Departamento de Ingeniería Eléctrica
Facultad de Ingeniería
Universidad de Concepción
Concepción, Chile.

549253-1 Taller de Aplicación TIC I

Proyecto Final

Profesor: *Vincenzo Caro Fuentes*

Profesor Ayudante: *Rodrigo Hernández*

Integrantes: *Javier Araya*

Martina Estrada

Matias Fernandez

18 de diciembre de 2025

Concepción, *18 de diciembre de 2025*

Resumen

En este informe del Proyecto Final para el Taller de Aplicación TIC I, se documenta paso a paso el desarrollo del videojuego "Guitar Raspi", junto con sus resultados, comentarios y el esquema del circuito utilizado para la implementación de botones físicos.

El desarrollo del proyecto se enfocó en la creación de un videojuego rítmico inspirado en Guitar Hero, implementado en Python. Se diseñó un sistema capaz de reproducir canciones sincronizadas con la aparición de notas musicales en pantalla, las cuales deben ser acertadas por el usuario en el momento preciso. Para ello, se implementó la lógica del juego, la gestión de puntaje y los efectos visuales asociados a los aciertos, además de una interfaz gráfica interactiva utilizando la librería PyQt6.

Asimismo, se desarrolló una estructura basada en carpetas para la gestión de canciones, permitiendo cargar dinámicamente archivos de audio, fondos e información de notas desde archivos de configuración en formato JSON. Para lograr una sincronización precisa entre la música y la jugabilidad, se implementaron herramientas capaces de convertir archivos CHART y MIDI a un formato compatible con el juego. Finalmente, se integraron botones físicos mediante los pines GPIO de la Raspberry Pi, permitiendo la interacción del usuario tanto a nivel físico como a través del teclado. El proyecto demuestra la aplicación práctica de conceptos de programación, manejo de interfaces gráficas y procesamiento de datos musicales.

Índice

1. Introducción **1**

2. Desarrollo Proyecto Guitar Raspi **2**
2.1. Descripción detallada del juego. 2
2.2. Estructura de la carpeta de canciones 2
2.3. Código del juego 3
2.3.1. Librerías 3
2.3.2. Parámetros 3
2.3.3. Notas Musicales 4
2.3.4. Carga de la canción 4
2.3.5. Variables de juego 5
2.3.6. Inicio de la canción 5
2.3.7. Bucle de juego 5
2.3.8. Detección de aciertos 6
2.3.9. Mensaje 6
2.4. Conversión de notas musicales 7
2.4.1. Archivo CHART 7
2.4.2. Archivo MIDI 9
2.5. Reproducción de audio 11
2.6. Interfaz Grafica 11
2.6.1. Menú del juego (Selección de canción) 12
2.6.2. Pantalla de juego 13
2.7. Botones Físicos 15
2.8. Esquema del circuito 16
2.9. Resultados y Comentarios 16

3. Conclusión **17**

1. Introducción

Las raspberry pi son una serie de mini computadoras de bajo costo y alto rendimiento desarrolladas por la *Raspberry Pi Fundation* en el Reino Unido. Estas están diseñadas para promover el aprendizaje en informática y programación siendo muy útiles gracias a sus características,como lo son el bajo costo, tamaño reducido y capacidad de ejecutar sistemas operativos completos que permiten su amplio uso en diversos proyectos.
El uso de las Rasberry pi dentro de las TICs pueden ser la creación de proyectos educativos, desarrollo de prototipos de hardware, automatización del hogar entre otras cosas. Todo esto gracias a su flexibilidad y capacidad de integración con otros sistemas además de la capacidad de utilizar diversos tipos de sensores.
Para este proyecto como grupo decidimos hacer nuestra propia versión del videojuego "Guitar Hero", el objetivo principal que nos propusimos fue recrear la misma mecánica del videojuego original, permitiendo introducir canciones a gusto de nosotros e imitar el sistema de reacción de tocar una nota cuando este cerca. Posteriormente se buscará dar un paso más allá y lograr utilizar botones físicos que cumplan la misma función.

2. Desarrollo Proyecto Guitar Raspi

2.1. Descripción detallada del juego.

Nuestro proyecto final corresponde a un videojuego rítmico inspirado en juegos del tipo Guitar Hero, implementado en Python y orientado a su ejecución en una Raspberry Pi. El juego consiste en la reproducción de una canción mientras una serie de notas visuales van cayendo por distintos carriles en la pantalla siguiendo un ritmo similar a la guitarra de la canción que fue escogida por el jugador. El objetivo principal del usuario es presionar el botón correspondiente al carril correcto en el instante preciso en el que la nota alcanza una línea de referencia.
El sistema permite la interacción mediante botones físicos conectados a los pines GPIO de la Raspberry Pi, así como también mediante el teclado, lo que facilita su jugabilidad tanto en un computador, como en una Raspberry Pi. El puntaje del jugador aumenta al acertar las notas y se muestra un efecto de "estallido" de la nota para mostrarle al jugador que a acertado, acompañado de un mensaje según que tan preciso fue al momento de acertar.
El juego incorpora un menú inicial que permite seleccionar distintas canciones, las cuales se cargan dinámicamente desde una estructura de carpetas, permitiendo así que se puedan agregar y/o eliminar canciones sin necesidad de modificar el código principal.

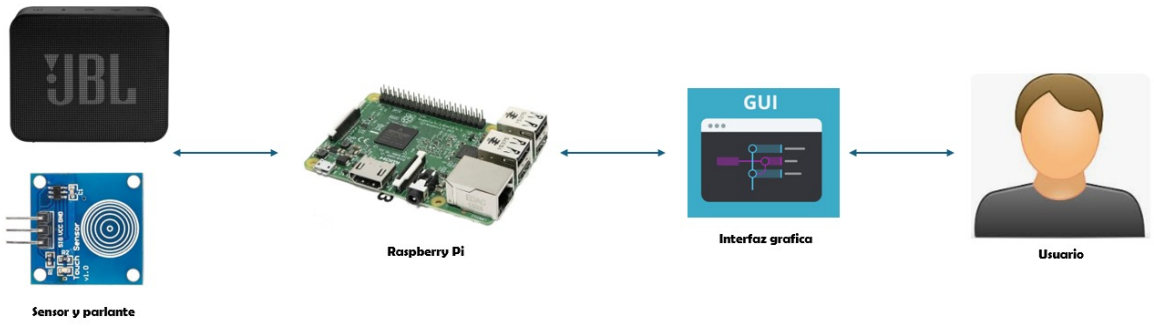

Fig. 1: Diagrama funcional básico.

2.2. Estructura de la carpeta de canciones

Como nombramos anteriormente el proyecto utiliza una estructura basada en carpetas independientes para cada canción. Todas ellas se almacenan dentro de una carpeta principal denominado "songs", ubicada en el mismo lugar que el código del juego, donde cada subcarpeta representa una canción distinta.

(a) Carpeta "songs". (b) Canciones.

Fig. 2: Estructura de carpetas y archivos de canciones.

Cada carpeta de canción contiene todos los recursos necesarios para su ejecución, tales como el archivo de audio, la imagen de fondo, el archivo de notas y un archivo de configuración. Este ultimo centraliza la información relevante de cada canción, indicando al programa qué archivos debe cargar y cómo sincronizar correctamente la reproducción.

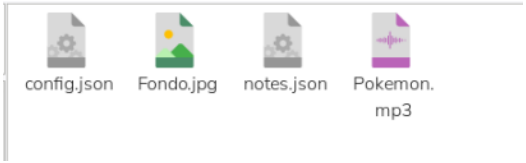

Fig. 3: Archivos necesarios para cada canción.

2.3. Código del juego

En esta sección explicaremos detalladamente el código principal del juego *Guitar Raspi*, esta se enfoca en la lógica del programa y su organización, dejando la descripción detallada de la interfaz gráfica para una sección posterior.

2.3.1. Librerías

Para comenzar con el desarrollo del código, es necesario importar diversas librerías que permiten la correcta ejecución del juego.

```
from PyQt6.QtWidgets import QApplication, QWidget, QLabel
from PyQt6.QtCore import Qt, QTimer, QTime
from PyQt6.QtGui import QPainter, QColor, QFont, QPen, QPixmap
from pathlib import Path
from gpiozero import Button
import json
import sys
import vlc
```

La librería `PyQt6` se utiliza para la creación de la ventana principal del juego (.QtWidgets), gestiona el tiempo y el uso de teclado, mouse, temporizador,etc (.QtCore). Por otra parte también se encarga de la visual tales como dibujo,colores,fuentes de letra e imágenes (.QtGui)
La librería `gpiozero` permite la lectura de botones físicos conectados a los pines GPIO de la Raspberry Pi.
`vlc` lo utilizamos para la reproducción del audio de las canciones.
Finalmente, las librerías `json`, `pathlib` y `sys` facilitan la lectura de archivos de configuración, el manejo de rutas y la ejecución general del programa.

2.3.2. Parámetros

```
LANES = 5
WINDOW_WIDTH = 600
WINDOW_HEIGHT = 800
HIT_LINE_Y = 550
NOTE_SPEED = 4
FRAME_MS = 16
SPAWN_Y = -60
BUTTON_PINS = [17,27,22,23,24]
```

Luego definimos una serie de parametros constantes que definen, la cantidad de carriles (LANES), el tamaño de la ventana de juego (WINDOW_WIDTH y WINDOW_HEIGHT), la posicion donde está

ubicada la linea de referencia (HIT_LINE_Y), la velocidad de caida de las notas y la posicion donde "nace" cada nota (NOTE_SPEED,FRAME_MS,SPAWN_Y) y por ultimo definimos los pines GPI de cada botón físico (BUTTON_PINS). Se definen como constantes para ajustar de mejor manera el juego sin alterar el código de manera significativa.

2.3.3. Notas Musicales

```
class Note:
def __init__(self, lane, y):
self.lane = lane
self.y = y
self.hit = False
```

Esta clase representa la nota musical que va descendiendo por la pantalla, almacena el carril, su posición y si fue o no acertada por el jugador

```
class HitEffect:
def __init__(self, lane, frames):
self.lane = lane
self.frames = frames
```

Por otro lado, en esta clase representamos el efecto visual asociado a los acierto,indicando el carril correspondiente y la duración del efecto. Esta idea fue implementada, ya que, al principio de la creación del juego no se tenia la certeza de si se había acertado o no a la nota.

2.3.4. Carga de la canción

```
def load_song(folder: Path):
    config_path = folder / "config.json"
    config = json.loads(config_path.read_text(encoding="utf-8"))
    audio_path = folder / config["audio"]
    background_path = folder / config["background"]
    chart_path = folder / config["chart"]
    chart_raw = json.loads(chart_path.read_text(encoding="utf-8"))
    chart = [(int(t), int(l)) for t, l in chart_raw]
    offset = int(config.get("offset_ms", 0))
    return {
        "name": config["name"],
        "audio": str(audio_path),
        "background": str(background_path),
        "chart": chart,
        "offset": offset,
    }
```

Esta función es la encargada de cargar la información de la canción que fue escogida por el jugador, ella lee el archivo llamado "config.json" de la Fig.3 el cual contiene lo siguiente:

```
{
    "name": "Pokemon",    #Nombre
    "audio": "Pokemon.mp3",  #Cancion en audio
    "background": "Fondo.jpg", #Fondo
    "chart": "notes.json",  #Notas Musicales
    "offset_ms": -120    #Desfase temporal de las notas (en ms)
}
```

2.3.5. Variables de juego

```
self.mode = "menu"
self.songs = []
self.selected_song = 0
self.current_song = None

self.notes = []
self.score = 0

self.chart = []
self.chart_index = 0
self.offset_ms = 0
self.travel_time_ms = 0
```

Creamos las variables principales para controlar la jugabilidad de la partida, como lo son el puntaje, las notas musicales y las posiciones correspondientes a cada cancion; y si el jugador está en el menú o en la pantalla de juego.

2.3.6. Inicio de la canción

```
def start_song(self, idx: int):
    self.current_song = self.songs[idx]
    self.chart = self.current_song["chart"]
    self.offset_ms = self.current_song["offset"]
    self.chart_index = 0
    self.notes = []
    self.hit_effects = []
    self.score = 0

    dist_pixels = HIT_LINE_Y - SPAWN_Y
    px_per_ms = NOTE_SPEED / FRAME_MS
    self.travel_time_ms = dist_pixels / px_per_ms

    self.start_time_ms = QTime.currentTime().msecsSinceStartOfDay()
    self.mode = "game"
```

Esta función “prepara” al juego para iniciar una canción, es la encargada de cargar los datos de la canción que fue seleccionada por el jugador, reinicia las variables del juego (esto sucede cuando el jugador cambia de canción luego de haber empezado por otra) y calcula el tiempo que tarde una nota en desplazarse desde que inicia hasta que choca con la linea de referencia, es la parte principal donde las notas se sincronizan con la música

2.3.7. Bucle de juego

```
def game_loop(self):
    if self.mode == "game":
        current_time_ms = QTime.currentTime().msecsSinceStartOfDay()
        current_ms = current_time_ms - self.start_time_ms

        lead_ms = self.travel_time_ms
        while (
            self.chart_index < len(self.chart)
            and current_ms >= self.chart[self.chart_index][0] - lead_ms + self.offset_ms
        ):
            _, lane = self.chart[self.chart_index]
            self.spawn_note_for_lane(lane)
```

```
            self.chart_index += 1

        for note in self.notes:
            note.y += NOTE_SPEED

        self.notes = [
            n for n in self.notes if n.y < WINDOW_HEIGHT + 80 and not n.hit
        ]
```

Este bucle se ejecuta continuamente y se encarga de actualizar el estado de la partida ¿Es necesaria? Es muy necesaria ya que lee las notas y genera las que vienen a continuación según la canción, mostrándolas en sus correspondientes carriles y eliminando a las que ya no son validas.

2.3.8. Detección de aciertos

```
def check_hit(self, lane):
    hit_window = 35
    best_note = None
    best_dist = hit_window

    for note in self.notes:
        if note.lane != lane or note.hit:
            continue
        dist = abs(note.y - HIT_LINE_Y)
        if dist <= best_dist:
            best_dist = dist
            best_note = note

    if best_note:
        best_note.hit = True
        self.score += 100
```

Aquí el código evalúa si se presiono el botón correcto cuando la nota musical toca la linea de referencia ¿Cómo? Este revisa si existe una nota del carril presionado que esté lo suficientemente cerca de la línea de golpe como para considerarse un acierto. Dependiendo de cuan preciso fue se le asigna un tipo de acierto y actualiza el puntaje (en caso de que sea un acierto).

2.3.9. Mensaje

```
   if best_dist <= 10:
        self.show_feedback("PERFECTO", QColor(0, 255, 0))
    else:
        self.show_feedback("BIEN", QColor(255, 215, 0))
else:
self.show_feedback("FALLASTE", QColor(255, 80, 80))
```

El código muestra un mensaje al jugador dependiendo de que tan efectivo fue su acierto o si no lo fue. Esta decisión fue netamente estética y para darle mas sentido de juego.

2.4. Conversión de notas musicales

Al principio, cuando empezamos a crear el juego, las notas musicales aparecían de forma aleatoria (con la librería "Random"), luego para que el juego tuviera mayor similitud con el juego "Guitar Hero" buscamos la forma de que las notas aparecieran en la pantalla de juego sincronizadas (o similar) a la canción que está sonando, al principio surgió la idea de hacerlo manualmente pero era muy complicado ya que queríamos hacer más de 5 canciones y nos tomaría mucho tiempo. Luego de buscar un poco encontramos una pagina llamada "CHORUS ENCORE" https://www.enchor.us, la cual corresponde a uno de los repositorios más extensos y utilizados por la comunidad de juegos rítmicos, especialmente por los de Clone Hero y Guitar Hero. Lo que mas nos llamó la atención de la pagina es que contiene archivos (.chart y .mid) los cuales que contienen información precisa sobre el tiempo de aparición de cada nota, su asignación a carriles específicos y la estructura rítmica de la canción. Aquí nos enfrentamos a uno de los primeros problemas, ya que nuestro código no leía el formato de estos archivos. Por lo tanto creamos dos códigos para convertir los archivos descargados a las notas que lee nuestro código.
Lo primero que hacemos es buscar la canción en la pagina.

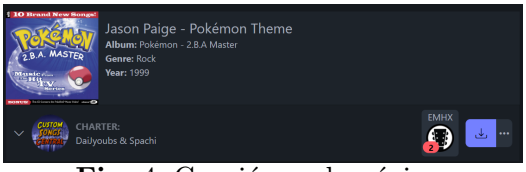

Fig. 4: Canción en la página.

Al descargarla viene una carpeta comprimida con las siguientes cosas.

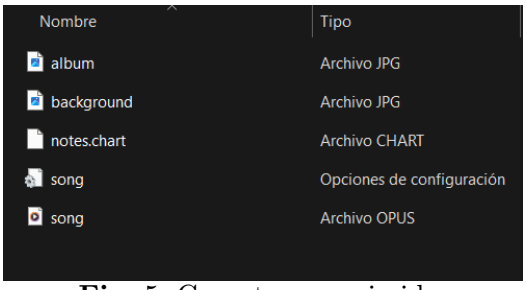

Fig. 5: Carpeta comprimida.

Aquí encontramos una serie de archivos, para nuestro caso tomamos el archivo CHART (notes.chart) y lo guardamos en una carpeta donde ahí está el código que nos convierte en el tipo de notas que lee el código principal del juego y nos devuelve un archivo .json con estas que luego pondremos en la carpeta de la canción (Fig. 2).

2.4.1. Archivo CHART

Lo primero que hacemos es importar las siguientes librerias.

```
from pathlib import Path
import json
```

La primera nos permitirá leer y procesar el primer archivo para poder convertirlo en lo que necesitamos, y la segunda es para que nos entregue las notas en formato json

```
def parse_chart(path, section_name="[ExpertSingle]"):
```

Definimos esta función para leer y analizar el archivo .chart, extrayendo de el tiempo y la nota, su función principal es transformar el tiempo que viene expresado en ticks a tiempo en mili-segundos.

```
lines = Path(path).read_text(encoding="utf-8", errors="ignore").splitlines()
```

Dentro de la función se lee linea por linea el archivo como si fuera un texto. Esto permite analizar manualmente cada parte del archivo, ignorando lo posibles problemas de codificación que podrían contener archivos de distintas fuentes.

```
resolution = 192
sync_ticks = []
in_song = False
in_sync = False
in_notes = False
notes_raw = []
```

Creamos variables para controlar que parte del archivo se está leyendo, además de estructuras para almacenar resolución, cambios de tiempo y notas encontradas.

```
if s.startswith("[") and s.endswith("]"):
    in_song = (s == "[Song]")
    in_sync = (s == "[SyncTrack]")
    in_notes = (s == section_name)
    continue
```

El archivo organizo sus secciones de la forma "[.....]", por lo tanto, el código identifica cada sección para trabajar unicamente con su información.

```
if in_song and "=" in s and s.lower().startswith("resolution"):
    parts = s.split("=")
    resolution = int(parts[1].strip())
```

Dentro de la sección "[Song]", se obtiene el valor de "resolution", que indica cuántos ticks representan un pulso musical. Este valor es fundamental para convertir ticks a mili-segundos.

```
if in_sync and "=" in s:
    left, right = s.split("=", 1)
    tick = int(left.strip())
    parts = right.strip().split()
    if len(parts) >= 2 and parts[0] == "B":
        bpm_raw = int(parts[1])
        bpm = bpm_raw / 1000.0
        sync_ticks.append((tick, bpm))
```

Por otro lado la sección "[SyncTrack]" nos indica la velocidad de la canción, si acelera o va mas lenta, este nos dice cuando lo hace y cual es la velocidad nueva, así el código lee y guarda la informacion de donde y cuando debe aparecer cada nota musical en el juego.

```
if in_notes and "=" in s:
    left, right = s.split("=", 1)
    tick = int(left.strip())
    parts = right.strip().split()
    if len(parts) >= 3 and parts[0] == "N":
        desc = int(parts[1])
        sustain = int(parts[2])
        notes_raw.append((tick, desc, sustain))
if not sync_ticks:
    sync_ticks = [(0, 120.0)]
```

En la seccion "[ExpertSingle]" donde por defectos están la notas, se extraen los evento de tipo "N" que representan a cada una de ellas, contienen un tick de inicio, el carril asociado y la duración (sustain). En caso de no contenerla se le da una por defecto.

```
segments = []
current_ms = 0.0
prev_tick = sync_ticks[0][0]
prev_bpm = sync_ticks[0][1]
ms_per_tick_prev = 60000.0 / (prev_bpm * resolution)
segments.append((prev_tick, current_ms, ms_per_tick_prev))
```

Se construye una lista de segmentos que relacionan ticks con tiempo real en mili-segundos. Cada segmento representa un intervalo donde el BPM (Beats Per Minute) conversión precisa de ticks a mili-segundos.

```
def tick_to_ms(tick):
    seg = segments[0]
    for s in segments:
        if s[0] <= tick:
            seg = s
        else:
            break
    tick_start, ms_start, ms_per_tick = seg
    return ms_start + (tick - tick_start) * ms_per_tick
```

Esta función es la encargada de convertir cualquier tick del chart en su tiempo real correspondiente en mili segundos a su vez, considera posibles cambios de BPM.

```
chart = []
for tick, desc, sustain in notes_raw:
    if 0 <= desc <= 4:
        t_ms = int(round(tick_to_ms(tick)))
        lane = desc
        chart.append([t_ms, lane])
```

Las notas leídas se convierten en pares de la forma [tiempo_ms, carril] filtrando únicamente los carriles válidos para el juego. El resultado se ordena temporalmente para facilitar su uso al momento de aparecer cada nota.

```
out_path.write_text(
    json.dumps(chart, indent=4),
    encoding="utf-8"
)
def main():
    chart_file = Path(__file__).with_name("notes.chart")
    chart = parse_chart(chart)
```

Finalmente la lista “chart” se guarda en un archivo .json el que es guardado en la misma carpeta donde está el archivo y el código que lo transcribe.

2.4.2. Archivo MIDI

Partimos importando las librerías que ocuparemos.

```
from pathlib import Path
import mido
import os
import json
```

al igual que en el codigo anterior usamos las librerias “pathlib” y “json” solo que ahora agregamos “mido” que sirve para leer y procesar archivos MIDI; y “os” que sirve para mostrar información de la carpeta.

```
MIDI_NOTE_TO_LANE = {
    60: 0,
    61: 1,
    62: 2,
    63: 3,
    64: 4,
}
```

A diferencia del archivo CHART, en los archivos MIDI las notas son números, por lo tanto definimos a que carril corresponde cada nota.

```
mid = mido.MidiFile(midi_path)
tpq = mid.ticks_per_beat
```

Se abre el archivo y obtenemos los "ticks por beat"(tpq) estos nos indican cuantos "pasos internos" tiene cada pulso musical. Esto nos ayudara a convertir el tiempo predeterminado por los archivos MIDI a mili segundos.

```
for i, track in enumerate(mid.tracks):
    name = ""
    for msg in track:
        if msg.type == "track_name":
        name = msg.name
        break
    print(f"  Track {i}: '{name}'")

if "guitar" in name or "part guitar" in name:
    part_track = track
if part_track is None:
    part_track = mid.tracks[0]
```

Un archivo de este tipo puede contener varias pistas de distintos instrumentos, este código en especifico busca donde está la pista de la guitarra y en caso de no estar el código usa la primera pista como respaldo.

```
tempo = 500000
ticks_acc = 0
chart = []
used_notes = set()
```

Definimos las variables necesarias para construir nuestras notas musicales, la velocidad de la canción, una que guarde el tiempo musical (ticks), una lista que guarde las notas del juego y un conjunto para registrar notas MIDI encontradas.

```
for msg in part_track:
    ticks_acc += msg.time

    if msg.type == "set_tempo":
        tempo = msg.tempo

    elif msg.type == "note_on" and msg.velocity > 0:
        note = msg.note
        used_notes.add(note)

        if note in MIDI_NOTE_TO_LANE:
            lane = MIDI_NOTE_TO_LANE[note]
            t_sec = mido.tick2second(ticks_acc, tpq, tempo)
            t_ms = int(round(t_sec * 1000))
            chart.append([t_ms, lane])
chart.append([t_ms, lane])
```

```
chart.sort(key=lambda x: x[0])
```

El codigo recorre todas la pista que fue seleccionada en el archivo (la guitarra), si el código detecta posibles cambios de tempo de la canción, actualizando el valor correspondiente cuando aparecen. Cuando se identifica un evento que indica el inicio de una nota (note_on con velocidad mayor que cero), el programa verifica si dicha nota está en alguno de los carriles del juego. En caso de serlo, el tiempo acumulado en ticks se convierte a mili-segundos utilizando la información del "tempo" y la resolución del MIDI, generando una nota valida para el juego y siendo ordenadas en orden para que se muestren en la partida.

```
with open(out_file, "w", encoding="utf-8") as f:
    json.dump(chart, f, indent=4)
```

Finalmente la lista se guarda en un archivo .json y este es guardado en la misma carpeta donde está el archivo y el código que lo transcribe.

2.5. Reproducción de audio

La librería encargada de reproducir el audio de la canción es la llamada "vlc".

```
self.vlc_instance = vlc.Instance()
self.player = self.vlc_instance.media_player_new()
self.player.audio_set_volume(80)
```

En la class llamada "GameWidget" creamos una instancia de VLC para luego crear un reproductor de audio funcional, el cual nos permite cargar y reproducir archivos de sonido durante la ejecución del juego.

```
self.player.stop()
media = self.vlc_instance.media_new(self.current_song["audio"])
self.player.set_media(media)
self.player.play()

self.start_time_ms = QTime.currentTime().msecsSinceStartOfDay()
```

Luego, cuando el jugador elige una canción del menú, se detiene cualquier audio anterior, se carga el archivo de audio asociado a la canción seleccionada y se reproduce. En simultaneo cuando se inicia la reproducción se guarda un tiempo de referencia (start_time_ms) esto sirve para sincronizar el juego con la canción.

```
self.player.stop()
```

Finalmente, si el jugador vuelve al menú o cierra la ventana, la canción se detiene.

2.6. Interfaz Grafica

```
class GameWidget(QWidget):
    def __init__(self):
        super().__init__()

            self.setFixedSize(WINDOW_WIDTH, WINDOW_HEIGHT)
            self.setWindowTitle("Guitar Raspi")
```

Definimos la clase donde crearemos la interfaz del juego, lo primero que creamos es la ventana principal del juego, con las medidas que definimos al principio y le agregamos el nombre.

```
def paintEvent(self, event):
    painter = QPainter(self)
```

Esta función es la encargada de "dibujar" los elementos visuales que utiliza el juego (notas musicales, carriles, puntaje). Se le llama cada vez que la pantalla debe actualizarse, por ejemplo, cuando se avanza del menú a la pantalla de juego.

2.6.1. Menú del juego (Selección de canción)

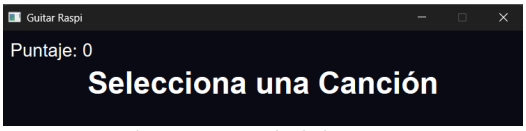

Fig. 6: Menú del juego.

```
    if self.mode == "menu":
        painter.fillRect(self.rect(), QColor(10, 10, 20))
        painter.setPen(QColor(255, 255, 255))
        painter.setFont(QFont("Arial", 26, QFont.Weight.Bold))
        painter.drawText(
            0,
            40,
            self.width(),
            40,
            Qt.AlignmentFlag.AlignHCenter | Qt.AlignmentFlag.AlignVCenter,
            "Selecciona una Cancion",
    )
self.score_label = QLabel("Puntaje: 0", self)
```

En la parte superior de nuestro menú se le indica al usuario que seleccione una canción de la lista para poder iniciar la partida y de igual forma se muestra permanentemente el puntaje en la esquina superior derecha de la ventana.

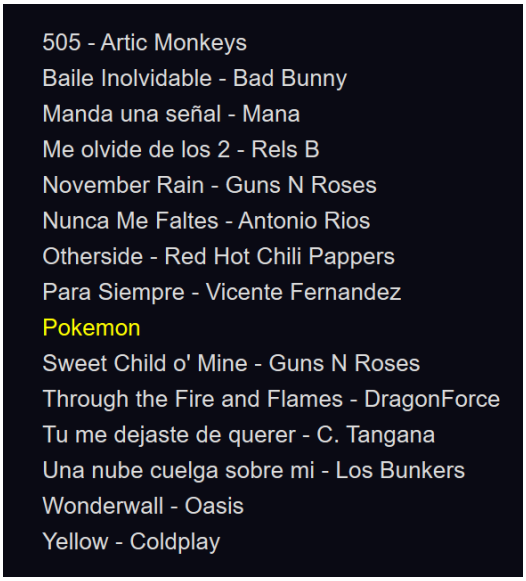

Fig. 7: Lista de canciones.

```
for i, song in enumerate(self.songs):
    color = QColor(255, 255, 0) if i == self.selected_song else QColor(220, 220, 220)
    painter.setPen(color)
    painter.drawText(
        60,
```

```
        y0 + i * line_h,
        self.width() - 120,
        line_h,
        Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignVCenter,
        song["name"],
        )
```

Luego, en la parte media del menú se le presenta al usuario la lista de canciones que están disponible para jugar. Además la canción que está actualmente seleccionada es resaltada con color amarillo facilitando así la navegación en la lista.

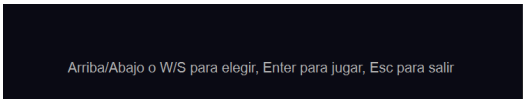

Fig. 8: Indicaciones.

```
    painter.setPen(QColor(180, 180, 180))
    painter.setFont(QFont("Arial", 12))
    painter.drawText(
        0,
        self.height() - 60,
        self.width(),
        20,
        Qt.AlignmentFlag.AlignHCenter | Qt.AlignmentFlag.AlignVCenter,
        "Arriba/Abajo o W/S para elegir, Enter para jugar, Esc para salir",
    )
return
```

Finalmente, en la parte superior de nuestro menú se encuentran las indicaciones de como navegar por la lista de canciones, como seleccionar la que está resaltada y como salir del juego

2.6.2. Pantalla de juego

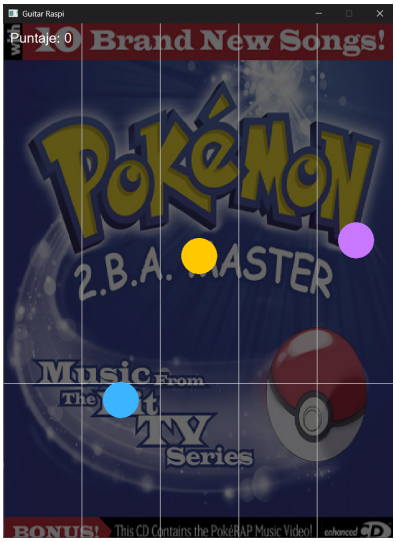

Fig. 9: Pantalla de juego.

```
if not  self.background_image.isNull():
    painter.drawPixmap(self.rect(), self.background_image)
else:
    painter.fillRect(self.rect(), QColor(20, 20, 20))
```

Durante la partida el fondo de nuestro juego es distinto para cada canción (En su mayoría la portada del álbum de cada canción). En caso de no existir (o estar mal escrita) se utiliza un fondo de color neutro.

```
lane_width = self.width() / LANES

for lane in range(LANES):
    x = lane * lane_width
    painter.fillRect(int(x), 0, int(lane_width), self.height(), QColor(0, 0, 0, 150))
    painter.drawLine(int(x), 0, int(x), self.height())
```

La pantalla se divide en 5 carriles verticales, por aquí se desplazan las notas musicales y son las encargadas de hacerle saber al jugador que botón presionar.

```
painter.setPen(QColor(255, 255, 255))
painter.drawLine(0, HIT_LINE_Y, self.width(), HIT_LINE_Y)
```

Integramos también la linea referencial, esta indica cuando el jugador debe presionar el botón para sumar puntaje.

```
for note in self.notes:
    color = self.lane_colors[note.lane]
    painter.setBrush(color)
    painter.setPen(Qt.PenStyle.NoPen)
    painter.drawEllipse(...)
```

Las notas que bajan por los carriles tienen una forma circular y cada una tiene un color distinto dependiendo del carril por donde caen, siguiendo la idea del juego Guitar Hero.

Fig. 10: Efecto.

```
for eff in self.hit_effects:
    painter.setPen(pen)
    painter.setBrush(Qt.BrushStyle.NoBrush)
    painter.drawEllipse(...)
```

Cuando el jugador acierta la nota se muestra un efecto, el cual definimos en la explicación del código principal (2.3.3).

Fig. 11: Mensajes .

```
        if self.feedback_frames > 0 and self.feedback_text:
            painter.setFont(QFont("Arial", 24, QFont.Weight.Bold))
            painter.setPen(self.feedback_color)
            painter.drawText(
                0,
                HIT_LINE_Y - 80,
                self.width(),
                40,
                Qt.AlignmentFlag.AlignHCenter | Qt.AlignmentFlag.AlignVCenter,
                self.feedback_text,
            )
```

Durante la partida se le muestran mensajes al jugador dependiendo de si acertó, que tan bien lo hizo o si falló, estos están centrados y desaparecen luego de un intervalo pequeño de tiempo.

2.7. Botones Físicos

Como describimos en las secciones 2.3.1 y 2.3.2, par ala implementacion de botones fisico necesitamos su libreria y sus Pines GPIO.

```
from gpiozero import Botton
BUTTON_PINS =[17,27,22,23,24]
```

```
self.buttons = [Button(pin) for pin in BUTTON_PINS]
```

Aqui el codigo recorre la lista de BUTTON_PINS y para cada uno crea un objeto de Button(pin) guardando los botones en una lista, esto seria self.buttons[0] el cual corresponde al botón del GPIO 17 self.buttons[1] seria el botón del GPIO 27
y así con los 5 botones.

```
self.button_prev_state = [False] * LANES
```

Una vez asignados todos lo botones se les da como estado el frame anterior, esta parte cumple con la función de controlar que el botón este o no presionado y que eso sea lo que se va a detectar.

```
for lane, button in enumerate(self.buttons):
    pressed = button.is_pressed
```

Aquí el button.is_pressed devuelve “True” si el botón esta presionado o en caso contrario “False”.

```
if pressed and not self.button_prev_state[lane]:

    self.check_hit(lane)
self.button_prev_state[lane] = pressed
```

Esto corresponde a la parte clave del programa en lo que respecta a botones, esto verifica que el botón acaba de ser presionado y cambia el estado de este a presionado.
El self.check_hit(lane) es el que verifica que el índice calce con el carril y así el juego intenta “golpear” la nota que va por ese carril.

2.8. Esquema del circuito

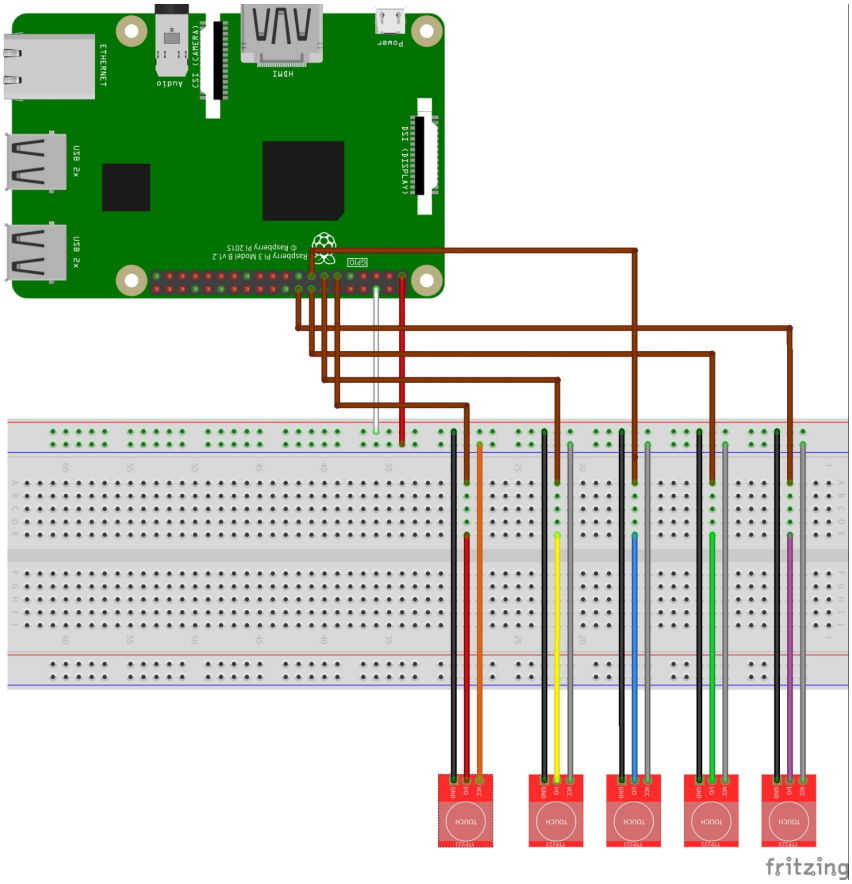

Fig. 12: Esquema del circuito.

2.9. Resultados y Comentarios

El desarrollo de este proyecto nos permitió implementar de manera exitosa una replica simple pero funcional del videojuego Guitar Hero en una raspberry pi logrando integrar tanto software como hardware.

Nuestro juego fue capaz de reproducir canciones, generar las notas musicales sincronizadas con el audio y detectar de manera correcta las acciones del usuario a través de los botones físicos.

En lo que respecta al funcionamiento, nuestra versión final del juego responde de manera estable durante su ejecución, no así sus primeras versiones, que mostraban una leve caída de fps la cual fue resuelta.

La sincronización de la música con las notas es uno de los aspectos mas relevantes del juego logrando un resultado más que satisfactorio gracias a que ocupamos una pagina utilizada por la comunidad del juego garantizando mayor efectividad.

Respecto a la interacción con los botones físicos esta fue correcta y no tuvo mayores complicaciones, la elaboración de un set up para que los cables no estorbaran fue uno de los puntos mas complicado, pero en cuanto a conexión e implementación no hubo problemas.

En general, si bien estamos conformes con los resultados de nuestro proyecto, si hemos identificados posibles mejoras a futuro, tales como la optimización del mismo proyecto, la incorporación de niveles de dificultad, quizás implementarle otro tipo de sensores como leds, que tenga un sistema de puntuación mas complejo y que se pueda ir registrando, entre otras cosas. Pero en si nuestros resultados obtenidos muestran que somos capaces de desarrollar algo de tal envergadura.

3. Conclusión

En este proyecto logramos desarrollar de manera exitosa Guitar Raspi, una réplica funcional de un videojuego rítmico inspirado enEn el presente proyecto se logró desarrollar de manera exitosa Guitar Raspi, una réplica funcional de un videojuego rítmico inspirado en Guitar Hero, implementado sobre una Raspberry Pi. A lo largo del desarrollo se aplicaron conocimientos de programación en Python, manejo de interfaces gráficas mediante PyQt6, procesamiento de archivos musicales y utilización de pines GPIO, cumpliendo con los objetivos propuestos inicialmente.

El sistema desarrollado permitió reproducir canciones de forma sincronizada con la aparición de notas musicales en pantalla, detectando correctamente las acciones del usuario tanto a través del teclado como de botones físicos. Además, se implementó una estructura modular, ya que separa claramente la gestión de canciones, la lógica del juego, la interfaz gráfica y la interacción con el los botones. Esta organización permite agregar nuevas canciones o modificar funcionalidades específicas sin afectar el funcionamiento general del sistema. La conversión de archivos “CHART” y “MIDI” a un formato compatible con el juego fue uno de los desafíos más relevantes, el cual se resolvió satisfactoriamente logrando una sincronización precisa entre música y jugabilidad.

En términos de desempeño, la versión final del juego presentó un funcionamiento estable, corrigiendo problemas iniciales de rendimiento y mejorando la experiencia de usuario. Asimismo, la interacción con los botones físicos resultó efectiva.

Finalmente, si bien el proyecto cumple con todos los requerimientos planteados, se identifican posibles mejoras futuras, tales como la incorporación de niveles de dificultad, un sistema de puntuación más avanzado, efectos visuales adicionales o el uso de otros sensores.