifies the UI and it is being updated.

5.6.2 TrackListFragment

The TrackListFragment is responsible for recording and playing back the audio tracks for the chosen song. It is possible to record up to eight tracks for a song. User is able to choose one or several tracks for the playback and control the volume. Each track has buttons for deletion and also the button that navigates to the EffectFragment, where the effects for the track could be set. *Figure 4* is showing the TrackListFragment with three audio tracks recorded to the Bamboo.wav song.

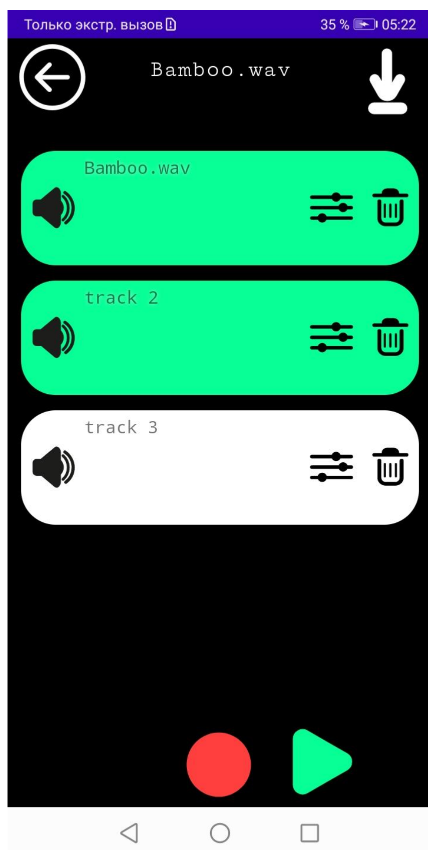

Figure 4. TrackListFragment, two tracks are chosen for the playback.

As could be seen in Figure 4, in the TrackListFragment user is able to select the tracks by simply clicking on the recycler view objects. The click is changing the colour of the object to green, which means that the track is ready for the playback. When there is any track ready for the playback the PLAY button is present in the user interface (UI).

The record button is available while there is no playback or recording ongoing in the application. When the record button is pressed, the AudioController gets the new state and the application starts to collect the data from the microphone and to stream it into the PCM file. The state of the UI while doing the recording without any tracks selected for the playback can be seen in Figure *5.*

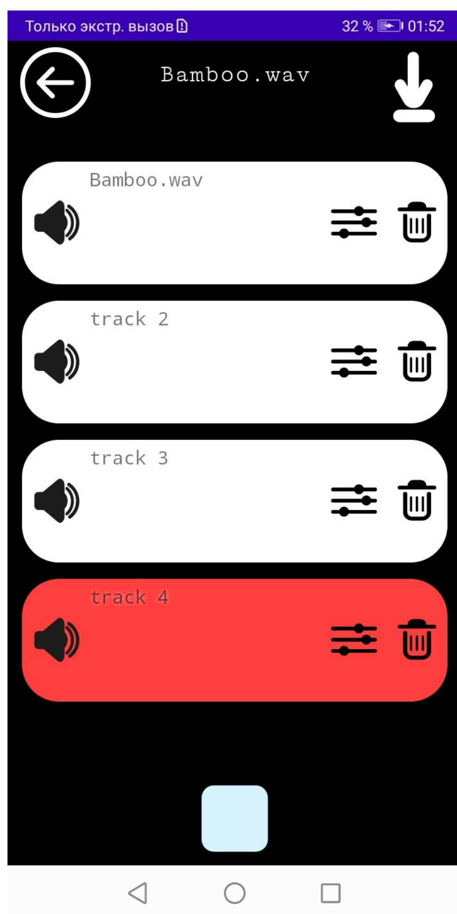

Figure 5. New audio track is being recorded for the song.

When recording starts, the AudioController is notified and changes its state accordingly. The AudioProcessor instance is created for the track and it takes all the necessary data for the playback or recording. The recording is done by creating the AudioRecord instance with the required audio properties. The application is recording the file with the following audio settings: 44.1KHz sample rate, Stereo, PCM 16 bit audio. While the state of the AudioController is set to record, AudioRecord is streaming the audio samples as pure PCM data to the file in the internal storage of the device.

The Track instance dummy is created to the room database and it is updated with the relevant data when the recording is stopped.

5.6.3 EffectFragment

The EffectFragment includes all the effect interfaces offered by the application. Three effects are currently offered: EQ, Compressor and Reverb. The view consists of the scrollview with a separate layout for each effect, which can be seen in *Figure 6*.

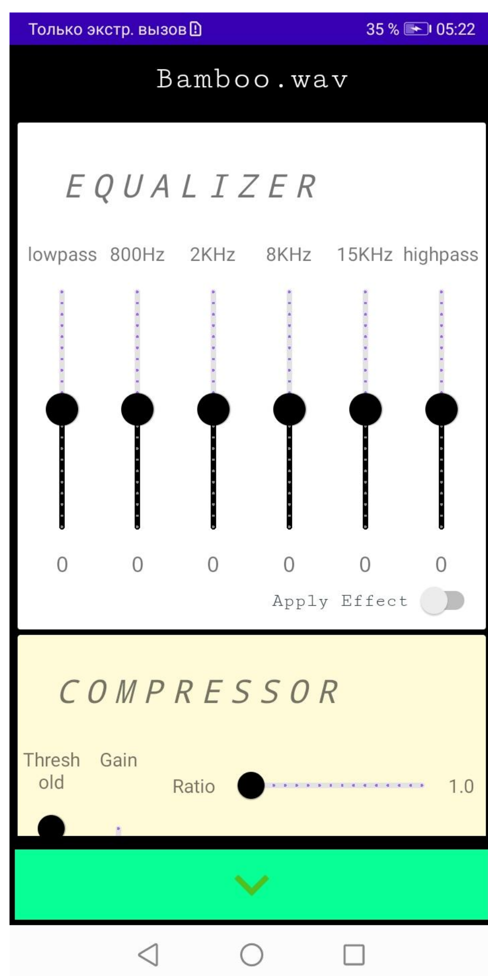

Figure 6. EffectFragment view.

By using the controls in Figure 7, the user can apply effect settings to a track and the values are dynamically updated in the database. Each effect can be individually auditioned and settings can be changed during playback. Effects instance values are stored as a string in the database and, on playback, are retrieved and converted to the desired data types.

6 Results

6.1 Testing and application problems

The application was tested on several Android devices of different generations. There were compatibility, performance and usability tests conducted. The application, as expected, showed the best results on the new models like Google Pixel and Samsung Galaxy.

With the old models there is still a possibility of delays and artefacts present in the audio playback, especially when a lot of audio processing is applied while playing back the audio. This problem was partially resolved by increasing the audio buffer for the AudioTrack while playing back the processed audio. The most efficient value was found at 128mb buffer size. Increasing the buffer size also helps with reducing the mixing time for large audio files.

Another problem is heavy UI fragments. The EffectFragment requires a lot of UI elements: global views, switches, sliders and dynamic text fields. Navigating to this fragment could be a bit slow on old devices. This problem could be resolved by rebuilding the EffectFragment in a more modular way. For example using the RecyclerView class could be the solution to this problem.

To increase the performance of the application it is also possible to rewrite the processing modules with c++ programming language.

Implementing the real-time audio processing was the hardest task in the project due to several reasons. Firstly, it required dealing with low-level audio APIs and buffer management, which can be challenging and error-prone. Secondly, it required designing and implementing complex signal processing algorithms, such as the equaliser and compressor, that could handle different types of audio signals and produce high-quality output. Thirdly, it required dealing with performance issues, as real-time audio processing must be fast and efficient to avoid audio glitches or latency.

Overall, real-time audio processing is a complex and challenging task that requires a deep understanding of audio programming, signal processing, and performance optimization. However, it is also a rewarding task that can lead to innovative and creative audio applications that provide unique user experiences.

6.2 Future of the application and upcoming features

At the moment the application stays in a state of a prototype, but there is a big potential for the future development. One of the future directions for the app could be implementing external audio input capability to allow for live recordings. Additionally, the implementation of external audio import would greatly enhance the app's functionality, allowing users to edit and manipulate pre-recorded audio. Another potential development would be enabling network connections for collaborative recordings, which would allow musicians to work together remotely in real-time. Furthermore, the app could be expanded to include audio synthesis capabilities, allowing users to create and edit their own sounds using a variety of synthesis methods. These improvements would undoubtedly increase the app's value to musicians and audio professionals alike, positioning it as a versatile and indispensable tool for all kinds of audio production tasks.

Conclusion

In conclusion, the purpose of this thesis was to develop a digital audio workstation for Android and to evaluate its effectiveness in meeting the needs of musicians and audio professionals. The thesis was based on a thorough analysis of existing digital audio workstations, as well as on the requirements and feedback from users in the target audience.

The outcome of the thesis was a functional digital audio workstation for Android, which was found to be effective in meeting the needs of its target audience. The objectives of the thesis were achieved to a significant extent, as the developed software was found to be user-friendly, versatile, and capable of handling a wide range of audio-related tasks.

Through this work, several findings were made, including the need for a more intuitive user interface, better integration with existing audio hardware, and the importance of providing a comprehensive set of features for music production. These findings have significant implications for the case company, as they suggest potential areas for improvement and development in their digital audio workstation software.

If this work had not been carried out, the need for a functional digital audio workstation for Android may have remained unmet. Therefore, the thesis provides a valuable contribution to the field of music production and audio engineering.

Based on the findings and outcomes of the thesis, recommendations for the case company include improving the user interface and expanding the feature set of the digital audio workstation, as well as further developing integration with existing audio hardware. Additionally, continued research in this area could help to improve the functionality and effectiveness of digital audio workstations for Android.

Overall, the thesis can be used as a valuable resource for musicians, audio professionals, and software developers in the field of music production.

References

[1] Rosenzweig, J., & Cogliano, M. 2018. Audio engineering in the smartphone era: An analysis of the evolution of mobile audio recording and production. Journal of Audio Engineering Society.

[2] Oliveira, E., Ribeiro, J., & Santos, C. 2018. Audio editing in mobile devices: A review of the state-of-the-art. In Proceedings of the 3rd International Conference on Design, User Experience, and Usability. Springer.

[3] Thomas D. Rossing, F. Richard Moore, Paul A. Wheeler. 2002. The Science Of Sound. Pearson Education.

[4] Rumsey, F., Zielinski, S., Kassier, R and Bech, S. On the relative importance of spatial and timbral fidelities in judgments of degraded multichannel audio quality. J. Acoust. Soc. Am., 2005, vol. 118

[5] Jarkko Punnonen. 2013. Quality Measurement of Stereophonic Audio Captured With Mobile Devices. Aalto University.

[6] David Miles Huber and Robert E. Runstein. 2017. Modern Recording Techniques. Routledge.

[7] Murphy, D. T., & Shadle, C. H. 2008. Effects of dynamic range compression on the perception of speech in noise. The Journal of the Acoustical Society of America, 123(4), 2248-2256. doi: 10.1121/1.2836784

[8] Smith, J. O. (2019). Physical Audio Signal Processing: Reverberation. https://ccrma.stanford.edu/~jos/pasp/Reverberation.html

[9] Mixing Secrets for the Small Studio (2011). Focal Press.

[10] Girish Gokul, Yin Yan, Karthik Dantu, Steven Y. Ko, Lukasz Ziarek. 2016. Real Time Sound Processing on Android. University at Buffalo, The State University of New York.

[11] Google and Open Handset Alliance n.d. Android API Guide. http://developer.android.com/guide/index.html. Accessed 25 October 2022.

[12] David Miles Huber and Robert E. Runstein. 2017.Modern Recording Techniques. Routledge.

[13] Android Developers. LiveData Overview. [Internet]. Available from: https://developer.android.com/topic/libraries/architecture/livedata. [Accessed 15 Jan 2023].

[14] Larman, Craig. 2017. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Pearson.

[15] Corbin D. Android Architecture Components: ViewModel [Internet]. 2017. Available from: https://medium.com/google-developers/android-architecture-components-viewmodel-e101a405b80. [Accessed 15 Jan 2023].

Appendix 1(1)

```
class Equalizer(val band1:Int,val band2:Int,val band3:Int,val
band4:Int,val band5:Int,val band6:Int) : Effect() {
   private var gains:Array<Int> =
arrayOf(band1,band2,band3,band4,band5,band6)

   override fun apply(floatArray: FloatArray): FloatArray {
       return parametricEqualizer(floatArray)
   }

   private val frequencyBands = arrayOf(
       Pair(0, 200),
       Pair(200, 1000),
       Pair(1000, 4000),
       Pair(4000, 10000),
       Pair(10000, 16000),
       Pair(16000, 22000)
   )

   private fun getNextPowerOfTwo(num: Int): Int {
       return 2.0.pow(ceil(ln(num.toDouble()) /
Math.log(2.0))).toInt()
   }

   private fun parametricEqualizer(audioSamples: FloatArray):
FloatArray {
       val fftSize = getNextPowerOfTwo(audioSamples.size)
       val fft = FloatFFT_1D(fftSize.toLong())

       val complexBuffer = FloatArray(2 * fftSize)
       for (i in audioSamples.indices) {
           complexBuffer[2 * i] = audioSamples[i]
           complexBuffer[2 * i + 1] = 0f
       }

       fft.complexForward(complexBuffer)
```

```
for (i in frequencyBands.indices) {
            val band = frequencyBands[i]
            val gain = 10.0.pow(gains[i].toFloat() /
20.0).toFloat()

            val startBin = (band.first * fftSize /
44100).toInt()
            val endBin = (band.second * fftSize /
44100).toInt()

            for (bin in startBin until endBin) {
                val re = complexBuffer[2 * bin]
                val im = complexBuffer[2 * bin + 1]

                val magnitude = sqrt(re * re + im *
im.toDouble()).toFloat()
                val phase = atan2(im.toDouble(),
re.toDouble()).toFloat()

                complexBuffer[2 * bin] = magnitude * gain *
cos(phase.toDouble()).toFloat()
                complexBuffer[2 * bin + 1] = magnitude * gain *
sin(phase.toDouble()).toFloat()
            }
        }

        fft.complexInverse(complexBuffer, true)

        val result = FloatArray(audioSamples.size)
        for (i in audioSamples.indices) {
            result[i] = complexBuffer[2 * i]
        }

        return result
    }
}
```

Appendix 2(1)

```
class Compressor(val limiterState:Boolean, val threshold:
Float, val ratio: Float, val knee: Float, val attackTime:
Float, val releaseTime: Float, val makeupGain: Float):
Effect() {


   override fun apply(floatArray: FloatArray): FloatArray {
       return compressAudio(floatArray)
   }


   private fun compressAudio(input: FloatArray): FloatArray {
       val output = FloatArray(input.size)
       var gain = 1f
       var filteredGain = gain
       val attackCoeff = exp(-1.0 / (44100 * attackTime))
       val releaseCoeff = exp(-1.0 / (44100 * releaseTime))
       val thresholdLevel = 10.0.pow(threshold / 20.0)

       val limiterEnabled: Boolean = limiterState

       for (i in input.indices) {
           val inputSample = input[i]
           val abs = abs(inputSample)
           val diff = abs - thresholdLevel.toFloat()
           val reduction = if (diff > 0) {
               diff * ratio
           } else {
               0f
           }
           gain = if (reduction > 0) {
               (1 / ratio).coerceAtLeast(1 - reduction / (knee
+ reduction))
           } else {
               (1 / ratio).coerceAtLeast(1 + diff / (knee -
diff))
           }
```

```
filteredGain = if ( gain > filteredGain) {
                (filteredGain * attackCoeff).toFloat() + ( gain
* (1 - attackCoeff)).toFloat()
            } else {
                (filteredGain * releaseCoeff).toFloat() + (gain
* (1 - releaseCoeff)).toFloat()
            }
            output[i] = if (makeupGain == 0f) inputSample *
filteredGain else inputSample * filteredGain * makeupGain
            if (limiterEnabled) {
                output[i] = output[i].coerceAtMost(0.99f)
            }
        }

        return output
    }
}
```

```
class Reverb(var delayInMilliSeconds: Int, var decayFactor: 
Float,
             var reverbPercent: Int, var feedbackFactor: Float) 
: Effect(){

   private val sampleRate = 44100
   private val combList = arrayOf(
       Comb(0.0f, 0.0f,feedbackFactor*1f),
       Comb( +11.73f, 0.1313f,feedbackFactor*0.4f),
       Comb( +16f, -0.2743f,feedbackFactor*0.8f),
       Comb( +7.97f, -0.31f,feedbackFactor*0.1f)
   )

   override fun apply(floatArray: FloatArray): FloatArray {
       return processReverb(floatArray)
   }

   private fun processReverb(floatAudio: FloatArray): 
FloatArray {
       if(reverbPercent == 0){
           return floatAudio
       }
       val mix = mixCombsWithDry(floatAudio)

       //Method calls for 2 All Pass Filters. Defined at the 
bottom
       val allPassFilterSamples1 = allPassFilter(mix)
       val allPassFilterSamples2 = 
allPassFilter(allPassFilterSamples1)


       //normalizeSamples(allPassFilterSamples2)

       val test = mix
       return allPassFilterSamples2
   }
```

```
inner class Comb(combDelay: Float, combDecay: Float, private
val feedback:Float) {
   private val delayInSamples =
       ((delayInMilliSeconds + combDelay) * (sampleRate /
1000)).toInt()
   private val decay = decayFactor + combDecay
   private val buffer = FloatArray(delayInSamples)
   private var bufferIndex = 0

   fun applyComb(audioInput:FloatArray): FloatArray {
       val combOutput = FloatArray(audioInput.size)
       for (i in audioInput.indices) {
           combOutput[i] = buffer[bufferIndex]
           buffer[bufferIndex] = audioInput[i] * decay +
buffer[bufferIndex] * feedback
           bufferIndex = (bufferIndex + 1) % delayInSamples
       }
       return combOutput
   }
}

private fun mixedCombsAsFloat(audioInput:
FloatArray):FloatArray{
   val mixedCombs = FloatArray(audioInput.size)
   for (comb in combList){
       val combOutput = comb.applyComb(audioInput)
       for(i in mixedCombs.indices){
           mixedCombs[i] += combOutput[i]
       }
   }
   return mixedCombs
}
```

```
private fun mixCombsWithDry(audioInput:
FloatArray):FloatArray{
   val mixedCombs = mixedCombsAsFloat(audioInput)
   val mixedWithDry = FloatArray(mixedCombs.size)

   for (i in mixedCombs.indices) {
       val outputCombByte = mixedCombs[i]
       if(i < audioInput.size) {
           val dryMixedByte = audioInput[i] * (100 -
reverbPercent) / 100
           mixedWithDry[i] = dryMixedByte + (outputCombByte *
reverbPercent / 100)
       }

   }
   return mixedWithDry
}
```

```
private fun allPassFilter(samples: FloatArray): FloatArray {
        val delaySamples =
            (89.27f * (sampleRate / 1000)).toInt() // Number of
delay samples. Calculated from number of samples per
millisecond
        val allPassFilterSamples = FloatArray(samples.size)
        val decayFactor = 0.131f

        //Applying algorithm for All Pass Filter
        for (i in samples.indices) {
            allPassFilterSamples[i] = samples[i]
            if (i - delaySamples >= 0) allPassFilterSamples[i]
+= -decayFactor * allPassFilterSamples[i - delaySamples]
            if (i + 20 - delaySamples >= 1)
allPassFilterSamples[i] += decayFactor *
allPassFilterSamples[i + 20 - delaySamples]
        }
        var value = allPassFilterSamples[0]
        var max = 0.0f
        for (i in samples.indices) {
            max += allPassFilterSamples[i] *
allPassFilterSamples[i]
        }
        max = sqrt(max / samples.size)
        for (i in samples.indices) {
            val currentValue = allPassFilterSamples[i] / max
            value = (value + (currentValue - value)) / max
            allPassFilterSamples[i] = value
        }
        return allPassFilterSamples
    }

}
```