String Master

Cody Malnor
CS480 Senior Project
Winter 2018

Introduction

String master is an Android app with the main function of helping the user memorize where notes are on the guitar fretboard. This is accomplished with a "trainer" which is a game to practice finding notes on each string. It also includes a guitar tuner and a settings page with preferences to adjust the difficulty and included notes of the trainer.

I created an Android app for my senior project because mobile platforms provide the perfect opportunity to build a tool like this with their portability and available sensors. The idea for this app came from when a coworker showed me a simple website with a script which displayed a random musical note that would change after a predefined time interval. This seemed like an interesting thing to make myself as an Android app, but I wondered how it could be improved upon.

The goal of the note trainer is to increase the speed with which the user can locate notes on the fretboard. While the website version had the option to increase note frequency, there was no penalty for missing a note, nor reward for quickly finding one. In String Master, the trainer listens to the user's guitar to determine if the correct note was played. Notes are served up as fast as they can be played and recognized. The user is tasked with playing as many correct notes in the allotted time as possible. This creates an environment where difficult notes will lower the overall score due to the time required to find them, while well practiced notes can be quickly dispensed resulting in a higher score.

Why is this useful?

Understanding where notes are on the different strings of a guitar is essential for things such as locating root notes and constructing scales. Most chord shapes on guitar are located by the root note, so the ability to quickly find it is important. Note locations are also useful from a theory perspective; understanding chord structure and constructing scales from note increments instead of memorizing shapes. The tuner portion of the app is useful for the obvious reason that a guitar must be in tune to sound as intended.

App structure

The app is built around the MainActivity; this is the only activity in the app. The MainActivity uses a navigation drawer to switch between the different pages of the app and each page is a fragment. The default fragment loaded is the TrainerFragment, which, as you would expect, contains the trainer game. There is no stack for the fragments; they restart whenever they are switched and the state is not saved. Relevant data such as high scores and changes to settings are saved automatically. The two other fragments in the app are the TunerFragment (guitar tuner portion), and the SettingsFragment (settings to adjust the trainer's functionality).

Technologies used:

- Android Studio IDE
- Java
- Pure Data for Android
- XML
- Android OS

Converting sound to MIDI notes

Both the TrainerFragment and TunerFragment require using the microphone of the Android device to listen to the notes being played on the user's guitar. While I originally considered writing my own code to find the dominant frequency of the incoming sound and convert that to a MIDI number, I quickly determined that this would be too much work on top of learning and building a full featured Android app. Pure Data for Android is the library I utilized for this task.

As its name implies, Pure Data for Android essentially allows Pure Data patches to be utilized in an Android app. Pure Data is an open source visual programming language for multimedia applications. Programs in Pure Data are called patches. The patch used in String Master is shown in Figure 1. As with many visual programming languages, in Pure Data functions are shown as objects which can be connected to each other to transfer values among them. The patch used for this app works by running input sound through an analog-to-digital converter, and then through a function which locates the dominant pitch and outputs it as a MIDI note. This patch is run by the Android app as part of a background service, and the variable which the MIDI value is sent to can be read into the app.

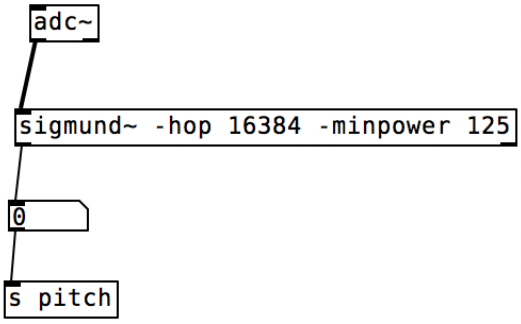

Figure 1: *Pure Data patch used to convert microphone sound into a MIDI note of the dominant pitch.*

Trainer

The trainer is a time-based game which presents random notes one at a time out of a list of available notes and listens for the user to play the correct note. The note list is constructed by starting at the first note of the selected string and working up the scale. The first consideration when constructing this list is the different parameters which can be set in the SettingsFragment such as the number of frets to test and whether sharp and flat notes are included. Due to the nature of music, each sharp note has a corresponding flat note with the same pitch. Because of this, I was unable to simply generate a list of note names and use their offsets to create MIDI notes. Instead, the note generator creates a LinkedHashMap, which allows finding the first note in the list, easily selecting a random entry in the list, and mapping multiple notes to the same offset.

Once a note is displayed for the user to play, and the user makes an attempt to play that note, the Fragment needs to determine if that value was correct or not. The log snippet below gives an example of what MIDI values the app is receiving (captured on a Nexus 6p emulator). The frequency of the readings is determined by the ‘hop’ value in the Pure Data patch and the sample rate of the phone. Because the guitar is a fretted instrument, I can assume anything less than a quarter step away from the correct note should count as correct. Frets are placed at every half step on a guitar, so there is no need to decide how close is ‘correct’ as on a violin or similar non-fretted instrument. MIDI values read in can then be considered correct when enough of them in the specified range are read sequentially. Currently, that amount is 3 and could easily be adjusted later.

```
04-22 14:46:18.983 4354-4354/com.example.android.string_master_01 D/TrainerFragment: setHighScore:
setting_high_score_11_0_30
04-22 14:46:18.984 4354-4354/com.example.android.string_master_01 D/MainActivity: getMIDINote: 48
04-22 14:46:19.164 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: 45.958652
04-22 14:46:19.329 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: 45.97633
04-22 14:46:19.490 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: 45.93083
04-22 14:46:19.673 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: 45.932396
04-22 14:46:19.839 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:20.015 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:20.184 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:20.345 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:20.524 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:20.691 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:20.857 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:21.033 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:21.202 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:21.381 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:21.543 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:21.715 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: 50.27658
04-22 14:46:21.892 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:22.058 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:22.221 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: -1500.0
04-22 14:46:22.403 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: 47.965218
04-22 14:46:22.566 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: 47.962276
04-22 14:46:22.751 4354-4354/com.example.android.string_master_01 I/TrainerFragment: pitch: 48.004562
```

The layout for the TrainerFragment is shown in Figure 2. To notify the user as to where the played pitch lies in relation to the pitch of the correct note there is a simple 3 block graphic. The middle block indicates when a correct note was played and lights up green. The red block on either side of it will light up red if the note played was higher or lower than the specified note. This system provides a hint to the user if they play an incorrect note, but no other guidance. Because the block colors are tied to the current MIDI note reading, a green block does not mean that the note has been accepted yet. Once enough correct MIDI notes are received in a row (as described earlier), the app will play a confirmation sound to alert the user to the correct note, and then proceed to the next note.

To earn the metrics points on my rubric, I implemented high scores into the trainer. The high score is defined as the most correct notes played in the time allotted for the given settings. The high score is recorded for each combination of settings as a change in settings would change the difficulty. To save each high score as a persistent value, a key is created which is unique to that combination of settings, and the key-value pair is saved using Android's SharedPreferences API.

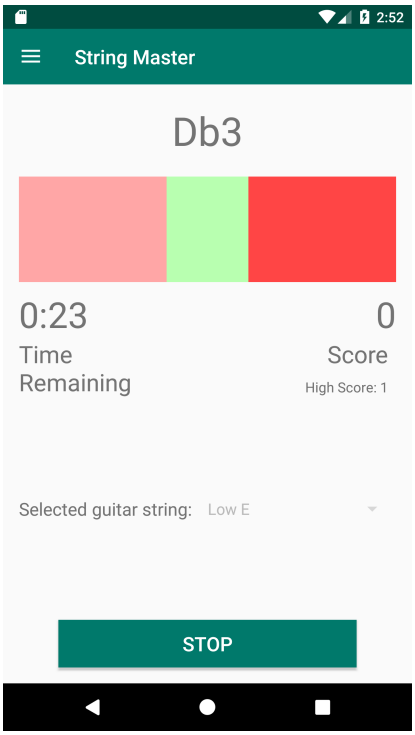

Figure 2: *TrainerFragment GUI.*

Tuner

The guitar tuner part of the app, TunerFragment, runs in a similar way to TrainerFragment. The TunerFragment runs the same Pure Data patch as the TrainerFragment, and MIDI notes are read the same way as well. The notes read in are used to determine the closest corresponding MIDI note available on guitar. This closest note is then set as the center value on the screen and a green line is drawn to represent the exact pitch of that note, as shown in Figure 3. Next, acting as the needle on a normal guitar tuner, the actual MIDI value read in is drawn as a blue line with its X-axis value offset adjusted based on its distance from the correct pitch. The blue needle is drawn continuously as MIDI notes are received, allowing the user to see the string's pitch change in real time as it is adjusted. To animate the movement of the needle, a ValueAnimator is used which 'steps' a value from one defined value to another, allowing for progressive movement.

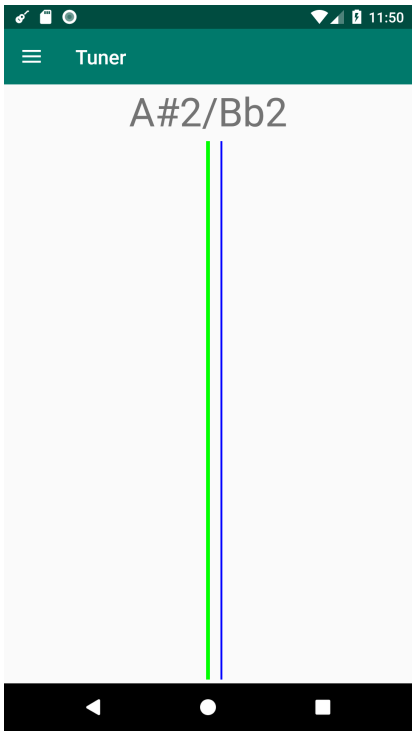

Figure 3: *TunerFragment GUI.*

Settings

I did not originally plan on building a settings page for this app. The app is simple, however there are some settings that would be very useful for a user who is learning note locations on guitar. The final settings page layout is shown in Figure 4. After creating an entire layout manually for this page, I discovered Android has a system for settings pages called Preferences. This page extends PreferenceFragmentCompat, and the layout is built out of different types of Preferences.

The preference views provide a way to easily add each setting using a single view item. As an example, the game length slider is simply a SeekBarPreference. Values for things like the text and icon can be specified in the layout of the preference while the locations and styles are predefined. The most difficult part of this page was figuring out how to create a custom DialogPreference to confirm the user wants to delete all high scores. The included DialogPreference classes do not cover this simple use case.

The new dialog is called ClearScoresDialog, shown in Figure 5. The layout file for the dialog is simply a text view, confirming the action of clicking on the ‘clear scores’ button. The two buttons at the bottom of the dialog are part of the DialogPreference style. This style is applied to the layout and the text of the buttons is set by overriding the style definition. For this layout resource to be used when the dialog is created, the DialogPreference is extended by ClearScoresDialogPreference. This class can then override

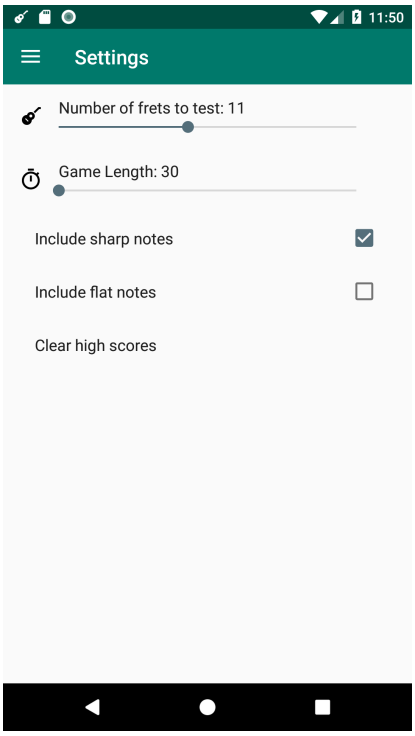

Figure 4: *SettingsFragment GUI.*

the getDialogLayoutResource method to use the custom layout, as well as call the clearScores method if the negative dialog button is clicked.

The preference now knows what layout to use and it contains a method to delete all high scores, but it does not know when to call this method nor does it call the dialog. Dialogs are fragments themselves, so a class extending PreferenceDialogFragmentCompat called ClearScoresDialogFragmentCompat is created. This class defines what to do when the positive dialog button is selected, in this case calling the method from the preference to delete high scores.

The last piece of the puzzle is creating the dialog fragment with the correct layout when the 'clear scores' button is pressed. This is accomplished in the SettingsFragment by overriding onDisplayPreferenceDialog. This method takes, as an argument, the preference that is trying to load a dialog, and by identifying an instance of ClearScoresDialogPreference, the ClearScoresDialogFragmentCompat can be loaded. The layout is specified in the preference, and when the positive dialog button is selected, will take the action defined in the fragment.

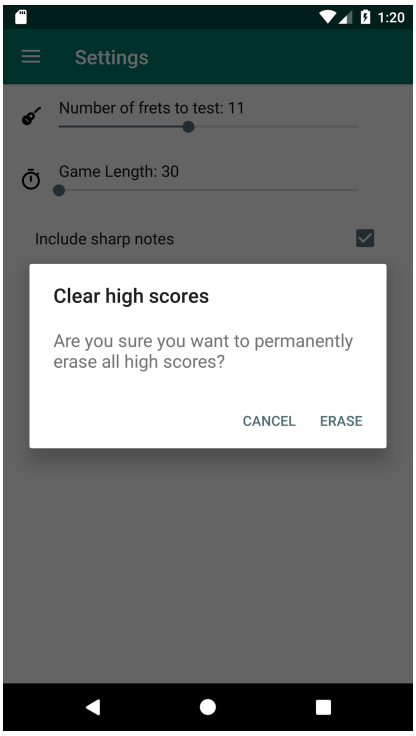

Figure 5: *ClearScoresDialog GUI.*

What I would do differently

All of the layouts in this app were originally built in XML manually. Towards the end of the project, I decided to convert them to ConstraintLayout. While writing the layouts manually allowed me to learn how layouts work and how they should be structured, it is no longer the recommended way to create layouts according to Google. ConstraintLayout provides performance and functionality benefits over RelativeLayout. Using ConstraintLayout flattens the view hierarchy, which increases performance as tree-traversal of the layout is not as complex. In the end, I would still recommend learning to create layouts manually with RelativeLayout and LinearLayout before switching to the GUI based ConstraintLayout system. However, for this app starting with ConstraintLayout would have been more efficient. Going forward I will exclusively use ConstraintLayout for its performance, versatility, and speed.

Android is a constantly changing platform, and since I began learning about it, they introduced Kotlin as a fully supported language next to Java. Many top apps are now written in Kotlin and the language is open-source as opposed to Java. Also, Kotlin is interoperable with Java, allowing for progressive adoption. In the future I would like to write an app in Kotlin.

Conclusion

With this project, I set out to create a useful tool while also building my first fully featured Android app. Now that the app is done, I feel much more comfortable working in the Android environment. Through the countless hurdles I encountered while working on this project, I've not only learned many essential things about Android, but also become more familiar with navigating the documentation and creating robust solutions to my problems. There are many things to consider when creating a program which runs on a phone. Through my increased experience with the environment, I now enjoy Android development much more than when I first started, and I look forward to building more apps and improving this one.

Grading

Overall, I accomplished everything that I set out to. The main criticism I have of my app is that the design is fairly simple. Along with improving the overall appearance, the animation for the tuner is not as smooth as users would expect from today's apps. Because of these things, 2 points have been

	Points earned	**Points possible**
Sound	**38**	**40**
Record and filter sound	18	20
Convert sound to MIDI	5	5
Note detection	15	15
Trainer	**38**	**40**
Note randomizer	5	5
GUI	13	15
Metrics	10	10
Game logic	10	10
Tuner	**8**	**10**
GUI	8	10
Navigation Drawer	**10**	**10**
Notifications	**10**	**10**
Total	**104**	**110**

deducted from each of the GUI portions in my rubric. The tuner also has difficulty creating a stable pitch reading of the Low E string. While this is most likely due to the quality of the phone's microphone, other apps are able to handle this better than mine, so I have deducted 2 points from the "Record and filter sound" section. The total comes out to 104/110 which is an A on the standard grading scale at 95%.

Extra Points

Adding a settings page was not in my original rubric, but it was necessary to make the app complete. It required quite a bit of time due to the implementation of the setting values and investigating the custom dialog. Based on the difficulty of the other items in my rubric, 15 points is a conservative value for this feature. This brings the total to 119/125 which does not affect the overall letter grade.

Settings	**15**	**15**
Proposed Total	**119**	**125**