UNIVERSIDAD DE CONCEPCIÓN

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

549305-1 – TALLER DE APLICACIÓN TIC 2

Profesor Vincenzo Caro

Miniproyecto 1

Catalina Ibaca

Christian Silva

Concepción, 11 de mayo de 2025

Tabla de Contenidos

Resumen: ¿Qué es un Arduino?

El Arduino es una placa de circuito (hardware) que puede programarse para realizar tareas automatizadas gracias al Entorno de Desarrollo Integrado del mismo nombre (software). Se podría decir entonces que el Arduino es el conjunto del hardware en forma de un mini circuito conocido como microcontrolador

Ítem 1.1 Actividad 1: Guitar Hero (on Arduino?)

Esta primera actividad consiste en implementar una versión básica del juego “Guitar Hero” utilizando una placa Arduino. Para ello, se desarrollará un piano digital que permita reproducir cinco notas musicales utilizando un buzzer, cinco botones y cinco LEDs RGB. La finalidad es que cada botón represente una tecla de la guitarra de Guitar Hero, los cuales se deberán presionar una vez se ilumine su respectivo LED, reproduciendo una nota musical específica.

Ítem 1.1

Para implementar esta primera fase del juego, consideren los siguientes módulos de su kit de 45 Sensores:

- *5 módulos LEDs (KY-009, KY-011, KY-016 o KY-029).*
- *5 botones (KY-004, KY-023, KY-040 u otros).*
- *1 buzzer pasivo (KY-006) o activo (KY-012).*

Con estos elementos, preparen un circuito en el que cada botón esté vinculado a un LED RGB y a una nota musical distinta. A partir de esta base, desarrollen un programa en Arduino que simule un piano de 5 teclas, considerando los siguientes aspectos:

1. *Cada vez que se presione uno de los botones, el sistema deberá:*

    - *Encender el LED RGB correspondiente con un color específico.*
    - *Reproducir una nota musical predefinida con una duración determinada a través del buzzer.*

- *Mostrar en el monitor serial un mensaje indicando qué botón fue presionado y qué nota fue reproducida.*

2. *Además, el código deberá implementar dos comandos especiales que se envían desde el monitor serial para añadir funcionalidad al sistema:*

    - *Comando Vxxx: Modifica el volumen de salida, siendo xxx un número entre 000 y 255 que representa el valor de PWM aplicado al buzzer.*
    - *Comando Tx: Cambia el banco de tonos utilizados. El índice actual del banco puede ser aumentado (T1) o disminuido (T0), permitiendo al usuario cambiar el set de 5 notas disponibles durante la ejecución del programa. Para mayor simplicidad al momento de trabajar con las notas, tomen como referencia el Anexo "**Reproducción de Notas**".*

Ítem 1.2

A partir del código base desarrollado en el ítem anterior, implementen ahora una versión básica del juego Guitar Hero, donde una fila de LEDs representa las notas musicales que el jugador deberá reproducir secuencialmente presionando los botones correspondientes.

Para ello, deberán considerar los siguientes aspectos en el desarrollo del circuito y del código en Arduino:

1. *Se debe crear un menú de selección inicial, visible en el monitor serial, donde el jugador pueda elegir:*

    - *La dificultad del juego: la dificultad modificará la velocidad a la que se reproducen las notas (es decir, el tiempo disponible para presionar el botón correcto).*
    - *Una canción predefinida: se deben definir al menos tres melodías diferentes. Estas pueden ser creadas o buscadas en internet (por ejemplo: Mario Bros, Harry Potter, Star Wars, entre otras).*

2. *Las notas de la canción seleccionada deberán almacenarse en una estructura secuencial (por ejemplo, una matriz) y reproducirse una a una, encendiendo el LED correspondiente a cada nota en el orden de la melodía. La velocidad de reproducción dependerá del nivel de dificultad seleccionado.*

3. *El jugador deberá presionar el botón correspondiente al LED que se encuentra iluminado en ese momento. Si acierta, el juego:*

    - *Suma una cantidad de puntos base.*
    - *Aumenta una racha de aciertos consecutivos. Las rachas multiplican los puntos base, en los siguientes niveles:*
        - *2x: al alcanzar 5 aciertos consecutivos.*

    - *4x: al alcanzar 10 aciertos consecutivos.*
    - *8x: al alcanzar 15 aciertos consecutivos.*
    - *16x: al alcanzar 20 aciertos consecutivos.*
  - *Reinicia la racha si el jugador comete un error.*

4. *Si el jugador presiona una tecla incorrecta, el juego:*

   - *Resta puntos del total.*
   - *Reinicia la racha.*
   - *Si el puntaje llega a 0 puntos, el juego termina automáticamente y se reproduce un tono de "Game Over" en el buzzer.*

5. *Una vez todas las notas de la canción han sido reproducidas, el juego finaliza y se deben contemplar las siguientes situaciones:*

   - *Si el jugador completó correctamente todas las notas, el código reproducirá una melodía de victoria y mostrará el puntaje total en el monitor serial.*
   - *Si el jugador no completó todas las notas, pero su puntaje no llegó a cero, se muestra únicamente el puntaje obtenido, sin reproducir ninguna melodía.*
   - *En ambos casos, se puede ofrecer reiniciar el juego o finalizar.*

Ítem 1.2 Desarrollo: Actividad 1

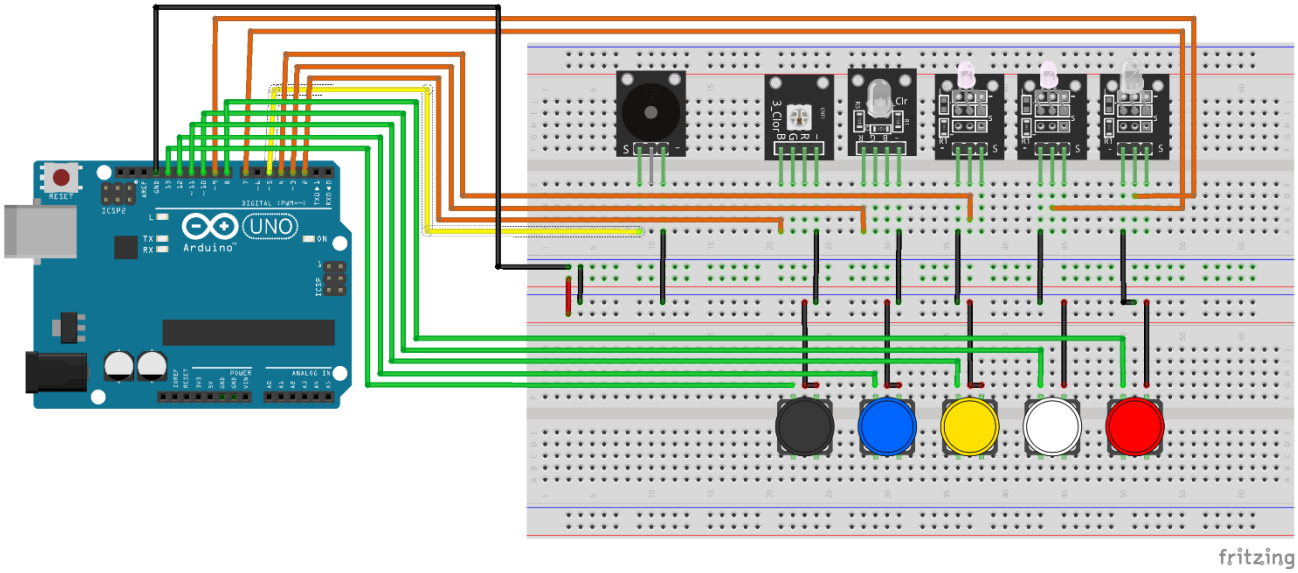
1.2.a) Parte 1: Identificación de Componentes

Comenzamos este proyecto identificando los components a utilizar:

- 1 buzzer pasivo (KY-006)
- 5 botones
- 5 LEDs (1x KY-009, 1x KY-016, 2x KY-011, 1x KY-029)

1.2.b) Parte 2: Esquemático de Circuito

Conectamos dos protoboards y realizamos la conexión de los componentes previamente mencionados de acuerdo al siguiente esquema:



Notemos que los cables negros son conexiones a tierra, los cables verdes son conexiones a botones, los cables naranja son conexiones a los LEDs y el cable amarillo es la conexión al buzzer pasivo.

1.1.1 Parte 3: Código Básico

Recurrimos a la herramienta virtual ChatGPT para obtener el código básico requerido por el enunciado y para tener una idea clara de la estructura del código en Arduino.

El código base obtenido es el siguiente:

```
const int botones[] = {7, 8, 9, 10, 11};  // Pines de los botones
const int leds[] = {2, 3, 4, 5, 6};       // Pines de los LEDs
const int buzzer = 12;                    // Pin del buzzer

void setup() {
    for (int i = 0; i < 5; i++) {
        pinMode(botones[i], INPUT_PULLUP); // Activa resistencias
internas pull-up
        pinMode(leds[i], OUTPUT);
    }
    pinMode(buzzer, OUTPUT);
}

void loop() {
    for (int i = 0; i < 5; i++) {
        if (digitalRead(botones[i]) == LOW) {  // Botón presionado
            digitalWrite(leds[i], HIGH);  // Enciende LED
            tone(buzzer, 200 + (i * 100)); // Suena una nota diferente
        } else {
            digitalWrite(leds[i], LOW);  // Apaga LED
        }
    }
     }
```

El código anterior crea dos arreglos con la misma dimensión (1x5) cuyos elementos son el número del pin que utilizan los botones (para el arreglo `const int botones[]`) y los LEDs (para el arreglo). El motivo por el cual el código guarda estos pines y de por qué los guarda en orden según su ubicación en el protoboard (de izquierda a derecha) es para que al botón "i" le corresponda el LED "i", simplificando el código.

NOTA: El código anterior utiliza el número 12 como el pin al cual el buzzer pasivo está conectado, pero más adelante se cambiará el pin del buzzer del 12 al 5 y por ende se cambiará el pin del segundo botón del 5 al 12. La razón detrás de este cambio es que el buzzer debe estar conectado en el pin 5 ó el pin 6 para poder controlar el volumen de dicho buzzer con el uso de una biblioteca externa llamada "Arduino Volume Control".

`void setup()` se encarga de configurar los pines de los botones y LEDs, utilizando un ciclo for que recorre en orden los índices de los arreglos para activar las resistencias internas pull-up (para evitar el uso de resistencias externas) para los botones, además de configurar el buzzer y los LEDs como salidas (outputs).

`void loop()` utiliza un ciclo for con la misma lógica anterior para monitorear el estado de los botones. Cuando el botón “i” se presione, el LED “i” se prende y el buzzer reproduce la frecuencia 200 + (i * 100).

1.2.c) Parte 4: Modificación del Código

El código anterior lo modificamos para corregir los números de pin a los cuales están conectados los componentes. El código modificado es el siguiente:

```
const int botones[] = {13, 12, 11, 10, 8};  // Pines de los botones
const int leds[] = {2, 3, 4, 5, 6};          // Pines de los LEDs
const int buzzer = 9;                        // Pin del buzzer
const int notas[] = {261, 294, 329, 392, 440}; // Frecuencias en Hz

void setup() {
    for (int i = 0; i < 5; i++) {
        pinMode(botones[i], INPUT_PULLUP); // Activa resistencias
internas pull-up
        pinMode(leds[i], OUTPUT);
    }
    pinMode(buzzer, OUTPUT);
}

void loop() {
    bool botonPresionado = false;

    for (int i = 0; i < 5; i++) {
        if (digitalRead(botones[i]) == LOW) {
            botonPresionado = true;
            digitalWrite(leds[i], HIGH);
            tone(buzzer, notas[i]);           // Reproduce la nota
correspondiente
            delay(500);                       // Espera 0.5 segundos
            noTone(buzzer);                   // Detiene el sonido
            digitalWrite(leds[i], LOW);       // Apaga el LED

            // Espera a que se suelte el botón antes de continuar
            while (digitalRead(botones[i]) == LOW);
            delay(50); // Pequeño retardo para evitar rebotes
            break; // Evita detectar otros botones durante este ciclo
        }
    }

    if (!botonPresionado) {
        noTone(buzzer); // Asegura que el buzzer esté apagado si no hay
pulsaciones
```

```
    }
  }
```

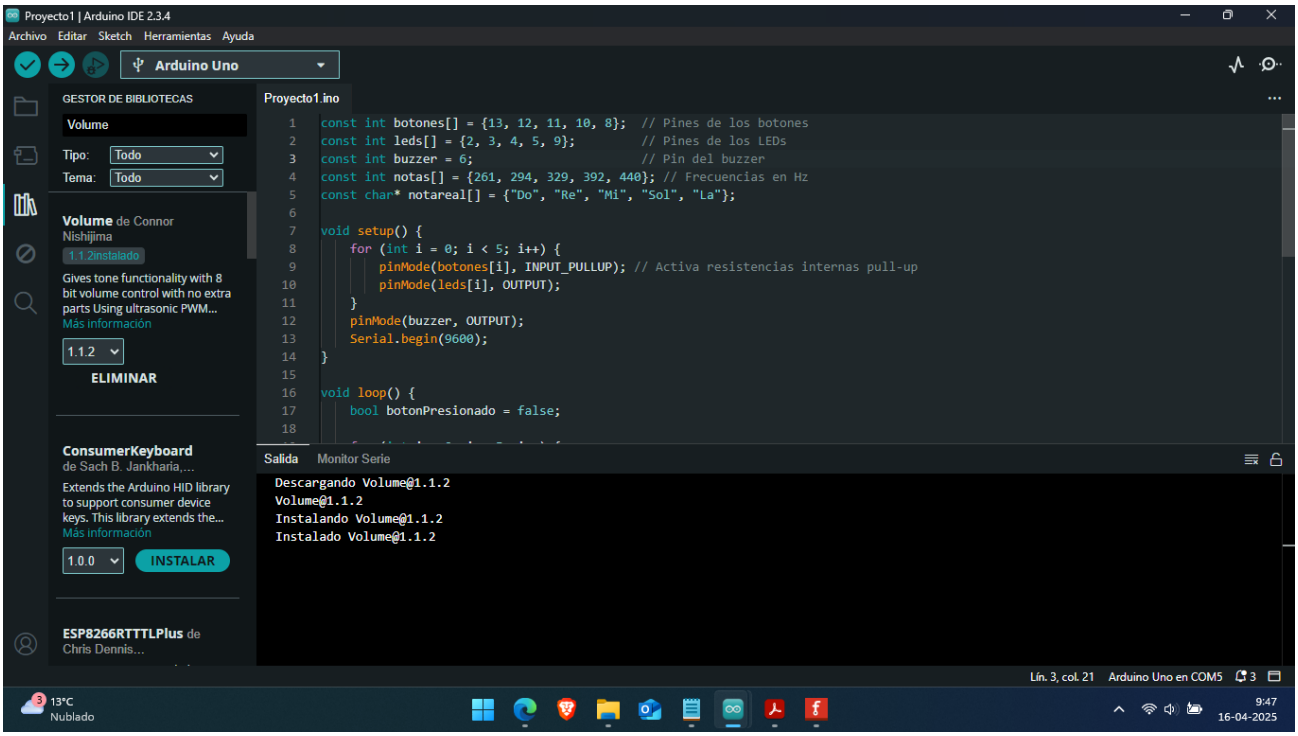
El código anterior creó el arreglo `notas[]` para definir las frecuencias que el buzzer reproducirá al presionar los botones.

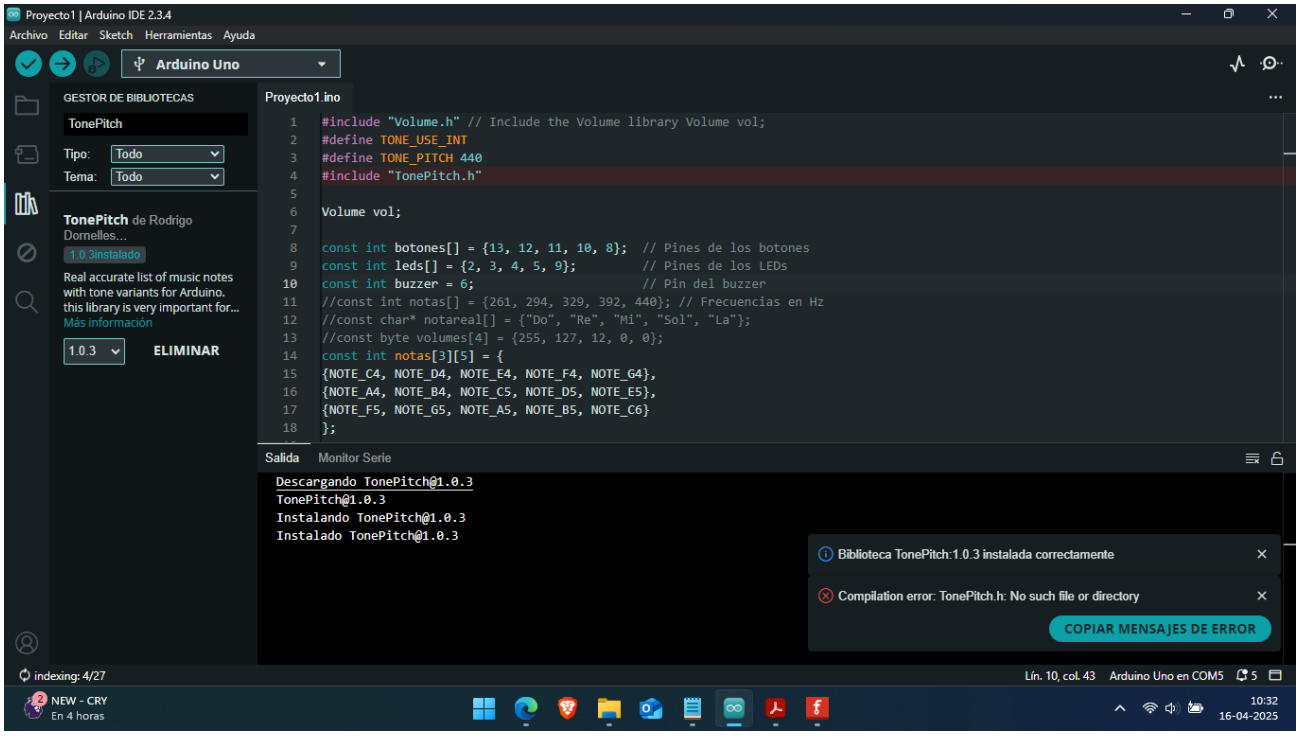
El `bool botonPresionado` corresponde a una variable que toma uno de dos valores: True o False. Es utilizado en conjunto con `tone(buzzer, notas[i])` y `noTone(buzzer)` para que al presionar el botón "i" el buzzer reproduzca la nota [i] dada por la frecuencia [i] por un intervalo de tiempo de 0.5 segundos (especificado por `delay(500)`).

El código además utiliza un ciclo While para que mientras que el botón se haya presionado, exista un retardo de 0,05 segundos para evitar que varios botones se presionen a la vez.

1.1.2 Parte 5: Instalación de Bibliotecas

Procedimos a instalar las bibliotecas Volume (de Connor Nishijima) y TonePitch (de Rodrigo Dornelles) para utilizarlas según lo indicado en el enunciado:





1.2.d) Parte 6: Código Final (Ítem 1.1)

El código definitivo que utilizaremos para completar el Ítem 1.1 es el siguiente:

```
#include "Volume.h"
#define TONE_USE_INT
#define TONE_PITCH 440
#include <TonePitch.h>

Volume vol;

const int botones[] = {13, 12, 11, 10, 8};  // Pines de botones
const int leds[] = {2, 3, 4, 7, 9};         // Cambié el LED que estaba
en el pin 5 al pin 7
const char* notareal[] = {"Do", "Re", "Mi", "Sol", "La"};

const int notasMatrix[3][5] = {
  {NOTE_C4, NOTE_D4, NOTE_E4, NOTE_F4, NOTE_G4},
  {NOTE_A4, NOTE_B4, NOTE_C5, NOTE_D5, NOTE_E5},
  {NOTE_F5, NOTE_G5, NOTE_A5, NOTE_B5, NOTE_C6}
};

int x = 0;        // Fila actual
int y = 255;      // Volumen

void setup() {
  vol.begin(); // ¡Sin parámetros!
```

```
  for (int i = 0; i < 5; i++) {
    pinMode(botones[i], INPUT_PULLUP);
    pinMode(leds[i], OUTPUT);
  }
  Serial.begin(9600);
  Serial.println("Sistema iniciado. Esperando comandos...");
}

void loop() {
  procesarBotones();
  procesarComando();
}

void procesarBotones() {
  for (int i = 0; i < 5; i++) {
    if (digitalRead(botones[i]) == LOW) {
      for (int j = 0; j < 5; j++) digitalWrite(leds[j], LOW);
      digitalWrite(leds[i], HIGH);

      vol.tone(notasMatrix[x][i], (byte)y);
      Serial.print("Volumen Seteado: ");
      Serial.print(y);
      Serial.print(" Botón Presionado: ");
      Serial.print(i + 1);
      Serial.print(". Nota: ");
      Serial.println(notareal[i]);

      vol.delay(300);
      vol.noTone();
      digitalWrite(leds[i], LOW);

      while (digitalRead(botones[i]) == LOW);
        delay(100);
        break;
    }
  }
}

void procesarComando() {
  if (Serial.available()) {
    String a = Serial.readStringUntil('\n');
    a.trim();

    if (a.charAt(0) == 'T') {
      int nuevoX = a.substring(1).toInt();
      if (nuevoX >= 0 && nuevoX < 3) {
        x = nuevoX;
        Serial.print("Fila de notas cambiada a: ");
        Serial.println(x);
      } else {
        Serial.println("Error: Fila fuera de rango (0-2)");
      }
    }
    else if (a.charAt(0) == 'V') {
      int nuevoY = a.substring(1).toInt();
      if (nuevoY >= 0 && nuevoY <= 255) {
        y = nuevoY;
        Serial.print("Volumen actualizado a: ");
```

```
        Serial.println(y);
      } else {
        Serial.println("Error: Volumen fuera de rango (0-255)");
      }
    }
    else {
      Serial.println("Comando inválido. Use T0-T2 o V000-V255");
    }
  }
}
```

El código funciona de la siguiente manera:

Se crea la matriz `const int notasMatrix[3][5]` que contiene los 3 grupos de 5 notas cada una. El `void setup()` incluye `vol.begin();` para utilizar la biblioteca "Volume" (de Connor Nishijima).

El `void loop()` contiene el llamado a dos funciones: `procesarBotones()` y `procesarComando()`. La función `procesarBotones()` responde al input que se obtiene al presionar el botón "i" (con "i" de 0 a 4 en el código, correspondiente al índice de los botones, LEDs y notas, todos de izquierda a derecha). La función, al presionarse el botón "i" activa el LED "i" y reproduce la nota correspondiente al elemento de la fila "x" columna "i" de la matriz `notasMatrix` (la variable "x" es por defecto 0, y en el siguiente párrafo veremos cómo cambiar el valor de "x").

La función `procesarComando()` se encarga de responder a los comandos "Tx" y "Vyyy" entregados mediante la consola serial. Tiene dos "if" para verificar que el comando ingresado empieze por "T" o por "V". Los comandos ingresados son:

- El comando "Tx" con "x" un número natural entre 0 y 2, cambia la fila de la matriz `notasMatrix` que usa el código para reproducir las notas (cambia la fila actual por la fila "x"), cambiando un grupo de 5 notas por otras 5 notas diferentes (utilizables gracias a la biblioteca "TonePitch" de Rodrigo Dornelles). Si escribimos el comando "Tx" fuera del rango 0 a 2, el código lo reconoce como un error, imprimiendo "`Error: Fila fuera de rango (0-2)`" y permitiendo introducir nuevamente otro comando.
- El comando "Vyyy" con "yyy" un número natural de tres cifras entre 000 y 255, modifica el volumen del buzzer (posible gracias a la biblioteca "Volume" de Connor

Nishijima). El volumen corresponde a los valores de “yyy” que representan la intensidad del PWM (es decir, la cantidad de enegría) entregado al buzzer. Los valores de “yyy” van de “000” (volumen mínimo, 0% de amplitud) a “255” (volumen máximo, 100% de amplitud), ya que Arduino usa PWM de 8 bits (256 niveles, del 0 al 255).

El código procesa los comandos ingresados utilizando `Serial.readStringUntil('\n')` para definir un string “a” que corresponde a todos los caracteres ingresados hasta que se presione “ENTER”. Luego, utiliza `a.trim()` para borrar “espacios” ingresados con “SPACEBAR”. También utiliza varios if (else if) cuyos argumentos utilizan `a.charAt(0)` para reconocer el primer caracter del string (para que el comando empieze por “T” o por “V”). Una vez los argumentos de algún if son correctos, el código crea un integer utilizando `a.substring(1).toInt()` para que ese integer corresponda al número ingresado después del primer carácter (“T” o “V”).

Una vez recibe el comando correcto, procede dependiendo del tipo de comando recibido. Si recibe “Tx”, redefine la variable integer “x” con el valor del número del comando ingresado (0, 1 o 2) para luego usar `vol.tone(notasMatrix[x][i], (byte)y)` donde `notasMatrix[x][i]` es la nota de la matriz `notasMatrix`, fila “x” columna “i” (“i” siendo el índice del botón presionado). Si recibe “Vyyy”, redefine la variable integer “y” con el valor del número del comando ingresado (número entre 0 y 255) para luego usar `vol.tone(notasMatrix[x][i], (byte)y)` donde `(byte)y` es el valor de la energía PWM que Arduino le entrega al buzzer, cambiando su volumen.

1.2.e) Parte 7: Link de “GitHub”

El link de GitHub es: https://github.com/ChrisX4Ever/TIC2_CICS/

1.2.f) Parte 8: Ítem 1.2

Intentamos modificar manualmente el código correspondiente a la actividad 1.1 sin utilizar ChatGPT, pero no obtuvimos resultados positivos, ya que la consola arrojaba múltiples errores.

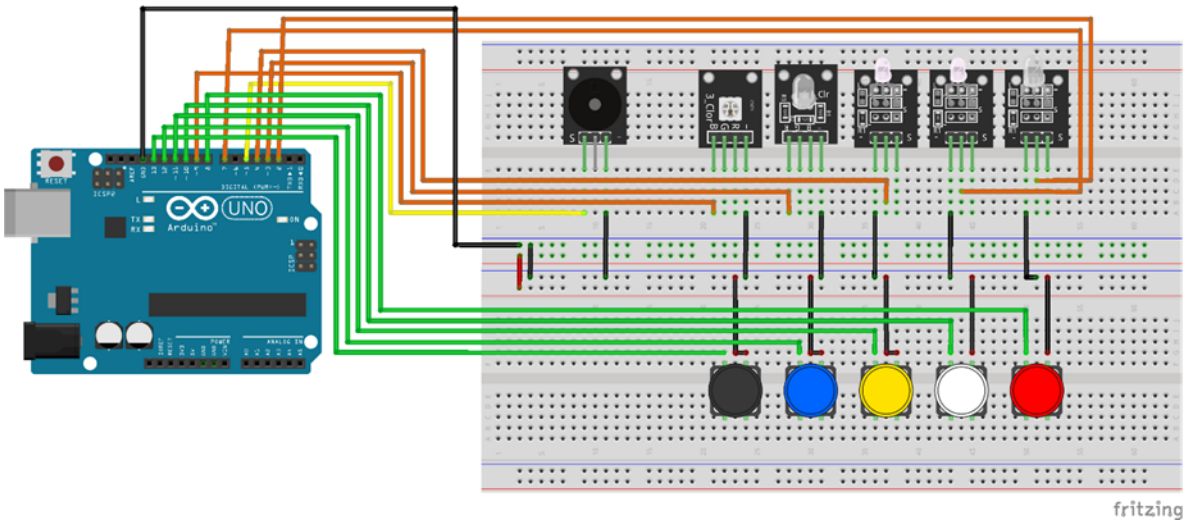
Posteriormente, optamos por recurrir a ChatGPT para obtener un código funcional, detallando los componentes utilizados, los pines a los que están conectados y el uso de la biblioteca "TonePitch" desarrollada por Rodrigo Dornelles.

Si bien los componentes siguen siendo los mismos que en la actividad 1.1, debido al progreso alcanzado en la actividad 2, al retomar la primera actividad, hubo algunos cambios en las conexiones del hardware.

Los cambios fueron minúsculos; como cambiar la conexión del LED 5 (KY-029) del pin 9 al pin 2 y cambiar la conexión del LED 1 (KY-009) del pin 2 al pin 9.

La primera versión del código que ChatGPT nos entregó devolvía los errores:

```
#error please choose your tune setting.
 #error please choose your tune setting.
  ^~~~~
c:\Users\Chris\Documents\Arduino\libraries\TonePitch\src/TonePitch.
h:120:2: error: #error invalid TONE_PITCH const macro value.
 #error invalid TONE_PITCH const macro value.
  ^~~~~
In file included from
c:\Users\Chris\Documents\Arduino\libraries\TonePitch\src/TonePitch.h:123:
0,
                 from D:\Program
Files\TIC2_CICS\Proyecto1Item1.2\Proyecto1Item1.2\Proyecto1Item1.2.ino:1:
c:\Users\Chris\Documents\Arduino\libraries\TonePitch\src/ToneVerify
.h:20:2: error: #error please choose your tone header.
 #error please choose your tone header.
```

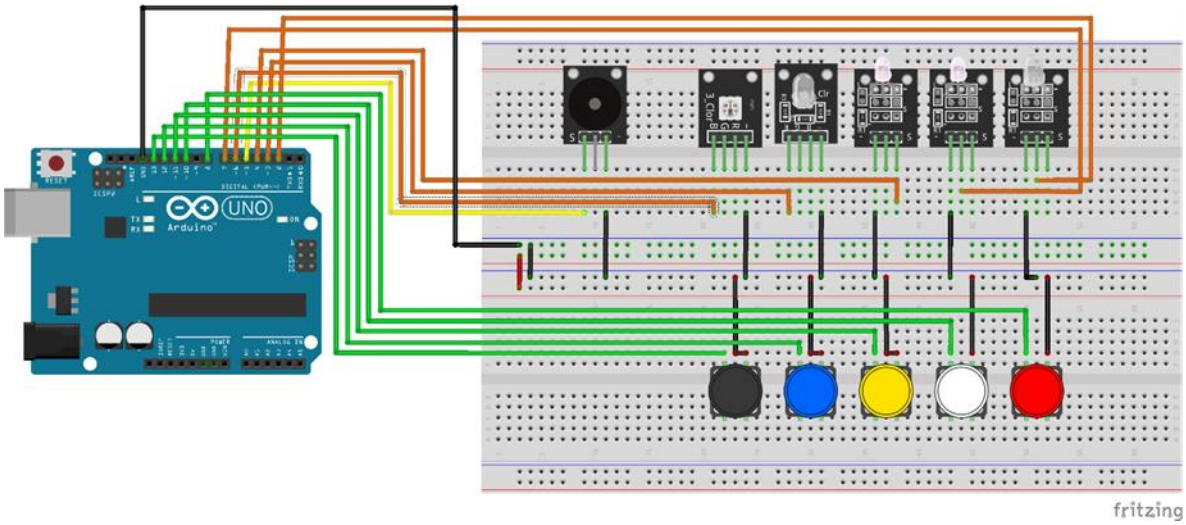


Por lo cual consultamos a ChatGPT por el motivo. La respuesta es que la biblioteca TonePitch necesita que se especifique un valor de "TONE_PITCH" (de referencia) antes de incluirla.

Después de modificar el código con ayuda de ChatGPT, logramos hacer que el código se compile correctamente. El juego permite que elijamos la dificultad (aunque sólo una vez al ejecutar el código por primera vez; después de elegir la canción el ganar o perder permitirá reiniciar y sólo nos dejará elegir la canción, no la dificultad), además de mostrar si ganamos o perdimos junto con el puntaje y la pregunta de que si queremos seguir jugando.

A pesar de que el código funcionaba, los LEDs 1 y 3 no prendían, por lo cual decidimos pedirle a ChatGPT un código de prueba de Arduino para que los LEDs se prendan al presionar sus respectivos botones. Después de prueba y error, nos dimos cuenta de que el LED 1 no estaba conectado al pin 9 sino que al pin 6; además, el LED 3 prendía al cambiar la conexión a otra “patita” (cambiamos de la “patita” 2 a la 3).

El cambio anterior resultó en el siguiente cambio en el esquemático:



Los únicos cambios fueron la correcta conexión del LED 1 (6, no 9) y la conexión del LED 3 (de la “patita” 2 a la 3).

El juego funcionaba bien, pero nos dimos cuenta de que al errar 3 notas consecutivas el juego no terminaba automáticamente, por lo que el código fue modificado (Después de la modificación nos dimos cuenta de que la regla de 3 errores consecutivos no existe en el enunciado, pero decidimos incluirla como bonus).

El código final funciona de la siguiente manera:

`#define TONE_PITCH 440` se usa para definir la frecuencia 440 como base para la biblioteca `TonePitch.h`.

`ledPins` y `buttonPins` son arreglos cuyos elementos son números que representan los pines de cada componente de izquierda a derecha. notes Es un arreglo cuyos elementos son las notas que el buzzer reproducirá al presionar los botones respectivos.

Las variables como velocidad, puntuacion, etc. son autoexplicativos y representan los valores por defecto del juego al iniciar o reiniciar una ronda.

`marioBros`, `starWars` y `harryPotter` son arreglos cuyos elementos representan los índices del arreglo notes para que el buzzer reproduzca las notas en orden, asimilando las respectivas canciones.

Al igual que el ítem anterior, `void setup()` se encarga de activar las resistencias internas `INPUT_PULLUP` para cada botón y de configurar los LEDs.

`void menuSeleccion()` es donde se define el comportamiento del menú de dificultad. La lógica que sigue la elección de dificultad (1 para fácil, 2 para medio y 3 para difícil) es usando el valor numérico seleccionado para insertar en la variable velocidad, la cual modifica la velocidad de reacción del videojuego.

`while (seleccion < 1 || seleccion > 3)` guarda la selección de la canción para poder ejecutar el juego en sí.

`iniciarJuego()` es donde se define cómo ocurren los aciertos o errores y qué otras funciones el juego debe ejecutar para procesarlas. Utiliza `startTime = millis()` para realizar el conteo de tiempo.

`procesarAcierto()` y `procesarError()` son autoexplicativas, son funciones que determinan las reglas del juego como las condiciones de término del juego, las rachas, etc.

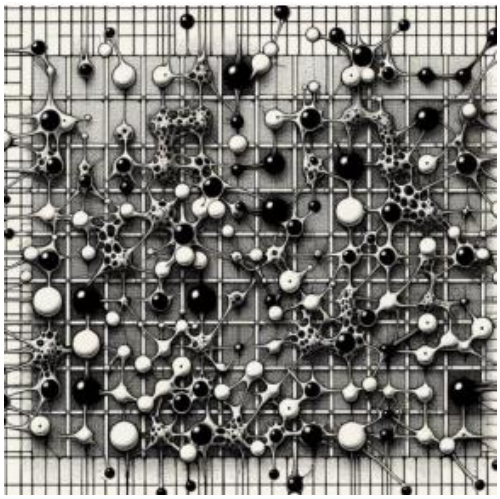
`finalizarJuego()` es donde aparece la pantalla final, mostrando el puntaje, la razón del término del juego y la posibilidad de que el jugador decida si volver a jugar o no.

Resultados y Comentarios

Ambas actividades fueron fundamentales para comprender el funcionamiento de botones, LEDs RGB, buzzer y comandos por monitor serial. A pesar de las dificultades iniciales en la actividad 1.1, logramos superarlas y entender la lógica del sistema. La actividad 1.2 fue más simple de lo esperado

y muy entretenida, permitiéndonos afianzar lo aprendido y prepararnos para desafíos más complejos.

Ítem 1.3 Actividad 2: The Game of life



The Game of Life fue creado por Conway como un modelo matemático de un sistema celular con el fin de explorar la idea de la vida y la muerte en un entorno abstracto. Conway buscaba un sistema con reglas simples, pero que pudiera producir comportamientos complejos y variados.
Las reglas originales del juego son simples y se aplican a una cuadrícula bidimensional de celdas, cada una de las cuales puede estar viva (1) o muerta (0). Dependiendo de la ubicación de cada célula y sus respectivos vecinos, se pueden dar alguno de los siguientes casos:

- *Sobrepoblación: Si una célula viva tiene más de tres vecinos vivos, muere debido a la falta de recursos.*
- *Subpoblación: Si una célula viva tiene menos de dos vecinos vivos, muere por soledad.*
- *Estabilidad: Una célula viva con dos o tres vecinos vivos permanece viva en la siguiente generación.*

- *Reproducción: Si una célula muerta tiene exactamente tres vecinos vivos, nace en la siguiente generación debido a la reproducción.*

Ítem 2.1

La idea detrás de esta actividad es que sean capaces de programar y alterar externamente una versión en Python de The Game of Life, para lo cual pueden considerar el código base mostrado en el Anexo "Código Base de The Game of Life", donde se implementa el juego para una cuadrícula de 100x100 con sus reglas originales.

De manera inicial, adapten dicho código para implementar el juego en una interfaz gráfica creada con la biblioteca PyQt6, apoyándose de la biblioteca matplotlib para construir la cuadrícula y de QTimers para actualizar el juego en tiempo real, con una tasa de actualización variable a definir por ustedes. Luego, utilizando como intermediario la comunicación por protocolo serial, integren un programa de Arduino que sea capaz de interactuar con su juego de la siguiente forma:

- *Cada un intervalo de 10 segundos, el programa de Python debe enviar hacia Arduino la cantidad de células vivas que se encuentran actualmente en la cuadrícula. Luego, definan en Arduino 3 intervalos para evaluar las condiciones de estabilidad, subpoblación y sobrepoblación de las células; vinculando cada estado con un color particular de un módulo LED.*
- *Desde Arduino, implementen un botón pulsador que sea capaz de reiniciar desde 0 el juego en Python. Para ello, cada vez que se presione el botón, envíen un carácter o un mensaje hacia Python para que la interfaz reaccione a esta acción.*

Ítem 2.2

En esta segunda parte de la actividad, deberán adaptar su juego del ítem anterior bajo el contexto de una invasión zombie, siguiendo nuevas reglas de comportamiento. Cada celda de la cuadrícula representará un individuo en el sistema, y puede encontrarse en uno de los siguientes tres estados:

0: zombie 1: persona viva 2: muerto

Adicionalmente, cada célula debe poseer un nivel de vida, con un valor máximo de 200 puntos.

Condiciones y reglas del juego Durante cada iteración del juego, los individuos evolucionan siguiendo las siguientes reglas:

• Una persona viva pierde 5 puntos de vida por cada zombie en su vecindad inmediata. Si su vida llega a 0, se transforma en zombie con 100 puntos de vida.

• Un zombie pierde 10 puntos de vida si está rodeado por menos de 2 personas viva y 30 puntos de vida si está rodeado por más de 3 personas vivas. Si su vida llega a 0, hay un 10% de probabilidad de que el zombie se cure y se transforme en persona con 100 puntos de vida. En caso contrario, se transforma en muerto.

• Una muerto puede revivir en los siguientes casos:

- *Si tiene 2 o más vecinos personas vivas, estas se reproducen y el muerto se reemplaza por una persona viva con 100 puntos de vida.*
- *Si tiene 2 o más vecinos zombies, revive como zombie con 100 puntos de vida.*
- *En caso de cumplir ambas condiciones, prevalece la conversión a zombie.*

Eventos especiales

El juego debe incluir cuatro tipos de eventos especiales, que pueden alterar el estado de la cuadrícula en zonas específicas:

1. ***Infección masiva (cuadrada):** transforma un bloque 5x5 lleno de personas vivas en zombies con 200 puntos de vida.*
2. ***Infección masiva mutada (forma pulsar):** transforma una figura pulsar de 13x13 (ver Anexo "Generación de pulsar") en el mapa en zombies con 200 puntos de vida.*
3. ***Ritual de purificación:** dentro de un área de 15x15, los zombies tienen un 80% de probabilidad de curarse y volver a ser personas vivas con 200 puntos de vida.*
4. ***I am atomic! (área)**: destruye completamente un área de 25x25, cambiando todas las células de la zona al estado muerto. Estos eventos especiales deben estar vinculados a botones dentro de la interfaz desarrollada con PyQt6. Cada botón debe permitir activar un único evento especial, y al hacerlo, debe actualizar inmediatamente la cuadrícula.*

Estos eventos especiales deben estar vinculados a botones dentro de la interfaz desarrollada con PyQt6. Cada botón debe permitir activar un único evento especial, y al hacerlo, debe actualizar inmediatamente la cuadrícula.

Interacción con Arduino (control físico)

Paralelamente, los eventos especiales también deben poder ser activados desde Arduino mediante botones físicos conectados a la placa. Para ello, deberán implementar un sistema de comunicación serial en el programa de Python, capaz de recibir e interpretar mensajes siguiendo la expresión regular "a-x", donde:

- *"a" identifica que se trata de un mensaje de ataque*

- *"x" es un número del 1 al 4 que representa el tipo de evento a ejecutar*

Ejemplos:

Mensaje recibido	***Evento asociado***
a-1	*Infección masiva*
a-2	*Infección masiva mutante*
a-3	*Ritual de purificación*
a-4	*I am atomic*
r	*Reinicio*

Condiciones de fin de juego y reinicio El juego debe detectar automáticamente si todo el mapa esta lleno de zombies o de personas vivas. En cualquiera de estos casos, se debe:

1. *Reproducir un sonido (audio de victoria o derrota) utilizando bibliotecas como playsound, pygame, QSound, u otra similar.*
2. *Reiniciar automáticamente el juego, volviendo a una configuración inicial aleatoria.*

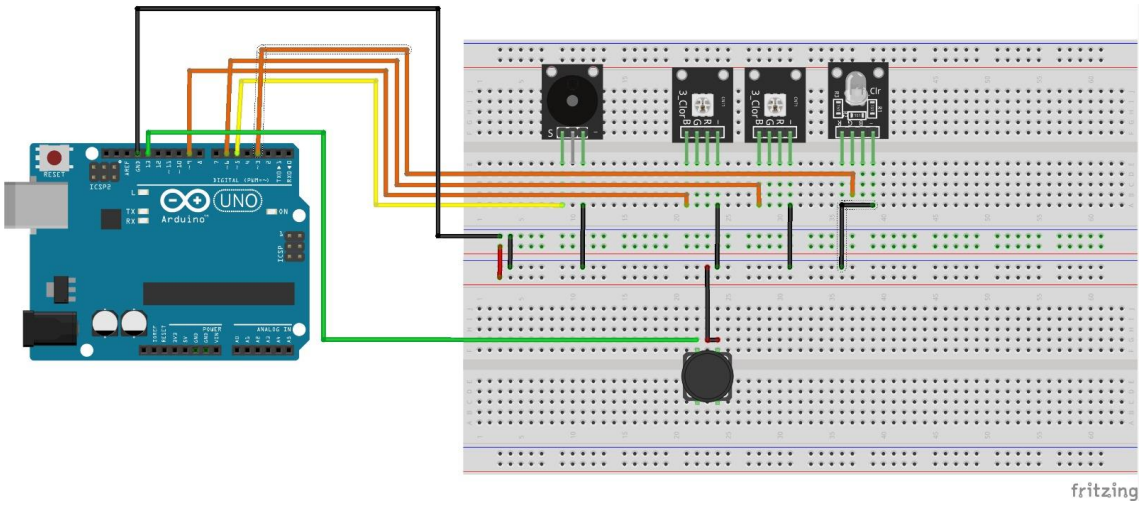
Ítem 1.4 Desarrollo actividad 2

Ítem 1.5 Desarrollo parte 2.1

1.5.a) Componentes utilizados

- 1 KY-016 RGB FULL COLOR LED MODULE
- 2 KY-009 RGB FULL COLOR LED SMD MODULE
- 1 KY-006 PASSIVE BUZZER MODULE
- 5 BOTON

1.5.b) Parte 2: Esquemático de Circuito 2.1



Conectamos dos protoboards y realizamos la conexión de los componentes previamente mencionados de acuerdo al siguiente esquema:

1.1.3 Parte 3: Código Básico

El código funciona de la siguiente manera:

El código carga la interfaz gráfica y la usa para representar gráficamente el videojuego en acción. Crea un tablero de 100 x 100 pixeles y genera una matriz de 100^2 (10.000) elementos que tienen un 80% de probabilidad de ser “muertos” o un 20% de probabilidad de ser “vivos”. Dicha matriz es utilizada por `FigureCanvas` para colocarla en el widget `plotWidget` de la interfaz gráfica.

Utiliza `imshow` para representar los datos de la matriz como imágenes, `interpolation` para que los pixeles no se mezclen y `axis` para eliminar los ejes visibles de la imagen final.

Utiliza la función `QTimer()` (específicamente `start`) para definir la velocidad a la que se ejecuta el juego (más adelante modificamos el código para que dicha velocidad sea controlada por un QSlider).

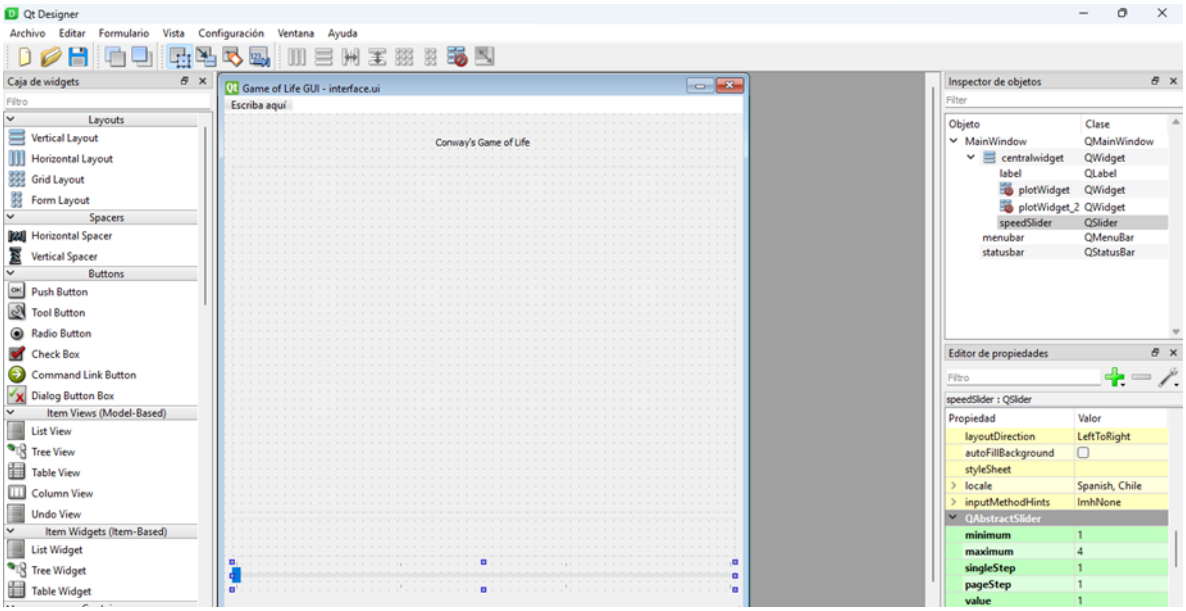
La función `update_grid(self):` es el juego en sí; utiliza la biblioteca `convolve2d` para usar convolución y así definir las reglas del juego (expresadas en el enunciado) y calcular los elementos “vivos” presentes en la imagen.

La función `send_alive_cells_to_arduino(self):` se encarga de enviar los datos al dispositivo Arduino (para que encienda los LEDs correspondientes a la cantidad de células

"vivas"), mientras que la función `listen_to_arduino(self):` se encarga de recibir los datos enviados por el Arduino (el presionado del botón).

Finalmente, la función `reset_grid(self):` "resetea" (restablece) el videojuego a una situación diferente (nueva partida con diferentes células).

Como el botón "reset" de la interfaz gráfica no funcionaba, decidimos eliminarlo. Además, el slider "speedSlider" no hacía nada, por lo cual lo modificamos para que se pueda mover 1/4 de la barra cada vez que se presiona izquierda o derecha, y además le asignamos los valores 1, 2, 3 y 4 para controlar la velocidad siendo los valores anteriores el equivalente a su multiplicador (1x, 2x, 3x y 4x).



Con respecto al slider, añadimos la línea `self.speedSlider.valueChanged.connect(self.on_slider_value_changed)` para declarar la existencia y el funcionamiento del slider (llamado speedSlider en la interfaz gráfica).

`self.` se refiere al objeto, es decir, al widget QSlider en sí.

`speedSlider` es el nombre que le puse al QSlider.

`valueChanged.` es una señal que el QSlider emite cada vez que se desliza y por ende cambie su posición y valor.

`connect(self.on_slider_value_changed)` se refiere a que al `speedSlider` se le asigna (conecta) una función (llamada `on_slider_value_changed`, la explicaremos más adelante) al slot, es decir, a un método que se ejecuta cada vez que se reciba la señal `valueChanged.`

Otra línea importante que agregué es `self.timer.start(self.speedSlider.value() * 100)`:

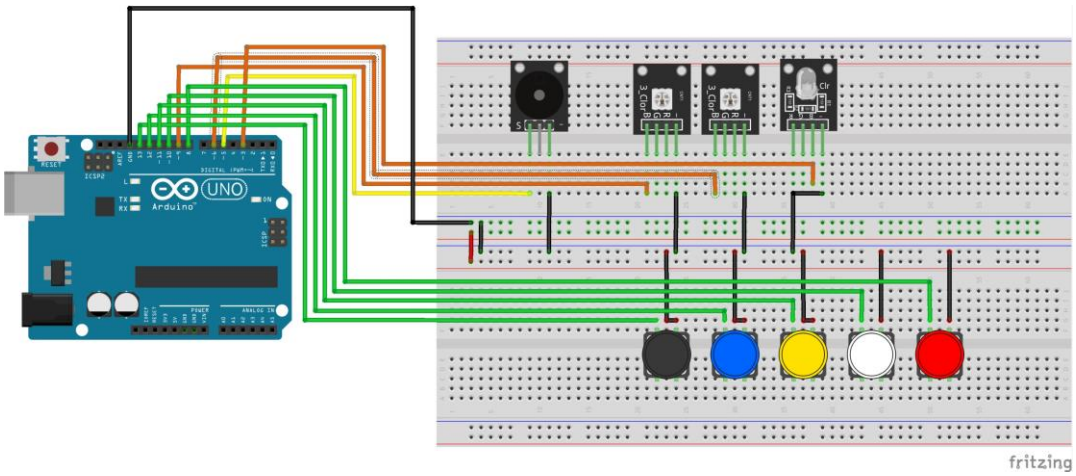
`self.` se refiere al objeto, es decir, a la función `timer.` (de PyQt5). La función es básicamente un temporizador que emite una señal (`timeout()`) en intervalos de tiempo regulares.

`start` hace que su argumento determine el intervalo de tiempo (en milisegundos) entre cada señal `timeout()` recibida. En este caso, el argumento es `self.speedSlider.value() * 100`, es decir, el valor emitido por el slider multiplicado por 100.

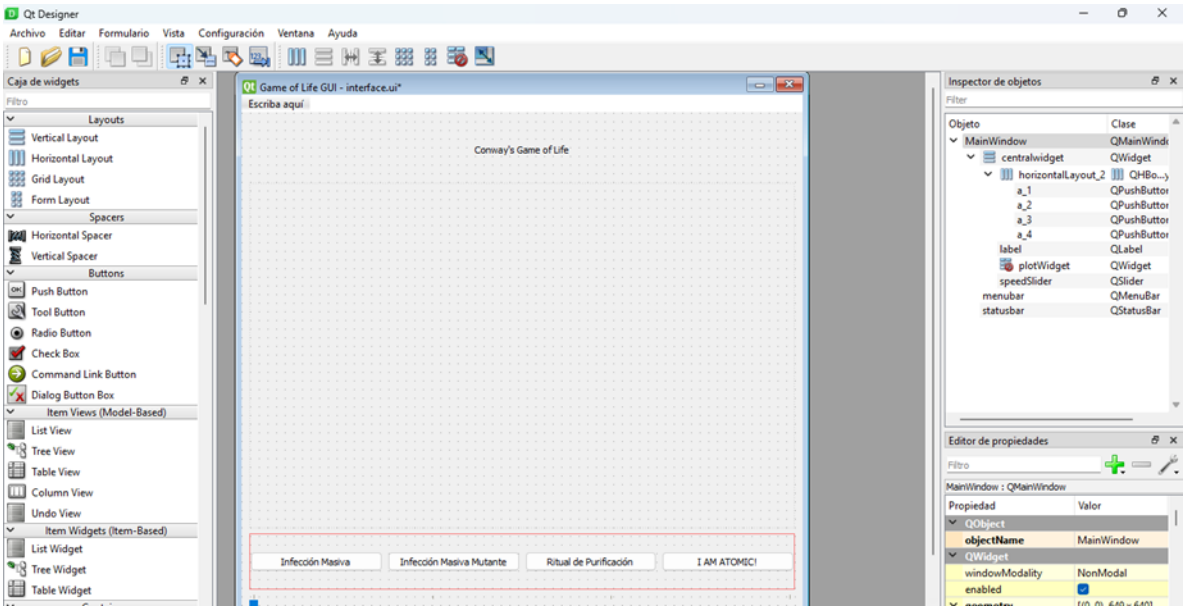
La última modificación al código es la función `on_slider_value_changed(self, value)` , que básicamente recibe el valor emitido por el slider y lo traduce a un multiplicador de velocidad del videojuego (es decir, recibe un “3” y lo usa para triplicar la velocidad a la que se ejecuta el juego).

1.5.c) ACTIVIDAD 2.2:

1.5.d) Parte 2: Esquemático de Circuito 2.2



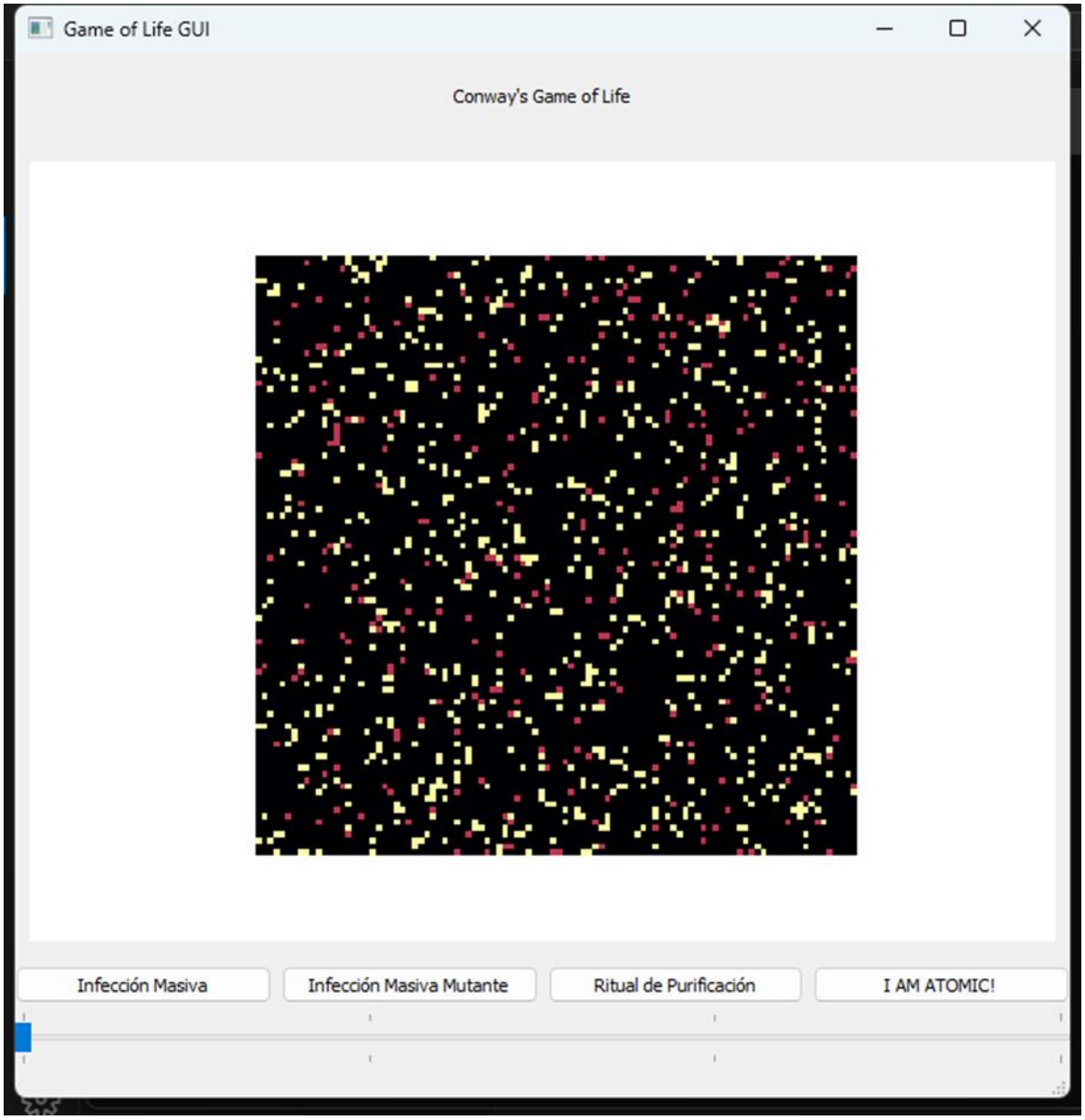
La interfaz gráfica del Game of Life la modifiqué de la siguiente manera :



Le añadimos los botones virtuales correspondientes a los eventos especiales descritos por el enunciado: “Infección masiva (cuadrada)”, “Infección masiva mutada (forma pulsar)”, “Ritual de purificación” y “I am atomic!”.

A pesar de que diseñamos los botones y los distribuimos horizontalmente, al ejecutar el videojuego los botones no estaban presentes. Después de mucho tiempo preguntándole a ChatGPT y abriendo por separado la interfaz con Qt Designer, al final nos dimos cuenta de que el layout de los botones estaban dentro del “QWidget” del Ítem anterior en lugar de dentro del “Main Window”. Rediseñamos los botones con la distribución guardándose en el “Main Window” y los botones virtuales lograron estar visibles en la ejecución del juego.

Lamentablemente, otro problema que surgió es que los colores del juego no quedaron como lo deseábamos inicialmente; “I am atomic!” creaba un cuadrado de pixeles negros en lugar de rojos y la mayoría de la pantalla estaba negra, creando confusión:



Le pedimos a ChatGPT que nos ayudara y utilizó la biblioteca “matplotlib” para asignar los colores que elegimos (Zombies de color negro, Humanos de color blanco y Muertos de color rojo) mediante el comando `self.cmap = ListedColormap(colors)`, definiendo `colors` como una matriz de colores: `colors = ['black', 'white', 'red']`.

Otro problema que encontramos era que el slider de multiplicador de velocidad hacía que el juego se actualizara tan seguido que resultaba en una secuencia de imágenes poco coherentes (no se lograba apreciar lo que estaba sucediendo con el juego con velocidad x2, x3 y mucho menos con x4). Al final el código utilizó la lógica de `speed = self.default_speed - (value * 10)`, siendo `self.default_speed = 200` y `value` el valor numérico que entrega el slider.

El código final funciona de la siguiente manera:

`self.grid = np.random.choice([0, 1, 2], self.grid_size**2, p=[0.2, 0.5, 0.3]).reshape(self.grid_size, self.grid_size)` se encarga de definir los valores que pueden tomar los pixeles del recuadro donde toma lugar el videojuego (Zombie = 0, Humano = 1, Muerto = 2), sus respectivas probabilidades de aparecer al iniciar el juego (0.2, 0.5 y 0.3) y su distribución aleatoria respecto de las dimensiones del tablero (siendo `self.grid_size = 100`).

`self.life_points` se encarga de establecer aleatoriamente los puntos de vida de cada pixel del juego.

`self.default_speed = 200` se utiliza para que el tiempo por defecto que cada cuadro tarda en cambiar al siguiente cuadro es 200 milisegundos (es decir, 5 cuadros por segundo).

`update_grid(self)` es la clase donde se actualiza el tablero. `kernel` es una matriz de 3x3 cuyo centro es cero (el pixel central) y los demás elementos son 1 (los vecinos que lo rodean). El código utiliza `convolve2d` de la biblioteca “SciPy” para realizar una convolución entre los vecinos (crea nuevos pixeles que representan como unos vecinos modifican otros).

Como el juego es básicamente una matriz de dos dimensiones, se utiliza un `for j in range(self.grid_size)` dentro de un `for i in range(self.grid_size)` para determinar el comportamiento de los pixeles basándose en la interacción entre los grupos de zombies, humanos y muertos. Los `if` y `elif` se utilizan para validar las reglas básicas del juego (cuándo un humano muere, cuándo un muerto revive, etc).

`listen_to_arduino(self)` es la clase que permite que el videojuego ejecutándose en Python reaccione a los comandos del Arduino ejecutándose en Arduino. Recibe las señales que envían los botones físicos conectados al Arduino.

`if __name__ == "__main__"` ejecuta el videojuego en Python.

Conclusiones actividad 2.2

La actividad 2.1 no fue particularmente difícil y resultó entretenida, ya que trabajar nuevamente con interfaces gráficas fue una experiencia intuitiva que nos permitió reforzar nuestras habilidades en Python, Arduino y el uso de Qt Designer. Aunque el enunciado parecía complejo al inicio, en la práctica logramos desenvolvernos con fluidez, consolidando nuestras competencias en el desarrollo de interfaces.

Por otro lado, la actividad 2.2 representó el mayor desafío del proyecto. Las múltiples instrucciones requerían más tiempo, esfuerzo y dedicación, lo que generó frustración y dificultades para completar todas las funcionalidades en el plazo establecido. Sin embargo, este proceso nos hizo

tomar mayor conciencia del compromiso que exige la asignatura y del valor de gestionar mejor nuestro tiempo, tanto en clases como fuera de ellas. A pesar de los errores y dificultades, nos quedamos con la certeza de que podemos mejorar, aprender de nuestros tropiezos y apoyarnos en el Profesor y el Ayudante para seguir desarrollando nuestras habilidades con herramientas tan versátiles como Arduino.

1.5.e) **Resumen del trabajo a realizar**

Este proyecto consistió en el desarrollo de dos actividades principales integrando Arduino, programación en Python y diseño de interfaces gráficas. En la actividad 1, se implementó un piano digital de cinco teclas inspirado en el juego *Guitar Hero*, utilizando botones, LEDs RGB y un buzzer. Se desarrollaron dos versiones: una básica que respondía a la interacción directa del usuario, y otra más compleja que incluía selección de dificultad, canciones predefinidas, sistema de puntaje y comandos por monitor serial.En la actividad 2, se trabajó con el juego *The Game of Life*, primero en su versión original y luego adaptándolo al contexto de una invasión zombi. Ambas versiones se desarrollaron con interfaz gráfica en PyQt6 y se integraron con Arduino mediante comunicación serial. El sistema permitía tanto enviar información desde Python hacia Arduino como controlar eventos especiales desde hardware físico.A lo largo del proyecto se enfrentaron desafíos técnicos, especialmente en la segunda actividad, pero también se reforzaron habilidades clave en programación, electrónica y diseño de interfaces, destacando la importancia de la planificación y del trabajo constante para cumplir los objetivos del curso.

Ítem 1.6 Bibliografía

[1] SFUPTOWNMAKER, “Using Flask to Send Data to a Raspberry Pi”, *https://learn.sparkfun.com/tutorials/using-flask-to-send-data-to-a-raspberry-pi/hardware-hookup*.

[2] A. NOVOZHILOV, “Imagen LED quemado”, *https://www.alamy.es/macro-primer-plano-viejo-pequeno-quemado-blanco-plastico-palido-diodo-emisor-de-luz-led-lampara-de-la-guirnalda-electrica-desenfoque-el-fondo-objeto-exterior-image456226110.html*.

[3] N. Poole y BBOYHO, “Light-Emitting Diodes (LEDs)”, *https://learn.sparkfun.com/tutorials/light-emitting-diodes-leds/all*.

Esta bibliografía fue hecha con Mendeley (https://www.mendeley.com/), una plataforma de gestión de referencias bibliográficas que se pueden importar desde diversas fuentes, como bases de datos académicas y páginas web, las que posteriormente se guardan en un formato estructurado que facilita la búsqueda y recuperación de información. Mendeley proporciona herramientas para la creación automática de citas y bibliografías en diferentes estilos de formato.

Ítem 1.7 Información Adicional

Ítem 1.8 Tipos de LEDs

1.8.a) RGB LEDs

¡Los LED RGB (Rojo-Verde-Azul) son en realidad tres LED en uno! Pero eso no significa que solo pueda hacer tres colores. Debido a que el rojo, el verde y el azul son los colores primarios aditivos, puede controlar la intensidad de cada uno para crear todos los colores del arco iris. La mayoría de los LED RGB tienen cuatro pines: uno para cada color y un pin común. En algunos, el pin común es el ánodo, y en otros, es el cátodo.

Figura Anexo **¡Error! No hay texto con el estilo especificado en el documento.**.1 LED de cátodo transparente común RGB. Fuente: [3]

A.1.1 LEDs con circuitos integrados

Algunos LED son más inteligentes que otros. Tomemos el LED de ciclismo, por ejemplo. Dentro de estos LED, en realidad hay un circuito integrado que permite que el LED parpadee sin ningún controlador externo. Aquí hay un primer plano del IC (el gran chip cuadrado negro en la punta del yunque) que controla los colores.

Figura Anexo **¡Error! No hay texto con el estilo especificado en el documento.**.2 Primer plano del LED de ciclo lento de 5 mm. Fuente: [3]

¡Simplemente enciéndalo y míralo ir! Estos son excelentes para proyectos en los que desea un poco más de acción, pero no tiene espacio para circuitos de control. ¡Incluso hay LED parpadeantes RGB que recorren miles de colores!

1.8.b) Paquetes de montaje en superficie (SMD)

Los LED SMD no son tanto un tipo específico de LED sino un tipo de paquete. A medida que la electrónica se hace cada vez más pequeña, los fabricantes han descubierto cómo meter más componentes en un espacio más pequeño. Las piezas SMD (dispositivo de montaje en superficie) son versiones pequeñas de sus contrapartes estándar. Aquí hay un primer plano de un LED direccionable WS2812B empaquetado en un pequeño paquete 5050.

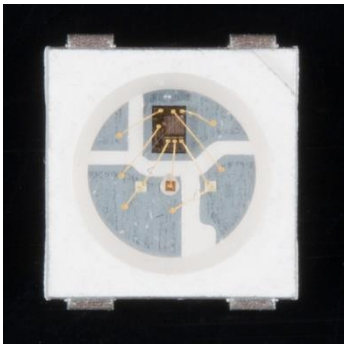


Figura Anexo **¡Error! No hay texto con el estilo especificado en el documento.**.1 Primer plano direccionable WS2812B. Fuente: [3]

Los LED SMD vienen en varios tamaños, ¡desde bastante grandes hasta más pequeños que un grano de arroz! Debido a que son tan pequeños y tienen almohadillas en lugar de piernas, no son tan fáciles de trabajar, pero si tiene poco espacio, podrían ser justo lo que recetó el médico.