Universidad de Concepción

Facultad de Ingeniería

Departamento de Ingeniería Eléctrica

Taller de aplicación TIC 1

Profesor: Vincenzo Caro

MiniProyecto 2

Integrantes:

Martín Johnson

Diego Sanhueza

Vicente Vásquez

Concepción, Noviembre 2025

Resumen

El presente informe describe el desarrollo del MiniProyecto 2 correspondiente al curso TIC I, cuyo objetivo fue integrar programación en Python, manejo de archivos, comunicación entre dispositivos y control de hardware mediante una Raspberry Pi. El proyecto se dividió en dos actividades principales.

La primera actividad consistió en implementar un sistema de juego colaborativo–competitivo inspirado en Among Us, compuesto por minijuegos locales, registro estructurado de eventos y coordinación mediante un host central. Para ello, se desarrollaron dos minijuegos de consola y dos minijuegos basados en sensores reales, utilizando botones, LEDs y un buzzer. Además, se implementó un sistema de logging unificado, la fase de Lobby con confirmación de conexión, dos rondas de minijuegos con asignación y sabotajes, y una ronda final de supervivencia basada en la gestión de vida del jugador.

La segunda actividad correspondió a la creación de una Pokédex interactiva utilizando PyQt5, capaz de mostrar información de diferentes Pokémon en base a archivos JSON, incluyendo su imagen, tipo y descripción, con navegación mediante botones y una interfaz gráfica inspirada en el diseño clásico del dispositivo.

Este proyecto permitió reforzar conceptos de programación modular, lectura y escritura de archivos, diseño de interfaces gráficas, control de hardware, y comunicación entre procesos. Asimismo, promovió el trabajo colaborativo y la organización sistemática del código, dando lugar a una solución completa que integra software, hardware y elementos interactivos.

Actividad 1.1 -Diseño e implementación de minijuegos

La primera parte de la actividad consistió en el diseño e implementación de cuatro minijuegos: dos desarrollados en consola y dos utilizando hardware real en la Raspberry Pi. Esta etapa nos permitió aplicar programación estructurada, modularización del código y, en el caso de los minijuegos con sensores, la integración práctica entre software y hardware. Como grupo, nos organizamos dividiendo la programación por módulos, lo que facilitó el trabajo colaborativo y nos permitió avanzar de manera ordenada.

Los minijuegos de consola representaron una extensión natural del trabajo realizado en el MiniProyecto 1, mientras que los minijuegos con sensores implicaron desafíos adicionales como el manejo de botones, LEDs y tiempos precisos de lectura, así como la sincronización entre la lógica del juego y la respuesta física del usuario. A continuación, se detallan cada uno de los minijuegos implementados.

1.1.1 Minijuego 1 — Ahorcado de colores (Consola)

1.1.1 Minijuego 1 — Ahorcado de colores (Consola)

Este minijuego corresponde a una variación del clásico “ahorcado”, utilizando únicamente palabras relacionadas con colores. El programa solicita al jugador dos parámetros iniciales: la cantidad de letras que debe tener la palabra y el número de intentos disponibles. A partir de estos datos, se filtra una lista predefinida y se selecciona una palabra aleatoria que cumpla con la longitud indicada.

Durante la partida, el usuario ingresa letras una por una. Si la letra está en la palabra, se revela su posición en la cadena de guiones; si no, se descuenta un intento. El juego termina cuando el jugador completa la palabra o se queda sin intentos. Esta actividad nos permitió reforzar el uso de ciclos, manejo de cadenas y validación de entradas.

En términos de integración con el sistema de logging del proyecto, este minijuego retorna un diccionario con el resultado ("Win.º "Lose") y un puntaje asociado (100 o 0), lo que facilita su registro por parte del módulo principal.

Listing 1: Núcleo del minijuego “Ahorcado de colores”.

```
def jugar_ahorcado():
    palabras = ["rojo","naranja","amarillo","verde","azul",
                "morado","rosado","celeste","blanco","negro","gris"]
    palabras = [p.lower() for p in palabras]

    nletras  = int(input("N mero de letras: "))
    intentos = int(input("N mero de intentos: "))

    candidatas = [p for p in palabras if len(p) == nletras]
    palabra = random.choice(candidatas)

    guiones = "_" * nletras
    intentos_rest = intentos
    resultado = "Lose"

    while intentos_rest > 0:
        letra = input("Letra: ").lower()

        if len(letra) == 1 and letra in palabra:
            guiones = "".join(letra if palabra[i]==letra else
               guiones[i]
                              for i in range(nletras))
            if guiones == palabra:
                resultado = "Win"
                break
        else:
            intentos_rest -= 1

    score = 100 if resultado == "Win" else 0
    return {"Result": resultado, "Score": score}
```

El comportamiento del programa fue estable en todas las pruebas. Las letras se revelaron correctamente en múltiples posiciones, y la estructura del código permitió un juego fluido e intuitivo. Como mejora, podríamos incluir una lógica para evitar que el jugador ingrese la misma letra varias veces, o agregar una versión con “modo difícil” añadiendo un temporizador.

1.1.2 Minijuego 2 — Adivina el Pokémon (Consola)

Este minijuego presenta una dinámica de adivinanza basada en pistas progresivas, utilizando una lista de Pokémon almacenada directamente en el código. Para cada partida, el programa selecciona aleatoriamente un Pokémon y entrega hasta tres pistas, permitiendo asignar un puntaje proporcional al desempeño del jugador.

Como grupo, este minijuego nos resultó sencillo y entretenido de implementar, ya que se basa en una estructura clara y fácil de modularizar. Su diseño también nos ayudó a practicar el uso de diccionarios y listas de Python, así como la validación de entradas por consola.

El flujo del juego es el siguiente:

1. Se selecciona un Pokémon aleatorio desde una lista predefinida.
2. Se entrega la **primera pista**, correspondiente a la **generación** del Pokémon.
3. Si el jugador falla, se muestra una **segunda pista** relacionada con su **color**.
4. Si aún no acierta, se muestra una **tercera pista** indicando su **tipo**.
5. Si adivina en alguna de las etapas, obtiene un puntaje según lo implementado en el código:
   - **100 puntos** si acierta con la primera pista.
   - **67 puntos** con la segunda.
   - **33 puntos** con la tercera.
   - **0 puntos** en caso de no acertar.
6. El minijuego retorna un diccionario con `Result"` y `"Score"`, lo que facilita su integración al sistema de logging del proyecto.

Este minijuego fue ideal para incorporar dentro de las rondas, ya que es rápido, no necesita hardware y permite diferenciar el rendimiento del jugador mediante puntajes escalonados.

Listing 2: Función principal del minijuego “Adivina el Pokémon".

```
def jugar_pokemon():
    pokemons = [
        {"name":"Bulbasaur","generation":1,"color":"Verde","type":"
           Planta"},
        {"name":"Charmander","generation":1,"color":"Naranja","type"
           :"Fuego"},
        {"name":"Squirtle","generation":1,"color":"Celeste","type":"
           Agua"},
        ...
    ]

    p = random.choice(pokemons)

```

```
    print("Pista 1: Generaci n", p["generation"])
    guess = input("Respuesta: ").capitalize()
    if guess == p["name"]:
        return {"Result":"Win","Score":100}

    print("Pista 2: Color", p["color"])
    guess = input("Respuesta: ").capitalize()
    if guess == p["name"]:
        return {"Result":"Win","Score":67}

    print("Pista 3: Tipo", p["type"])
    guess = input("Respuesta: ").capitalize()
    if guess == p["name"]:
        return {"Result":"Win","Score":33}

    return {"Result":"Lose","Score":0}
```

El minijuego se comportó de manera estable durante las pruebas y su diseño progresivo mantuvo la atención del usuario. Una dificultad menor fue estandarizar la capitalización de la entrada para compararla correctamente con el nombre del Pokémon, lo que se solucionó utilizando el método capitalize(). Como mejora futura, podríamos ampliar la lista de Pokémon o incluir pistas adicionales como habilidades o descripciones breves.

1.1.3 Minijuego Sensor 1 — Secuencia de LEDs (Simón dice)

Este minijuego basado en hardware utiliza:

- **3 LEDs** (rojo, amarillo, azul)
- **3 botones físicos** asociados a cada LED
- **1 buzzer** para señales de acierto y error

El programa genera una secuencia de tres colores (permitiendo o no la repetición), la muestra mediante los LEDs y luego solicita al jugador reproducirla mediante los botones. Cada error añade una penalización de tiempo, y al completar correctamente la secuencia, el jugador gana.

El resultado final (`True`/`False`) se convierte en `"Win"` o `"Lose"` al momento de registrar el log.

Listing 3: Funciones principales del minijuego Sensor 1 (Simón dice)

```

# Secuencia aleatoria de 3 colores
def gen_secuencia(allow_repeat):
    base = [1,2,3]          # 1=rojo, 2=amarillo, 3=azul
    seq = []
    while len(seq) < 3:
```

```
        x = random.choice(base)
        if allow_repeat or x not in seq:
            seq.append(x)
    return seq

# Lectura de bot n (devuelve 1, 2 o 3)
def leer_boton_una_vez():
    while True:
        if btn_r.is_pressed:
            while btn_r.is_pressed: time.sleep(0.005)
            return 1
        if btn_y.is_pressed:
            while btn_y.is_pressed: time.sleep(0.005)
            return 2
        if btn_b.is_pressed:
            while btn_b.is_pressed: time.sleep(0.005)
            return 3

# N cleo del minijuego
def juego():
    Ttotal = pedir_tiempo("Tiempo total del juego")
    permitir_rep = pedir_bool("  Permitir   repetici n?")
    seq = gen_secuencia(permitir_rep)

    input("Presiona ENTER para comenzar...")
    t0 = time.time()
    progreso = 0
    penal = 0.0

    while True:
        trans = time.time() - t0
        t_rest = max(0, Ttotal - trans - penal)
        imprimir_tiempo_restante(t_rest)

        if t_rest <= 0:
            melody_gameover()
            return False

        boton = leer_boton_una_vez()
        esperado = seq[progreso]

        if boton == esperado:
            bip_ok()
            progreso += 1
            if progreso == 3:
                melody_victory()
                return True
        else:
```

```
            bip_fail()
            progreso = 0
            penal += 1.0
```

1.4 Minijuego Sensor 2 — Botón de los 10 segundos

El segundo minijuego basado en hardware se centra en la percepción del tiempo por parte del jugador. En este caso sólo se utiliza un botón físico conectado a la Raspberry Pi, pero la lógica del juego depende de medir con precisión el intervalo transcurrido entre el inicio y la pulsación.

El flujo del minijuego es el siguiente:

1. El jugador inicia la prueba desde la consola, presionando ENTER para comenzar.
2. En ese momento se almacena el tiempo inicial mediante `time.time()`.
3. El sistema queda a la espera de que el jugador presione el botón físico.
4. Cuando se detecta la pulsación, se registra el tiempo final y se calcula el intervalo transcurrido.
5. Se compara dicho intervalo con el objetivo de **10 segundos** y se calcula el error absoluto.
6. En función de ese error, se asigna un puntaje y un resultado (`"Win"` o `"Lose"`), que se devuelve al módulo principal en forma de diccionario con las claves `Result"` y `"Score"`.

Este minijuego es rápido de ejecutar y no requiere más que un único botón, por lo que resultó ideal para integrarlo dentro de las rondas del juego. Además, permite diferenciar el desempeño de los jugadores según su capacidad para estimar el tiempo con precisión.

Listing 4: Funciones principales del minijuego Sensor 2 (botón de los 10 segundos)

```
1
2  BUTTON_PIN = 17
3  button = Button(BUTTON_PIN, pull_up=True)
4
5  def juego_tiempo_10s():
6      input("Presiona ENTER para comenzar...")
7
8      t_inicio = time.time()
9      button.wait_for_press()
10     t_fin = time.time()
11
12     tiempo = t_fin - t_inicio
13     diferencia = abs(tiempo - 10.0)
14
15     if diferencia < 0.2:
```

```
        result, score = "Win", 100
    elif diferencia < 0.5:
        result, score = "Win", 70
    elif diferencia < 1.0:
        result, score = "Win", 40
    else:
        result, score = "Lose", 0

    return {"Result": result, "Score": score}
```

Este minijuego fue uno de los más sencillos de implementar, pero también uno de los más interesantes en términos de precisión. El uso de un único botón hizo que el armado del circuito fuera muy rápido, permitiéndonos enfocarnos en la medición del tiempo y en la asignación del puntaje. Durante las pruebas observamos variaciones comunes en la estimación de los 10 segundos, lo cual hizo que el sistema de puntajes graduales fuese adecuado y motivador. Como posible mejora, podríamos añadir retroalimentación sonora o un modo avanzado que solicite varios intervalos consecutivos.

Actividad 1.2 — Registro de jugadores (Lobby)

Esta actividad tuvo como propósito implementar la fase inicial del juego, conocida como **Lobby**, donde cada jugador se registra ante el host y confirma su disponibilidad para participar en la partida. A diferencia de los minijuegos, esta sección requirió integrar comunicación remota mediante SSH, envío de archivos por SFTP y lectura coordinada de logs entre el jugador y el host central.

El flujo básico del Lobby se resume en los siguientes pasos:

1. El jugador genera un registro local indicando la acción `"Join"`.
2. El archivo de eventos es enviado al host mediante SSH y SFTP.
3. El jugador queda a la espera de un mensaje del host con la acción `."Accepted"`.
4. Una vez recibido, el jugador registra la acción `Ready"` y vuelve a enviar su log.

Este mecanismo permite asegurar que todos los jugadores están conectados y listos antes de iniciar las rondas, evitando desincronizaciones o inicios prematuros.

Listing 5: Funciones principales del proceso de Lobby (Join → Accepted → Ready)

```

def conectar_con_host():
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh.connect(hostname=HOST, username=USER, password=PASS)
    return ssh, ssh.open_sftp()

def lobby_connection():
```

```
    ssh, sftp = conectar_con_host()

    # 1. Registrar llegada del jugador
    log_event("Lobby", "Join")
    enviar_log_al_host()

    # 2. Esperar "Accepted" desde el host
    while True:
        sftp.get(REMOTE_STATUS, "game_status_local.log")
        with open("game_status_local.log") as f:
            for line in f:
                data = json.loads(line)
                if data.get("Action") == "Accepted" and data.get("
                   PlayerID") == PLAYER_ID:

                    # 3. Confirmar disponibilidad del jugador
                    log_event("Lobby", "Ready")
                    enviar_log_al_host()
                    return
        time.sleep(1)
```

La implementación del Lobby fue una de las partes más interesantes del proyecto, ya que implicó trabajar con comunicación remota y coordinación entre distintos dispositivos. Al principio resultó desafiante manejar la lectura de los archivos del host sin generar conflictos, pero la estrategia de revisar los logs línea por línea permitió obtener un funcionamiento estable. En general, esta actividad nos ayudó a comprender de mejor manera el flujo cliente–servidor y la importancia de un sistema de registro bien estructurado.

Actividad 1.3 — Despliegue de rondas (R1 y R2)

En esta actividad se integraron los minijuegos desarrollados en la sección 1.1 dentro de un flujo de juego estructurado en dos rondas: **Ronda 1 (R1)** y **Ronda 2 (R2)**. Cada ronda está compuesta por tres minijuegos, los cuales son asignados por el host mediante registros en el archivo de estado. El jugador, por su parte, lee estas asignaciones, ejecuta el minijuego correspondiente y registra el resultado en su propio log.

Esta etapa del proyecto fue clave para pasar desde programas aislados a un sistema coordinado, donde intervienen tanto el host como el jugador y los distintos módulos de minijuego. Como grupo, nos vimos obligados a mantener una buena organización del código para que la lógica del flujo fuera clara y fácil de depurar.

El flujo general de cada ronda puede resumirse en los siguientes pasos:

1. El host escribe en el archivo de estado una asignación de minijuego para el jugador, indicando un `GameID` asociado a la ronda (R1 o R2).

2. El jugador lee periódicamente dicho archivo hasta encontrar una asignación dirigida a su `PlayerID`.

3. Según el `GameID` recibido, el jugador ejecuta el minijuego correspondiente (consola o sensor).

4. Al finalizar el minijuego, se obtiene un diccionario con `Result"` y `"Score"`, el cual se registra en `player_events.log` mediante la función `log_event()`.

5. El archivo de log se envía al host por SFTP, para que éste pueda actualizar el estado global de la partida.

6. Este proceso se repite tres veces por cada ronda, completando así el conjunto de minijuegos asignados al jugador.

De esta forma, las rondas R1 y R2 permiten orquestar el uso de todos los minijuegos implementados, manteniendo un registro detallado de las acciones realizadas por cada jugador.

Listing 6: Ejecución de minijuegos y flujo general de las rondas R1 y R2

```
1
2  def ejecutar_minijuego(game_id, stage):
3      if game_id == 1:
4          res = Minijuego1.jugar_ahorcado()
5      elif game_id == 2:
6          res = Minijuego2.jugar_pokemon()
7      elif game_id == 3:
8          res = Minijuegosensor1.juego()
9      elif game_id == 4:
10         res = Minijuegosensor2.juego_tiempo_10s()
11     else:
12         print("GameID desconocido:", game_id)
13         return
14
15     log_event(stage, "Result",
16               game_id,
17               res["Result"],
18               res["Score"])
19     enviar_log_al_host()
20
21
22 def jugar_rondas():
23     for ronda in ["R1", "R2"]:
24         log_event(ronda, "Start")
25         enviar_log_al_host()
26
27         for _ in range(3):
28             # Leer asignaci n desde el archivo de estado del host
29             sftp_client.get(REMOTE_STATUS, "game_status_local.log")
30             game_id = None
31             with open("game_status_local.log") as f:
32                 for line in f:
```

```
                    data = json.loads(line.strip())
                    if data.get("game_stage") == ronda \
                       and data.get("Action") == "Assign" \
                       and data.get("PlayerID") == PLAYER_ID:
                        game_id = data.get("GameID")

            if game_id is not None:
                ejecutar_minijuego(game_id, ronda)
```

La implementación de las rondas fue una de las partes más desafiantes del proyecto, ya que implicó coordinar la comunicación con el host, la selección de minijuegos y el registro correcto de cada resultado. En las primeras pruebas surgieron problemas al leer asignaciones antiguas desde el archivo de estado, por lo que fue necesario filtrar los registros según la ronda y el `PlayerID`. Una vez resuelto esto, el flujo se volvió más estable y permitió integrar todos los minijuegos de forma ordenada. Como posible mejora, consideramos que el uso de sockets en lugar de archivos podría hacer la comunicación más directa, aunque para los objetivos del curso el enfoque basado en logs fue suficiente y fácil de depurar.

Actividad 1.4 — Ronda Final

La última etapa del sistema de juego corresponde a la **Ronda Final**, en la cual el jugador debe mantener un nivel de vida superior a cero durante un tiempo determinado. A diferencia de las rondas anteriores, aquí no se ejecutan minijuegos tradicionales, sino que el jugador se enfrenta a eventos que pueden aumentar o disminuir su vida, según lo indique el host mediante los registros del archivo de estado.

Esta ronda fue diseñada para generar un cierre dinámico y distinto a las rondas R1 y R2, con un flujo basado en lectura continua de eventos más que en ejecución de minijuegos.

El comportamiento general de la Ronda Final puede resumirse en los siguientes pasos:

1. El host inicia la ronda escribiendo un evento con la acción `"FinalStart"`.
2. El jugador establece su vida inicial en un valor determinado, por ejemplo `100`.
3. El jugador revisa periódicamente el archivo de estado en busca de instrucciones del host, como:
   - `"Life"`: disminuir vida en cierta cantidad.
   - `Recover"`: aumentar vida en cierta cantidad.
4. El jugador registra cada modificación de vida mediante `log_event()` y envía el log al host mediante SFTP.
5. Si la vida llega a cero, se registra derrota en el log y la ronda termina.
6. Si el jugador mantiene vida positiva hasta recibir la acción `"FinalEnd"`, se considera victoria.

Esta ronda se centra en la capacidad del sistema para coordinar eventos entre el host y el jugador, y entregar un desenlace dependiente del estado de vida gestionado durante el proceso.

Listing 7: Lógica principal del jugador durante la Ronda Final

```
1
2  def ronda_final():
3      vida = 100
4      log_event("Final", "Start", extra={"Life": vida})
5      enviar_log_al_host()
6
7      while True:
8          # Leer estado del host
9          sftp_client.get(REMOTE_STATUS, "game_status_local.log")
10
11         with open("game_status_local.log") as f:
12             for line in f:
13                 data = json.loads(line.strip())
14
15                 # Evento de da o
16                 if data.get("Action") == "Life" and data.get("
                       PlayerID") == PLAYER_ID:
17                     vida -= data.get("Amount", 0)
18                     log_event("Final", "Life", extra={"Life": vida})
19                     enviar_log_al_host()
20
21                 # Evento de curaci n
22                 if data.get("Action") == "Recover" and data.get("
                       PlayerID") == PLAYER_ID:
23                     vida += data.get("Amount", 0)
24                     log_event("Final", "Recover", extra={"Life":
                           vida})
25                     enviar_log_al_host()
26
27                 # Fin de la ronda final
28                 if data.get("Action") == "FinalEnd":
29                     resultado = "Win" if vida > 0 else "Lose"
30                     log_event("Final", "End", result=resultado)
31                     enviar_log_al_host()
32                     return
```

La implementación de la ronda final resultó diferente al resto del proyecto, ya que no involucra minijuegos directos, sino la lectura continua de instrucciones por parte del host y la actualización del estado de vida del jugador. Fue necesario verificar cuidadosamente que cada evento del host se interpretara correctamente y no generara inconsistencias en el nivel de vida. También observamos que enviar logs después de cada modificación ayudó a mantener el sistema sincronizado. En general, esta actividad permitió cerrar el proyecto con una dinámica distinta y basada en estados, complementando las rondas de minijuegos previas.

Actividad 2 — Pokédex Interactiva

La segunda parte del proyecto consistió en desarrollar una aplicación gráfica inspirada en una **Pokédex**, utilizando la librería `PyQt5`. El propósito de esta actividad fue complementar la primera parte del proyecto incorporando una interfaz visual, lectura de archivos externos y el despliegue estructurado de información mediante botones y elementos gráficos.

A diferencia de las rondas y minijuegos de la Actividad 1, esta sección se enfoca en programación orientada a interfaces gráficas, organización de datos mediante archivos `JSON` y manejo de imágenes en una ventana interactiva. Para ello, se utilizó una estructura modular donde cada Pokémon se encontraba definido por:

- Nombre
- Tipo
- Descripción
- Ruta a una imagen asociada

Estos datos se almacenaron en un archivo `pokedex.json`, desde el cual la aplicación carga la información al iniciar. Mediante botones "siguiente" y "anterior", el usuario puede navegar entre las distintas entradas y ver su información en tiempo real.

Imagenes de ejemplo(Pokédex)

Circuito utilizado en la Pokédex

Para complementar la experiencia visual de la Pokédex, se implementó un pequeño circuito que utiliza un **LED RGB** conectado directamente a la Raspberry Pi. Este LED permite indicar visualmente ciertos estados de la aplicación (por ejemplo, cambios de Pokémon o acciones específicas dentro de la interfaz). El circuito fue armado sobre un módulo LED RGB que integra las resistencias necesarias.

En la Figura 1 se presenta el esquema general de la conexión, donde se aprecian claramente los pines correspondientes a los tres colores del LED, así como la conexión a tierra (GND).

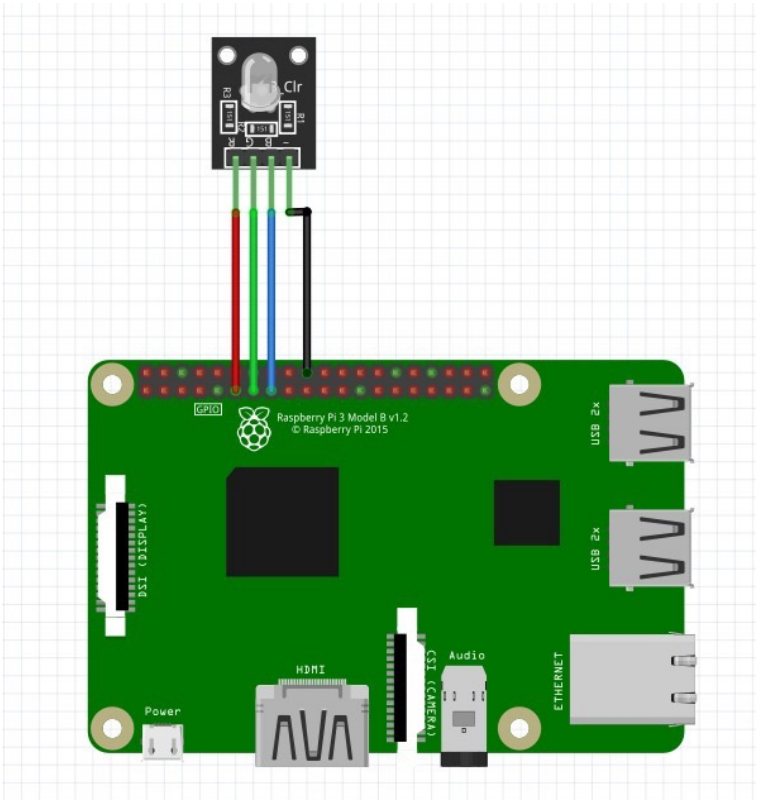

Figura 1: Conexión del LED RGB utilizado en la Pokédex.

Color	**Pin del módulo**	**GPIO Raspberry Pi**	**Descripción**
Rojo (R)	R	GPIO 17	Canal rojo del LED RGB
Verde (G)	G	GPIO 27	Canal verde del LED RGB
Azul (B)	B	GPIO 22	Canal azul del LED RGB
GND	–	Pin 6	Tierra (referencia común)

Cuadro 1: Asignación de pines del LED RGB utilizado en la Pokédex.

El circuito utilizado para la Pokédex fue sencillo de implementar gracias a que el módulo LED RGB incluye las resistencias necesarias y se conecta directamente a los pines GPIO de la Raspberry Pi. Este componente permitió añadir retroalimentación visual a la aplicación, haciendo la experiencia más dinámica. El armado fue rápido y sin dificultades, lo cual nos permitió centrarnos en integrar correctamente las salidas desde PyQt5 hacia el control del LED.

Circuito para minijuegos

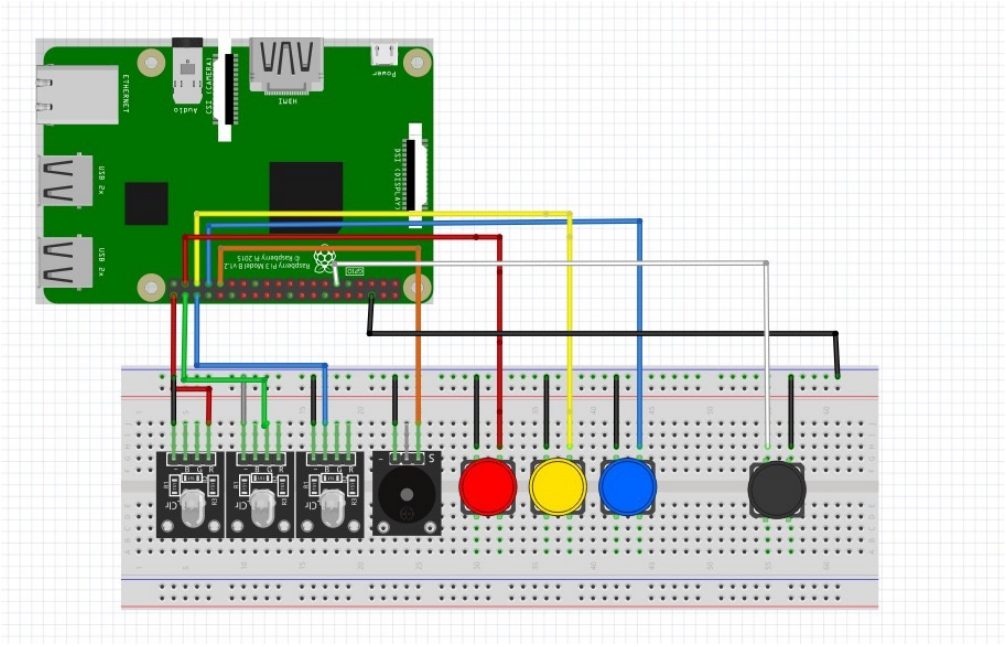

Figura 2: Esquema del circuito empleado en los minijuegos con sensores.

Componente	GPIO	Descripción
LED Rojo	GPIO 21	Indicador de color rojo en “Simón dice”
LED Amarillo	GPIO 20	Indicador de color amarillo
LED Azul	GPIO 16	Indicador de color azul
Botón Rojo	GPIO 26	Entrada del jugador para color rojo
Botón Amarillo	GPIO 19	Entrada del jugador para color amarillo
Botón Azul	GPIO 13	Entrada del jugador para color azul
Buzzer	GPIO 6	Emisión de sonidos (aciertos, errores, etc.)
Botón Negro (10 s)	GPIO 17	Botón usado en el minijuego de los 10 segundos

Cuadro 2: Asignación de pines GPIO en los minijuegos con sensores.

Listing 8: Funciones principales de la Pokédex desarrollada en PyQt5

```
1
2  class Pokedex(QWidget):
3      def __init__(self):
4          super().__init__()
5          self.index = 0
6          self.data = self.cargar_pokedex("pokedex.json")
7
8          self.lbl_img = QLabel(self)
9          self.lbl_name = QLabel(self)
10         self.lbl_type = QLabel(self)
11         self.lbl_desc = QLabel(self)
12
13         btn_prev = QPushButton("Anterior")
14         btn_next = QPushButton("Siguiente")
15         btn_prev.clicked.connect(self.prev_pokemon)
16         btn_next.clicked.connect(self.next_pokemon)
17
18         layout = QVBoxLayout()
19         layout.addWidget(self.lbl_img)
20         layout.addWidget(self.lbl_name)
21         layout.addWidget(self.lbl_type)
22         layout.addWidget(self.lbl_desc)
23         layout.addWidget(btn_prev)
24         layout.addWidget(btn_next)
25         self.setLayout(layout)
26
27         self.mostrar_pokemon()
28
29     def cargar_pokedex(self, archivo):
30         with open(archivo, "r") as f:
31             return json.load(f)
32
33     def mostrar_pokemon(self):
34         p = self.data[self.index]
35         self.lbl_name.setText(f"Nombre: {p['name']}")
36         self.lbl_type.setText(f"Tipo: {p['type']}")
37         self.lbl_desc.setText(p["description"])
38         pixmap = QPixmap(p["image"])
39         self.lbl_img.setPixmap(pixmap.scaled(250, 250))
40
41     def next_pokemon(self):
42         self.index = (self.index + 1) % len(self.data)
43         self.mostrar_pokemon()
44
45     def prev_pokemon(self):
46         self.index = (self.index - 1) % len(self.data)
47         self.mostrar_pokemon()
```

La Pokédex fue una actividad muy distinta a las rondas y minijuegos de la primera parte del proyecto. Trabajar con `PyQt5` nos permitió reforzar conceptos de interfaces gráficas, lectura de archivos `JSON` y manipulación de imágenes dentro de una ventana interactiva. Las funciones de navegación entre Pokémon resultaron sencillas de implementar y el uso de layouts facilitó organizar los elementos en pantalla. En general, esta actividad fue intuitiva y entretenida, y nos entregó un acercamiento práctico al diseño de GUI en Python.