<table>
<tr><th>Code</th><th>Explanation</th></tr>
<tr><td>

```
#include <Wire.h>
#include <hd44780.h>
#include <hd44780ioClass/hd44780_I2Cexp.h>
#include <avr/pgmspace.h>

// ===== LCD (I2C) =====
hd44780_I2Cexp lcd;
#define LCD_COLS 20
#define LCD_ROWS 4
```

</td><td>

This section sets up the display system used in the project.

- #include directives: These lines import the necessary libraries. Wire.h enables I2C communication, which the LCD uses to communicate with the Arduino. hd44780.h and hd44780_I2Cexp.h provide the functions needed to control the LCD through an I2C interface. avr/pgmspace.h is a special library that allows the program to store data in the Arduino's program memory.
- hd44780_I2Cexp lcd; This initializes the LCD object. This object represents the screen and gives the program the ability to write text to it using I2C communication.
- #define LCD_COLS 20 and #define LCD_ROWS 4
These define the size of the LCD. The values indicate that the display has 20 columns and 4 rows.

</td></tr>
<tr><td>

```
// ===== Pins =====
#define BUZZER     10
#define BTN_LEFT   A0
#define BTN_RIGHT  A1
const bool BTN_ACTIVE_HIGH = false;

// ===== Pitches (Hz) =====
#define C4 262
#define D4 294
#define E4 330
#define F4 349
#define G4 392
#define A4 440
#define B4 494
#define C5 523
#define D5 587
#define E5 659
#define F5 698
#define G5 784
```

</td><td>

Pins and Pitches

- This section of the code sets up the hardware inputs along with the musical notes used in the project. It assigns readable names to important pins and defines the frequencies for each tone the buzzer will create.

The Pins – Hardware Connections

- This part labels the Arduino pins used in the circuit. The #define command allows the program to use clear names instead of raw pin numbers.
- BUZZER is the output pin connected to the sound device.
- BTN_LEFT and BTN_RIGHT represent the two input buttons used during gameplay.
- BTN_ACTIVE_HIGH indicates how the buttons behave electrically when pressed.

The Pitches – Note Frequencies

- This section lists the frequencies of the musical notes in hertz. Each note name, such as C4 or E4, is assigned its corresponding pitch value.
- This improves readability by letting the program refer to musical notes by name

</td></tr>
</table>

```
// ===== Non-blocking Tone Management =====
unsigned long toneEndTime = 0;

void startTone(uint8_t pin, uint16_t freq,
               uint16_t dur) {
  tone(pin, freq);
  toneEndTime = millis() + dur;
}

void updateTone(uint8_t pin) {
if (toneEndTime > 0 && millis() >= toneEndTime)
  {
    noTone(pin);
    toneEndTime = 0;
  }
}

// ===== Songs =====
const int PROGMEM hb_melody[] = {
  G4, G4, A4, G4, C5, B4,
  G4, G4, A4, G4, D5, C5,
  G4, G4, G5, E5, C5, B4, A4,
  F5, F5, E5, C5, D5, C5
};
const int PROGMEM hb_beatMs[] = {
  400, 400, 800, 800, 800, 1600,
  400, 400, 800, 800, 800, 1600,
  400, 400, 800, 800, 800, 800, 1600,
  400, 400, 800, 800, 800, 1600
};

const int PROGMEM jingle_melody[] = {
E4, E4, E4, E4, E4, E4, E4, G4, C4, D4, E4, F4,
  F4, F4, F4,
F4, E4, E4, E4, E4, D4, D4, E4, D4, G4, C5, C5,
  B4, A4, G4
};
const int PROGMEM jingle_beatMs[] = {
300,300,600,300,300,600,300,300,300,300,600,
  300,300,600,300,
300,300,300,300,300,300,300,300,300,600,
  400,400,300,300,800
};

const int PROGMEM blue_melody[] = {
G4, G4, A4, B4, A4, G4, E4, D4, G4, G4, A4, B4,
```

rather than by number.

Non-blocking Tone Management

- This section handles sound playback without pausing the program. It keeps track of when a tone should end so that the game can continue running smoothly.

Tone Timing

- A timer variable is created to record when the current tone will finish.
- The `startTone` function begins a tone and sets the time when it should stop.
- The `updateTone` function checks the timer and turns the tone off when its duration has passed.
- This allows sound to play while the rest of the program continues updating.

Songs

- This section contains the musical data used in the project. Each song is made of two parts: a melody array and a beat-duration array.

Melody and Beat Arrays

- Each melody is stored in an array that contains a sequence of note values.
- Each beat array contains the duration for every note in milliseconds.
- The `PROGMEM` keyword places these arrays in program memory, conserving dynamic memory.
- These arrays define complete songs that the buzzer can play.

```
A4, G4, E4, D4,
D4, E4, F4, G4, F4, E4, D4, G4, A4, B4, C5, B4,
A4
};
const int PROGMEM blue_beatMs[] = {
400,400,300,300,300,400,400,400,
400,400,300,300,300,400,400,400,
400,400,300,300,300,400,800,
400,300,300,400,300,800
};

const int PROGMEM nocturne_melody[] = {
E5, D5, C5, D5, E5, G5, F5, E5, D5, C5, B4, C5,
D5, E5, D5, C5,
B4, A4, B4, C5, D5, C5, B4, A4, B4, C5, B4, A4
};
const int PROGMEM nocturne_beatMs[] = {
500,300,300,300,500,400,400,400,
300,300,300,300,500,400,300,300,
300,300,300,300,500,400,800,
400,300,300,300,800
};

const int PROGMEM waltz_melody[] = {
C4, E4, G4, E4, C4, G4, C4, D4, F4, A4, F4, D4,
A4, D4,
E4, G4, B4, G4, E4, G4, E4, C5, B4, A4, G4
};
const int PROGMEM waltz_beatMs[] = {
400,200,200,400,400,400,800,
400,200,200,400,400,400,800,
400,200,200,400,400,400,800, 400,300,300,800
};

struct Song {
const char* name;
const int* melody;
const int* beatMs;
int len;
};

Song songs[] = {
{ "Happy Birthday", hb_melody, hb_beatMs,
(int)(sizeof(hb_melody)/sizeof(int)) },
{ "Jingle Bell", jingle_melody, jingle_beatMs,
(int)(sizeof(jingle_melody)/sizeof(int)) },
{ "Blue Danube", blue_melody, blue_beatMs,
(int)(sizeof(blue_melody)/sizeof(int)) },
```

Song Structure and List

- This section defines how songs are stored and organized in the program. It groups related song data into a clear and reusable format.
- **`struct Song:`** This part declares a structure named `Song`.

Each Song has:

- a `name` for the title of the song,
- a `melody` pointer to the array of notes,
- a `beatMs` pointer to the array of note durations,
- and `len`, which stores how many notes the song contains.
- `Song songs[] = { ... }` This array holds all the songs used in the project.
- Each entry provides the title and connects it to its melody and beat arrays.
- The `sizeof(...)/sizeof(int)` expression calculates how many elements are in each melody array and stores that number in `len`.
- `const int NUM_SONGS = sizeof(songs)/sizeof(Song);` This line automatically computes how many songs are in the playlist.
- `int selectedSongIdx = 0;` This variable tracks which song is currently selected.
- `int highScores[5] = {0,0,0,0,0};` This array stores the high score for each song.

Difficulty Settings

- This section defines the difficulty levels

```
    { "Nocturne Op9", nocturne_melody,
            nocturne_beatMs,
  (int)(sizeof(nocturne_melody)/sizeof(int)) },
   { "Waltz Mirror", waltz_melody, waltz_beatMs,
    (int)(sizeof(waltz_melody)/sizeof(int)) }
                    };
const int NUM_SONGS = sizeof(songs)/sizeof(Song);

        int selectedSongIdx = 0;
      int highScores[5] = {0,0,0,0,0};

     // ===== Difficulty Settings =====
   enum Difficulty { EASY, MEDIUM, HARD };
    Difficulty selectedDifficulty = HARD;
  const char* difficultyNames[] = {"Easy  ",
          "Medium", "Hard  "};
 const float difficultyMultipliers[] = {1.0f,
              1.5f, 2.7f};
```

available in the game and how they are represented.

- `enum Difficulty { EASY, MEDIUM, HARD };` This line creates an enumeration for the three difficulty levels.
- `Difficulty selectedDifficulty = HARD;` This variable keeps track of the difficulty level currently chosen.
- `const char* difficultyNames[] = {"Easy ", "Medium", "Hard "};` This array stores the text labels that will be shown on the display for each difficulty.
- `const float difficultyMultipliers[] = {1.0f, 1.5f, 2.7f};` This array provides a multiplier value for each difficulty level.

```
        // ===== Timing =====
      const int MOVE_DT_MS = 90;
const int BASE_TRAVEL_TIME_MS = (LCD_COLS - 1) *
                MOVE_DT_MS;
   int travelTimeMs = BASE_TRAVEL_TIME_MS;
     const int HIT_WINDOW_MS = 180;
      const int CORRECT_MS = 220;
        const byte BLOCK = 255;

     // ===== Schedule arrays =====
       unsigned long startMs[64];
       unsigned long spawnMs[64];

      // ===== Active notes =====
           struct Active {
              bool used;
              byte row;
             int noteIdx;
           int col, prevCol;
         bool inWindow, resolved;
    unsigned long spawnedAt, windowStart;
                };
       const byte MAX_ACTIVE = 14;
       Active active[MAX_ACTIVE];
```

Timing

- This section defines the main timing values used in the game.
- `MOVE_DT_MS` controls how often notes shift on the screen.
- `BASE_TRAVEL_TIME_MS` sets how long a note takes to move across the display.
- `HIT_WINDOW_MS` determines how close the timing must be for a note to count as a hit.

Schedule Arrays

- These arrays store the exact times when notes should spawn and when the song begins. They allow the game to schedule notes in advance.

Active Notes

- This structure keeps track of notes that are currently moving on the screen. Each active note stores its row, position, timing, and whether it has been hit.

```
// ===== State =====
bool isPlaying = false;
unsigned long tStart = 0;
int nextToSpawn = 0;
int curScore = 0;
int totalNotes = 0;
```

- `MAX_ACTIVE` sets the maximum number of notes that can be on screen at once.

State

- This section holds the main variables that control gameplay. It tracks whether a song is playing, when it started, which note spawns next, and the scoring information.

```
// ===== Helpers =====
inline bool rawPressed(int pin) {
int v = digitalRead(pin);
return BTN_ACTIVE_HIGH ? (v == HIGH) : (v ==
LOW);
}

void playErrorSound() {
tone(BUZZER, 200, 60);
delay(70);
tone(BUZZER, 150, 80);
toneEndTime = millis() + 80;
}

// ===== LCD helpers =====
inline void put(byte c, byte r, char ch) {
lcd.setCursor(c,r);
lcd.write(ch);
}

inline void drawBorders() {
lcd.setCursor(0,0);
lcd.print(F("===================="));
lcd.setCursor(0,3);
lcd.print(F("===================="));
}

inline void drawTarget() {
put(0,1,'P');
put(0,2,'P');
}

inline void drawConfirmBarBottomSmooth(int
filledCols) {
if (filledCols <= 0) {
lcd.setCursor(0,3);
```

Helpers

- `rawPressed` checks if a button is currently pressed.
- `playErrorSound` plays a short two-tone sound used for mistakes.

LCD Helpers

- `put` writes a character to a specific LCD position.
- `drawBorders` prints the top and bottom borders of the screen.
- `drawTarget` draws the target markers in the two gameplay rows.
- `drawConfirmBarBottomSmooth` draws or updates the progress bar at the bottom.

```
        lcd.print(F("====================="));
                        return;
                          }
        if (filledCols > 20) filledCols = 20;
        for (int col = 0; col < 20; col++) {
                lcd.setCursor(col, 3);
        lcd.print(col < filledCols ? (char)255 : '
                        ');
                          }
                          }

    inline void clearCell(byte c, byte r) {
                if (c != 0) put(c,r,' ');
                          }

    inline void drawBlock(byte c, byte r) {
                    put(c,r,BLOCK);
                          }

        inline int btnForRow(byte r) {
        return (r == 1) ? BTN_LEFT : BTN_RIGHT;
                          }
```

- `clearCell` removes a character from a cell.
- `drawBlock` displays a note block on the screen.
- `btnForRow` returns which button corresponds to a given row.

```
            // ===== Game Logic =====
            void computeSchedule() {
                  float softMul =
pow(difficultyMultipliers[selectedDifficulty],
                        0.5f);
    travelTimeMs = (int)(BASE_TRAVEL_TIME_MS /
                      softMul);
                unsigned long t = 0;
        int len = songs[selectedSongIdx].len;
                const int* beats =
          songs[selectedSongIdx].beatMs;
            for (int i=0; i<len; i++) {
                    startMs[i] = t;
            spawnMs[i] = (t >= (unsigned
 long)travelTimeMs) ? (t - travelTimeMs) : 0;
       // multiply beat duration inversely by
 difficulty to speed up the song (difficulty
                        logic)
          unsigned int baseBeat = (unsigned
          int)pgm_read_word(&beats[i]);
    // scale beats with a softened multiplier to
       speed up songs less aggressively
```

Game Logic

- This section describes how the game controls note timing, movement, inputs, and the overall flow of a song.
- computeSchedule(): Sets up the full timing plan for the song. Decides when each note appears and when it should reach the target.

```
    float scaledBeat = (float)baseBeat / softMul;
      t += (unsigned long)(scaledBeat + 0.5f);
                        }
                      }

             void resetActive() {
        for (byte i=0; i<MAX_ACTIVE; i++)
    active[i] = {false,0,0,0,-1,false,false,0,0};
                nextToSpawn = 0;
                 curScore = 0;
      totalNotes = songs[selectedSongIdx].len;
                      }

              void startSong() {
                  lcd.clear();
                 drawBorders();
                  drawTarget();
                computeSchedule();
                  resetActive();
        tStart = millis() - travelTimeMs;
                isPlaying = true;
                      }

    void spawnReadyNotes(unsigned long now) {
     int songLen = songs[selectedSongIdx].len;
  while (nextToSpawn < songLen && (now - tStart)
           >= spawnMs[nextToSpawn]) {
        for (byte i=0; i<MAX_ACTIVE; i++) {
               if (!active[i].used) {
                active[i].used = true;
         active[i].row = 1 + (nextToSpawn % 2);
            active[i].noteIdx = nextToSpawn;
             active[i].col = LCD_COLS - 1;
                active[i].prevCol = -1;
              active[i].inWindow = false;
              active[i].resolved = false;
               active[i].spawnedAt = now;
                         break;
                           }
                         }
                   nextToSpawn++;
                        }
                      }

     void updateMovement(unsigned long now) {
        for (byte i=0; i<MAX_ACTIVE; i++) {
           if (!active[i].used) continue;
```

- resetActive(): Clears all active notes before the song begins.
  Resets score counters and prepares empty slots.

- startSong(): Starts the round and prepares the screen.
  Loads the timing schedule and sets the starting time.

- spawnReadyNotes(): Releases notes when their scheduled time is reached. Place each note in its correct lane.

- updateMovement():Moves notes across the display based on elapsed time. Stops their movement when they reach the hit area.

```
            if (active[i].prevCol >= 0)
  clearCell(active[i].prevCol, active[i].row);
      if (active[i].inWindow) { drawBlock(0,
          active[i].row); continue; }

            long elapsed = (long)(now -
             active[i].spawnedAt);
    int steps = (elapsed <= 0) ? 0 : (elapsed /
                 MOVE_DT_MS);
       int newCol = (LCD_COLS - 1) - steps;
            if (newCol < 0) newCol = 0;

              active[i].col = newCol;
     drawBlock(active[i].col, active[i].row);
        active[i].prevCol = active[i].col;

              if (active[i].col == 0) {
              active[i].inWindow = true;
             active[i].windowStart = now;
                          }
                        }
                      }

    void checkWindows(unsigned long now) {
       for (byte i=0; i<MAX_ACTIVE; i++) {
    if (!active[i].used || !active[i].inWindow)
                  continue;

        int btn = btnForRow(active[i].row);

         if (!active[i].resolved && (now -
      active[i].windowStart <= (unsigned
            long)HIT_WINDOW_MS)) {
               if (rawPressed(btn)) {
                    int noteFreq =
pgm_read_word(&songs[selectedSongIdx].melody[acti
                ve[i].noteIdx]);
          tone(BUZZER, noteFreq, CORRECT_MS);
         toneEndTime = millis() + CORRECT_MS;
              active[i].resolved = true;
                      curScore++;
                          }
                        }

    if (now - active[i].windowStart > (unsigned
            long)HIT_WINDOW_MS) {
             if (!active[i].resolved) {
                  playErrorSound();
```

- checkWindows(): Checks timing accuracy in the hit window. Registers hits, adds score, or marks missed notes.

```
                    }
            put(0, active[i].row, 'P');
              active[i].used = false;
                        }
                      }
                    }

  bool songFinished(unsigned long now) {
  int songLen = songs[selectedSongIdx].len;
  if (nextToSpawn < songLen) return false;
unsigned long endTime = startMs[songLen-1] +
    HIT_WINDOW_MS + travelTimeMs + 50;
  if (now - tStart < endTime) return false;
    for (byte i=0; i<MAX_ACTIVE; i++) if
      (active[i].used) return false;
                return true;
                    }
```

- songFinished(): Determines when the song is fully complete. Checks that all notes have spawned and no active notes remain.

```
        void waitForStartPress() {
                lcd.clear();
                delay(100);
              drawBorders();
            lcd.setCursor(2,1);
     lcd.print(F("Press any button"));
            lcd.setCursor(6,2);
         lcd.print(F("to start"));

     bool prevL = rawPressed(BTN_LEFT);
    bool prevR = rawPressed(BTN_RIGHT);
              while (true) {
       bool l = rawPressed(BTN_LEFT);
      bool r = rawPressed(BTN_RIGHT);
    if ((l && !prevL) || (r && !prevR)) {
                    break;
                  }
            prevL = l; prevR = r;
                delay(10);
                  }

                delay(500);
               lcd.clear();
                delay(100);
              drawBorders();
              drawTarget();
                  }

        void songSelectionMenu() {
```

Start Screen and Song Selection

- This section explains how the game waits for player input to begin and how the player chooses a song.

waitForStartPress():

1. Shows the start message on the screen.Waits for the player to press either button.
2. Only reacts when a button is newly pressed, preventing accidental holds.
3. Once a press is detected, it clears the screen and sets up the game display.

```
      while (rawPressed(BTN_LEFT) ||
  rawPressed(BTN_RIGHT)) { delay(10); }
              delay(100);

              lcd.clear();
             drawBorders();
          lcd.setCursor(2,1);
     lcd.print(F("Select Song..."));

 const unsigned long LONG_PRESS_MS = 1000;
const unsigned long PROGRESS_SHOW_MS = 400;
          lcd.setCursor(0,2);
   lcd.print(F("                    "));
          lcd.setCursor(3,2);
  lcd.print(songs[selectedSongIdx].name);
              delay(150);

        bool confirming = false;
          while (!confirming) {
        if (rawPressed(BTN_LEFT)) {
        unsigned long t0 = millis();
       while (rawPressed(BTN_LEFT)) {
     unsigned long held = millis() - t0;

        if (held >= PROGRESS_SHOW_MS) {
           unsigned long progWindow =
  (LONG_PRESS_MS > PROGRESS_SHOW_MS) ?
(LONG_PRESS_MS - PROGRESS_SHOW_MS) : 1;
           unsigned long rel = held -
           PROGRESS_SHOW_MS;
      int filledCols = (int)((float)rel /
   (float)progWindow * 20.0f + 0.5f);
      if (filledCols > 20) filledCols = 20;
     drawConfirmBarBottomSmooth(filledCols);
                } else {
         drawConfirmBarBottomSmooth(0);
                    }
   if (held >= LONG_PRESS_MS) { confirming =
           true; break; }
                delay(30);
                   }
       drawConfirmBarBottomSmooth(0);
           if (confirming) break;
   selectedSongIdx = (selectedSongIdx - 1 +
        NUM_SONGS) % NUM_SONGS;
            lcd.setCursor(0,2);
    lcd.print(F("                    "));
            lcd.setCursor(3,2);
```

songSelectionMenu():

1. Displays the list of songs the player can choose from.

2. Shows the current song name on the screen.Pressing left or right moves through the song list.

3. Holding a button starts a progress bar to confirm selection.

4. A long press selects the song and shows a “Selected” message.

5. After confirming, the chosen song is stored and the menu ends.

```
    lcd.print(songs[selectedSongIdx].name);
                delay(150);
                    }
          if (rawPressed(BTN_RIGHT)) {
           unsigned long t0 = millis();
         while (rawPressed(BTN_RIGHT)) {
        unsigned long held = millis() - t0;
          if (held >= PROGRESS_SHOW_MS) {
             unsigned long progWindow =
  (LONG_PRESS_MS > PROGRESS_SHOW_MS) ?
(LONG_PRESS_MS - PROGRESS_SHOW_MS) : 1;
             unsigned long rel = held -
            PROGRESS_SHOW_MS;
         int filledCols = (int)((float)rel /
   (float)progWindow * 20.0f + 0.5f);
       if (filledCols > 20) filledCols = 20;

drawConfirmBarBottomSmooth(filledCols);
                    } else {
           drawConfirmBarBottomSmooth(0);
                        }
      if (held >= LONG_PRESS_MS) { confirming
            = true; break; }
                   delay(30);
                      }
         drawConfirmBarBottomSmooth(0);
             if (confirming) break;
    selectedSongIdx = (selectedSongIdx + 1) %
              NUM_SONGS;
             lcd.setCursor(4,2);
      lcd.print(F("                    "));
             lcd.setCursor(3,2);
     lcd.print(songs[selectedSongIdx].name);
                delay(150);
                   }
           lcd.setCursor(2,1);
      lcd.print(F(" Hold to Select  "));
              delay(50);
                 }

              lcd.clear();
            drawBorders();
           lcd.setCursor(3,1);
        lcd.print(F("Selected:"));
           lcd.setCursor(3,2);
  lcd.print(songs[selectedSongIdx].name);
              delay(1000);
                  }
```

```
void difficultySelectionMenu() {
  while (rawPressed(BTN_LEFT) ||
rawPressed(BTN_RIGHT)) { delay(10); }
              delay(100);

              lcd.clear();
             drawBorders();

const unsigned long LONG_PRESS_MS = 1000;
const unsigned long PROGRESS_SHOW_MS = 400;
              delay(150);
          lcd.setCursor(0,2);
  lcd.print(F("                    "));
          lcd.setCursor(1,2);
lcd.print(difficultyNames[selectedDifficulty]);

          bool confirming = false;
            while (!confirming) {
              lcd.setCursor(1,1);
      lcd.print(F("Difficulty (R):  "));
         if (rawPressed(BTN_RIGHT)) {
          unsigned long t0 = millis();
        while (rawPressed(BTN_RIGHT)) {
       unsigned long held = millis() - t0;
         if (held >= PROGRESS_SHOW_MS) {
            unsigned long progWindow =
  (LONG_PRESS_MS > PROGRESS_SHOW_MS) ?
(LONG_PRESS_MS - PROGRESS_SHOW_MS) : 1;
           unsigned long rel = held -
          PROGRESS_SHOW_MS;
       int filledCols = (int)((float)rel /
   (float)progWindow * 20.0f + 0.5f);
      if (filledCols > 20) filledCols = 20;
     drawConfirmBarBottomSmooth(filledCols);
                  } else {
          drawConfirmBarBottomSmooth(0);
                    }
          if (held >= LONG_PRESS_MS) {
                confirming = true;
                  break;
                   }
                delay(30);
                   }
        drawConfirmBarBottomSmooth(0);
           if (confirming) break;
            selectedDifficulty =
```

Difficulty Selection Menu

- The difficulty menu handles how the game presents the available difficulty levels and how the player confirms a choice.

difficultySelectionMenu():

- Shows the difficulty options and lets the player choose a level using the right button.

- A short press cycles through the available difficulties, while a long press triggers a confirmation bar at the bottom of the screen.

- Once the long-press threshold is reached, the chosen difficulty is locked in and displayed briefly before continuing.

```
(Difficulty)((selectedDifficulty + 1) % 3);
                lcd.setCursor(0,2);
        lcd.print(F("                    "));
                lcd.setCursor(1,2);

lcd.print(difficultyNames[selectedDifficulty]);
                    delay(200);
                        }
                    delay(50);
                        }
                  lcd.clear();
                 drawBorders();
               lcd.setCursor(1,1);
        lcd.print(F("Difficulty Set:"));
               lcd.setCursor(1,2);
 lcd.print(difficultyNames[selectedDifficulty]);
                  delay(1000);
                      }
```

```
            bool playAgainMenu() {
                  lcd.clear();
                 drawBorders();
               lcd.setCursor(5,1);
          lcd.print(F("Play again?"));
               lcd.setCursor(4,2);
         lcd.print(F("L=YES  R=MENU"));

         while (rawPressed(BTN_LEFT) ||
          rawPressed(BTN_RIGHT)) {}
                  delay(50);

                while (true) {
           if (rawPressed(BTN_LEFT)) {
                    delay(40);
          while (rawPressed(BTN_LEFT)) {}
                  return true;
                      }
           if (rawPressed(BTN_RIGHT)) {
                    delay(40);
          while (rawPressed(BTN_RIGHT)) {}
                  return false;
                      }
                    }
                  }
```

Play Again Menu

- Manages what happens after a song ends and lets the player choose whether to restart or return to the main menu.

playAgainMenu():

- Shows a “Play again?” screen with choices for YES or MENU.
- Waits for a clean button release so old inputs don’t interfere.
- A press on the left button confirms replay and returns true.
- A press on the right button returns false and sends the game back to the selection menus.

```
            void setup() {
```

Setup

```
pinMode(BUZZER, OUTPUT);
pinMode(BTN_LEFT, INPUT_PULLUP);
pinMode(BTN_RIGHT, INPUT_PULLUP);

int status = lcd.begin(LCD_COLS, LCD_ROWS);
if(status) {
pinMode(13, OUTPUT);
while(1) {
digitalWrite(13, HIGH);
delay(200);
digitalWrite(13, LOW);
delay(200);
}
}

lcd.clear();
drawBorders();
lcd.setCursor(4,1);
lcd.print(F("RHYTHM GAME"));
lcd.setCursor(3,2);
lcd.print(F("Press to Start"));
delay(2000);

songSelectionMenu();
difficultySelectionMenu();
waitForStartPress();
startSong();
}
```

- Handles the game's initialization phase and prepares everything needed before the first round begins.

setup():

- Initializes the buzzer, buttons, and LCD to establish the game's starting hardware state.
- Includes an LCD startup check that signals an error if the display fails to initialize.
- Shows the title screen to introduce the game before interaction starts.
- Loads the song and difficulty menus so the game begins with the player's chosen settings.

```
void loop() {
if (!isPlaying) {
delay(10);
return;
}

unsigned long now = millis();
updateTone(BUZZER);

spawnReadyNotes(now);
updateMovement(now);
checkWindows(now);

if (songFinished(now)) {
isPlaying = false;

if (curScore > highScores[selectedSongIdx]) {
```

Loop

- Controls the main game cycle while a song is active and manages what happens when the song ends.

loop():

- Updates sound, note movement, spawns, and hit checks while the song is active.
- Stops gameplay when the song is finished and updates the high score if beaten.
- Shows the score screens, then uses the replay menu to decide whether to restart or return to the selection menus.

```
      highScores[selectedSongIdx] = curScore;
                        }

                    delay(500);
                   lcd.clear();
                    delay(100);
                  drawBorders();
                lcd.setCursor(7,1);
              lcd.print(F("Score:"));
                lcd.setCursor(7,2);
        lcd.write('0' + (curScore / 10));
        lcd.write('0' + (curScore % 10));
                  lcd.write('/');
      lcd.write('0' + (totalNotes / 10));
      lcd.write('0' + (totalNotes % 10));
                  delay(3000);

                   lcd.clear();
                    delay(100);
                  drawBorders();
                lcd.setCursor(5,1);
          lcd.print(F("Best Score:"));
                lcd.setCursor(7,2);
  lcd.write('0' + (highScores[selectedSongIdx]
                    / 10));
  lcd.write('0' + (highScores[selectedSongIdx]
                    % 10));
                  lcd.write('/');
      lcd.write('0' + (totalNotes / 10));
      lcd.write('0' + (totalNotes % 10));
                  delay(3000);

         while (rawPressed(BTN_LEFT) ||
   rawPressed(BTN_RIGHT)) { delay(10); }
                  delay(200);

         bool again = playAgainMenu();
                  if (again) {
            difficultySelectionMenu();
               waitForStartPress();
                   startSong();
                    } else {
               songSelectionMenu();
            difficultySelectionMenu();
               waitForStartPress();
                   startSong();
                        }
                    return;
```

```
        }

        delay(MOVE_DT_MS);
    }
```