OctaPitch – Guitar Tuner

A Senior Project Created By:

Taylor Plambeck

Professor Lin
ECE481 & ECE482 & ECE467

California State Polytechnic University, Pomona
Electrical & Computer Engineering Department

Development Timeline: September 2016 – June 2017

Table of Contents

Forward

This document is detailing the development of my Android app *OctaPitch*. I have been playing guitar for over 14 years, and so this project is less of a school assignment, and more of a passion project, one that probably would have happened at some point whether or not I was being graded for it. I worked alone, so every single aspect of this project was done by yours truly, from *OctaPitch's* inception to its final uploading to the public app store known as the Google Play Store.

Aside from the hours of coding time on a platform I had never worked with, I also had to become proficient at several different skills that weren't just programming. This includes (but is not limited to) drafting a Privacy Policy to make *OctaPitch's* audio recording legal, and resizing a vector several different ways to provide correct image resolutions for the poster, app, and Play Store advertisement banner. Making *OctaPitch* as professional as possible was something that added a significant amount of development time that is hard to account for, but the impact of those small details can't be understated.

I have tried to structure this document in a way that makes sense to the reader and isn't overly technical. All of the code written for *OctaPitch* is located in the Appendix. Check the Table of Contents to see where each section has its corresponding code. I have included pictures to supplement every section, as well as a slideshow of *OctaPitch* through its various development stages. I have also included the detailed list of what changed throughout every one of the 30+ versions of *OctaPitch,* recorded through the development.

- Taylor Plambeck

Project Overview

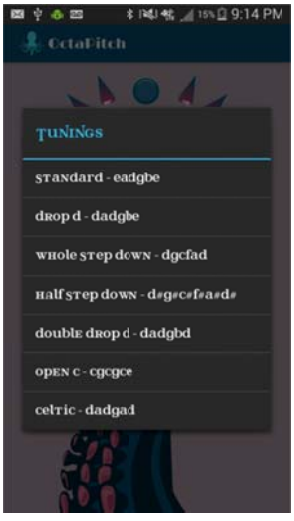
OctaPitch is an app that uses the microphone on your Android device to recognize pitch! When someone plays the guitar, the piano, or even when they sing, we hear a musical note, and each note has a corresponding frequency that is referred to as its pitch. Different types of instruments may have different ranges of notes, but all instruments will have a method to keep those notes in tune. That is, a method to make sure the notes they create are "tuned" to the corresponding pitch that they should be. A tuned guitar can make a terrible player sound great, and an out-of-tune guitar can make a great player sound terrible.

Tuning instruments can be very different. For example, you would only need to tune a piano a few times a year, but when you do, you need a specialist, and even then it takes a couple of hours. This is not the case on the guitar! You want to tune your guitar before every playing session, if not more. Depending on the quality of your guitar, it may go out of tune *after one song.* The good news is we don't need to hire anyone, we just need a smart phone, and we can tune our guitar in seconds.

This is where *OctaPitch* comes in. The *OctaPitch* user interface will update constantly as it hears audio, changing the interface contents to show the name of the closest note, how far you are from it, and whether you are above it or below it. This gives us all the information we need to tune the guitar.

OctaPitch Main Features

- Pitch Recognition using Autocorrelation
- Finds note closest to recognized pitch and guides user to it with the display
- Display has several denominations, middle circle is "Perfectly Tuned"
- Indicates when a recognized pitch belongs to a certain string
- Tuning Menu to change which notes these indicators belong to
- Emboss Filters applied to text and shapes to give a wet and 3D look
- Linear Gradients applied to triangles to mix and fade between two colors
- Custom fonts applied to menus and application title
- Tested on multiple devices, app layout is consistent across screen size



How it Works

Every phone is equipped with a microphone, which is freely available for use within an app. When we start recording, the audio is "sampled" and stored in a format that is manageable by a computer. This audio is processed using the *Yin Algorithm* which uses a Fourier transform to perform an autocorrelation of the signal. Different notes will have different periods, and so by iterating through these periods we can see how well they "fit" to the audio the app hears. Once a fit is found, we can also measure how strong the correlation was to that note. This is used as a way for *OctaPitch* to decide if the recognized pitch is accurate enough to display to its user.

If there is a strong correlation, then there was probably minimal background noise and the app is very sure that the pitch that it recognized is correct. However, if there was too much noise there will be a very weak correlation. When this happens, the *OctaPitch* interface will not be updated! Only when the correlation is above 90% will the display update. Any pitch judged with a correlation of less than 90% will not only be ignored, but the app will continue to display the *previous* note with a correlation *above* 90%. This prevents the interface from appearing jumpy, and eliminates the impact background noise can have on the user experience. *OctaPitch* can record and process audio faster than we can even read the changes in the UI, so trimming any jumpiness possible provides a better user experience.

On the next page is a detailed example of exactly what happens to every sample taken from the microphone, and how the pitch is found from those samples.

Autocorrelation Theory

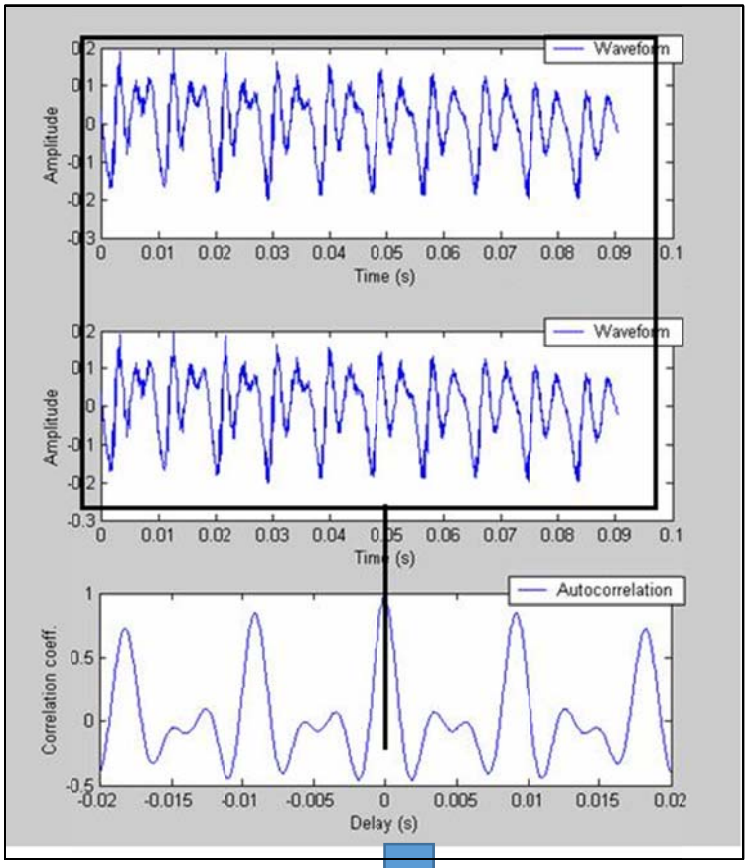
Suppose the waveform on the top right is one of the samples detected by *OctaPitch*. We can instantly see where the waveform might be repeating, so this example will be easy to showcase how the autocorrelation function works.

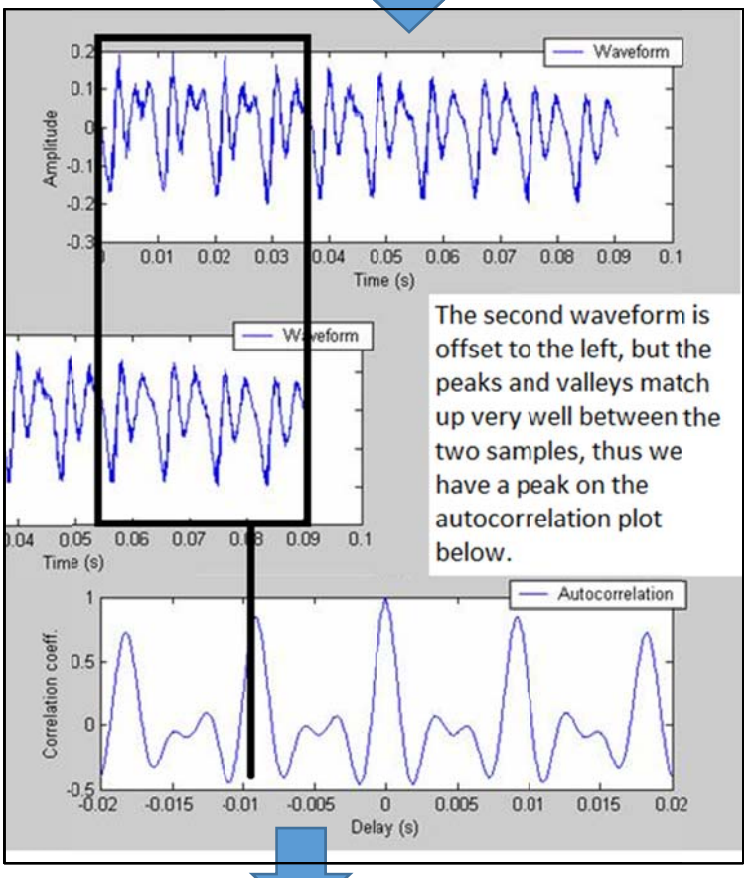
Below the waveform is just a second copy of the sample, and at the bottom we have the plot of our autocorrelation function. We can imagine the autocorrelation function as a plot of how well the two sample copies line up with each other. The X-axis describes how far the sample-copy is offset from the original, and the Y-axis describes how well the two samples line up. If the peaks line up very strongly, then the Y value will be high. If the peaks of the two samples do not line up at all, then there is a lower Y value.

The X-axis of the autocorrelation plot is referred to as the delay or the lag, which describes how the sample-copy is placed with respect to the original sample. When the lag is zero, we see that the autocorrelation plot reaches its maximum. This is because the sample-copy has not been shifted at all. When there is a negative X value, the copy was shifted to the left, and when X is positive the copy was shifted to the right.

The Y-axis of the autocorrelation is the correlation coefficient. If the sample and the sample-copy line up well, this will result in a higher correlation coefficient, and if they do not line up well this results in a lower correlation coefficient. When we have the finished autocorrelation, we can measure the distance between its maximums and perform some quick math to find the fundamental frequency of the initial waveform. This fundamental is then used by *OctaPitch* to find the note.

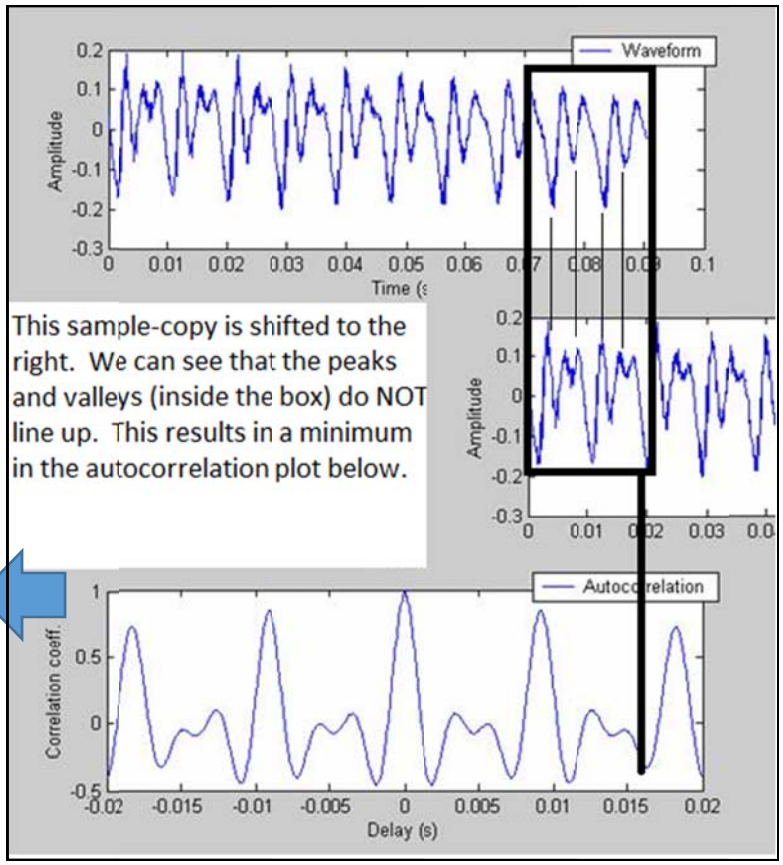
The picture on the bottom right is the beginning of the offset performed in Autocorrelation. The middle waveform starts on the left side and progresses to all the way until it is completely offset to the right. This is shown on the next page.



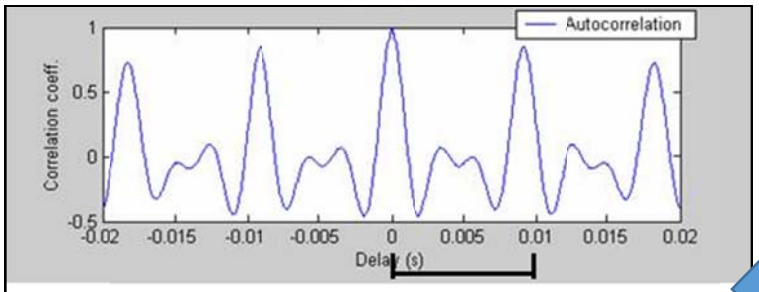


Autocorrelation Theory Continued

On the right, we see the second waveform continue shifting to the right. The autocorrelation continues to be drawn based on how well the peaks of the two waveforms line up as they are shifted across each other.



This sample-copy is shifted to the right. We can see that the peaks and valleys (inside the box) do NOT line up. This results in a minimum in the autocorrelation plot below.



From this autocorrelation plot, we can find the period of the sample by checking how far the peaks are from each other. In this example, the sampling period is about 0.01 seconds.

The sampling period is related to the waveform period in this way:

period= sampling period * (1 / sample rate)

fundamental frequency = 1 / period

The fundamental frequency can then give us the pitch.

Above is the final result. In the middle, we see the Delay is equal to 0. This is a maximum, and this is always the case. The Delay is also referred to as the Lag, and its value shows the offset of the second waveform in respect to the first. So with a Delay of 0, there is a maximum because there is virtually no difference between the waveforms, because they haven't been shifted.

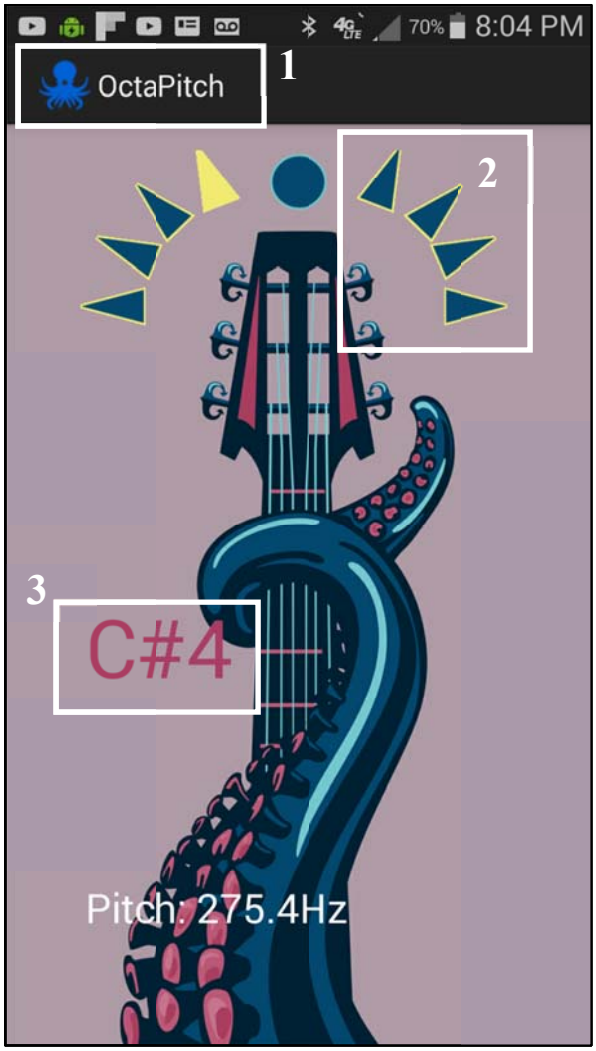
But as you continue shifting we see peaks and valleys develop. The distance between peaks is the sampling period. From that sampling period, we can do some quick math to find the fundamental frequency, as shown to the bottom right.

The final value is the fundamental frequency, which is used as the pitch.

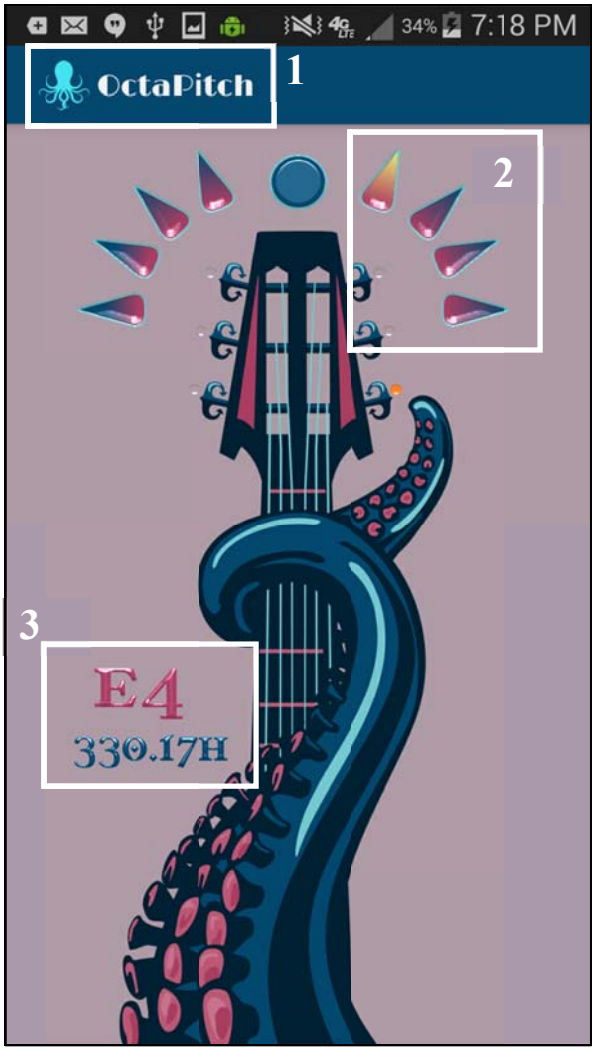
The fundamental frequency is the pitch heard by OctaPitch. This autocorrelation example is performed on every audio sample taken from the microphone, and OctaPitch takes 22,000 samples per second. Thus, the pitch is calculated constantly and continuously, as long as the app is running.

Graphic Features

Creating the display for *OctaPitch* involved a lot more than just the pitch recognition. There are several graphical effects that were applied to make *OctaPitch* look more professional, and more like it was drawn by a graphic artist instead of a computer. This is best exemplified by these two images of the app, the left picture taken about one month prior to the picture of the final version of *OctaPitch,* taken in June of 2017. There are three areas I would like to draw your attention to, as shown by the boxes below. I will go over these in detail on the next page.



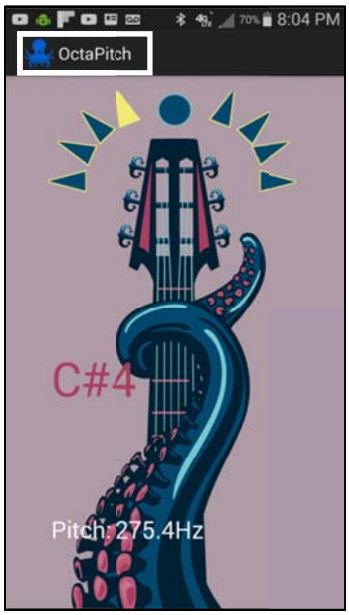
Old version of *OctaPitch* taken on May 14th 2017.



Final version of *OctaPitch* taken on June 9, 2017. This is with custom fonts, linear gradients and emboss filters applied.

Graphic Features Continued

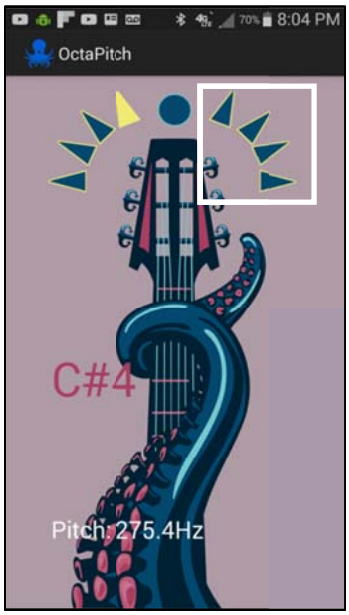
Here we can see the use of Custom Fonts, applied to the app title. Android only comes with a handful of fonts to choose from, so if we want to use something else we must import it and create a custom typeface with its own Class. This class can be found in the Appendix A.ii



Here we can see the use of Linear Gradients applied to the triangles. In the old version, we can see that the triangles were a solid blue color, with a yellow outline.

In the final version of *OctaPitch,* Linear Gradients have been applied to make the triangles fade from purple to blue, and then each was given a light blue outline. The linear gradient code accepts two different colors and two points, and from those two points a line is drawn and everything in that line is faded between those two colors. In order to control the triangles independently, a separate linear gradient was used for each. (Eight Linear Gradients total) Also, each linear gradient needed to take into account the specific location of the triangle it was applied to, in order to make each triangle's fade consistent with each other.

The Linear Gradient code is in Appendix A.iii



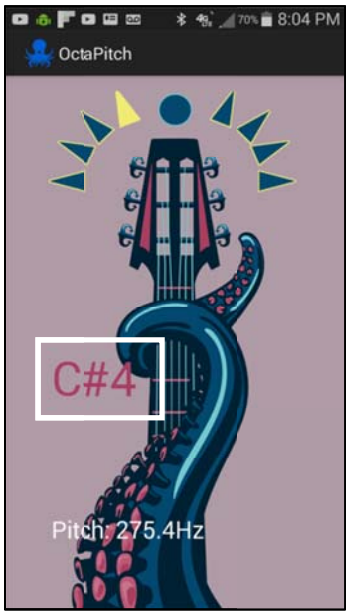
Graphic Features Continued

Here we can see the use of an Emboss Filter. An emboss filter applies a light white shading on an object to give it a three dimensional feel. We can see this most demonstrated by the "E4" in the picture on the right. At the top of the "E4" we can see there is a white line, and because of this the text appears a little 3D or even a little wet.

These are simple graphic filters that essentially emulate a fake light source. You can change the position this light source is in respect to the object you are lighting. This emboss filter was also applied to the triangles. You can see that each triangle has a small white shade on the bottom, as if a light was illuminating from the guitar below them.

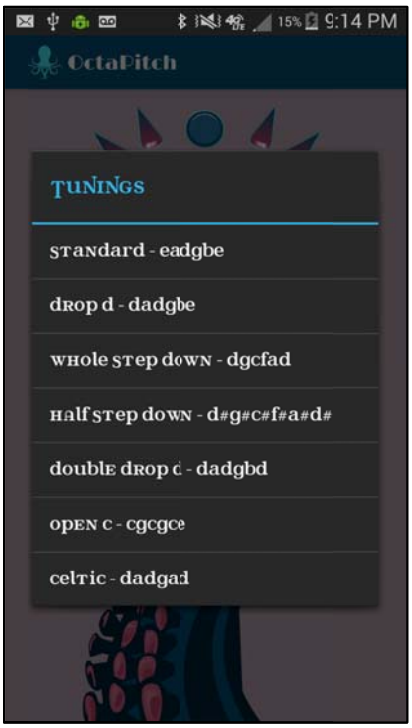
The circle was also given an emboss filter to make it appear more like a 3D button, instead of just a circle.

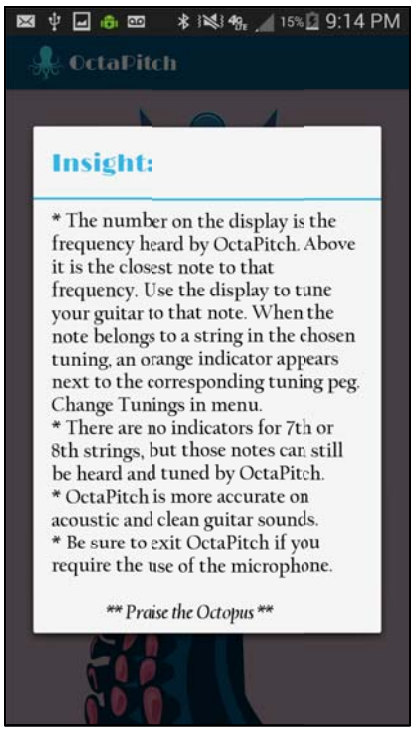
The code for the Emboss Filter is in Appendix A.iv



OctaPitch User Menus

OctaPitch includes 3 menus. The first is a Tuning Menu, which allows the user to select different tunings for the tuner. This changes the notes that are synced to each string. Each string has an indicator on the display, so the user knows when the note belongs to a certain string. This indicator prevents confusion because even though the guitar string may be the note E, the octave matters, and in this case it may be E4 we want to tune to, and not E3. The Tuning Menu allows the user to change this. This menu is shown below on the left.





In the middle is the Insight menu. This is simply a "How to use OctaPitch" screen that pops up when the user selects the menu item.

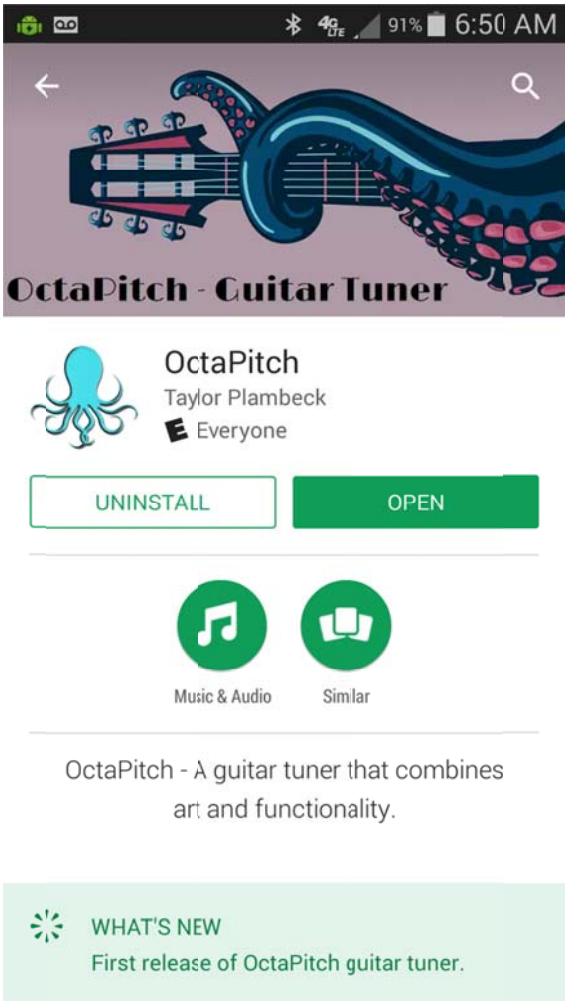
On the right is the "About OctaPitch" screen that pops up when the user selects it in the menu. It includes my name and email address, as well as an acknowledgement to the credits in the Play Store in case someone needs them

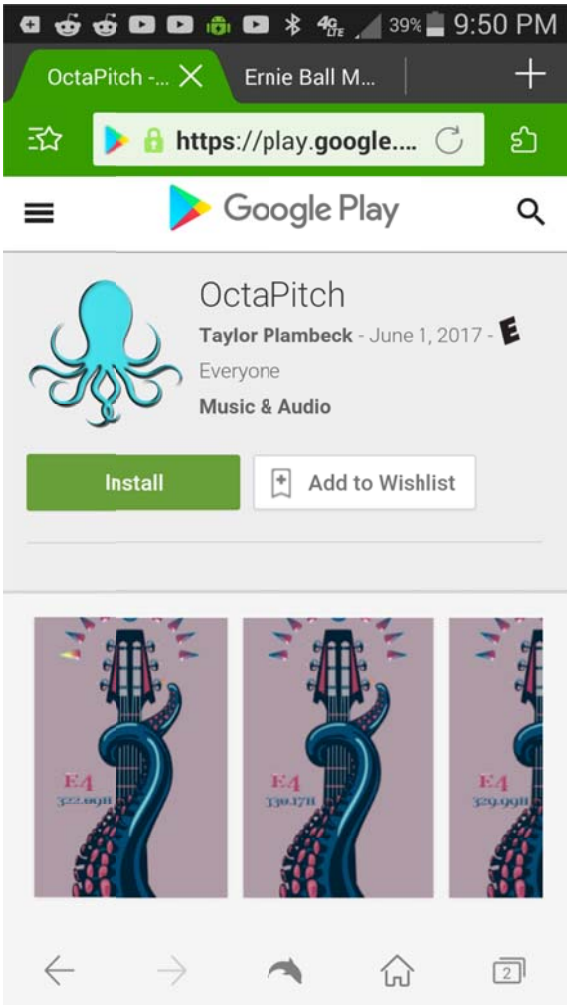
The code for the menus can be found in the Appendix A.vi

Publishing OctaPitch to the Google Play Store

In my opinion, *OctaPitch* would not have seemed complete if I did not upload it to the Play Store for everyone else to enjoy. This resulted in several extra steps, which I will briefly cover below.

- *OctaPitch* was tested on several different devices using an emulator. An emulator was used to create a virtual machine for various Android devices. The app was tested with many different screen sizes and Android API versions.
- A Google Developer Account had to be created, which is a one-time $25 fee in order to upload apps to the Play Store.
- Pictures had to be created to support the Play Store page, including an app icon and the page banner.
- A Privacy Policy needed to be created because *OctaPitch* uses the microphone. This means that when the app is installed it informs the user that *OctaPitch* needs access to their microphone, and because I could technically record a user's personal conversation I had to create a Privacy Policy that holds me liable that I will use the microphone only for pitch recognition.





Publishing OctaPitch to the Google Play Store Continued

On the page for *OctaPitch* in the Play Store there is a part where you need to write the description of the app. This was essentially a brief advertisement that had to be written to entice user's to use *OctaPitch* over the other guitar tuner apps. This "ad" is below.

OctaPitch – Guitar Tuner

App or Art?

OctaPitch combines a high standard for pitch recognition and an art style of unwavering mystique. Seek counsel from the Octopus for any musical insight you may need. Why *not* let your tuner inspire you?

OctaPitch comes with no ads, no loading screens, and no charge. OctaPitch recognizes every note in the guitar frequency spectrum and offers preset tunings to choose from.

The elegance of OctaPitch transcends into its functionality as well. The number on the display is the frequency heard by OctaPitch. Above it is the note closest to that frequency. The display guides you so the guitar is perfectly tuned to that note. When the note belongs to a string in the chosen tuning, an indicator will turn orange next to the corresponding tuning peg. You can use the Tuning Menu to change which group of notes are synced to the indicators on the pegs. Because of the design on the artwork, there are no indicators for the 7^{th} and 8^{th} strings, but those notes can still be heard and tuned by OctaPitch.

*** Praise the Octopus ***

Work Breakdown

This is a general overview of the development cycle of *OctaPitch.* This was a senior project, so the project spanned over three, ten week quarters. Getting the pitch recognition working proved to be problematic, and wasn't solved until later into the second quarter. The last quarter is where 99% of the user interface and display was created.

A: Fall Quarter 2016 –

- Research regarding pitch recognition. Figuring out if I should use a library or code it myself. Different techniques of pitch recognition, and theory behind it. More research involving what is difficult about coding such algorithms.
- Acquisition, installation, and familiarization of all necessary software:
    - Android Studio – For all coding work
    - Adobe Photoshop – For all graphic work
    - FontForge – Custom Font Creator
- Creation of a OneDrive to store all *OctaPitch* versions.
- Creation of a regular Google Account in preparation for the Developer Account
- Learning Android app creation. How the AndroidManifest works, how layouts and views work.

B: Winter Quarter 2017 –

- Trying to figure out a specific method to find pitch, ended up not being able to code it myself, as I had to learn how to use Android Studio and learn the basic coding of an Android app in general. Because of these two unfamiliarity's, I chose to find a library that could provide an FFT that I could use for my Autocorrelation.
- Implemented the audio recording of my app, was able to successfully pass the audio samples to the FFT library and perform my Autocorrelation.
- This important step took several weeks of Spring quarter. At the end of Spring quarter, *OctaPitch* could listen to an audio and output the pitch as a floating point number onto the screen.

Work Breakdown Continued

C: Spring Quarter 2017 –

- Now that *OctaPitch* could hear and interpret pitch, I developed an array that held every single note that could be played on a guitar. This also includes notes found on down-tuned 8 string guitars, and going as high as the 24th fret on the highest string, which nobody would ever tune, but nonetheless, *OctaPitch* can hear it and recognize it.
- From this array of notes, *OctaPitch* was then able to see how far your guitar was from the correct note. This was calculated with a simple distance function, and finds not only how far you are from it, but in which direction. The degree of separation (and in what direction) was now coded so *OctaPitch* could dynamically change the display to explain to the user how they should tune the guitar.
- The display was then completed:
    - Found background image used in app on Shutterstock
    - Created 8 triangles, all angled in different directions around the guitar to indicate which way the user should tune their guitar
    - Drew a circle to indicate when the guitar is tuned
    - Drew 6 small circles around the guitar's headstock, to indicate when a Note belongs to a certain string in the current chosen tuning
    - Added the Note Name to the screen, to compliment the frequency
    - Added Linear Gradients to the Triangles to fade between colors
    - Added Emboss Filters to all the shapes and all texts to give a 3D look
    - Added custom fonts used for the app title, the text on the screen, as well as fonts used for the Menus
    - Added Menus
        - How to Use OctaPitch
        - About OctaPitch
        - Tunings: Change which notes the small 6 circles are synced to
- Created the App Icon
- Created the App Name
- Created and printed the poster for the Symposium
- Created the page for the Play Store:
    - Page Banner
    - App Icon
    - App Description
- Created Privacy Policy

Development Cost

- $15 - License for the background picture. Purchased from Shutterstock.com
- $60 - Cost of canvas and poster display
- $25 - Google Play Developer Account (One-time fee to host apps on Play Store)
- A couple hundred hours of Taylor's free time

References

- TarsosDSP Library for Android: Includes FFT used to recreate the YIN Algorithm
- Background Image used in the app was created by user "ne2pi" on Shutterstock. Image ID# 328968350
- Fonts used in the Application (and on poster):
    - **LimeLight Font** - Copyright (c) 2011 by Sorkin Type Co (www.sorkintype.com)
    - **WATER STREET FONT** - Darren Rigby (dart@puzzlers.org)
    - **WATER STREET DETOUR FONT**- Darren Rigby (dart@puzzlers.org)
- OctaPitch app icon – Created by Felix Bronnimann from The Noun Project
- De Cheveigne, Alain, and Hideki Kawahara. "YIN, a Fundamental Frequency Estimator for Speech and Music." The Journal of the Acoustical Society of America 111.4 (2002): 1-14.
- Software:
    - Android Studio
    - Adobe Photoshop
    - FontForge
    - OneDrive
- Special thanks to Dr. Lin for being the Project Advisor

Appendix

Appendix A: Java Code

(i) Autocorrelation Class

```
public final class Yin implements PitchDetector {
		private static final double DEFAULT_THRESHOLD = 0.20;
		public static final int DEFAULT_BUFFER_SIZE = 2048;
		public static final int DEFAULT_OVERLAP = 1536;
		private final double threshold;
		private final float sampleRate;
		private final float[] yinBuffer;
		private final PitchDetectionResult result;
		/**
		 * Create a new pitch detector for a stream with the defined sample rate.
		 * Processes the audio in blocks of the defined size.
		 *
		 * @param audioSampleRate
		 *			The sample rate of the audio stream. E.g. 44.1 kHz.
		 * @param bufferSize
		 *			The size of a buffer. E.g. 1024.
		 */
public Yin(final float audioSampleRate, final int bufferSize) {this(audioSampleRate, bufferSize,
DEFAULT_THRESHOLD);}
		/**
		 * Create a new pitch detector for a stream with the defined sample rate.
		 * Processes the audio in blocks of the defined size.
		 *
		 * @param audioSampleRate
		 *			The sample rate of the audio stream. E.g. 44.1 kHz.
		 * @param bufferSize
		 *			The size of a buffer. E.g. 1024.
		 * @param yinThreshold
		 *			The parameter that defines which peaks are kept as possible
		 *			pitch candidates. See the YIN paper for more details.
		 */
public Yin(final float audioSampleRate, final int bufferSize, final double yinThreshold) {
			this.sampleRate = audioSampleRate;
			this.threshold = yinThreshold;
			yinBuffer = new float[bufferSize / 2];
			result = new PitchDetectionResult();
		}
	//The main flow of the YIN algorithm. Returns a pitch value in Hz or -1 if
		// no pitch is detected. return a pitch value in Hz or -1 if no pitch is detected.
```

Autocorrelation Class Continued

```
public PitchDetectionResult getPitch(final float[] audioBuffer) {
        final int tauEstimate;
        final float pitchInHertz;
        // step 2
        difference(audioBuffer);
        // step 3
        cumulativeMeanNormalizedDifference();
        // step 4
        tauEstimate = absoluteThreshold();
        // step 5
        if (tauEstimate != -1) {
            final float betterTau = parabolicInterpolation(tauEstimate);
            // step 6
            // TODO Implement optimization for the AUBIO_YIN algorithm.
            // 0.77% => 0.5% error rate,
            // using the data of the YIN paper
            // bestLocalEstimate()
            // conversion to Hz
            pitchInHertz = sampleRate / betterTau;
        } else{
            // no pitch found
            pitchInHertz = -1;
        }
        result.setPitch(pitchInHertz);
        return result;
    }
```

Step 2:

```
private void difference(final float[] audioBuffer) {
        int index, tau;
        float delta;
        for (tau = 0; tau < yinBuffer.length; tau++) {
            yinBuffer[tau] = 0;
        }
        for (tau = 1; tau < yinBuffer.length; tau++) {
            for (index = 0; index < yinBuffer.length; index++) {
                delta = audioBuffer[index] - audioBuffer[index + tau];
                yinBuffer[tau] += delta * delta;
            }
        }
    }
```

Autocorrelation Class Continued

Step 3:

```
private void cumulativeMeanNormalizedDifference() {
int tau;
yinBuffer[0] = 1;
float runningSum = 0;
for (tau = 1; tau < yinBuffer.length; tau++) {
        runningSum += yinBuffer[tau];
        yinBuffer[tau] *= tau / runningSum;
}
}
```

Step 4:

```
private int absoluteThreshold() {
// Uses another loop construct
// than the AUBIO implementation
int tau;
// first two positions in yinBuffer are always 1
// So start at the third (index 2)
for (tau = 2; tau < yinBuffer.length; tau++) {
        if (yinBuffer[tau] < threshold) {
                while (tau + 1 < yinBuffer.length && yinBuffer[tau + 1] <
yinBuffer[tau]) {
                        tau++;
                }
                // found tau, exit loop and return
                // store the probability
                // From the YIN paper: The threshold determines the list of
                // candidates admitted to the set, can be interpreted as the
                // proportion of aperiodic power tolerated
                // within a periodic signal.
                // Since we want the periodicity and and not aperiodicity:
                // periodicity = 1 - aperiodicity
                result.setProbability(1 - yinBuffer[tau]);
                break;
        }
}
// if no pitch found, tau => -1
if (tau == yinBuffer.length || yinBuffer[tau] >= threshold) {
        tau = -1;
        result.setProbability(0);
        result.setPitched(false);
} else {
        result.setPitched(true);
}

return tau;
```

Autocorrelation Class Continued

Step 5:

```
        private float parabolicInterpolation(final int tauEstimate) {
        final float betterTau;
        final int x0;
        final int x2;

        if (tauEstimate < 1) {
                x0 = tauEstimate;
        } else {
                x0 = tauEstimate - 1;
        }
        if (tauEstimate + 1 < yinBuffer.length) {
                x2 = tauEstimate + 1;
        } else {
                x2 = tauEstimate;
        }
        if (x0 == tauEstimate) {
                if (yinBuffer[tauEstimate] <= yinBuffer[x2]) {
                        betterTau = tauEstimate;
                } else {
                        betterTau = x2;
                }
        } else if (x2 == tauEstimate) {
                if (yinBuffer[tauEstimate] <= yinBuffer[x0]) {
                        betterTau = tauEstimate;
                } else {
                        betterTau = x0;
                }
        } else {
                float s0, s1, s2;
                s0 = yinBuffer[x0];
                s1 = yinBuffer[tauEstimate];
                s2 = yinBuffer[x2];
                // fixed AUDIO implementation, thanks to Karl Helgason:
                // (2.0f * s1 - s2 - s0) was incorrectly multiplied with -1
                betterTau = tauEstimate + (s2 - s0) / (2 * (2 * s1 - s2 - s0));
        }
        return betterTau;
    }
}
```

Appendix A: Code

(ii) Custom Typeface Class

```
package be.tarsos.taylor.plambeck.android.octapitch;

/*
                    ***  Created By:  ***
                        Taylor Plambeck
                      OctaPitch - Guitar Tuner
                Built from September 2016 - June 2017
     California Polytechnic University of Pomona
   College of Computer Engineering Senior Project
           (Pitch Recgonition Credit to TarsosDSP!)

 */

import android.annotation.SuppressLint;
import android.graphics.Paint;
import android.graphics.Typeface;
import android.text.TextPaint;
import android.text.style.TypefaceSpan;

@SuppressLint("ParcelCreator")
public class CustomTypefaceSpan extends TypefaceSpan {

    private final Typeface newType;

    public CustomTypefaceSpan(String family, Typeface type) {
        super(family);
        newType = type;
    }

    @Override
    public void updateDrawState(TextPaint ds) {
        applyCustomTypeFace(ds, newType);
    }

    @Override
    public void updateMeasureState(TextPaint paint) {
        applyCustomTypeFace(paint, newType);
    }
```

Custom Typeface Class Continued

```
private static void applyCustomTypeFace(Paint paint, Typeface tf) {
        int oldStyle;
        Typeface old = paint.getTypeface();
        if (old == null) {
            oldStyle = 0;
        } else {
            oldStyle = old.getStyle();
        }
        int fake = oldStyle & ~tf.getStyle();
        if ((fake & Typeface.BOLD) != 0) {
            paint.setFakeBoldText(true);
        }
        if ((fake & Typeface.ITALIC) != 0) {
            paint.setTextSkewX(-0.25f);
        }
        paint.setTypeface(tf);
    }
}
```

Appendix A: Code

(iii)Linear Gradient

Each triangle was given two different shaders, each made from a Linear Gradient. Each triangle has its default Blue->Pink gradient, and then a Pink->Yellow gradient when the triangle is "Illuminated"

```
//  -- SHADERS –

Shader LT1yellowToPink = new
LinearGradient(186,0,198,65,Color.parseColor(YELLOW_PALE), Color.parseColor(PINK),
Shader.TileMode.CLAMP);

Shader LT1blueToPink= new LinearGradient(186,0,198,65, Color.parseColor(BLUE_DARK),
Color.parseColor(PINK), Shader.TileMode.CLAMP);

//set shader to default; pink bottom with blue pointed tip
paintLT1.setShader(LT1blueToPink);

//draw all paths. This results in filled triangles using shaders above
canvas.drawPath(L1path, paintLT1);
```

(iv)Emboss Filter

Emboss filters were used to make each object on the display appear more 3D. Below is an example of applying the emboss filter to the text on screen, so the Note Name and the Pitch Frequency appear 3D and wet.

```
final EmbossMaskFilter textEmbossFilter = new EmbossMaskFilter
(new float[]{0f, 0.3f, 0.3f},0.85f,3f,4f);


//set the textViews to textEmbossFilter above
textNoteName.getPaint().setMaskFilter(textEmbossFilter);

textPitch.getPaint().setMaskFilter(textEmbossFilter);
```

Appendix A: Code

(v) Draw Triangle at an Angle

This code accepts the triangle's center, triangle's base and height, and the angle as parameters and draws the triangle accordingly. This code had to be written 8 ways, because there are 8 different triangles each with a different location and angle.

```
float angle = (float) Math.toRadians(-22); // Angle to rotate, 0 degrees is pointing up, assuming base > height
        float height =72;//was 78
        float width =42;//was 45
        float centerX = 200;// Display coordinates where triangle will be drawn
        float centerY = 34;//used to be 30
        float x1 = centerX;// Vertex's coordinates before rotating
        float y1 = centerY - height / 2;
        float x2 = centerX + width / 2;
        float y2 = centerY + height / 2;
        float x3 = centerX - width / 2;
        float y3 = y2;

float a0x = (float) ((x1 - centerX) * Math.cos(angle) - (y1 - centerY) * Math.sin(angle) + centerX);//3 points

float a0y= (float) ((x1 - centerX) * Math.sin(angle) + (y1 - centerY) * Math.cos(angle) + centerY);

float b0x = (float) ((x2 - centerX) * Math.cos(angle) - (y2 - centerY) * Math.sin(angle) + centerX);

float b0y = (float) ((x2 - centerX) * Math.sin(angle) + (y2 - centerY) * Math.cos(angle) + centerY);

float c0x= (float) ((x3 - centerX) * Math.cos(angle) - (y3 - centerY) * Math.sin(angle) + centerX);

float c0y = (float) ((x3 - centerX) * Math.sin(angle) + (y3 - centerY) * Math.cos(angle) + centerY);

L1path.moveTo(a0x, a0y);
L1path.lineTo(b0x, b0y);
L1path.lineTo(c0x,c0y);
L1path.lineTo(a0x,a0y);
L1path.lineTo(a0x,a0y);//redundant call, but Path would not connect otherwise

canvas.drawPath(L1path, paint);
```

Appendix A: Code

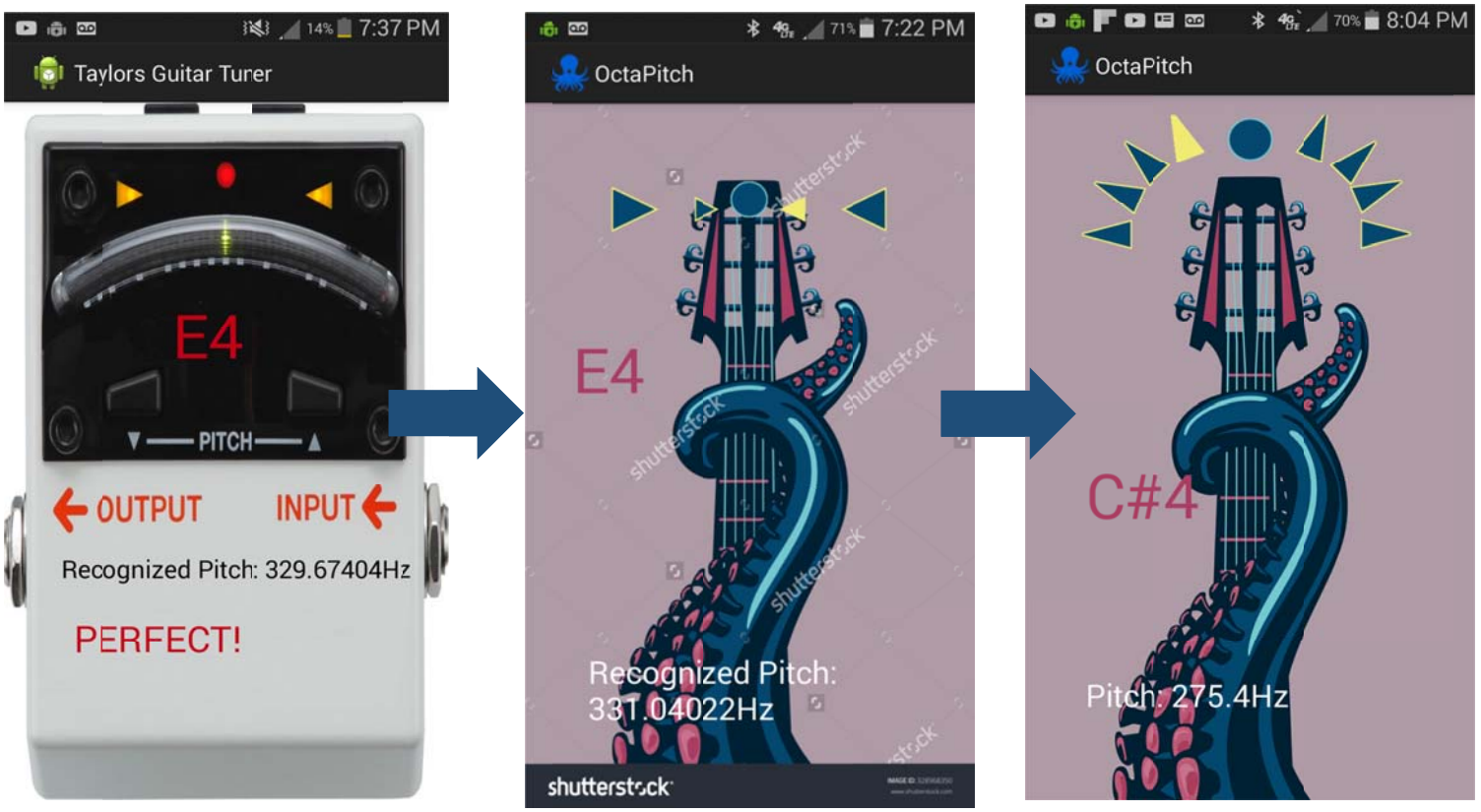
(vi)Menu Code

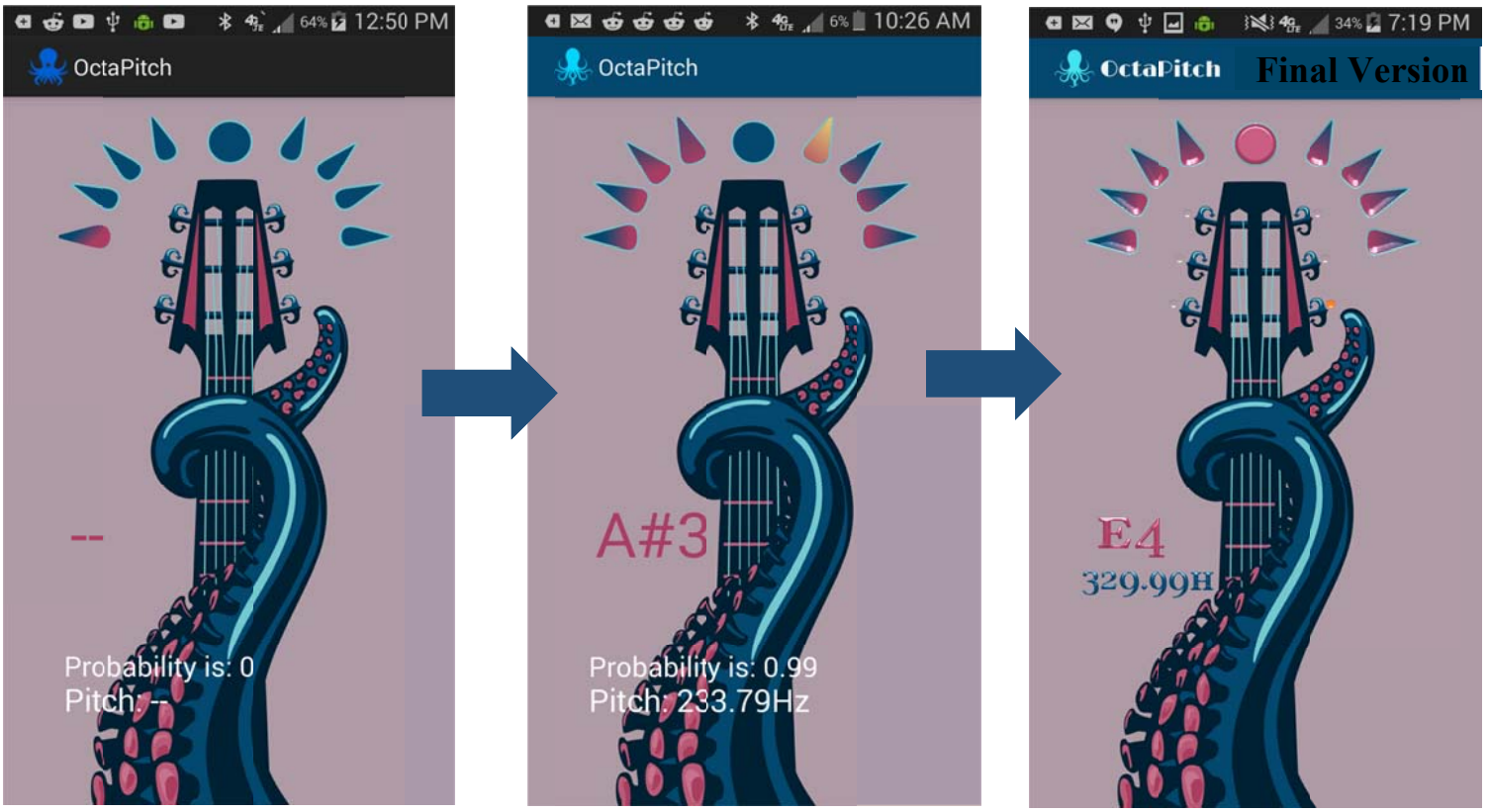
Below is the code for just one of the menus. This is in the actionListener() that checks to see fi the user pressed the menu item. This code is what executes the dialog box that pops up on the screen when the user presses the menu item "About OctaPitch".

```
if(selectedTuning==R.id.action_about)     //if the About OctaPitch is pressed, create alertDialog
{

AlertDialog.Builder aboutDialogBuilder = new AlertDialog.Builder(OctaPitch_Main.this);
        //build in context of activity

Typeface TitleFont=Typeface.createFromAsset(getAssets(), "limelight.ttf");         //declare fonts
Typeface WaterContent_Font = Typeface.createFromAsset(getAssets(), "water2final.ttf");

//creates string for title, line below applies font
SpannableString dialogTitleSpanString= new SpannableString("About OctaPitch:");
dialogTitleSpanString.setSpan(new CustomTypefaceSpan("",TitleFont),0,
dialogTitleSpanString.length(), Spanned.SPAN_EXCLUSIVE_INCLUSIVE);

aboutDialogBuilder.setTitle(dialogTitleSpanString);

SpannableString dialogContentSpanString= new SpannableString("* CreaTOr:  taylOr
PlambeCk\n  taylOr@ligHtHOuseCOmm.COm\n\n* CrediTs CaN be fOuNd in tHe
dOCumeNTATiOn On tHe GOOGle Play STOre");

dialogContentSpanString.setSpan(new CustomTypefaceSpan("",WaterContent_Font),0,126,
Spanned.SPAN_EXCLUSIVE_INCLUSIVE);         //apply content font

aboutDialogBuilder.setMessage(dialogContentSpanString).setCancelable(true);    //set content
AlertDialog aboutDialog = aboutDialogBuilder.create();      //create the dialog box
aboutDialog.show();   //show dialog
aboutDialog.setCanceledOnTouchOutside(true);      //we can end the dialog box with the BACK
button OR just by touching somewhere else

}
```

Appendix B: OctaPitch Version Slideshow

Below are several pictures taken through the development of OctaPitch, to help showcase how much work was actually done to get the app to the professional polish it has in the final version.





Appendix C: Version Documentation

Below is the documentation that shows every change made to the 30+ different versions of OctaPitch through its development. This is simply to show how much work was really performed. Most of the document corresponds to the last 5 weeks of development before the symposium, and before the release on the Play Store.

Version 0.1 Pitch being displayed to the user, using TarsosDSP.
Date: 3/26/2017 PM
Version 0.2 slightly edited for initial demo to Lin in February 2017
Date: 4/17/2017 PM
Version 0.3 Added textviews, organized basic project structure
Date: 4/19/2017 PM
Version 0.4 NoteFinder class implemented
Date: 4/19/2017 PM
Version 0.5-0.6 Shapes can now be drawn onto background. MASSIVE DESIGN CHANGE: DYNAMIC SHAPE DRAWS INSTEAD OF CHANGING BACKGROUND IMAGES
Date: 4/27/2017 AM
Version 0.7 Shapes are dynamic, background is now Octapus
Date: 4/27/2017 PM
Version 0.8 Four triangles are spaced and dynamic. Aligned flat with horizon
Date: 5/6/2017 PM

0.9.0 - Added changes discussed with Dad on 5/12. This includes
-Decimal Format for Pitch output is now #.##
-Changing NoteName and Pitch from N/A and -1 to --
-Now added purchased background with 720x1280 resolution
-Added the backend algorithm for the extra triangles, we now support 4 triangles on each side, with smaller denominations
-Cleaned up comments and code, moved non-used code to bottom
(This is the last version before I try to code in the 4 extra triangles)

0.10.0 Changes on 5/14 Includes:
-All functionality for the triangles has been split into 8 triangles, instead of 4 triangles
-Their denominations are in the code, 49%,35%,20%,15% measurements

0.11.0 - Changes on 5/15 Includes:
-Code has been thoroughly cleaned up. All old replaced code has been commented en masse at bottom of .java's
-Found probability, will implement later

0.12.0 Changes on 5/17 Includes:
-Probability is now displayed. Will return .99 when it is sure, background noise can contribute to it falling below 0.9
-Will implement code to prevent display update if we are below the 90% certainty threshold
-Pitch recognition algorithm changed from Fast-Yin to Yin. Algorithm will be explained in documentation
-Triangles are now rounded. Edges are smooth, CornerPatheffect implemented heavily
-Gradient effect is applied on the L4 triangle as a proof of concept. Will implement the Shaders on the other seven triangles later

0.13.0 Changes on 5/18 Includes:
-Added Gradients to all triangles. This includes making 8 SEPARATE paint brushes
-Triangles change gradients now instead of the solid colors
-App has new (temp?) icon, as well as the banner on top being a more relevant color
-Android Manifest should refresh and lock portrait mode, as well as close when home button is pressed
-Gradients need to be perfected

0.14.0 Changes on 5/18 Includes:
-Gradients nearly perfected
-Probability is still not implemented
-Made several new Icons: cropped, background removed and in various colors
-Changed Android Manifest: Screen is locked on portrait mode, and doesn't allow the app to suspend, IE closes on BACK or HOME button presses -CHANGED LATER
-Added an OnStop() method to close the dispatcher line when the app is exited
-When I saved 0.14.0 it would not run. I think it is because of some OnStop() or OnDestroy() bug. Fix in next version

0.15.0 Changes on 5/20 AM Includes:
-Probability partially implemented. Currently works for anything over 90% but when no pitch is captured it still shows the last recognized name and frequency
-Will have the app draw the basic GUI always. Then if above 90% we can light up triangles and display pitch name and value
-This requires a large restructuring of the program, moving around the parts that will always run and the parts that are conditional on the 90%

0.16.0 Changes on 5/20 PM Includes:
-Probability is now implemented. Any pitch with a likelihood of UNDER 90% being correct is now IGNORED
-The UI acts as though it is not detecting anything if we have a lower probability
IE: The UI is on triangle LT1. Pitch is 100, note is A.
If we were to have a background noise, or somebody talk over it, there is no triangle and Pitch and Note are "--"
-Design choice: Do I keep the blank UI on sub-90% recognition, or do I have it display the PREVIOUSLY >90% RECOGNIZED PITCH?

0.17.0 Changes on 5/20 PM Includes:
-Probability version 2 has been written. It is currently commented out and UNTESTED
-Code is written to keep previous shape displayed through a misread. This uses a previousFlag, as well as a firstRunFlag
-This firstRunFlag also allows for a fade-in of the shapes upon startup, Animation is at end of UIThread
-Removed OnStop() and the Manifest suspension options from 0.14. These are now commented out, I could not get the app to run consistently! (portrait kept)

0.18.0 Changes on 5/22 PM Includes:
-Probability Version 2 that I wrote in 0.17 has now officially been implemented in the app
-If the app misreads a pitch and probability is below 90%, not only is the UI NOT updated, it displays OLD info from the PREVIOUSLY SUCCESSFULLY READ PITCH
-These changes prevent the rapid flickering that the app used to do. It flickers less because we ignore misreads, and when misreads (inevitably) happen we don't
change the UI at ALL. This leads to a pretty stable reading, that doesn't go TOO insane with crosstalk or background noise
-Finally added the TUNING INDICATORS. These are small circles next to the tuning pegs on the background guitar. These light up when the reference note matches
up to the corresponding string that each indicator is tied to. This will allow for different tuning behaviors (NEED TO IMPLEMENT SETTINGS MENU)
Having these indicators means the app doesn't require the user to have memorized which octave the notes are for various tunings
-These tuning indicators are referred to MINI_CIRCLES. They have their own paintbrush, and all location and radii are coded via declared constants at the top
-Moved UI around: Pitch # is now right below the NOTENAME, both bold and large, with EMBOSSFILTER applied. Removed the "--" for the Pitch (Still on NOTENAME)
-You can add different algorithms on the same dispatcher. Mildly slows down App but does prove that they are relatively similar in effectiveness

0.19.0 Changes on 5/23 Includes:
-Random change: Night after i finished 0.18. Emboss filter is now added to triangles to give a little dimension to them. Looking good..
-Added Font to the Action Bar so OctaPitch reads in the Limelight font. NOTENAME is in the water2 font
-Class was added for the sole purpose of handling the custom font on the action bar
-Added an emboss filter to the watery NOTENAME so it could have the wet look and the other shapes can have theirs
-Experimented with the emboss filters for an array of different looks. Notes are inside comment in program

0.20.0 Changes on 5/24 PM Includes:
-Added a Tunings Menu, included various tunings and implemented how they can change the MINI_CIRCLES
-Added a popup dialog that simply displays the ABOUT OctaPitch info. My name, Github, credits
-All Menus have appropriate fonts. The bioshock-esque font is used for the Title of the app and the ABOUT menu
-Merged fonts into my own custom font with FontForge. Combining the two water street fonts allow for perfect implementation of the tunings menu. (Water2fixed)
-Moved the MINI_CIRCLES into better locations. Should I change color?
-Did not change the emboss filters from yesterday, still needs to be done.

0.21.0 AND 0.22.0 Changes on 5/25 PM Includes:
-Fixed fonts for menus
-Added padding with DP in manifest, this makes the textNoteName and textPitch more consistent in placement
-Did not end up scaling my bitmap in anyway. Seems to work well enough on the screens 1 or 2 levels above mine
-Installed genymotion and many different Android emulators for testing
-Ordered poster display and guitar strap for acoustic

Testing Note:
-APP DOES NOT WORK ON SCREEN THAT ISNT PREDOMINANTLY VERTICAL
-NEXUS 9 1536x2048
-After research, we see that tablets are not going to work for OctaPitch in its current form

0.24.0 Changes on 5/25 PM Includes:
-Finalized the placement for all 8 triangles
-Finalized the gradient for all 8 triangles
-The emboss filters have been split up for each shape, emboss filters are VERY CLOSE TO BEING FINAL
-Added in some screen compatability stuff in the manifest
-Found the exact points that make up every triangle
-Tested various screens and phones. In its current state, OctaPitch seems ready for deployment for normal handheld devices, the background prevents tablet viability
-Took off starting animation
-Changed the pitchInHZ to the dark blue
-Took out a lot of commented leftover code
-Moved what I could to the OnCreate(). The emboss filters can be moved back when I finalize them. FONTS DO NOT WORK OUTSIDE OF THEIR FUNCTIONS
-Rearranged some code and also got rid of some warnings
-Changed the targetSDK to 25, as I tested it up to that Android API

0.25.0 Changes on 5/28 AM Includes:
-Finalized positions for the MINI_CIRCLES
-Finalized the emboss filters and the gradients for all shapes
-Finalized the location for the textviews

0.26.0 Changes on 5/28 PM Includes:
-Made a function to set all parameters for the paintbrushes, called preparePaintbrushes() called at bottom of activity
-Moved the filter declarations to the top of the activity, and then when Oncreate() is called, we instantly call preparePaintbrushes()
-This makes it so we never have to change any of the paint parameters again after runtime. We dynamically switch shaders for tuning
-Note: This is the last save before I try to modify the build path and the activity and all that shit.

0.27.0 Changes on 5/31 PM Includes:
-Startup Animation removed
-Probability removed from UI
-Fonts and texts finalized
-This is the last version before i change package names and clean out the comments!
-Added more notes to accomodate 8 strings
-Added Open C and Celtic tunings

0.28.0 Changes on 6/1 PM Includes:
-Renamed package name to: be.tarsos.tarsos.plambeck.android.octapitch
-Added a version number
-Did the last lookover of the app!
-THIS IS THE FINAL VERSION BEFORE I CLEAN EVERYTHING OUT

0.29.0 Changes on 6/1 PM Includes:
-Cleaned out all unused code, added comments and organized the project for deployment
- THIS IS AN APP THAT CAN BE ON THE PLAY STORE

0.30.0 Changes on 6/1 PM Includes:
-Signed APK Version
-Removed Screen Limiter and also cleaned up the manifest

1.0.0
-Changed compile to 23
-Changed package to: be.tarsos.taylor.plambeck.android.octapitch
-CHANGED THE targetSDKversion BACK TO 20. For some reason 25 doesn't work
-This is the final working app store version, as used in OctaPitch - Guitar Tuner v3 in Play Store