Philosophische Fakultät III
Sprach- , Literatur- und Kulturwissenschaften
Institut für Information und Medien, Sprache und Kultur (I:IMSK)
Lehrstuhl für Medieninformatik

Forschungsseminar Interaktionstechniken und -technologien
Modul: MEI-M32.1 (M.sc.)
SS 2021
Leitung: Jürgen Hahn, Andreas Schmid, Raphael Wimmer

DIPPID Melody Game

Marco Beetz
Matr.-Nr.: 1779119
1. Semester M.sc. Medieninformatik
E-Mail: marco.beetz@stud.uni-regensburg.de

Joshua Benker
Matr.-Nr.: 1773061
1. Semester M.sc. Medieninformatik
E-Mail: joshua.benker@stud.uni-regensburg.de

Abgegeben am 23.07.2021

Contents

1 Introduction

With the choice of Python as programming language there are many possible approaches to develop software with the large number of packages and libraries available. When interacting with a computer system the user typically looks at the screen or listens to it. Visual and audible senses are combined when you play a music game with a desktop application. We present *DIPPID Melody Game*, a novel music game that uses the DIPPID application as user input to process sensor data into different sounds. With seven possible sounds and one the user needs to reproduce well-known melodies by turning the phone with the DIPPID application running like a clock. All sounds are created with the *Synthesizer* module. The application requires working with DIPPID via smartphone and gazing at the UI on a laptop simultaneously. Thus, two essential interaction technologies were implemented – pointing and clicking on the UI and physically handling DIPPID.

The development was created by Joshua Benker and Marco Beetz. Both team members discussed several ideas on the procedure of the game, which modules work best for the problem on hand and how it should be implemented. The UI was designed and implemented by Marco Beetz. The logic and sequence, especially the game loop with all included functions and the transformation of sensor data into MIDI tones was implemented by Joshua Benker. The manual development of melodies from musical tones and fitting it to DIPPID data states was discussed by both team members and implemented by Joshua Benker. Outsourcing songs into melodies.py was created by Marco Beetz. The present documentation was written by Marco Beetz.

2 Packages and setup

Please make sure that the packages mentioned below are installed. If not please install each package in Terminal. The user interface is created by using *PyQt5* as a widget toolkit. Early stages of the project involved *Pygame* as primary interface. Several sub-libraries of PyQt5 need to be imported. We suggest to simply import the entire PyQt5 framework. The DIPPID application is used on a common Android smartphone and needs to be running with the correct IP address and the enabled button to send data. Furthermore, an open port to pass the data between phone and computer is required. The primary input data from DIPPID is generated by using its accelerometer.

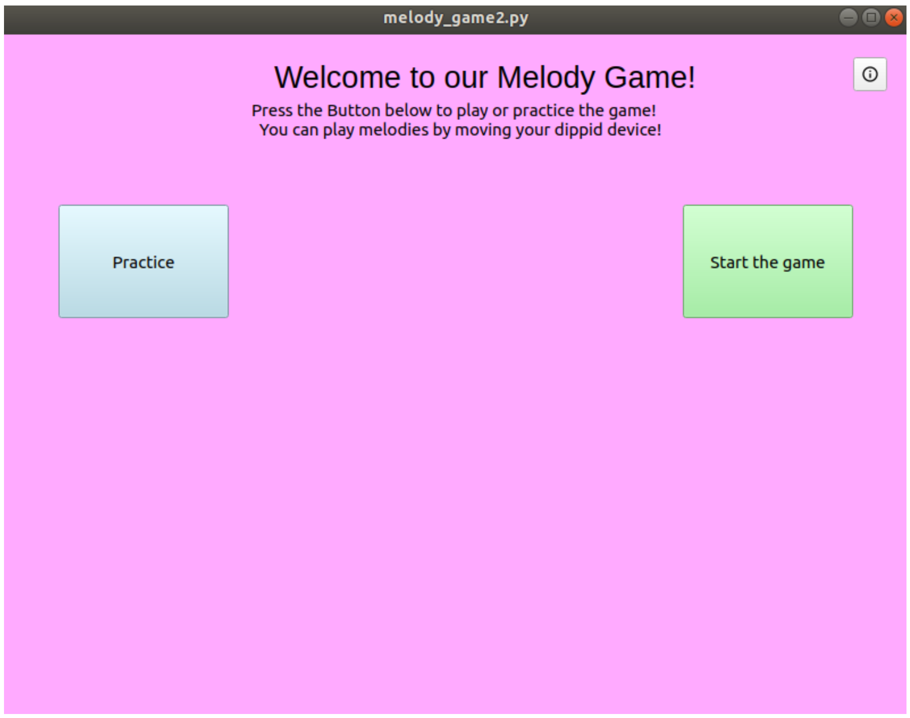
```
import time
import sys
import melodies

from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PyQt5.QtCore import *
from synthesizer import Player, Synthesizer, Waveform
from enum import Enum
from DIPPID import SensorUDP
```

With music as an important part of our application we conducted some research to find a suitable package to create sounds. Since the application is meant to be a game, we decided for a playful way to create music and chose the *Synthesizer* package. Previously we discussed picking another package to play sounds like on a piano. The melodies.py module is a self-written python file that contains all songs with their tones and potential DIPPID values. It is already in the same folder as the other files so nothing needs to be installed or relocated.

3 Implementation

We begin with a brief explanation of the UI and how we developed it. A simple *QMainWindow* is the basic structure, and everything happens inside this window. Some text labels at the top and two major buttons to enable one mode or another. A small but visible icon on the top right gives instructions to the user how to play the game. By clicking it, a *QDialog* with several more text labels, appears.

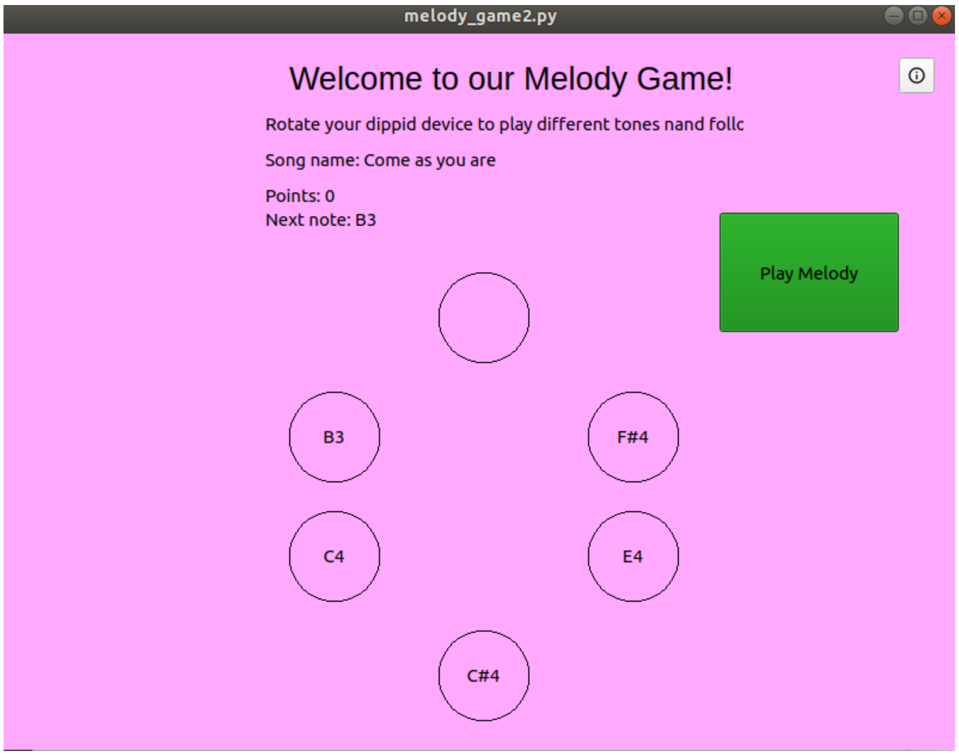

melody_game2.py

Welcome to our Melody Game!

Press the Button below to play or practice the game!
You can play melodies by moving your dippid device!

Figure 1: User Interface at the very beginning

By clicking on the right button, the UI changes and now contains a clock-like representation of MIDI notes that the user needs to play in a specific order. Each element of the "sound clock" is created with *QPainter*. The right button changes its functionality so when the user clicks on it a random melody is played. After the melody is played the user needs to be ready for turning DIPPID in the specific directions. To provide more stability and error tolerance the user can take as much time as he needs to. But if he wants to play a note, he needs to press a specific button of DIPPID. We used button 1, but it does not matter. A sound is played, and the user can proceed to the next tone. While playing the user can see which note is next with a *QLabel* at the top.


melody_game2.py

Welcome to our Melody Game!

Rotate your dippid device to play different tones nand follc

Song name: Come as you are

Points: 0
Next note: B3

Figure 2: User Interface while playing the game

The practice mode is very similar to the playing mode mentioned above. The main and only difference is that the user can play notes without counting it up in a scoring system. We implemented the same methods for both modes and called them twice, once in playing and in practice mode.

We now proceed with a step by step (or maybe function by function) walkthrough for some important sections of our implementation. Only the most important lines of code are mentioned below. The imported *Enum* module is used to create different states for our game. There are many options how to implement this, but an Enum is easy to use and clearly structured.

```
class GameState(Enum):
    INTRO = 1
    START = 2
    PRACTICE = 3
   DONE = 4
```

A class *Game* which inherits from *QMainWindow* contains several variables that are first declared and later filled. In its constructor, *Game* initializes the UI components, the Synthesizer module, a *QTimer* and updates the *GameState*. With passing *self* as a parameter of functions we ensure that our data can be passed from one point to another fluently inside our source code.

```
class Game(QMainWindow):
  def init(self):
        super().init()
        self.initUI()
        self.init_timer_game_loop()
        self.synthesizer = Synthesizer(…)
        self.timer = QTimer(self)
```

Initializing the UI also includes preparing the buttons for event handling, therefore connecting them with functions. Three functions for three buttons (practice, start, help) are written. Inside the help-function a *QDialog* is created which contains several lines of text with a *QLabel*. After defining several labels the box is opened. Another function called *paintEvent* is written and uses *QPainter* to draw the clock-like rectangles. Handling the entire UI requires at least 200 and at most 300 lines of code and overlaps with the now presented logic of our game.

The main function of our application is called *game_loop*. After using the aforementioned Enum conditionally the notes are displayed and the sensor data from DIPPID is collected. This happens in separate own functions. If not every note has been played, the next note is visualized on the screen. If the last note is played, the loop ends with a score and asks the user to play another round.

```
def game_loop(self):
      if self.GameState == GameState.START:
            self.display_next_note()
            self.get_sensor_data()
```

These functions are followed by further conditions that check if the user has pressed the button on DIPPID. If he has the data is collected for the time of pressing the button. If the button is not pressed no DIPPID data is collected. The DIPPID data is transformed into a sound in the function *play_tone*. We outline the simple structure exemplary below. With the x and y parameters of the DIPPID device we can create different angles with the phone and match it to different tones. If the DIPPID values are suitable a specific tone is played by the synthesizer. Finally, the tone is appended to a list of already played tones which is required later on to create the score system of the entire game. The code below is simplified and requires some further error handling if the DIPPID data and therefore the user's behaviour does not match any tones.

```
def play_tone(x,y,z):
      if(0.6 < x > 1) and (0.6 < y > 1):
            played_tone = 1
            synth.play_tone(played_tone)
            played_tones.append(played_tone)
```

The scoring system directly corresponds to the list of played tones with DIPPID and compares it with the given list of tones – the melodies. Another file *melodies.py* was created to separate our representation of each melody from the actual game. This gives us the option to add or delete further melodies to extend our game. Inside melodies.py each song is stored in a dictionary. The key of a song is simply its name and the value of a song is a list of tuples that contains the individual sequence of notes. Each tuple represents a note with a float value (which is used in the DIPPID data comparison) and the tone itself on the major scale of tones (e.g. "D4"). An entry of the dictionary *d* is passed from one file to another with the *select_new_song* function

```
def select_new_song():
      song = random.choice(list(d.keys()))
      values = d.get(random_song)
      return song, values
```

When the function is called inside *melody_game2.py* the tuple is unpacked, and the song is available for the entire game process. The game loop also handles the scoring system. For instance, if a given song contains 20 tones the user needs to play all 20 tones and is afterwards informed how many tones he completed successfully. Another round can be played, or the window can be closed. The functions *new_game* and *new_round* handle the game subsequently.

4 Playing the game

To compete successfully some major thoughts were put into the circumstances of the game. First the user needs to put his device steady in front of the screen (approximately DIPPID values of x = 0, y=1 and z= 0.2). Since data from DIPPID is collected at all times of no further limitations are developed we designed our game in a way that the user can turn his phone fast or slow, it does not matter for the outcome of the game. When pressing button 1, data is collected only once. If the user holds the button in a pressed state and moves around with his phone, the user may crash our entire game logic. Thus, we strongly advice the user to press the button once if he found the position accordingly to the clock-like tone.

The game is mostly a handling skills game that tests the user's capability to provide several subsequent turns with his phone, pressing a button and gives this movement a fun addition with the sounds played. With very few changes the game could also be modified into a memory game where the user needs to recreate a specific order of tones. With entire melodies consisting of more than 10 tones this would be way too difficult based on our selected songs. The 8-bit sound of the synthesizer and some well-known melodies complete our gaming experience.

References

Riverbank Computing (2021). PyQt5 (5.15.4) [Software]. Riverbank Computing.

https://riverbankcomputing.com/software/pyqt/download