T.R.

GEBZE TECHNICAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

GUITAR PLAYER ASSISTANT

AHMET TUĞKAN AYHAN

SUPERVISOR
DR. GÖKHAN KAYA

GEBZE
2023

T.R.
GEBZE TECHNICAL UNIVERSITY
FACULTY OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

GUITAR PLAYER ASSISTANT

AHMET TUĞKAN AYHAN

SUPERVISOR
DR. GÖKHAN KAYA

2023
GEBZE

This study has been accepted as an Undergraduate Graduation Project in the Department of Computer Engineering on 19/01/2023 by the following jury.

JURY

Member
(Supervisor) : Dr. Gökhan Kaya

Member : Dr. Yakup Genç

ABSTRACT

Guitar is one of the most popular instruments in the world, however, many people struggle with the steep learning curve and the physical discomfort of playing the instrument, such as pressing the frets all the time. There is also a chance of not finding the tutorial or tabs[1] of the song people want to play. This lack of information and mentors discourages people from learning and continuing to play guitar. To address this problem, this graduation project presents the design, implementation, and evaluation of the Guitar Player Assistant, an iOS application aimed at making learning guitar easier.

The application can convert any MIDI file of a song into a human-readable note/duration view, providing real-time audio recognition to compare the user's performance. It is developed using the Swift programming language and various libraries such as SwiftUI, Fretboard, PopupView, UIPilot, AudioKit, AudioKitUI, AudioKitEx, SoundpipeAudioKit, and AVFoundation.

The results showed that the application was able to accurately convert MIDI files and match them with the user's own audio recordings. This thesis demonstrates the potential of the Guitar Player Assistant as a useful tool for guitar players to improve their skills.

Keywords: Guitar, MIDI files, usability, effectiveness, improving skills, real-time audio recognition/conversion, Swift.

[1] A form of notation that shows the position of notes on the guitar for playing songs, and does not require musical knowledge, often preferred by beginners

ÖZET

Gitar, dünyadaki en popüler enstrümanlardan birisi. Fakat birçok insan gitarın dik öğrenme eğrisinden ve gitarı çalarken karşılaşılan fiziksel zorluklardan(perdelere sürekli basmanın parmakları acıtması gibi) dolayı bu enstrümanı öğrenmekten çekiniyor.

Bir diğer problem ise, insanlar aradıkları şarkıların notalarını, tablarını[1] veya nasıl çalacaklarını gösterecek eğitim videolarını bulmakta zorlanabiliyorlar. Bu bilgi ve mentor eksikliği insanların gitar çalma isteğini ciddi oranda azaltabiliyor. Guitar Player Assistant ise bu problemi çözmek için ortaya çıkarılmış bir proje. Bu bitirme projesi raporunda Guitar Player Assistant uygulamasının dizayn, geliştirme ve değerlendirme kısımları anlatılmaktadır.

Guitar Player Assistant, herhangi bir MIDI dosyasını daha kullanıcı dostu bir görünüme çeviren, hangi notaların çalınacağını, ne uzunlukta çalınacağını gösteren ve kullanıcının mikrofon kaydını alarak çalınan notayla, çalınması gereken notayı karşılaştırabilen bir IOS uygulamasıdır. Uygulamanın geliştirilmesi Swift dili kullanılarak yapılmıştır ve geliştirme ortamı olarak XCode kullanılmıştır. Kütüphane olarak SwiftUI, Fretboard, PopupView, UIPilot, AudioKit, AudioKitUI, AudioKitEx, SoundpipeAudioKit ve AVFoundation kullanılmıştır.

Proje sonunda yapılan testler uygulamanın başarılı bir şekilde MIDI dosyalarını çevirebildiğini ve kullanıcı sesiyle gecikme olmaksızın karşılaştırma yapabildiğini göstermiştir. Bu rapor aynı zamanda Guitar Player Assistant uygulamasının gitar kullanıcılarının yeteneklerini geliştirebilecek bir uygulama olduğunu göstermektedir.

Anahtar Kelimeler: Gitar, MIDI dosyası, kullanılabilirlik, yararlılık, yetenek gelişimi, gerçek zamanlı ses tanımlama ve karşılaştırması, Swift.

[1]Şarkıyı çalabilmek için notaların gitar üzerindeki pozisyonlarını gösteren, nota bilgisi gerektirmediği için yeni başlayanlar tarafından sıklıkla tercih edilen bir alternatif nota gösterim şekli

ACKNOWLEDGEMENT

I would like to express my gratitude to my supervisor, Dr. Gökhan Kaya, for his guidance throughout the research and writing process of this graduation project. I would also like to thank my family and friends for their support throughout this process.

Ahmet Tuğkan Ayhan

LIST OF SYMBOLS AND ABBREVIATIONS

Symbol or Abbreviation	:	**Explanation**
MIDI	:	Musical Instrument Digital Interface
TPF	:	Ticks per frame
TPB	:	Ticks per beat
BPM	:	Beats per minute
MVVM	:	Model, View, View-Model
Hz	:	Hertz
FFT	:	Fast Fourier Transform

CONTENTS

Abstract iv

Özet v

Acknowledgement vi

List of Symbols and Abbreviations vii

Contents ix

List of Figures x

List of Tables xi

1 Introduction 1

1.1 Related Works 1

1.2 Success Criteria 1

2 Design and Implementation 2

2.1 User Interface 2

2.1.1 Folder Design 2

2.1.2 Home View 3

2.1.3 Play View 6

2.1.4 Fretboard View 10

2.2 Working With MIDI 11

2.2.1 What is MIDI? 11

2.2.2 Reading MIDI File 11

2.2.2.1 Contents 11

2.2.2.2 Implementation 13

2.2.3 Playing a MIDI File 14

2.3 Audio Recognition 15

2.3.1 Receiving Audio 15

2.3.2 Note Recognition 16

2.3.3 Chord Recognition 17

3 Evaluation 18

4 Conclusion **19**

Bibliography **20**

LIST OF FIGURES

2.1 First part of folder design 2
2.2 Second part of folder design 3
2.3 Home view when nothing is selected 4
2.4 File importer view 4
2.5 Home view after importing a MIDI file 5
2.6 Home view after selecting a track 5
2.7 Play view after pressing the play button inside Home View 6
2.8 Play view after pressing the play button inside Play View 7
2.9 PlayView after user misplays a note 8
2.10 Pop-up view when user finishes the song 9
2.11 Fretboard view shown in Play View 10
2.12 Inside of guitar.json file 10
2.13 Header chunk of a MIDI file 12

LIST OF TABLES

2.1 Last two members object of AudioRecognition class. 16

1. INTRODUCTION

MIDI, or Musical Instrument Digital Interface, is a well-established method for storing musical information. Despite its age, it has stood the test of time and remains a widely used format. The Guitar Player Assistant aims to leverage the information contained in MIDI files to extract musical events from a song and present them in a more user-friendly format, such as note names, chord names, a fretboard view, and so on...

1.1. Related Works

There are many applications available on the market or on web that either convert MIDI files into visual representations of notes and chords, or receive real-time audio and display the corresponding musical notes based on frequency or pitch.

During the development of this project, several of these applications were examined and their source codes were researched to gain an understanding of their inner workings.

The sequencer web application [1] converts MIDI files into piano rolls, allowing users to play the song with their choice of tempo(BPM). Since MIDI files do not store any audio information, the application generates the audio itself. Additionally, users can choose from a variety of instruments such as piano, electric guitar, and drums.

The MacOS application [2] receives user audio and converts it into musical note names in real-time. It is available as both an AUv3 plugin and a standalone application.

However, there is currently no application on the market that combines these two features.

1.2. Success Criteria

- The application must be able to run without delay. Therefore, the application will have near real-time detection and feedback generation.
- The application must be able to recognize chords as well as notes.
- The app should be able to convert the received audio into note or chord with at least %90 accuracy.

2. DESIGN AND IMPLEMENTATION

In this chapter, the design process and implementation of the application will be discussed. As a design choice, MVVM is used as the architecture. While MVVM may be considered overkill for simple applications, it was considered a suitable choice for this project due to the amount of functionality it contains.

2.1. User Interface

2.1.1. Folder Design

Before getting into user interface, it is important to understand how the application is structured. As seen in the figures2.12.2, there are multiple folders such as App, Core, Models, Extensions and so on.

App is the main folder which contains two main files created with the project. Nothing important is implemented here other than the navigation structure2.1

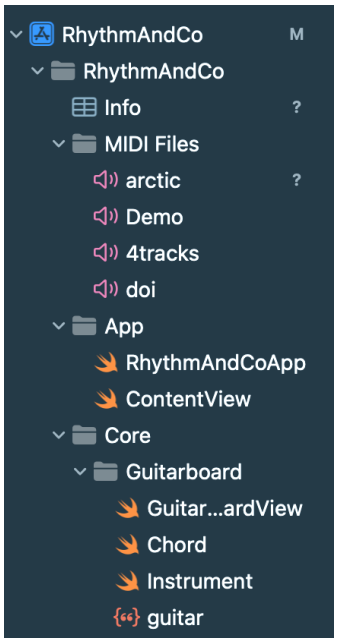

Figure 2.1: First part of folder design

Core is the folder where any view or business logic-related items are stored. Within this folder, there are three subfolders that will be explained further in the report.

Models folder is used to store models, which can be thought of as interfaces. The NoteSyntax stores all note name values, such as ”A,” ”C#,” and so on. The NoteInfo stores all note-related information, such as the note name, note start and end time, and note duration, etc. These files are also will be explained further in the report.

Utils and Extensions contain code snippets to prevent code repetition. Boilerplate code becomes a big problem when a project gets bigger. Therefore, such structuring becomes necessary to maintain the project. 2.2

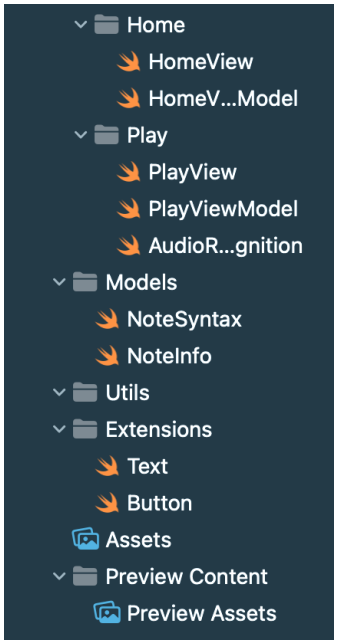

Figure 2.2: Second part of folder design

2.1.2. Home View

Home View is one of the three subfolders mentioned in the folder design. It is the first view that users interact with as they open the application. When the application is first executed, users see the image below. 2.3

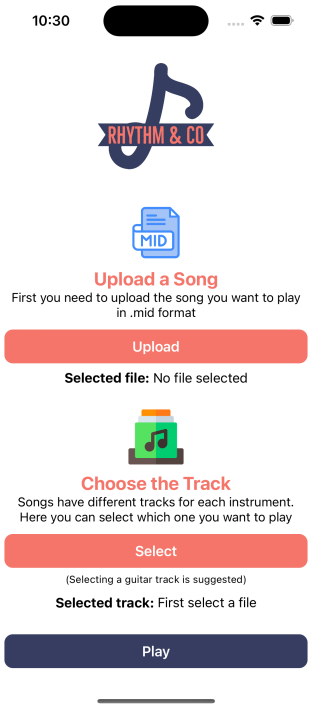

Figure 2.3: Home view when nothing is selected

At this stage, the user cannot interact with the select or play buttons as they are inactive. Once the user presses the Upload button, a file importer view will appear, and it will look like this 2.4

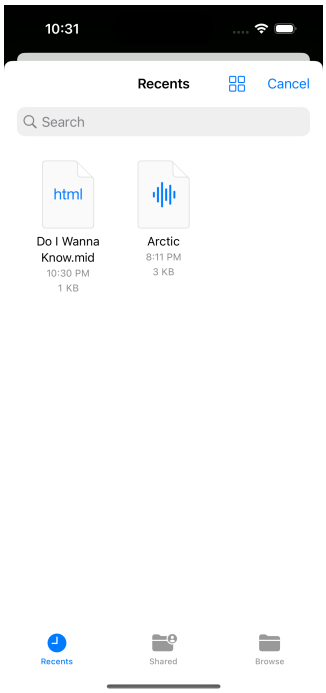

Figure 2.4: File importer view

Here, the user can see the downloaded apps on their phone and select any MIDI file. Once they select one of the files, the home view will look like this 2.5

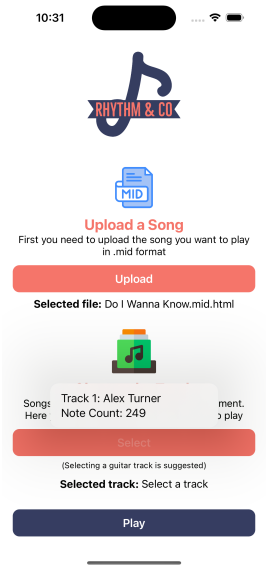

Figure 2.5: Home view after importing a MIDI file

After this point, the user can select a track and track options are listed in 2.5. For this file, the only option is Alex Turner, but in some MIDI files, there can be up to 6-7 tracks to choose from. Below the track name, the note count is shown. This value indicates how many notes are in this specific track. When the user chooses a track, the home view looks like this. 2.6

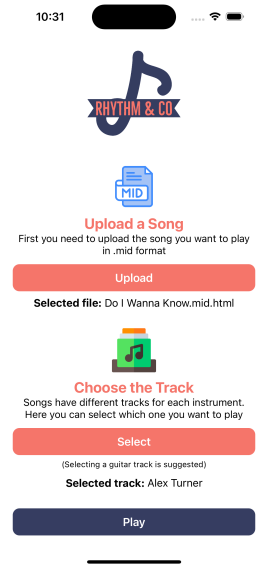

Figure 2.6: Home view after selecting a track

Once user presses the play button, application navigates to the Play View.

2.1.3. Play View

Inside the play view, there are multiple buttons and subviews for the user to interact with. These include: the file title, the correct and misplay values the user played, a MIDI sequencer (represented as a black box in the middle), a fretboard, a microphone input, and a scroll view displaying all the note names in the selected MIDI track. Additionally, there are pause, start, reset, and home buttons.

File title value is extracted by using the lastPathComponent value of the URL object. URL object is received when the user selects the MIDI file they want to play.

Correct and Misplay values indicate how many notes the user played correctly or incorrectly. These values are stored as @State objects, as they are not static values. Inside the PlayViewModel, the note name received from the microphone and the current note to be played are compared. Based on the result, the corresponding value (Correct or Misplay) is incremented. The sum of these two values becomes equal to the total note count once the user finishes the song. 2.7

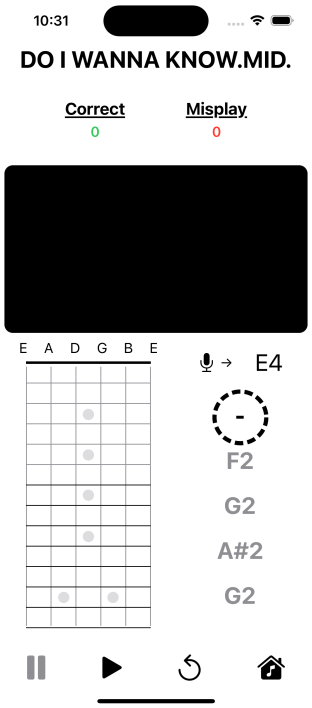

Figure 2.7: Play view after pressing the play button inside Home View

MIDI sequencer is the black box displayed in the middle of the view. When the user presses the play button, notes start flowing from right to left. To stop the sequencer, the user presses the stop button, which is located in the bottom-left corner. Inside the MIDI sequencer, some notes appear longer than others, indicating how long the note needs to be played. This sequencer is implemented using the MIDITrackView struct. The details of the implementation are discussed later in the ”Implementation” subsection of the ”Working With MIDI” chapter 2.8

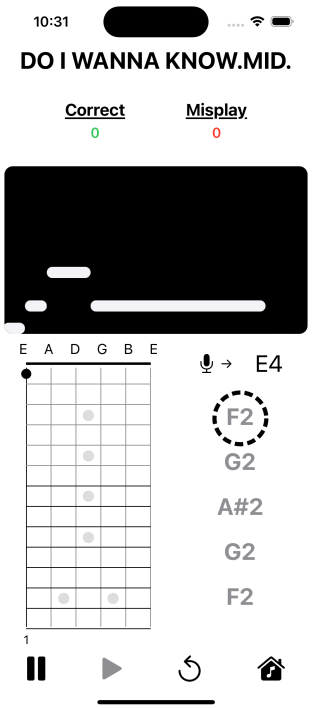

Figure 2.8: Play view after pressing the play button inside Play View

Fretboard is a minimal representation of guitar frets. A detailed implementation of this will be discussed in the Fretboard subsection of this section, but for now, it is important to know that this fretboard shows the possible fret points to produce the a note.

Once the user goes to the next note, the fretboard refreshes and shows the corresponding fret points for that note. Some notes are not even possible to play on a guitar, and if the imported song contains these kinds of notes, an empty fretboard will be shown to the user.

Other than out-of-range notes, some notes can be played at multiple places, and some notes can only be produced from one point on the fretboard. The user can choose at which place they want to play according to their liking. Points shown on the fretboard produce the same exact frequencies, with no octave difference.

Microphone input is received from a phone microphone using the SoundpipeAudioKit library. The conversion of pitch and amplitude values to note names is discussed in the Audio Recognition chapter. The user interface section explains the purpose of the microphone object.

The right side of the microphone icon displays the corresponding note name of the received pitch value from the microphone. This value is constantly refreshed as the user produces sound. The microphone does not receive audio until the user presses the play button. The user can also press the pause button to stop using the microphone.

Scroll view is located below the microphone output. It is a vertical scroll view that goes from top to bottom. The user cannot scroll this view. The application iterates through the list as the user moves on to the next note. The note name displayed inside the circle indicates the current song that needs to be played.

Possible ways to produce the note inside the circle are shown on the fretboard, and the duration for which it needs to be played is shown in the sequencer view.

Initially, the color of the note name inside the scroll view is gray, but once the user plays the note correctly or not, it changes the color of the current note to either green or red.

Later, when the user finishes the song, the scroll view becomes scrollable and the user can see which notes they played correctly and which notes they have made mistakes on.2.9

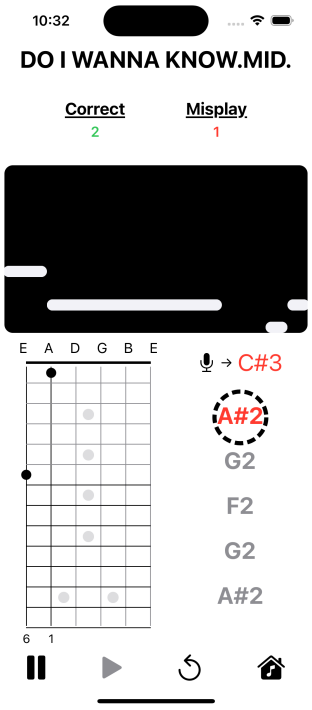

Figure 2.9: PlayView after user misplays a note

Buttons are located at the bottom of the screen. The pause button (bottom-left corner) is initially pressed when the user first opens the Play View. The purpose of this button is to stop sending audio information to the microphone and to stop the MIDI sequencer.

The implementation of these buttons is made in the PlayView file and the pause button only toggles the play status of the microphone engine and the MIDI sequencer object.

Users can resume/start the application by pressing the start button located to the right of the pause button. Users need to press this button in order to start playing the song. After pressing it, it becomes disabled until the pause button is pressed. It does the same things as the pause button but in reverse.

The reset button is next to the play button and can be pressed at any time. It dismisses the current play view and creates a new instance. For this purpose, I used the UIPilot library which makes navigation between screens a lot easier.

At the bottom-right corner, there is a home button. This button is also available at all times. Instead of creating a new instance of the play view like the reset button, it sends the user back to the Home View where they can select a new song or track.

Pop-up view is shown when the user finishes a song. Inside the pop-up view, a celebratory text is displayed and options for what the user can do next are explained. The PopupView library was used for the pop-up view. 2.10

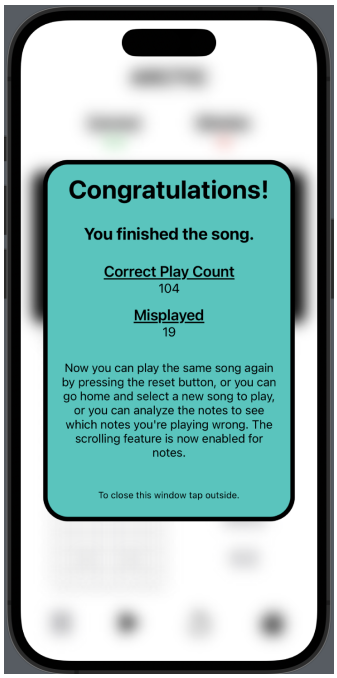

Figure 2.10: Pop-up view when user finishes the song

2.1.4. Fretboard View

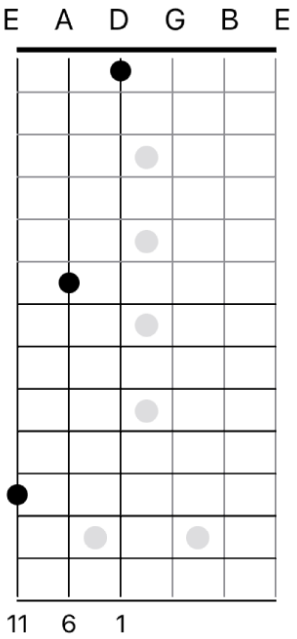

Figure 2.11: Fretboard view shown in Play View

Fretboard View is one of the most complex parts of this project. It uses a json file called 'guitar.json' and stores all possible fret positions for a note from E2 to B6 (the maximum range for a normally tuned 6-string guitar).2.12

```
{
    "key": "C",
    "suffix": "4",
    "positions": [
        {
            "frets": [ -1, -1, 10, 5, 1, -1 ],
            "fingers": [ 0, 0, 10, 5, 1, 0 ],
            "baseFret": 0,
            "barres": [],
            "midi": [ 48, 51, 55, 60, 67 ]
        },
    ]
},
```

Figure 2.12: Inside of guitar.json file

After examining each note and its position on the fretboard, I plotted them in json. To get the fretboard view, I used an existing library called Fretboard, but it was designed to show a small percentage of frets and could only show chord positions. I rewrote most of the parts to show note points and extended the view to show all frets at

once.

Fretview struct works by taking fret and finger positions as parameters. It then finds the corresponding points on the fret and places a black circle.2.11 Below the curtain board, the user is shown some numbers. These values represent the fret position. If this number is 3, the user can place their fingers directly on the 3rd fret indicator, which is shown by a different colored circle on each guitar. Here it is shown in opaque gray. 2.11

2.2. Working With MIDI

2.2.1. What is MIDI?

MIDI, is a file format that stores musical information, such as the notes played, tempo, and instrumentation. It is a sequence of timing events, and can be played using a MIDI synthesizer or a software instrument.

MIDI files do not contain audio data, but rather sequence of instructions for a device to play back the music. Because of it's design choice, it can also be used as a general purpose timing protocol.

2.2.2. Reading MIDI File

Even though MIDI format doesn't transmit any audio, it stores a significant amount of information that can be used to communicate between instruments in a meaningful way. Although not all of this information is necessary for this project, understanding the contents of the MIDI format can provide a valuable perspective.

2.2.2.1. Contents

To simplify the demonstration, the Audiokit library will be used to showcase the contents of the MIDI format. When a file URL is provided to the MIDIFile constructor, it returns a MIDIFile object. The contents of the object are as follows:

Tracks are the most complicated aspect of a MIDI file. They store a collection of events that are played by an instrument. Each track has its own set of MIDI events, tempo and other performance settings, which allows for multiple layers of music and different instruments to be played simultaneously in a MIDI file.

Each track has its own name to differentiate which track is playing which instrument. The name of the person who plays the track or the instrument being played on

the track is often used as the name of the track.

Time Division of a MIDI file is a value that determines how the timing of events in the file is represented. There are two types of time division: ticks per beat and frames per second. This is one of the most important value in a MIDI file. It is always needed when using a MIDI sequencer. Without a time division value, it is not possible to know at which BPM the song needs to be played. [3]

The header chunk of a MIDI file contains some important information about the MIDI file. The last two bytes of this header file contain the time division value in delta times. If the top bit of the 16 bits is 0, the time division is in ticks per beat (or pulses per quarter note). If the top bit is 1, the time division is in frames per second. 2.13

Description	Length in bytes	Starts at byte	Value
chunk ID	4	0x00	The character string "MThd"
size	4	0x04	6
format type	2	0x08	0, 1, or 2
number of tracks	2	0x0A	1 - 65,535
time division	2	0x0C	Various as described below

Figure 2.13: Header chunk of a MIDI file

Ticks Per Beat/Frame per Second are time division values used in MIDI files that are intended to be played back with a specific tempo. These values represent the number of MIDI ticks that occur in one beat/each frame of the MIDI file.

It is important to note that a MIDI file can only use one of the two time division method, either ticks per frame or frames per second, but not both. And also, the higher the value of frames per second or ticks per beat, the more precise the timing of the events in the file will be.

Events are actions that occurs at a specific time in the file. Events include things like note number, note duration, note start/end time, note on and note off messages and so on.

Channels are virtual pathway through which MIDI events are sent. Each channel can have its own set of instruments, volumes, and other settings. There are 16 channels in a standard MIDI file, each of which can be used to control a separate instrument or group of instruments.

Note Number is a value used to represent a specific note in the MIDI format. The note number corresponds to a specific pitch on a musical scale, and is used in note

on and note off messages to specify which note is being played. It is important to know that, instruments are able to create only a limited number of note sounds.

Note Start/End Time are the time at which a note on/off message are sent in a MIDI file. These values are measured in MIDI ticks, and are used to specify when a note should begin/stop playing.

Note Duration is the length of time between a note on message and a corresponding note off message in a MIDI file. This value is measured in MIDI ticks.

Note Range is the range of notes that an instrument or group of instruments can play. For example, a piano has a larger note range than a guitar. In MIDI, the note range is represented by the lowest and highest note numbers that can be played by an instrument.

As the guitar is the primary focus of this project, it's important to note that the typical range of a 6-string guitar, when tuned to standard tuning, is between E2 (82.41 Hz) and B6 (1174.66 Hz).

2.2.2.2. Implementation

The implementation of the file reading process includes the use of the Audiokit library. Listed below are all the structs and classes used while reading a MIDI file:

MIDIFile is a structure used to initialize a MIDI file. It uses a URL object to get the location of the file and returns an instance of this structure. This instance is then used to get the MIDIFileTrackNoteMap object.

MIDIFile stores all the information that a MIDI file can contain, but not all of it is directly accessible. The only useful information directly available for this project is the track count. Tracks are stored as a MIDIFileTrack object inside of MIDIFile struct and inside of MIDIFileTrack events are stored as MIDIEvent.

The disadvantage of MIDIEvent is that only one event stores the note number value. Other important information such as note duration, note interval, note start and end time are not stored.

To solve this problem, it is possible to instantiate another structure called MIDIFileTrackNoteMap using MIDIFile and the track number the user wants to play.

MIDIFileTrackNoteMap is a struct that contains a lot of useful information about a MIDI file such as hiNote(highest note in the track), loNote(lowest note in the

track), note range and note list. Note list is an array of MIDINoteDuration struct. This array is then used to get create NoteInfo struct.

MIDINoteDuration stores every information this project requires from a note. The problem with MIDINoteDuration struct is, it doesn't contain any identification value. That means it is possible to create two or more exact instances of this struct. It is important to solve this problem because in the user interface part, SwiftUI doesn't know how to differentiate these structs. To solve this problem, I created NoteInfo struct.

NoteInfo is the structure used to store note information for this project. It has 3 different member: 'id' for identification, 'noteName' for name representation of a note number and 'MIDINoteDuration' to access other necessary informations(note number, note start/end time and note range).

As the NoteInfo array is created, reading part for the project is finishes. This array is created in PlayViewModel then used inside the PlayView file which draws the user interface. To keep track of which note is need to displayed at any moment, a note index value is used inside PlayViewModel.

2.2.3. Playing a MIDI File

In order to offer better user experience, the person using this application needs to know how long they need to play a note. To achieve this, it would be a good idea to simultaneously play a MIDI file using a sequencer.

For this part, one struct which named MIDITrackView and one class which named MIDITrackViewModel is used.

The MIDITrackView creates a card view within the frame and uses a scroll view to display all the notes of a MIDI file horizontally. These notes are represented as bold strings, and their length varies according to the duration of the note.

To use the MIDITrackView, a MIDITrackViewModel object is created within the PlayView file. This view model is then added as an environment object to the view. When the PlayView frame appears, the MidiTrackViewModel is initialized by calling its startEngine and loadSequencerFile functions.

The startEngine function is used to connect the MIDITrackView and MIDITrack-ViewModel. The loadSequencerFile function is used to load the MIDI file that will be played into the view model.

If the user wants to start playing, the start function of the MIDITrackViewModel is called. If the user leaves the PlayView, the stopEngine and stop functions of the MIDITrackViewModel are called.

2.3. Audio Recognition

In this section, the discussion will focus on how audio is received and converted into notes or chords.

2.3.1. Receiving Audio

Guitar Player Assistant uses the phone microphone to receive audio data. Phone manufacturers provide APIs for developers to access the phone's functionalities. Developers can either write their own functions or use existing libraries to access these APIs.

In this project, the SoundpipeAudiokit library is used to receive audio data. By making changes to the AudioRecognition file, audio data is received in real-time, along with note conversion

AudioRecognition class uses the Audiokit, AudiokitEx, and SoundpipeAudioKit libraries. It has six important members: data, engine, mic, tracker, note frequencies, and note names with sharps.

The first object, data, is a TunerData object that contains important information about the incoming audio, such as pitch, amplitude, and note name. This data object is continuously updated as long as the audio engine is working.

The second object, engine, is an AudioEngine object that is used to access the microphone API. If it is unable to find any usable microphone, it throws a fatal error.

The third object, mic, is used as a shortcut. Once the engine accesses the microphone API, it assigns it to the AudioEngine.inputNode class. The mic is also an AudioEngine.inputNode object, and after engine accesses the microphone API, inputNode is assigned to this mic object to make access easier.

The next object, tracker, is a PitchTap object that is initialized with a mic (AudioEngine.inputNode) object. The tracker then follows any changes in pitch or magnitude in the mic and returns these values. The tracker also asynchronously calls the update function to change the data(TunerData) values with the values it returns.

The last two objects, note frequencies and note names, are integer and string arrays. The note frequencies store the approximate frequency values that correspond to a note name.

Table 2.1: Last two members object of AudioRecognition class.

noteNames	noteFrequencies
C	16.35
C#	17.32
D	18.35
D#	19.45
E	20.6
F	21.83
F#	23.12
G	24.5
G#	25.96
A	27.5
A#	29.14
B	30.87

2.3.2. Note Recognition

Conversion of pitch value into note/chord is done inside update function. This function is called with each trigger of tracker(PitchTap) object.

Update function first compares the amplitude value to prevent noise or fluctuating data. If the amplitude value is bigger than the threshold, then the function continues.

It first assigns pitch and amplitude values which comes from tracker into data member variables 'data.pitch' and 'data.amplitude'. Then it creates a frequency variable and assigns pitch value to it. There 2 steps until frequency value can be used.

First, it divides the frequency value with 2 until it becomes less than the biggest value of noteFrequencies array.

Then, update function multiplies it with 2 until it becomes bigger than the lowest value of the noteFrequencies array. By doing so, frequency value always stays between the given frequency values and it won't cause out of boundary exceptions.

After getting the frequency value, update function finds closest value to frequency inside the noteFrequencies array in a for loop. When it finds the closest distance value, it saves the index of that frequency value.

To find octave value, update function takes the base 2 logarithm of pitch / frequency and assigns this value as octave value.

Finally update function changes data.noteName value with the "noteNames WithSharps[index] + octave" value.

2.3.3. Chord Recognition

Before getting into chord recognition, knowing what chord actually is and how it differs from musical notes is important.

A chord is a group of two or more musical notes played at the same time, while a musical note is a single sound played at a specific pitch. Chords are created by playing multiple notes together, while a single note is played one at a time. Chords can be made up of any combination of notes, while a single note is just one pitch.

For example, a C major chord is made up of the notes C, E, and G played at the same time. Another example A minor chord, which is made up of the notes A, C, and E. But a single note, such as middle C, is just one pitch played by itself.

Since chords are made of multiple of notes played at the same time, it makes them harder to identify. Current technologies in the market uses artifical intelligence to guess the chord.

Another solution would be using fast fourier transform to identify notes. By applying the FFT to the recorded chord, it is possible to obtain a representation of the frequency content of the chord. The individual notes in the chord will appear as distinct peaks in the frequency spectrum, corresponding to their specific frequencies.

The problem with this technique, accuracy rate of chord recognition will be affected by quality of the recording and complexity of the chord.

This section is not yet implemented, but I still working on it.

3. EVALUATION

First success criteria of the project was: The application must be able to run without delay. Therefore, the application will have near real-time detection and feedback generation.

As can be seen in the Demo video, time delay of the conversion of audio to note name is indeed near real-time. Users don't notice any delay.

Second success criteria of the project was: The application must be able to recognize chords as well as notes.

This part was the trickiest part of the project, since it requires multiple note recognition at the same time. As Mr. Gökhan Kaya suggested, fast fourier transform will be used on the audio data. If I can identify individual notes in the chord it is possible to develop a working sytem. These chords most likely will appear as distinct peaks in the frequency spectrum, corresponding to their specific frequencies.

As this was the last part of the roadmap, there was not enough time to implement a successful chord recognition technique for this project.

Last success criteria of the project was: The app should be able to convert the received audio into note or chord with at least %90 accuracy.

Noise and data fluctuation is a problem when dealing with audio but as seen in the Demo the success rate of the audio to note translation is above %90.

4. CONCLUSION

In conclusion, the results of user testing showed that the application was able to accurately convert MIDI files and match them with the user's own audio recordings. These results demonstrate the potential of the Guitar Player Assistant as a valuable tool for anyone looking to improve their skills in playing guitar.

It is important to note that despite the application's success in converting MIDI files, this project was limited by the lack of information on the specific type of project and the challenges it presented(e.g. chord recognition). However, this project serves as a foundation for future work in this area, and with further research and development, the application can be improved and expanded to include additional features.

Overall, this graduation project has shown that with the help of technology, it is possible to make the process of learning to play guitar easier and more effective, and can be a valuable tool for anyone looking to improve their skills.

BIBLIOGRAPHY

[1] Jacob Morgan, George Burdell, "Online sequencer," 2013. [Online]. Available: https://onlinesequencer.net/import.

[2] Nilson Ltd., "A2m real-time audio to midi," 2021. [Online]. Available: https://apps.apple.com/tr/app/a2m-real-time-audio-to-midi/id1569569145.

[3] Recording Blogs, "Header chunk of a midi file," 2011. [Online]. Available: https://www.recordingblogs.com/wiki/header-chunk-of-a-midi-file.