**4/23/2017**

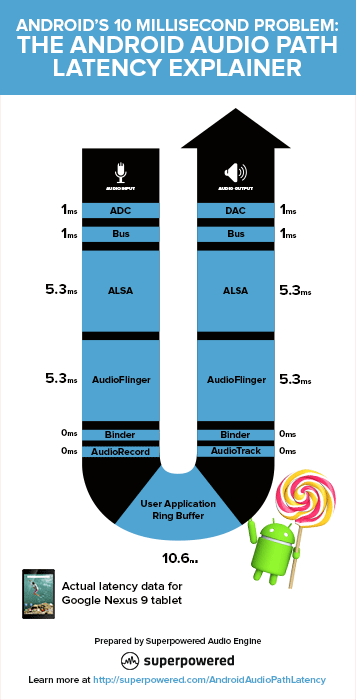
**Intro**

For this project, I decided to expand my mobile software development abilities and create an audio related app. At the time, I had an Android, so was interested in programming for Android. I had made a few experiments for Android before, just testing compilers and figuring out how to work on it, but I had never had a real project on Android before, which is another reason why I thought it would be perfect. The idea originally was to create transcription software, something that will write music as you sing it. When I realized how much more limited for time I would be, I pivoted from that to the current iteration of my app, RapiTune, an extremely quick Android tuner.

**The concept/purpose**

The idea behind creating a quick Android tuner came from my discovery of the Superpowered SDK and Android’s 10ms latency issue. If you’ve ever perused through the app store and play store, you’ll notice an obvious difference is the sheer difference in amount of audio related apps for iOS compared to Android. Many audio tech companies have entire divisions dedicated to their mobile development teams, producing iOS apps for every audio purpose you could imagine. The few audio apps on Android are plagued by bad latency, poor design, and abandoned development. As far as I know, there are no major apps on the play store that have yet utilized the Superpowered SDK, but this is just because I couldn’t find anyone who said they used it. The software is free to download however, requires no reference, and does not keep track of who is using it in anyway.

The Superpowered SDK team tested a few apps for Android to compare their latency, and found that some have almost 100ms of output latency and 200ms from input to output. This means entire industries, such as VR, will never find a market for Android. My goal was to create a tuner using this SDK, and make the tuner as fast as possible. Unfortunately, due to time being limited by many issues during development, there is still much work to do to bring the app to the level I want, and as I continue to understand how the Android operating systems works better, I am able to make changes to the app to make it better.

Tuner apps are a dime a dozen, but there are surprisingly few that get the job done. My favorite app, TE Tuner, is the most solid and well put together tuner app that I could find. The completeness of the app over many of the other ones could be because it is a port from an iOS app, and if there are any serious development teams, they would usually avoid working with Android altogether. However, because the best apps are ports from iOS, they have hefty UIs which are not correctly ported and run UI operations on the CPU, and use the standard android audio input and output, meaning that the latency is off the charts for these apps. This means that if someone could bypass the latency issue and a less CPU intensive UI, they could easily create the fastest tuner on the market, even without using the fastest pitch detection algorithm.

The purpose of this app and readme is not to educate on why there is so much latency on Android or how to fix it, but instead to see how easily the Superpowered SDK can be implemented into fairly simple software, and to see if it could easily be expanded upon to change the Android market and allow for more music apps. Google has actually allowed for a natural bypass to the slow audio path by using the Android’s native-code toolset, Android NDK. However, due to the complex implementation of the NDK, I found that there was a very minimal crowd of NDK users and therefore a significant reduction in the amount of relevant guides and FAQs online that would assist me.

**Materials**

The first resource I needed was a phone and a way to connect it to the computer. Interestingly, Motorola decided that they would not produce an official USB cord for my phone, the Droid Turbo 2, and because of this decision, this meant that there was no manufacturer’s guarantee that any cord would work for USB connectivity. Once I started testing, I discovered that there were actually no cords that worked correctly, which set back my first round of debugging by about two weeks. After updating my BIOs, trying multiple computers, and even performing a hard reset on my phone, I found the answer on some old hidden Motorola forum post. Back in 2016, Motorola had independent driver software for every phone on their line, and many older ones, but then replaced it with a new form of driver software that replaces the driver software for all of their phones they’ve ever made, but only supports drivers for the latest few phones that they’ve produced. Using an archived version of a year old copy of the Motorola website to acquire a download, I was finally moving forward.

Another essential part of my development was the use of the Superpowered SDK which uses the Android NDK. Installing the Superpowered SDK was simple, it’s just downloading files and placing them in my project directory, but installing the Android NDK involved changing the way that Android Studio builds the compiler for the app. There were a few guides to help with the NDK, but I could not find a single user online with a guide for completely implementing the SDK into the compiler as well, and the guide from Superpowered has been dated since the release of the latest version of Android(7.0 Nougat).

In order to start development, the first thing I had to think of was which IDE to use. There aren’t many options for IDE’s for android development, but after any amount of research, it becomes increasingly obvious that Android Studio is the most compatible option, and somewhat less importantly, had an easy to use GitHub integration. Android Studio is a very well designed piece of software that really helps as much as possible with programming a single application for thousands of different devices with different specs and requirements.

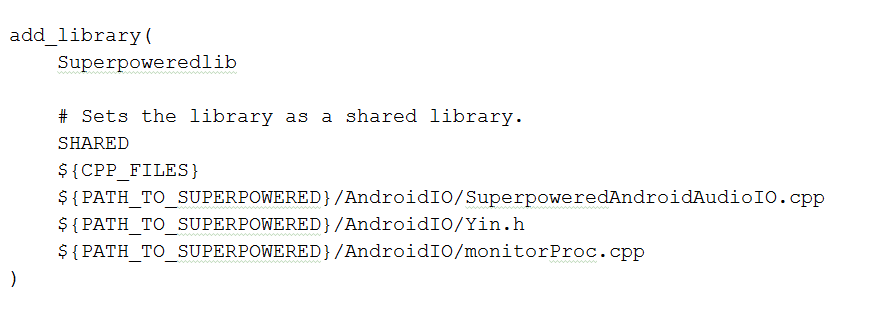
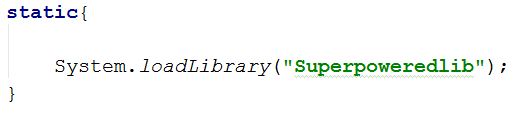
**The design/operation**

The app has completely changed its design twice since I started writing the software. In fact, since the time of writing this sentence, I would say that about 75% of all the running code has been completely changed in the last two weeks. In its current iteration however, the app itself is very simple, and therefore easy to improve and understand. The code is divided up into 3 groups, Java code for the base Android OS operation and UI animating, C++ for the audio input loop and pitch detection, and Android-specific XML code for the UI. Integrating the XML into the Java was mostly done by Android Studio, however, getting the native code and Java to communicate was the issue that single handedly delayed my progress on the app more than anything else.

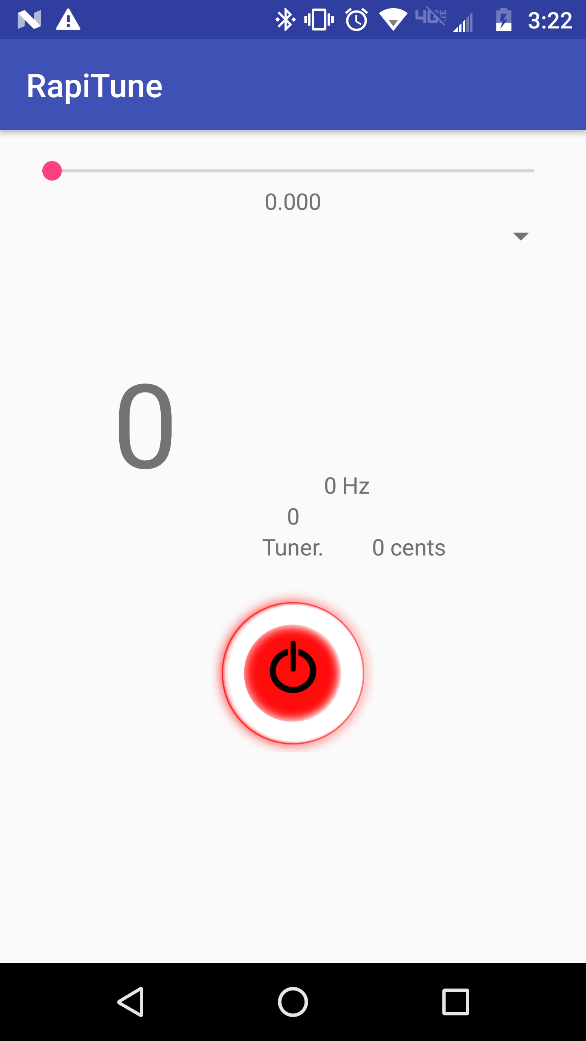
The Java code was initially 5 classes and somewhere around 600 lines total, but has been condensed and completely rewritten down to 1 single main class:

**public class** Trainer **extends** AppCompatActivity {

The first thing this class does, is load my native library. On the left is how I load the library, then on the right is how I created the library in a CMakeLists.txt file.

After that, the actual Trainer activity gets created, then, Android calls the onCreate function. This function should only ever be called once, when the app is first opened. This is where I do some basic tasks for android specifically; getting user permission to use the microphone, measuring the sample rate of the phone, finding the minimum input buffer size for the phone that I can use, setting a spot on the phone’s memory to save the recording from the microhphone which is mostly for future development, defining global variables, such as the container layout for the UI. This is also where I do some minimal UI work, I attatch onClickListeners to anything you need to interact with and finally, update the UI once it’s ready for the user to see it. The last thing the app does before waiting for you to interact with it is it creates, but doesn’t start, the thread for the audioloop. At this point, the screen should look like this:



1

4

2

3

5

1 – The sensitivity adjustment slider and monitor, this sets a number in the function that determines how certain the algorithm must be that it heard a certain tone in order for it to display the value, in the future, this will be controlled by swiping up or down on the top half of the UI.

2 – The current note you are playing, displays 0 when it is not detecting a pitch or is off.

3 – (Left to Right) Octave of the pitch, A4 = 440,

Frequency (in Hz) of the suspected pitch

The amount of cents sharp it is

4 – Custom power button which turns into the Intonation monitor, a visual representation of your pitch (this feature currently has memory issues so isn’t active in the GitHub release v1.0 yet)

5 – The pagename, this will become relevant in further development as a number of features, including a metronome, tone generator, and a simple recorder, the pages will be changed by swiping laterally across the screen.

6 – This arrow indicates a feature for V1.1 where you will select an instrument, this modifies the algorithm to better suit your instrument, as well as adjusts the pitch for keys.

6

**monitorProc.cpp**

The next stage of the app is where things begin to get complicated. One of the major issues with the app is how the Java communicates with C++. After loading the C++ library into java, it allows me to initialize native functions in my Trainer class:



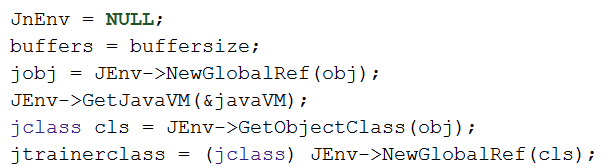
These functions technically belong to the Java program, but are then defined in the c++ program. The way this works is very complicated in the backend, but is actually fairly easy to implement. When called, the Trainer.java sends a copy of the Java environment (JNIEnv) to the Java virtual machine (JavaVM). All parameters except arrays can be passed through these functions, and show up on the other side as Java object equivalents (jobject). So an int becomes a jint, and a double becomes a jdouble. These variables are automatically cast to the C++ equivalent if set to equal it. This means that the java classes can communicate most of its variables and instances directly to C++, and using returns from the method, can also communicate back to the Java class. The StartRec function above, becomes the StartRec function below, once in C++:



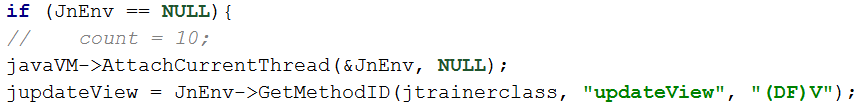
The JNIEnv object that gets passed is basically a copy of the current instance of the java environment, in terms of a pointer to an array of pointers to address for various functions and variables. Knowing that, it means I could get any class, or any function that existed at the time of calling the native code, this should make communication back and forth between the java and C++ code easy. The issue is that the Java environment only exists for the duration of the current thread. The whole purpose of the native code however, was to start an audio loop using Google’s recommended low-latency, native-only input. This means that not only will I be starting a new thread, but I won’t be able to return any value from the infinite recording loop, thereby severing communication with the java code. Technically, I could set the JNIEnv as a static variable in the native code, but once a new thread is started, all the address change.

What I had to do was save the entirety of the JavaVM as a static variable, attach the new thread to the old VM, and then search through the array for a method I specifically created to send information from the native code to the UI save that methodID as a static variable, and call it every time the loop goes.

startRec()



audioProcessing()





**Yin.cpp**

The next and last class in the program is the Yin class. Yin is a method of analyzing audio buffer data to find a pitch. It is based off of the assumption that across two sample periods, the smallest difference between individual samples should always be across samples which are at the same location in the period. In other words, if you took one sample and kept calculating the difference between that sample and another sample, while incrementing the index of the second sample by one every time, the index gap between the two samples for the smallest difference calculated, should represent the length of one sample, Tau, which is 1/Hz.

Since this method requires you to calculate across two different sample periods, the maximum tau, or minimum fs = 1/tau, can only be at half the buffer length. Which creates a natural limit for how low of a frequency can be analyzed, keeping the tuner from being tricked into thinking the note is actually half of the actual frequency in certain ranges where that is bound to occur. Also due to this method of analysis, it means that the tuner works best with imperfect sounds, such as voices or instruments, but is very easily tricked by perfectly repeating waves with no modulation.

The actual pitch detection occurs by means of four separate functions in the Yin class.

**void** Difference(Yin\* yin, int16\_t\* buff){

the first function that’s called is the easiest to understand, but the part of the audio recording loop that single handedly creates the largest amount of latency. The function must add every single two sample combination acccross the buffer, up to a sample range of half the buffer size. This works by means of a double for loop, simply finding the squared result of the calculations, to solve for negatives. I was able to speed up this function a little by decreasing the amount of Taus that it checks for, figuring it doesn’t need to check for super high frequencies. In V1.1, the user will select an instrument on the main page which limits the range of values that this function will search for, as well as make a smaller buffer size for higher pitched instruments, making it even faster.

**void** Meannormalized(Yin \*yin){

this function goes through the buffer again, averaging all the values against the values it’s already calculated. Most of the results are between 0 and 1, but a few frequency values that are way off can end up with values much higher than this, however we only care about values below 1 anyway, and more specifically values below the threshold.

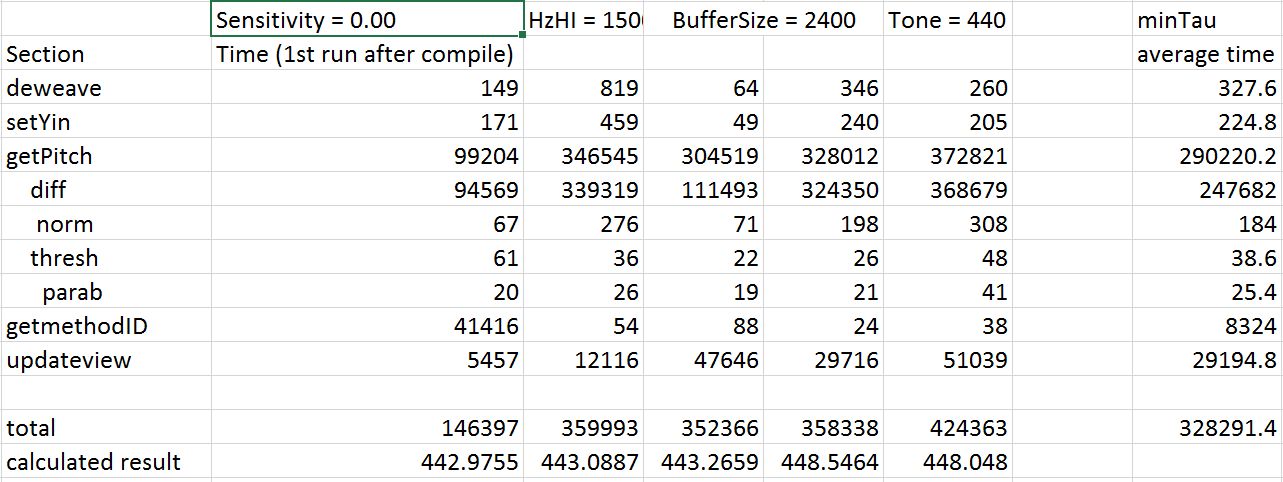
int16\_t absThreshold(Yin \*yin){

This is the first function where a plausible answer is returned. The threshold that the function refers to is the sensitivity number at the top of the GUI in V1.0. That exact number is sent through the rest of the program to the Yin, whenever the slider is adjusted. The buffer is incremented through again, however this time, the first answer that is below the threshold is chosen. Then, the algorithm keeps choosing the next sample until the one after it is greater than the current sample.

**float** Yin\_parabolicInterpolation(Yin \*yin, int16\_t tauEstimate) {

If any plausible tau is found, then it is passed to this function to determine the exact number, since the correct tau is usually between two samples. The number returned from this function will determine what is shown on the screen next time the UI updates. Right after this function returns, the value received is sent back to the java file via the JMethodID lines above.

I implemented a timing function to see which of these functions creates the most hang up, and found that the Difference function creates so much latency, that the latency from the other methods could be completely neglected when timing the app. Most of my future efforts in increasing speed will involve that function, as well as some other things such as the UI. Below you can see some measurements I made of the Yin class, so you can see just how much time is spent in the difference function compared to the others:



**Issues**

Software development for Android presents a lot of issues with audio and compatibility. Aside from the regular 10ms issue outlined by Superpowered, some cheaper and older phones will develop issues by other means. Gathering the sample rate and minimum buffer size is easy, but there are some phones, usually ones released before Android 6.0 (Marshmallow) that have much larger minimum buffer sizes, which cannot be circumvented and will cause a naturally high latency no matter what. Some phones with much cheaper microphones also have large amounts of noise in their signal, making the Yin formula less effective as well as simply decreasing the frequency range of the microphone. Preparing for all the different setups in Android is basically impossible with the amount of devices, new and old, running the OS. The only option is to prepare for as many different devices as possible.

Even though there are so many different devices, allowing for them in java is not that difficult, but designing the UI is much more difficult for a number of reasons. There is the much more obvious issue of screen resolution and orientation as well as differences in dpi that make planning the UI difficult, but different devices can even handle and decode the UI differently. On certain devices, certain animation classes don’t work, usually for purposes of graphical memory. As I was only able to test out the software on my phone and emulators (which use my computers GPU), I could not prepare for phones with very low graphical memory.

There were a number of other issues that were mentioned in the previous section, but one that hasn’t been mentioned yet is the way that Android handles threads. UI updates can only be applied on the main thread, however, if the main thread is really busy, Android will sometimes shut down background threads instead of allowing them to hang. With the newest release of Android 7.0 (Nougat), they have somewhat fixed this issue with java threads, but it seems as though Android still has issues recognizing the importance of threads that are running in native code.

**Thing I Learned**

After making the program, I changed my mind