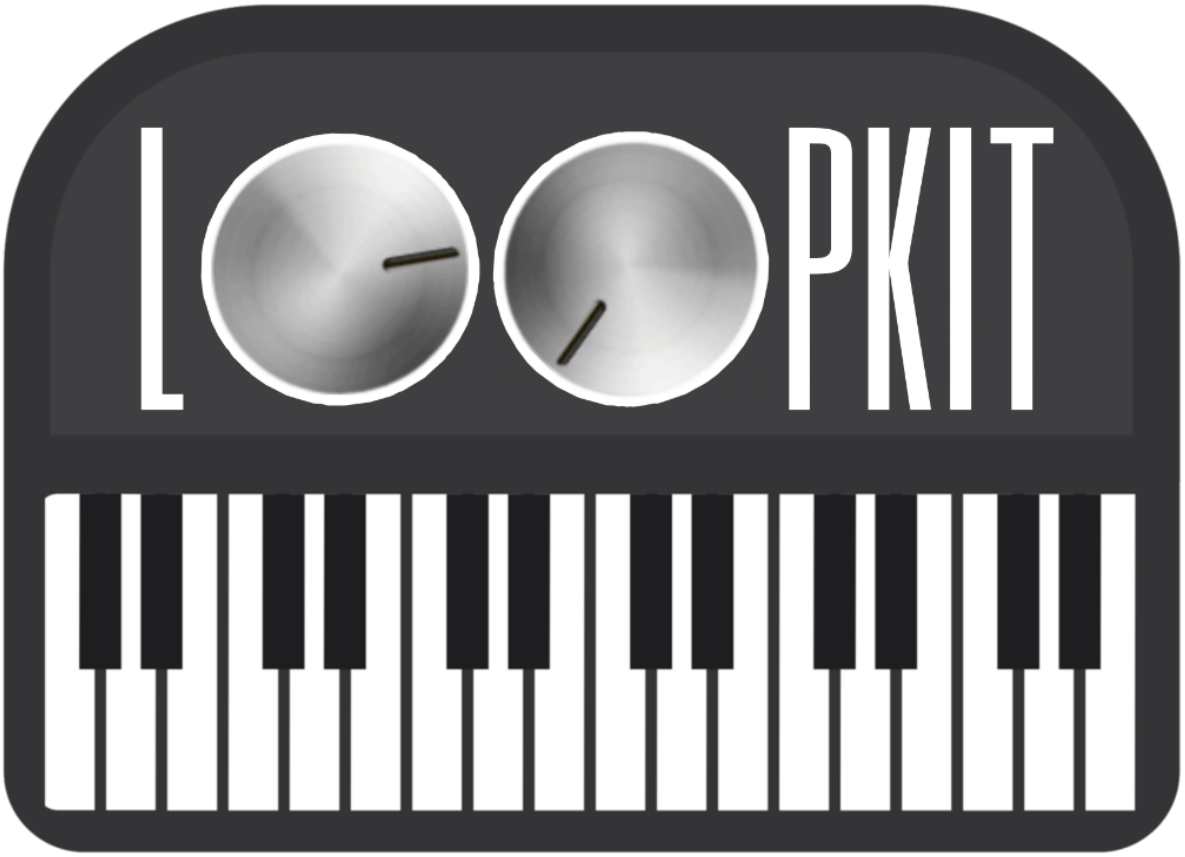


LoopKit

Technical Manual

Developed by:

Patrick Gildea (17374013) & Connell Kelly (17480902)

Supervised by:

Dr. Dónal Fitzpatrick

Document Completed: 07/05/21

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Abstract

Loopkit is a digital audio workstation (DAW) application for Android devices. The app provides users with the means to record audio (e.g. vocals, guitar chords, beatboxing) with their device and loop it to create original music to play along to. Users can also use an in-built drum machine that allows them to play sampled audio in whatever sequences that they choose. These features in combination with one another allow musicians to record and work with music on-the-go in an accessible way.

1.2. Motivation

As avid musicians ourselves, we found great disappointment in the lack of an adequate application for looping audio on our Android devices. Most applications which did exist to fill this role were either long deprecated or non-functional on our devices and none of those that did offered a drum machine or sequencer to accompany user recorded music. We found that when we would get musical ideas that came to us when we were without the proper environment to record them (i.e. laptop with DAW) that we would have to try and record the idea in an audio recorder. But inspiration is fickle and when you have an idea it is much better to have the option to build on it through looping and adding layers.
LoopKit's goal as a project is to allow musicians like ourselves to capture ideas when they strike. While some of our motivation may be of personal interest, we anticipated that others may have felt the same way and confirmed this sentiment with an anonymous survey which we ran during the first semester. Numerous members of DCU's Alternative & Indie Music Society came forward and answered our survey questions, revealing that there is a well-defined desire for a portable digital audio workstation for Android devices. Motivated by our own interests, the interests of others and a genuine hole in the market, our rationale for creating LoopKit was sound.

1.3. Research

Research for the project began with various investigations into the viability of developing a simple audio recorder and subsequent looping functionality. We discovered a variety of useful sources on the development of applications similar to ours which helped us understand some of the challenges we may encounter. We also tested several digital audio workstation apps already available in the Play Store like 'Loopify', but faced considerable difficulty in running them and noted potential areas where we could improve on ourselves.

In order to better establish which features we should focus on implementing, we put together an anonymous online survey asking a variety of questions in regards musicians and what they'd wish to see in an application like this. The survey was created using Google Forms. We were also fortunate enough to be aided by the DCU Alternative & Indie Music Society which would distribute the survey to their varied membership of student musicians.

Survey questions included the following:

1. Do you consider yourself a musician?
2. If so, what kind of instrument/s do you play?
3. Do you write music?
4. Do you record any music?
5. If so, what do you use to record/produce music?
6. Have you used an audio looper before?
7. Would you be interested in using a mobile application to record and loop music on the go or in a pinch?
8. If so, what kind of features would interest you in an audio looping app?
9. Based on that, what would you primarily use an audio looping app for?
10. What factor is most *important* to you in an audio looping mobile app?
11. Is there anything else you'd wish to see in a mobile app like this or any points you'd like to suggest?

Survey results included the following:

- Nearly all participants considered themselves active musicians.
- The most commonly played instruments amongst them includes guitar and piano. Vocals were also a notable musical element participants selected. This provided us an indication of which instruments we should specialise and test the app for.
- 90% of participants actively put time aside to record music. All did so as a hobby, which would play well into our intention to make LoopKit as accessible and easy-to-use as possible.
- Most participants used Audacity or the pre-loaded audio recorder on their phone whenever they recorded music. Audacity being an audio editing application for PCs indicated that there is a genuine lack of digital audio workstations to expand upon the simple utility of simple pre-downloaded audio recorders.
- All participants indicated that they were interested in a purpose built application for portable and easy-to-use audio looping.
- The most sought after features included the ability to easily record sound and to do so with the help of a drum sequencer.
- 40% of participants indicated that they would intend to use the app for musical experimentation and 30% indicated that they would use the app for recording rough drafts and demos to publish or work with later.
- 80% of participants selected “ease-of-use” as the most important feature they’d hope to find in ‘LoopKit’. This information made it clear that we would need to prioritise this aspect.

Along with a certain degree of validation for our initial idea, the survey gave us an improved idea of what features musicians would like to see along with what kind of instruments we should prioritise testing with. We knew to prioritise the development of looping and drum functionality of the app, along with ensuring an accessible design for ease-of-use. With this information, development for ‘Loopkit’ could proceed in earnest.

1.4. Glossary

Beats Per Minute (BPM): The speed, pace or tempo of a given piece of music.

DAW: Digital Audio Workstation; software application where music is recorded, edited and produced. Some popular ones include ableton, garageband and FL studio.
Integrated Development Environment (IDE): Software application which provides useful facilities to computer programmers for software development. For example, Android Studio.

Layer: An individual audio track which can be played simultaneously with other audio tracks.

Loop: A seamless audio track that plays continuously over and over again and the totality of all the layers playing in conjunction with the sequencer on 'LoopKit'.

PCM (Pulse-Code Modulation): An uncompressed audio format that uses a "sampling rate" that describes how often the original audio was sampled, and a "bit depth" which describes how many bits are used to define each sample.

Sampler: A digital instrument that uses sound recordings of real excerpts from recorded songs or found sounds.

Sequencer: Play back music, by handling note, sample and performance information.

2. SYSTEM ARCHITECTURE

2.1. Design

After performing thorough research throughout the first semester and referring to the information collected from our anonymous survey on ‘LoopKit’s potential features, we determined a priority list for the most important features we would like to implement.

Priority List:

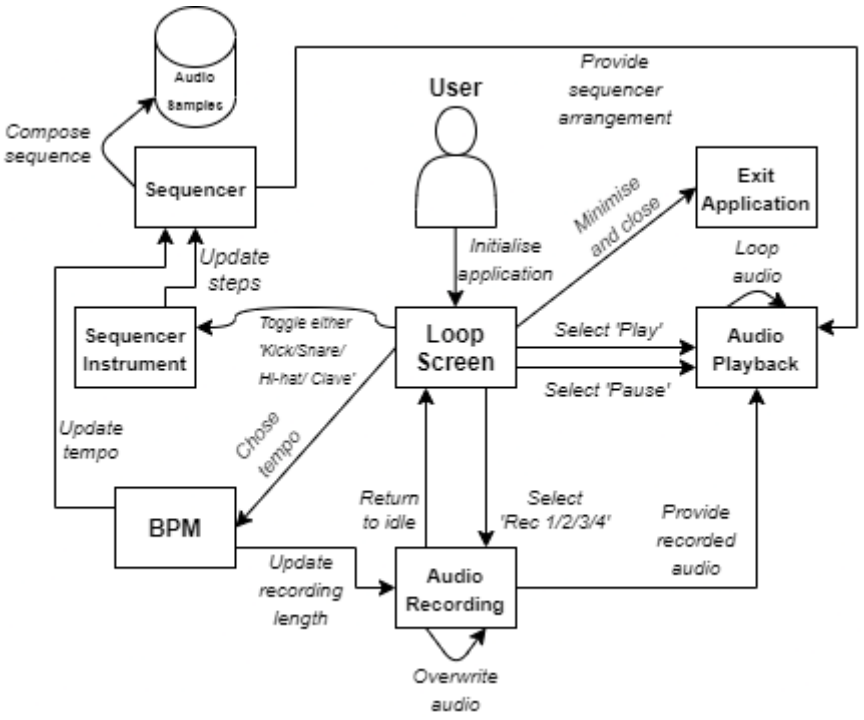
1. Audio Recording
2. Audio Looping
3. Drum Sequencer
4. BPM Control
5. Multiple Sampled Instruments
6. Synthesiser

We determined that the top three features are core to ensuring completion of our original goal for the project and to keep in line with the most sought after features from anonymous feedback. We deemed audio recording to be the most important feature as it would be core to allowing for creative expression with the app and would develop further into audio looping. Together these would be the most technically complex parts of the project’s development for a variety of reasons which are described later in the document.

The drum sequencer was our most popular feature suggestion and we made sure to select this as our next most important priority in the development of ‘LoopKit’. A sequencer would create a lot more icons on the screen, so it would require intuitive design, careful colour contrast and clearly indicated buttons and modules to ensure adequate accessibility. Various other features like BPM control and multiple sampled instruments would improve the overall quality and versatility of the app’s music production capabilities.

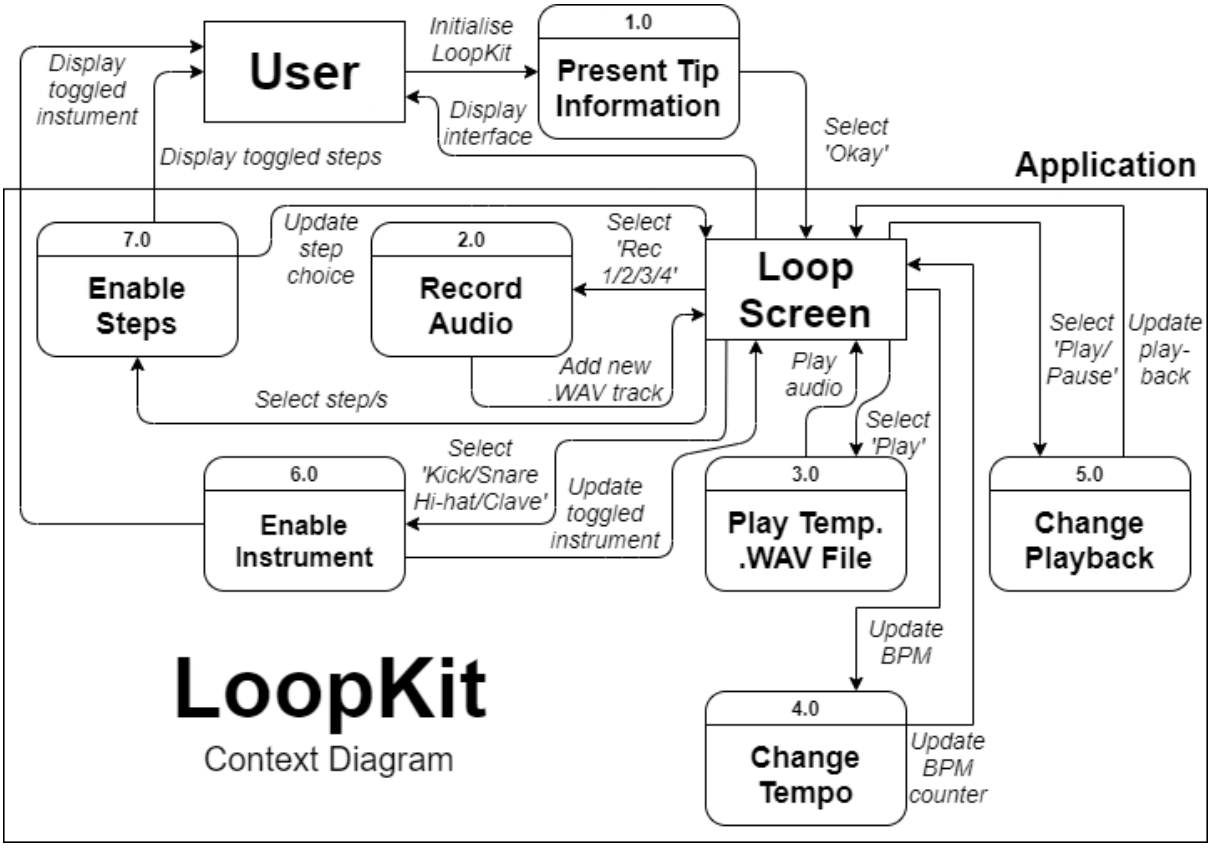
The following design documentation included updated diagrams describing the structure of ‘LoopKit’. Other diagrams were prepared as part of our Functional Specification and we used them to inform the overall structure of the app during development. We would come to understand the several aspects could not be fully integrated successfully and there were certain concepts we didn’t fully understand during pre-production, but the roadmap it provided was important and much of it could still be implemented.

2.2. System Architecture Diagram



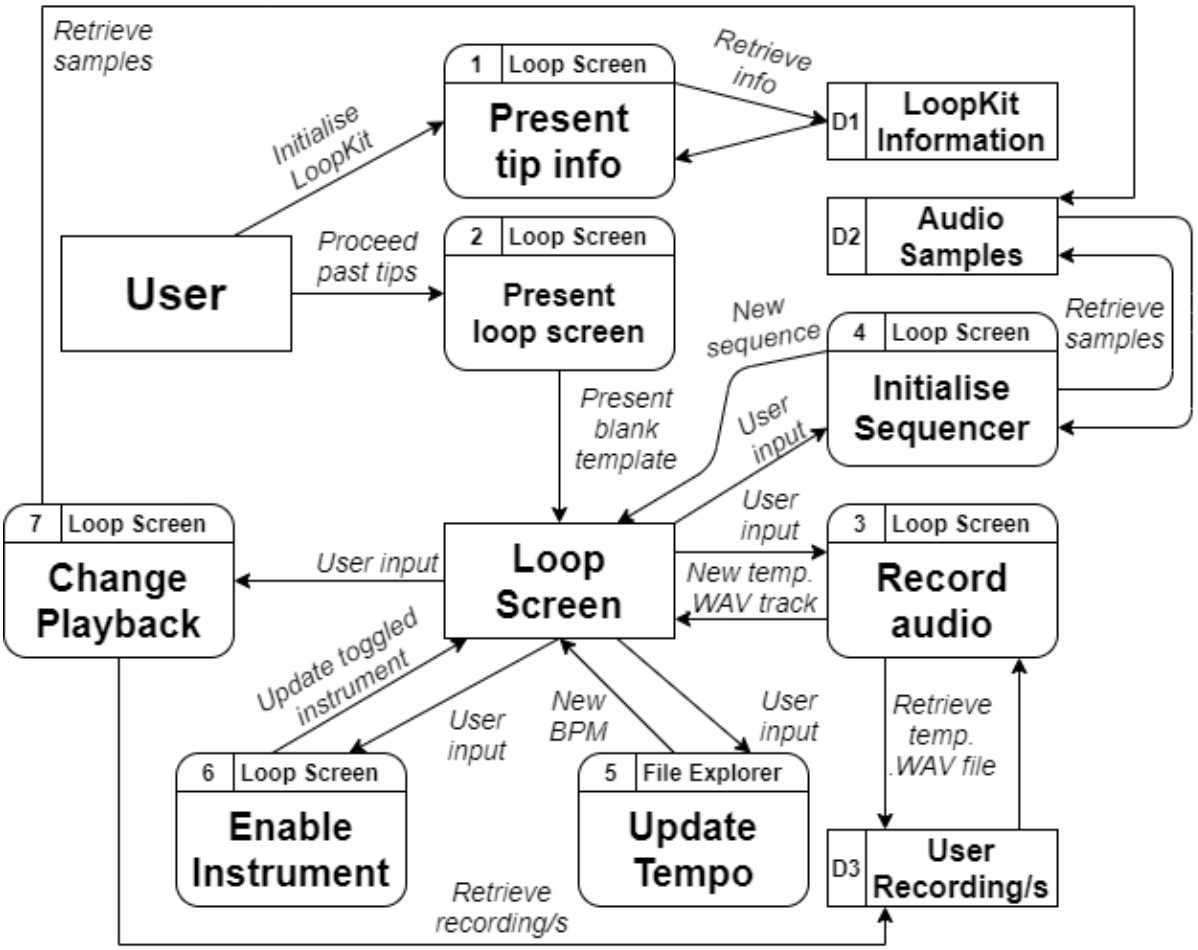
Summary:
The System Architecture Diagram above is a slightly updated version of the diagram we had created for our Function Specification. This was done to provide a blueprint that more closely reflects the final state of the application. The Loop Screen acts as the hub for all activity carried out on ‘LoopKit’. The app was optimised for landscape mode on mobile devices in an effort to replicate the rectangular design of many digital audio workstations which were designed for 16:9 desktop resolution ratios. From the loop screen, all features available in ‘LoopKit’ can be accessed and toggled. These features are composed of a number of interconnected classes and modules.

2.3. Context Diagram



Summary:
The Context Diagram expands upon many of the functions available to a potential target audience on ‘LoopKit’. It identifies the nature of the audio formats involved in recording audio and features like that depend on the loop screen to function.

2.4. Data Flow Diagram



Summary:
The Data Flow Diagram above describes the flow of data through a potential ‘LoopKit’ use case. Data accessed and implemented by functions within the project are available, created and stored for later use for song production purposes. Some information pre-loaded onto the app such as the ‘LoopKit’ tip information and audio samples for the sequencer, but spaces are allocated for data which will be obtained from the user such as any audio recorded.

The data flow diagram shows how data flows through the average Juke use case. This diagram ended up having more databases than necessary. Instead of having an individual song queue database, the song queue is simply part of each venue document on firestore. The accessibility database was unnecessary as Juke was built with accessibility in mind, so there was no need to turn on an option. To achieve this, there were close considerations of the recommendations in Inclusive Components by Heydon Pickering[1] while designing the app to ensure the developers did not fall into common accessibility traps or pitfalls.

3. IMPLEMENTATION

3.1. Audio Recording

The audio recording refers to the user recorded audio to accompany the drum beats that are loaded from the device. Audio is first recorded as raw pcm using the audio record module, then this pcm is converted to wav.

record1func()

The first step to recording audio is taken by the record function associated with each layer, pictured is a fragment of record1func, which starts the process of recording to a layer when the record button is pressed while the loop is running.

The time taken for the drum loop to arrive back at the start is calculated using the drum machine's bpm, and the recorder waits this long before starting the recording so that it is in sync with the drum beat.

Once the recording has started (startRecording1()) record time determines how long it should record for, that being the length of the of the loop.

Once the recording has stopped we tell the app that the first layer has been recorded (layer1setup = true)

Fragment from record1func()

```
if (playing) {

   try {

       //needs to start at the start of the drum machine line
       //wait till we're at the start of the next drum loop

       Log.d(TAG, "record pressed at: " + drum_position);//drum position on time
of record press

        //get the bpm value to calculate wait time
        //record for the exact length of the drum loop
        int waittime = ((60000/(beats_per_minute*4)) * 16) - drum_position - 400;
        TimeUnit.MILLISECONDS.sleep(waittime); //***may need to change added value

        Log.d(TAG, "recording started at: " + drum_position);
```

```
        startRecording1();
        //Log.d("Toledo", "recording started at..." + drum_position);
        int recordtime = ( ( (60000/(beats_per_minute*4) ) * 16) + 213);
////////////////////////////// 120 bpm 113 was good what happened?
        TimeUnit.MILLISECONDS.sleep(recordtime); //***may need to change added
value
        //Log.d(TAG, "Recording stopped: " + recordtime);
        stopRecording1();
        layer1setup = true;


    } catch (IllegalStateException ise) {
        // Exception
    }

    //tapping when paused will mute/unmute the layer


}
```

startRecording1()

The startRecording1() function handles the first part of the setup for using audiorecorder, by creating a new recorder and creating within which to call the next function, writeAudioDataToFile1()

Fragment from startRecording1()

```
recorder = new AudioRecord(MediaRecorder.AudioSource.MIC,
RECORDER_SAMPLERATE, RECORDER_CHANNELS,
RECORDER_AUDIO_ENCODING, BufferElements2Rec * BytesPerElement);

recorder.startRecording();
isRecording = true;
recordingThread = new Thread(new Runnable() {
    public void run() {
        writeAudioDataToFile1();
        //Toast.makeText(getApplicationContext(), "writeaudiodata",
Toast.LENGTH_LONG).show();
    }
```

```
}, "AudioRecorder Thread");
recordingThread.start();
```

writeAudioDataToFile1()

This function uses shorts and a file output stream to write to the required file path, making use of shorttobyte()

Fragment from writeAudioDataToFile1()

```
String filePath = Environment.getExternalStorageDirectory().getAbsolutePath() +
"/recording1.pcm"; //where we're writing
short sData[] = new short[BufferElements2Rec];

FileOutputStream os = null;
try {
   os = new FileOutputStream(filePath);
} catch (FileNotFoundException e) {
   e.printStackTrace();
}

while (isRecording) {
   // gets the voice output from microphone to byte format

   recorder.read(sData, 0, BufferElements2Rec);
   //System.out.println("Short writing to file" + sData.toString());
   // Toast.makeText(getApplicationContext(), "writing to file",
Toast.LENGTH_LONG).show();
   try {
       // // writes the data to file from buffer
       // // stores the voice buffer
       byte bData[] = shorttobyte(sData);
       os.write(bData, 0, BufferElements2Rec * BytesPerElement);
   } catch (IOException e) {
       e.printStackTrace();
   }
```

shorttobyte()

Calculation is done to convert recorded shorts to the byte format necessary for pcm files.

Fragment from shorttobyte()

```
int shortArrsize = sData.length;
byte[] bytes = new byte[shortArrsize * 2];
for (int i = 0; i < shortArrsize; i++) {
    bytes[i * 2] = (byte) (sData[i] & 0x00FF);
    bytes[(i * 2) + 1] = (byte) (sData[i] >> 8);
    sData[i] = 0;
}
return bytes;
```

stopRecording1()

The recording is stopped and the generated pcm file is converted to wav using…

Fragment from stopRecording1()

```
if (null != recorder) { //huh
    isRecording = false;
    recorder.stop();
    recorder.release();
    recorder = null;
    recordingThread = null;
    File f1 = new File(Environment.getExternalStorageDirectory().getAbsolutePath()
+ "/recording1.pcm"); //loc of pcm file
    File f2 = new File(Environment.getExternalStorageDirectory().getAbsolutePath()
+ "/recording1.wav"); // The location where you want your WAV file
    try {
        rawToWav(f1, f2);
    } catch (IOException e) {
        e.printStackTrace();
    }
    //Toast.makeText(getApplicationContext(), "stopped recording",
Toast.LENGTH_LONG).show();
}
```

rawToWav()

The audio is converted to wav by the adding of a wav header to the raw pcm data

Fragment from rawToWav()

```
byte[] rawData = new byte[(int) rawFile.length()];
DataInputStream input = null;
try {
   input = new DataInputStream(new FileInputStream(rawFile));
   input.read(rawData);
} finally {
   if (input != null) {
       input.close();
   }
}

DataOutputStream output = null;
try {
   output = new DataOutputStream(new FileOutputStream(waveFile));
   // WAV header
   writeString(output, "RIFF"); // chunk id
   writeInt(output, 36 + rawData.length); // chunk size
   writeString(output, "WAVE"); // format
   writeString(output, "fmt "); // subchunk 1 id
   writeInt(output, 16); // subchunk 1 size
   writeShort(output, (short) 1); // audio format (1 = PCM)
   writeShort(output, (short) 1); // number of channels
   writeInt(output, 44100); // sample rate
   writeInt(output, RECORDER_SAMPLERATE * 2); // byte rate
   writeShort(output, (short) 2); // block align
   writeShort(output, (short) 16); // bits per sample
   writeString(output, "data"); // subchunk 2 id
   writeInt(output, rawData.length); // subchunk 2 size

   // Audio data
   short[] shorts = new short[rawData.length / 2];

ByteBuffer.wrap(rawData).order(ByteOrder.LITTLE_ENDIAN).asShortBuffer().get(shorts
);
   ByteBuffer bytes = ByteBuffer.allocate(shorts.length * 2);
   for (short s : shorts) {
```

```
        bytes.putShort(s);
    }

    output.write(fullyReadFileToBytes(rawFile));
} finally {
    if (output != null) {
        output.close();
    }
}
```

fullyReadFileToBytes()

This file is then written, the bytes are returned.

Now that the audio for this layer has been stored as wav it is ready to be played in a loop.

Fragment from fullyReadFileToBytes()

```
int size = (int) f.length();
byte bytes[] = new byte[size];
byte tmpBuff[] = new byte[size];
FileInputStream fis= new FileInputStream(f);
try {

    int read = fis.read(bytes, 0, size);
    if (read < size) {
        int remain = size - read;
        while (remain > 0) {
            read = fis.read(tmpBuff, 0, remain);
            System.arraycopy(tmpBuff, 0, bytes, size - remain, read);
            remain -= read;
        }
    }
}  catch (IOException e){
    throw e;
} finally {
    fis.close();
}

return bytes;
```

3.2. Audio Looping

The audio loops by calling the audioloop function

In the first part of the function the layers that have been recorded (as checked with if (layer1setup) etc) are prepared and played with mediaplayer, with there being 2 players for each layer that alternate to alleviate the delay present when mediaplayer starts. The setup is done each time so that if the layer has been written over that the new audio is properly prepared.

First part of audioLoop()

```
if (layer1setup) {

   //set up the 2 mediaplayers needed for one layer of the loop

mediaPlayer1a.setDataSource(Environment.getExternalStorageDirectory().getAbsoluteP
ath() + "/recording1.wav");
   mediaPlayer1a.prepare();

mediaPlayer1b.setDataSource(Environment.getExternalStorageDirectory().getAbsoluteP
ath() + "/recording1.wav");
   mediaPlayer1b.prepare();
   //their own looping does not suffice, we're doing it our own way as you will
see below
   mediaPlayer1a.setLooping(false);
   mediaPlayer1b.setLooping(false);
   mediaPlayer1a.start();
   layer1playing = true;
}

if (layer2setup) {


mediaPlayer2a.setDataSource(Environment.getExternalStorageDirectory().getAbsoluteP
ath() + "/recording2.wav");
   mediaPlayer2a.prepare();

mediaPlayer2b.setDataSource(Environment.getExternalStorageDirectory().getAbsoluteP
ath() + "/recording2.wav");
```

```
    mediaPlayer2b.prepare();
    //their own looping does not suffice, we're doing it our own way as you will
see below
    mediaPlayer2a.setLooping(false);
    mediaPlayer2b.setLooping(false);

    //start playing layers using mediaplayer
    mediaPlayer2a.start();
    layer2playing = true;

}

if (layer3setup) {


mediaPlayer3a.setDataSource(Environment.getExternalStorageDirectory().getAbsoluteP
ath() + "/recording3.wav");
    mediaPlayer3a.prepare();

mediaPlayer3b.setDataSource(Environment.getExternalStorageDirectory().getAbsoluteP
ath() + "/recording3.wav");
    mediaPlayer3b.prepare();
    //their own looping does not suffice, we're doing it our own way as you will
see below
    mediaPlayer3a.setLooping(false);
    mediaPlayer3b.setLooping(false);

    //start playing layers using mediaplayer
    mediaPlayer3a.start();
    layer3playing = true;

}

if (layer4setup) {


mediaPlayer4a.setDataSource(Environment.getExternalStorageDirectory().getAbsoluteP
ath() + "/recording4.wav");
    mediaPlayer4a.prepare();

mediaPlayer4b.setDataSource(Environment.getExternalStorageDirectory().getAbsoluteP
ath() + "/recording4.wav");
    mediaPlayer4b.prepare();
```

```
   //their own looping does not suffice, we're doing it our own way as you will
see below
   mediaPlayer4a.setLooping(false);
   mediaPlayer4b.setLooping(false);

   //start playing layers using mediaplayer
   mediaPlayer4a.start();
   layer4playing = true;

}
```

In the second half of the audioloop function timers are set up for the necessary layers so that when one of the two recorders for one of the layers is about to finish the other one is started, this happens in a loop until the user pauses, then the drum sequence will be paused and reset back to the start of the bar as will the user audio, and the timers will be reset.

Second part of audioLoop()

```
Timer HACK_loopTimera = new Timer(); //timer1
TimerTask HACK_loopTaska = new TimerTask()
{
   @Override public void run() {

       if (layer1setup) {
           mediaPlayer1b.start();
       }
       if (layer2setup){
           mediaPlayer2b.start();
       }
       if (layer3setup){
           mediaPlayer3b.start();
       }
       if (layer4setup){
           mediaPlayer4b.start();
       }
   }
};

//starts about 350 milliseconds early to account for delay
//however because the time it takes varies some fallout is inevitable
```

```
//get the duration of mediaplayer because that layer is recorded first
final long[] waitingTimea = {mediaPlayer1a.getDuration() - 350};
HACK_loopTimera.schedule(HACK_loopTaska, waitingTimea[0], waitingTimea[0]);



//alternate back to mediaplayer 1 before meiaplayer 2 has technically
finished...and so on
Timer HACK_loopTimerb = new Timer(); //timer2

TimerTask HACK_loopTaskb = new TimerTask()
{
   @Override public void run() {
       if (layer1setup) {
           mediaPlayer1a.start();
       }
       if (layer2setup){
           mediaPlayer2a.start();
       }
       if (layer3setup){
           mediaPlayer3a.start();
       }
       if (layer4setup){
           mediaPlayer4a.start();
       }
   }
};

final long[] waitingTimeb = {mediaPlayer1b.getDuration() - 350};
HACK_loopTimerb.schedule(HACK_loopTaskb, waitingTimeb[0], waitingTimeb[0]);
```

stopaudioseq()

Finally stopaudioseq() is called on pause to stop the user audio part of the loop. On play the loop can be resumed.

Fragment of stopaudioseq()

```
//only stopping that which is actually playing, not just setup
Log.d(TAG, "stopSeq2");
Log.d(TAG, "layer 1" + layer1playing);

try {

   if (layer1setup & layer1playing) {
        mediaPlayer1a.stop();
        mediaPlayer1a.reset(); //allows the loop to be played again
        mediaPlayer1b.stop();
        mediaPlayer1b.reset(); //allows the loop to be played again

        HACK_loopTimera.cancel();
        waitingTimea = 0;
        HACK_loopTimerb.cancel();
        waitingTimeb = 0;

   }
   if (layer2setup & layer2playing) {
        mediaPlayer2a.stop();
        mediaPlayer2a.reset(); //allows the loop to be played again
        mediaPlayer2b.stop();
        mediaPlayer2b.reset(); //allows the loop to be played again
   }

   if (layer3setup & layer3playing) {
        mediaPlayer3a.stop();
        mediaPlayer3a.reset(); //allows the loop to be played again
        mediaPlayer3b.stop();
        mediaPlayer3b.reset(); //allows the loop to be played again
   }

   if (layer4setup & layer4playing) {
        mediaPlayer4a.stop();
        mediaPlayer4a.reset(); //allows the loop to be played again
        mediaPlayer4b.stop();
        mediaPlayer4b.reset(); //allows the loop to be played again
   }
```

```
    Log.d("Sequence", "Stopped");
    //mySeq.stop();


}

catch (IllegalStateException e) {
    e.printStackTrace();
}

drum_position = 0;
playing = false;
layer1playing = false;
layer2playing = false;
layer3playing = false;
layer4playing = false;
```

3.3. Sampler

The sampler class functions in tandem with the Sequencer class to allow for streaming of the sound samples. It incorporates a number of useful media classes such AudioAttributes[3], AudioManager[4] and SoundPool[5]. AudioAttributes assists in encapsulating a collection of attributes which describe various pieces of detailed information about an audio stream. AudioManager provides necessary access to volume control. SoundPool allows for the management and play of short audio resources and, as a result, is perfectly suited to playing the sound samples we've made available for the sequencer. When constructing the sampler, the app takes into account whether or not the Android version on the device is Lollipop (5.0) or above. The SoundPool class is deprecated from Lollipop onwards, so we accommodate this and only incorporate the class for devices below.

Fragment from Sampler.java

```
try{

    this.mycontext = appcontext;

    // Detect if Android version is Lollipop.
    if(Build.VERSION.SDK_INT>=Build.VERSION_CODES.LOLLIPOP){
```

```
      attr = new AudioAttributes.Builder()
              .setContentType(AudioAttributes.CONTENT_TYPE_UNKNOWN).
                      setUsage(AudioAttributes.USAGE_MEDIA).
                      build();
      sp = new
SoundPool.Builder().setMaxStreams(4).setAudioAttributes(attr).build();

  }
  // If Android version is below Lollipop
  else{
      sp = new SoundPool(10, AudioManager.STREAM_MUSIC,0);
  }
```

The audio samples are then loaded into private integers from their location in the *res/raw* folder. Their IDs are recorded so they can be called individually later.

Fragment from Sampler.java

```
// Loading Sounds from res/raw
         kickId = sp.load(mycontext, R.raw.kick , 1);
         snareId = sp.load(mycontext, R.raw.snare, 1);
         percId = sp.load(mycontext, R.raw.perc, 1);
         hhId = sp.load(mycontext, R.raw.hh,1);
```

The returnDrum() and play() functions work together to return the correct sound sample with the associated sequencer step that was selected by the user. Using SoundPool, volume settings, sound priority and playback rate (frequency) are set for audio samples being played at a particular step defined by a dictionary keeping track of this parameter.

Fragment from Sampler.java

```
public void play(int[] curStep){

       for (int i = 1; i<5;i++){

           if (curStep[i] ==1){
               sp.play(returnDrum(i), 1, 1, 1, 0, 1);

           }

       }

   }
```

```
    // returns drum based on row.
    public int returnDrum(int drum){
        switch (drum){
            case 1:
                drum=kickId;
                break;
            case 2:
                drum =hhId;
                break;
            case 3:
                drum = snareId;
                break;
            case 4:
                drum = percId;
                break;
        }
        return drum;
    }
```

3.4. Sequencer

The Sequencer plays with the sequenceloop() function

The first thread start the sequencer after an initial delay of 400ms to account for the time it takes the audio loop to start up

```
playing = true;

//drum sequence thread
Thread myThread1 = new Thread(new Runnable() {
   @Override
   public void run() {


       try {
           TimeUnit.MILLISECONDS.sleep(400); //wait for the vocal loop to start,
right now everytime but ideally when the audio has been added
       } catch (InterruptedException e) {
           e.printStackTrace();
       }
```

```
        mySeq.play();


        handler.post(new Runnable() {
            @Override
            public void run() {

            }
        });


    }
});
myThread1.start();
```

The second thread starts a timer that counts for the length of the sequence loop and then resets, this is used to inform the audio recorder of where the sequencer is in its loop so they can sync together.

```
//drum position tracker
thread//////////////////////////////////////////////////////////
Thread myThread2 = new Thread(new Runnable() {
    @Override
    public void run() {

        //start timer counts to length of the the loop dependant on bpm, then
resets to 0 when done public timer


        Log.d(TAG, "start");
        while (playing) {

            try {
                //Log.d("toledo", String.valueOf(drum_position));
                TimeUnit.MILLISECONDS.sleep(1);
                drum_position += 1;

            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }

            if (drum_position == ((60000 / (beats_per_minute * 4)) * 16)) //if i ==
length of loop i.e. we have counted till we are back at the start length of beats
though? time it takes to play though?
            {
                drum_position = 0;
            }


            handler.post(new Runnable() {
                @Override
                public void run() {

                }
            });


        }
    }

});
myThread2.start();
```

The Seq.java file deals with the setting up of the sequencer, handling of the bpm and the beats. The sequencer works through its 16 beats, using the bpm to determine how long it should sleep between each beat.

```
 public void play(){
        playing = true;



        //while (playing == true) {
        //long a = System.nanoTime();

        //
        // Log.d("time", String.valueOf(a));
    while(playing == true) {
```

```
    for (int i = 0; i < 16; i++) {

        //need to keep track of when i is zero
        mySampler.play(Sequet[i]);
        step = i;
        if (playing == false) {
            break;
        }
        try {
            Thread.sleep(60000 / (bpm * 4));

            if (i == 15) i = -1;



        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
     /** long b = System.nanoTime();

        Log.d("time", String.valueOf(b));

        long result = (b - a);

        Log.d("time", String.valueOf(result));
        */

        }

  // }

    public int returnStep() {
        return step;
    }


    public void stop(){

        playing = false;

    }
    // setting bpm for sequence
```

```
    public void setBpm(int bpmIn){
        bpm = bpmIn;
    }

    public void outputSequence(){

        for (int j = 0; j<16; j++){
            System.out.println("Step"+ j);

            for (int z =0; z<5; z++){
                System.out.println(Sequet[j][z] + " ");
            }
            System.out.println(" ------ ");

        }
    }
```

4. TESTING PROCEDURES

4.1. Unit Testing

Unit testing presented an issue as our applications functions were often void or called other functions or dealt with files that were only created in the context of the application. However there were some situations where we were able to test whether a function had worked correctly, even if it was void by checking if other values that were created in the function were set up correctly

```
public classstartRecordingTest{

      @Test
      public void startRecording1() throws Exception
      {


            boolean expected = true;
            double delta = .1;
        recorder = new AudioRecord(MediaRecorder.AudioSource.MIC,
                RECORDER_SAMPLERATE, RECORDER_CHANNELS,
                RECORDER_AUDIO_ENCODING, BufferElements2Rec *
BytesPerElement);
```

```
        recorder.startRecording();
        isRecording = true;
        recordingThread = new Thread(new Runnable() {
            public void run() {
                writeAudioDataToFile1();
                //Toast.makeText(getApplicationContext(),
"writeaudiodata", Toast.LENGTH_LONG).show();
            }
        }, "AudioRecorder Thread");
        recordingThread.start();


            output = recordingThread.isAlive();



     //contents, calculations


     assertEquals(expected, output, delta);


     }

     @Test
     public void startRecording2() throws Exception
     {


            boolean expected = true;
            double delta = .1;
        recorder = new AudioRecord(MediaRecorder.AudioSource.MIC,
                RECORDER_SAMPLERATE, RECORDER_CHANNELS,
                RECORDER_AUDIO_ENCODING, BufferElements2Rec *
BytesPerElement);

        recorder.startRecording();
        isRecording = true;
        recordingThread = new Thread(new Runnable() {
```

```
            public void run() {
                writeAudioDataToFile2();
                //Toast.makeText(getApplicationContext(),
"writeaudiodata", Toast.LENGTH_LONG).show();
            }
        }, "AudioRecorder Thread");
        recordingThread.start();


            output = recordingThread.isAlive();



      //contents, calculations


      assertEquals(expected, output, delta);


      }

      @Test
      public void startRecording3() throws Exception
      {


            boolean expected = true;
            double delta = .1;
        recorder = new AudioRecord(MediaRecorder.AudioSource.MIC,
                RECORDER_SAMPLERATE, RECORDER_CHANNELS,
                RECORDER_AUDIO_ENCODING, BufferElements2Rec *
BytesPerElement);

        recorder.startRecording();
        isRecording = true;
        recordingThread = new Thread(new Runnable() {
            public void run() {
                writeAudioDataToFile3();
                //Toast.makeText(getApplicationContext(),
"writeaudiodata", Toast.LENGTH_LONG).show();
```

```
            }
        }, "AudioRecorder Thread");
        recordingThread.start();


            output = recordingThread.isAlive();



      //contents, calculations


      assertEquals(expected, output, delta);


      }


      @Test
      public void startRecording4() throws Exception
      {


            boolean expected = true;
            double delta = .1;

        recorder = new AudioRecord(MediaRecorder.AudioSource.MIC,
                RECORDER_SAMPLERATE, RECORDER_CHANNELS,
                RECORDER_AUDIO_ENCODING, BufferElements2Rec *
BytesPerElement);

        recorder.startRecording();
        isRecording = true;
        recordingThread = new Thread(new Runnable() {
            public void run() {
                writeAudioDataToFile4();
                //Toast.makeText(getApplicationContext(),
"writeaudiodata", Toast.LENGTH_LONG).show();
            }
        }, "AudioRecorder Thread");
```

```
        recordingThread.start();


            output = recordingThread.isAlive();



      //contents, calculations


      assertEquals(expected, output, delta);


      }



}
```

4.2. Log Testing

Of great help was the log.d command.
It allowed us to test what values were coming into functions and keep track of where we were in the execution of the code when viewing the log. We did not have the time to document each of these tests as they were happening constantly.

E.g.

```
Log.d(TAG, "stopSeq2");
Log.d(TAG, "layer 1" + layer1playing);
```

4.3. User Testing

After receiving ethical approval from DCU, we were able to begin preparations for a number of vital user tests that would assist us in developing ‘LoopKit’. A plain language statement and informed consent form was provided to potential participants. Due to the restrictive nature of user testing during the COVID-19 pandemic, we were limited to doing our tests online and this factor discouraged many potential participants from getting involved, leaving us with three participants who we were able to test with. These tests were still thorough and

informative as they involved online interviews with participants where they would provide commentary on their use of ‘LoopKit’ and answer pre-determined questions.

Users had fun testing out ‘LoopKit’s’ available features at the time and enabled us to note many important observations from the results we recorded. Our user testing procedures involved interviewing three participants thoroughly and each participant demonstrated varying degrees of audio engineering and musical skills. The latter detail was valuable as it provided us with a microcosm of all potential users: skilled, semi-skilled and inexperienced. The following results were recorded from the testing procedures.

Results:

1. Without prior explanation, certain aspects of the sequencer such as the step functionality and the instrument descriptions were confusing. We would take these observations on-board and improve the naming conventions found for each instrument, along with a popup on application start-up which informs the user of all the functions available and how to use them.
2. The recording audio is somewhat confusing and participants desired a more intuitive way to do so.This was another issue that we attempted to remedy with a comprehensive written guide on start-up.
3. Some of the UI design and colour choices were found to be unintuitive by several participants, so we made sure to improve placement of each production module to better fit the device screen and investigated better combinations of colours[6] that kept visual fidelity and accessibility in mind.
4. ‘LoopKit’ can face audio recording issues depending on the version of Android that it is running on. Thanks to these user tests, we could establish that audio recording functionality works as expected up to and including Android Pie (9.0). While all sequencer functionality remained beyond Android Pie, attempting to record any audio would result in ‘LoopKit’ crashing.
5. Participants noted the lack of a built-in ‘clear’ button for reverting the sequencer and audio recorder back to a blank template. The only way to do this originally was to restart the app, so we attempted to implement a new feature that would allow users to clear all variables at play and restart the sequencer from scratch without having to manually leave and reopen the app.
6. Some participants wanted more instruments to choose from than the four we allocated. We were happy with our instrument selection and the development time we had remaining was short so we chose to leave the audio sampler as it was, but this reassured us in our modular LinearLayout design of the UI which would allow for relatively easy changes to components like the instruments.
7. Some degree of audio processing (e.g. reverb, distortion) was also desired by participants who wanted to produce their audio tracks in more depth.

These are some of the more notable results we obtained from users tests and many of the issues mentioned above could be alleviated. A tip box was added on start-up to inform new users on how to use the app and several UI elements were changed and improved to accommodate user observations. The varying levels of user experience with audio engineering and music meant that, despite the relatively small number of participants, we received a broad range of perspectives on how the app is utilised by different mindsets. A

built-in ‘clear’ button for reverting the app back to its default state was also requested, though it’s implementation into the app buggy and will receive subsequent fixes.

However, some potential fixes and improvements required further investigation and in some cases, could not be implemented before the project deadline. The audio recording bugs related to the Android version present on the phone was an elaborate problem that would require much more time than we had remaining to solve and the complexities about how it unfolded our details in our ‘5.1 Problems’ section. There were several requests for more audio samples, which is something that the app was designed to allow for, but an observation we placed low priority on as there were already four included and more could be added later as detailed in ‘6.2 Future Work’. Deeper audio processing beyond tempo control as requested by a participant is something we would have liked to integrate into our project, but major development hurdles in ensuring working audio recording prevented us reaching a position where we could safely incorporate these features.

User testing was a fruitful and informative addition to the testing process for ‘LoopKit’. We came to understand how users of varying skill levels would approach our app and we were provided with perspectives and observations we may not have come to on our own. Users enjoyed their time using the app and when asked how they would rate the app on a numerical scale, they gave us positive feedback.

4.4. Formative User Testing

Performing any informal forms of testing was difficult without an active lab environment. In an effort to circumvent this challenge, we asked family members to assist us in several aspects of the development process. Data would be collected from their usage of various builds of ‘LoopKit and it provided us with critical insight into aspects we could investigate over online user testing such instrumentation testing and device resolution tests. The obtained the following pieces of valuable information.

Formative User Test Improvements and Observations:

- Various screen sizes and ratios were tested with Patrick’s family members who each owned different kinds of Android devices (Samsung Galaxy A50, Google Pixel 3A, Redmi Note 7). This helped establish which screen size was universally best suited for the development with. We concluded that the Google Pixel 3A provided the best resolution for app development
- Connell’s brother, who had an understanding of music and digital audio workstations, provided insight during the instrument sample selection process. We determined that a kick drum, snare drum and hi-hat was essential and added a clave as the fourth option. This process assisted in helping us narrow down which instruments would be the most useful for users and which samples had the highest-fidelity.
- Connell’s brother assisted the development process by allowing Connell to record him playing various instruments with a test build of ‘LoopKit’. This allowed us to test how various instruments sounded after being recorded by our modules and methodology. The results of this prompted us to abandon the MediaRecorder[1] audio recording module in favour of the AudioRecorder and AudioTrack modules.

Ad hoc testing was useful in solving several issues in the project, especially in regards to sound quality. Despite the process being somewhat inhibited by lockdown conditions, family members and friends still provided us with useful outside perspectives and observations.

5. PROBLEMS AND RESOLUTIONS

5.1. Problems

1. Looping gap
2. Looping was not in sync
3. Looping was not recorded at the right time
4. Mediarecorder low quality
5. Pcm unplayable
6. App Crashing on Recording

1. Looping gap

Although the mediaplayer module is well supported and has a built in looping method (set looping true) this results in audio that has a noticeable delay every time the loop restarts. This was not acceptable and we sought to eliminate the delay.

2. Looping was not in sync

Getting the loop to be in sync proved very difficult as there are up to 5 layers (4 user + drum sequence), the device can slow down and a piece of code to do with timing can take longer to execute than expected.

3. Looping was not recorded at the right time

We could loop audio but the recording would start when the button was pressed, but recorded audio always starts playing at the start of the bar, leading to offset drum and audio layers.

4. Mediarecorder low quality

We started the project by recording audio with Mediarecorder, but soon found that its audio quality was not lossless and as such was unsatisfactory.

5. Pcm unplayable

When we switched to AudioRecorder we found that it recorded pcm files which are unplayable by MediaPlayer.

6. App Crashing on Recording

After extensive user testing and running the app on whatever Android devices we had available in our homes, we established that ‘LoopKit’ will crash when users attempt to record audio from their microphone a device with an Android version past Pie (9.0). Given how dated most usable online guides and Stack Overflow examples were in regards to Android

audio recording and that both project investigators lacked a device past Pie, this was an issue that eluded us until late in development. Later we learned that overhauls to file permission systems on various Android versions after the release Android Q (10.0) would cause unforeseen bugs when ‘LoopKit’ attempted to operate built-in devices. Most learning material on dealing with audio on Android devices that we studied was dated well before the release of Q.

5.2. Solutions

1. Using two media players per layer
2. Fine tuning
3. Sequence timer
4. Switching to AudioRecorder
5. Pcm to wav converter

1. Using two media players per layer

To solve the issue of the gap when playing a file with MediaPlayer we implemented two media players instead that alternate, starting each early to compensate for the delay to start, which requires measuring how long that line of the code took to execute.

2. Fine tuning

Getting the loop to sync as well as possible required fine tuning of all the values relating to how long the audio was recorded for, when it starts playing etc. We did numerous log tests to see how long certain pieces of code were taking to execute, and whether the time at which we were starting recording/playing resulted in an optimum loop.

3. Sequence timer

To ensure that user audio is played as the user recorded it in sync with the drum sequence, a drum timer was created in a separate thread that would keep track of the position of the drum sequence and use this to tell the any recorders trying to record to wait till the drum timer reached 0 and then start recording.

4. Switching to audio recorder

We have already detailed this but we discovered that lossless recording could be done with audio recorder and set to work redoing the code that concerned recording.

5. Pcm to wav converter

We were able to implement a file converter, which works by adding wav heading data to the pure byte array of the pcm file. The created wav file can then be played with MediaPlayer.

5.3. Limitations

There were a number of factors at play that wrecked havoc with the development of ‘LoopKit’ and in turn created a number of limitations that hampered our work. The most notable limitation was that of the COVID-19 pandemic and the consequences that the ensuing series of lockdowns lead to. Given that we selected and prepared many aspects of the project’s development assuming we’d be back on campus for the second semester, we felt it was important to list these limitations as they played a significant negative role.

Without access to a lab computer as a result of living in Sligo and being confined to it as a result of various lockdowns, Connell’s hardware capabilities for app development were severely limited. Android Studio 4.2 is a surprisingly demanding IDE [2] and Connell’s only computer available for project work was an underpowered laptop. This presented considerable challenges in both running Android Studio and test builds of ‘LoopKit’. Android Studio could take up to 95% of available memory, causing hitches, crashes and hurting workflow. Similarly Patrick’s computer was unable to emulate and had to rely on

Because of this the only two phones we had to test were both running earlier versions of android (8,9) which became apparent as an issue during our user testing, but as emulation was not an option for a long time I don’t see how we could have avoided this problem.

Not being able to work together in person caused a number of issues in regards to our code integration, Git pushing and user testing. Not being able to develop ‘LoopKit’ side-by-side on lab machines hampered our ability to efficiently work together and write code that adhered to shared formats and principles. Despite our best efforts, integrating fragments of our code together into one cohesive build was difficult to perform solely online and without any personal interaction. We also faced a lot of trouble initially when pushing builds of the project to our Gitlab repo, given relative inexperience pushing work to it outside of on-campus lab computers. This inexperienced led to it taking extensive research and time to determine how to perform regular Git commits and pushes. The inability to perform user tests in-person with potential users prevented us from capturing more informative data on their usage and how ‘LoopKit’ performs on their devices.

6. CONCLUSION

6.1. Results

Due to the constraints described we decided to focus on the core of the project's priorities. In this manner we were successful. We successfully constructed a simple, but functional digital audio workstation for Android devices. LoopKit comes fully featured with a versatile drum sequencer that can be adapted to different tempos and played simultaneously with user recorded audio. There are several layers which can be recorded over or muted and the drum machine pattern can be changed on the fly. Allowing for audio looping was a considerable challenge that surprisingly lacked in recent online guides, so we chose to design the app based on ideas and concepts that may be dated in some formats. Despite this, 'LoopKit' still functioned as intended, aside from a handful of bugs that can and will be fixed in the future.

Despite pandemic limitations, it was tested on as many devices as we feasible could and established many important observations and made as many improvements as we could based on what we had noted.

6.2. Future Work

Solving any bugs and getting the looper as accurate as possible is a priority, as is Addressing more of the feedback given to us during user testing: such as, improvements to the UI, making things more clear to the user, adding of more sequencer sounds and audio processing are examples of user requests.

In addition working on features that we initially intended to implement such as menu screen for saving and exporting loops and a synthesizer would be of interest to us. Our LinearLayout design provided LoopKit with a very modular design for the layout, meaning with more work we could easily slot in new plugins, instruments and tools.

6.3. Installation

Any user who wishes to use 'LoopKit' can download a build of it in the form of an APK (Android Application Package). An APK is the package file format used by the Android operating system to distribute and allow for the installation of mobile applications, mobile games and middleware.

Prerequisites:

- Android Device (4.4.+)

Steps:

1. Access the 'app-debut.apk' package from the link below on your Android device. https://drive.google.com/drive/folders/19YiEWxQI8rBqP0qZFbv8dgWjPyrnBVWj?usp=sharing
2. Accept permission requests in regards to device microphone access.
3. Navigate to 'LoopKit' on your device again if you wish to launch it again

Note: Should you face bugs as a result of having a newer version of Android, we recommend downloading the Blue Stacks 5 Android emulator on your desktop and running the APK through it. Audio recording was tested to work on that.

6.4. Appendices

1. MediaRecorder[1]
2. Android Studio [2]
3. AudioAttributes [3]
   https://developer.android.com/reference/android/media/AudioAttributes
4. AudioManager[4]
   https://developer.android.com/reference/android/media/AudioManager
5. SoundPool[5]
   https://developer.android.com/reference/android/media/SoundPool
6. Colour Contrast[6]
   https://davidmathlogic.com/colorblind/#%23D81B60-%231E88E5-%23FFC107-%23004D40