CSCI188 Final Project Write-up – Pitchpipe for Android – Alisa Peters

The motivation behind this project was to learn a new platform, namely, the one that my personal phone runs on, and to create an application that I would find useful. Hence the Pitchpipe for Android idea was born. This final write-up will follow the structure of tasks which I described in the proposal.

The Basics

The first thing was to download the Android SDK, install the Eclipse plug-in, and get a "Hello, World" application running in the emulator. This all went quite smoothly, and the documentation was adequate. The next thing was to get the sample application running on my phone. That was more of a challenge. Finding the appropriate driver for my phone, so that my computer and Eclipse would recognize it as a smartphone available to deploy code to and not just a USB-storage device was not trivial. Documentation was weak, but after a search of several Android developer forums, I found sufficient advice to guide me in the right direction for my own trial and error. For my computer, a Sony VAIO, PCG-6G4L, once I finally got the correct driver to take precedence, the successful order of operations to enable on-device bugging is: 1) enable the Settings >> Applications >> Development >> USB debugging option on the phone *prior* to plugging it in; 2) plug in the phone via USB; 3) open Eclipse (it's possible that Eclipse could already be running and this would work, but I took no chances); 4) debug and select the active Android device. Even though it took a long time, I really wanted to get on-device debugging up and running prior to starting further application development due to the fact that one of my key application features – blowing into the microphone to activate the pitch pipe – would not be able to be tested using the emulator.

Playing Sound

The central functionality of a pitch pipe is the ability to play pitches to allow singers and/or instruments to tune-up to the same key. Thus, this was the first feature to be developed. I remembered that during the media lecture, there was a method which gave the ability to play tones at specified frequencies. I looked long and hard for such a method provided by the Android platform, but to no avail. However, the other thing was, I really wanted it to sound like a pitch pipe. My application probably fits more in to the novelty category, but still provides functionality, as well. I took my real pitch pipe, recorded the sounds on to my phone (with the exception of B-flat which is a piano key due to my pitch pipe being broken for that pitch), downloaded the recordings from my phone to my computer, and added them as resources to my project. Obviously, I could have just accessed the original recordings on my phone from the file system, but I needed to have a set of pitches that I could distribute with the application.

Once I had the pitches, I wanted a simple way to play them, just so I could test out that one part of functionality. The first iteration, `PitchPipeTouchButton` in the included code, utilized `android.widget.Buttons` to play the sound "onDown" and stop playing the sound "onUp" (see code snippet below).

```
@Override
public boolean onTouch(View v, MotionEvent event) {
    TextView tv = (TextView)findViewById(R.id.text);
    if(v instanceof Button){
        Button button = (Button)v;
        Integer music = musicButtons.get(button);
        if(event.getAction()==MotionEvent.ACTION_DOWN){
            tv.setText(button.getText());
            currPitchStreamId = sp.play(music, 0, 1, 0, -1, 1);
        }
        else if (event.getAction()==MotionEvent.ACTION_UP) {
            sp.stop(currPitchStreamId);
        }
    }
   return true;
}
```

Figure 1: Iteration 1 - PitchPipeTouchButton.onTouch()

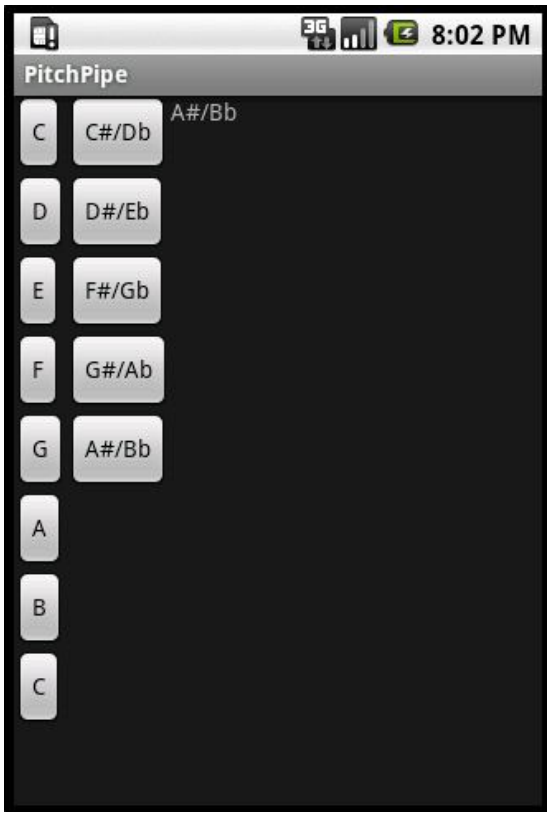

Figure 2: Iteration 1 - PitchPipeTouchButton screenshot

At first, I only implemented one pitch using an instance of `android.media.MediaPlayer` for the pitch. After succeeding at that, I created `MediaPlayer`'s for each pitch. Clearly, there exists an inherent limit on the number of `MediaPlayer` instances one can have, which, thinking about it after the fact made complete sense. This led me to discover the `android.media.SoundPool` class, which is described as being able to "manage the number of audio streams being rendered at once." While I wasn't playing more than one audio stream at a time, I needed something which could hold and manage the different pitches.

Enabling Pitch "on Blow"

The next iteration, `PitchPipe` in the included code, aimed to adapt the first iteration to, instead of playing the pitch "on hold" of a button, play the pitch utilizing the phone microphone to trigger the playing of the pitch. The first adaptation was to change the Button widgets from the first iteration to `android.widget.ToggleButton` widgets. The `ToggleButton`s I implemented stay "held" after you press them until they are pressed again. The pressing of the button would trigger the microphone to start recording, using the `android.media.MediaRecorder` class (see code in Figure 3 below).

```
@Override
public void onClick(View v) {
    TextView tv = (TextView)findViewById(R.id.text);
    //uncheck the current button, note this will allow repeat
    //button pushes
    if(currCheckedButtonId != 0)
        ((ToggleButton)findViewById(currCheckedButtonId)).setChecked(false);
    if(v instanceof ToggleButton){
        ToggleButton button = (ToggleButton)v;
        Integer music = musicButtons.get(button);
        if(!button.isChecked()){
            //in case we do have the same button, we must recheck it
            button.setChecked(true);
        }
        tv.setText(button.getTextOn());
        currCheckedButtonId = button.getId();

        //then do recording playback thinger
        startRecorder(music);
    }
}

private void startRecorder(int music){
    btpt.initialize(music);
    btpt.run();
}
```

Figure 3: Iteration 2 - PitchPipe.onClick()

The `btpt` object in the above figure is an instance of the `BlowToPitchTranslator` class I created. The initialize method sets up the

`MediaRecorder` and sets the correct pitch to be played by the `SoundPool`. The `run` method is shown below in Figure 4. The `run` method periodically polls the sound coming into the microphone by using the `MediaRecorder.getMaxAmplitude()` method. NOTE: In order to utilize the microphone in an application you must define the <uses-permission android:name=*"android.permission.RECORD_AUDIO"*/> in the manifest. Thresholds for determining if the user is blowing into the microphone were set by trial and error. The `listenIn` variable which determines how often the microphone is polled was also set by trial and error to be 150ms. Anything much below 150ms seemed to be too fast for the `run` method to work appropriately.

```
//@Override
public void run() {
    int maxAmp = mr.getMaxAmplitude();
    //if we have a large positive change, then the user is blowing
    //so play the pitch
    if(maxAmp > (prevAmp + 8000)){
        currPitchStreamId = sp.play(soundId, 0, 1, 1, -1, 1);
        prevAmp = maxAmp;
        try {
            Thread.sleep(listenIn);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        run();
    }
    //else if we have a large negative change, the user stopped blowing
    //kill the recorder and soundpool
    else if (maxAmp < (prevAmp - 8000)){
        sp.stop(currPitchStreamId);
        mr.stop();
    }
    else{
        prevAmp = maxAmp;
        try {
            Thread.sleep(listenIn);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        run();
    }
}
```

Figure 4: BlowToPitchTranslator.run()

At first, I had the `BlowToPitchTranslator` running in a different thread, using an instance of `android.os.Handler` to control the communication between the threads. Unfortunately, and most likely due to “operator error”, the button which was pressed representing the current pitch to be played did not show as being pressed on the UI until after the pitch was blown; it appeared as though the

previous pitch which had been pressed was the one playing. So, I changed the `BlowToPitchTranslator` to run in the main thread. This makes sense in the context of my application, since the user wouldn't want to be doing anything else with the pitch pipe while they were blowing the pitch. There is still an issue where in very few cases (happened to me twice out of many, many trials), some kind of infinite loop with the `SoundPool` happens so that the pitch just plays over and over even after killing the activity and process. This is definitely a barrier to publishing the app to the market place today.

Spinning a Pitch Pipe

The final iteration as demonstrated to the class, `PitchPipeSpinner` in the included code, adapted the second iteration so that instead of being a bunch of buttons, the user could actually "spin" the pitch pipe to the desired pitch before blowing. In prior iterations, the `SoundPool` ids for each pitch were associated with the `Button` widgets representing them. In this iteration, degrees of rotation of the pitch pipe graphic are associated with the `SoundPool` ids for each pitch. The degrees of rotation were determined by trial and error. My initial plan was to immediately start polling the microphone once the pitch pipe was settled on a pitch, but this event happened far too frequently, and the user might not necessarily want to blow the pitch right away. Thus, I implemented a button to allow the user to set the pitch, and then click the Blow Pitch button when they are ready to blow (see Figure 5 below).

```
@Override
public void onClick(View v) {
    if(v instanceof Button){
        //start listening for blowing
        startRecorder(pitchToSoundId.get(degrees));
    }
}
```

Figure 5: Iteration 3 - PitchPipeSpinner.onClick()

The hardest piece of this puzzle was to get the math right for enabling a user to spin the pitch pipe both clockwise and counterclockwise, and to swipe the touch screen from wherever they want. To be able to handle this functionality, I split the pitch pipe into quadrants, and then determined if the user had swiped within one quadrant, or across quadrants, and set the direction of spin accordingly. After calculating (using fairly simple geometry) the degree of rotation based upon the users swipe on the screen, the actual degrees of rotation passed to the draw method are "snapped" to the closest pitch's defined degrees of rotation. The `onTouchEvent` method handles the calls to all the math stuff (see Figure 6 below). Many of these math methods could likely be used again for an application which had to rotate a figure based upon user swipe.

```
@Override
public boolean onTouchEvent(MotionEvent e){
    if(e.getAction()==MotionEvent.ACTION_DOWN){
        if(centerX == 0){
            setImageCenterAndBounds();
        }
        setDownCoordinates(e.getX(), e.getY());
    }
    else if(e.getAction()==MotionEvent.ACTION_UP){
        float deltaDegrees = getDegreesFromUpDown(e.getX(), e.getY());
        if(deltaDegrees < 0){
            deltaDegrees = 360 + deltaDegrees;
        }
        degrees += deltaDegrees;
        if(degrees > 360)
            degrees = degrees - 360;
        snapDegreesToPitch();
        draw(degrees);
    }
    return true;
}
```

Figure 6: PitchPipeSpinner.onTouchEvent()

Next Steps

I really would like to publish this application to the Android market place. In order to feel comfortable doing that, I will need to figure out the cause of the “infinite loop” playing of the pitch situation. I also would want to enhance the pitch sounds to be a little louder, and get a better, maybe slightly larger, image for the pitch pipe. Some of my optional features I had in my proposal I feel would still be nice to haves, especially implementing the “men’s” range of pitches (low F to high F) and letting the user decide which they’d like to use. Overall, this project has been a great experience, and I’m glad I challenged myself to learn a new platform, which while being able to apply lessons I’ve learned throughout the semester, presented new challenges in and of itself.