Student: **Gokularamanan R S** Roll Number: **23B1854**

Project Title: Doodluino: Doodle Jump on Arduino

DOODLUINO

DOODLE JUMP ON ARDUINO

Project Abstract –

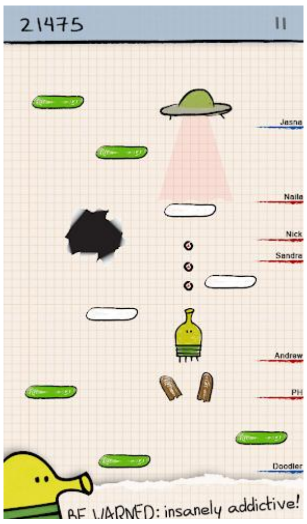


Figure 1: The Actual Game

Credits: Lima Sky (game dev)

- Introduction – Doodle Jump is a classic arcade game where players control a small character (the "Doodler") that continuously jumps on randomly spawning, dynamic and stationary platforms, trying to ascend as high as possible without falling. Using a Joystick to move the Doodler left or right, we aim not to fall or get attacked by obstacles like enemies, moving platforms with spikes, and black holes that appear along the way. There are also power-ups, such as springs and jetpacks that help us jump higher. Ultimately, the goal is to survive and to achieve the highest score.

- The Grand Goal – TO obtain and simulate the complete version of the game on an Arduino UNO board, respecting all the game physics and simulations involved.
  - Collision detections
  - SPI update rate
  - Multi-object dynamics
  - Interrupts - for performing separate tasks simultaneously, or with minimum delay (in my case, simultaneously playing background music during gameplay through Arduino connected speaker)
  - EEPROM Memory Allocation
- The main inputs and outputs –
  - Inputs:
    - Joystick – Left and Right Control, Top and Bottom, and Switch
  - Outputs:
    - TFT Display to display the game
    - Speaker to play game-play music
- During the DEMO, I would like to demonstrate my game, by playing each of the Game Levels, showcasing the various features that have been built into it – like EEPROM utilization, Interrupts, etc.
- Role of the Arduino – It is the brain of the game. It gets joystick input, outputs the game mechanics to the TFT display and audio via the speaker, powered by the

LM386 low-power amplifier.

Project Detail –

- The major components used –

1. Arduino UNO (32 kB storage and 2 kB RAM, with 1 KB allocated for EEPROM)
2. Adafruit ILI9341 2.8'' TFT SPI Display

The important pins:

1. CS (Chip Select) - Selects ILI9341 for communication
2. DC - Data/Command
3. RST - Reset
4. MOSI - Master Out Slave In
5. SCK – Serial Clock, operates at 16 MHz, same as Uno

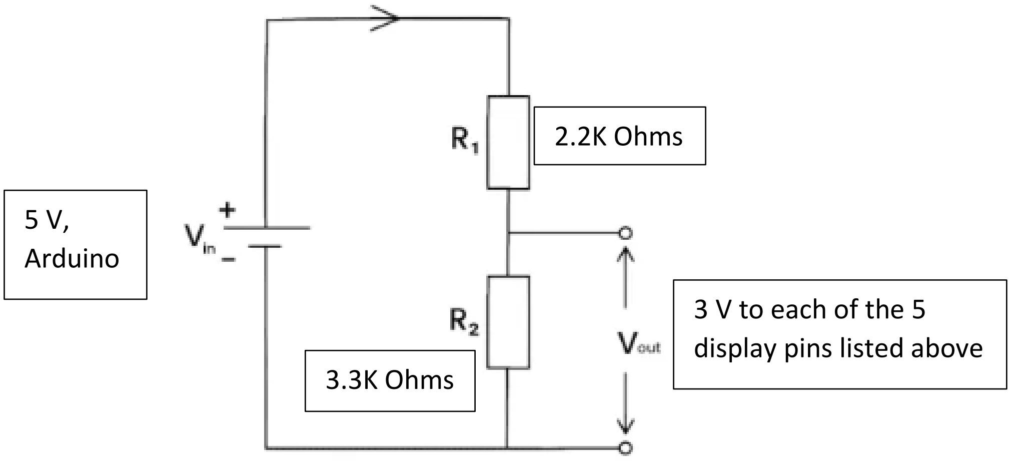
3. Analog Joystick (Left-Right, Top-Bottom, and Select button/ Switch)
4. Speaker (8 Ohm, 0.5 W)
5. LM386, Low Power Amplifier

LM386 amplifier –

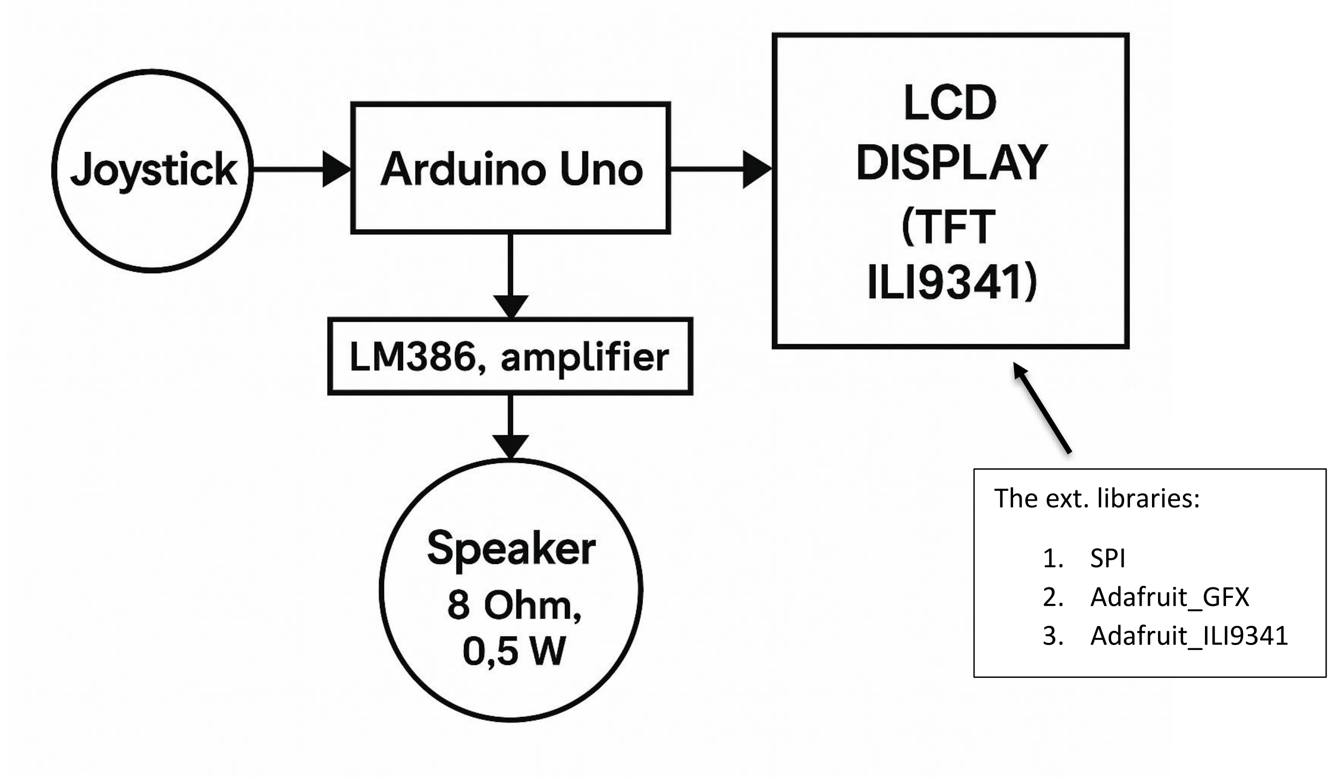
1. operates on a low voltage
2. adjustable gain control, using the potentiometer knob

6. Potential Divider Circuit (Used 2.2 Ohm and 3.3 Ohm resistors to precisely deliver 3 V from Arduino 5 V to TFT display, to make sure that it doesn't get damaged due to heavy load)
Ref: https://cdn-shop.adafruit.com/datasheets/ILI9341.pdf

POTENTIAL DIVIDER EQUATION: $V_{out} = \frac{R_2}{R_1+R_2} V_{in}$



- Hardware block diagram –



The ext. libraries:

1. SPI
2. Adafruit_GFX
3. Adafruit_ILI9341

- Final Circuit Diagram – (next page)

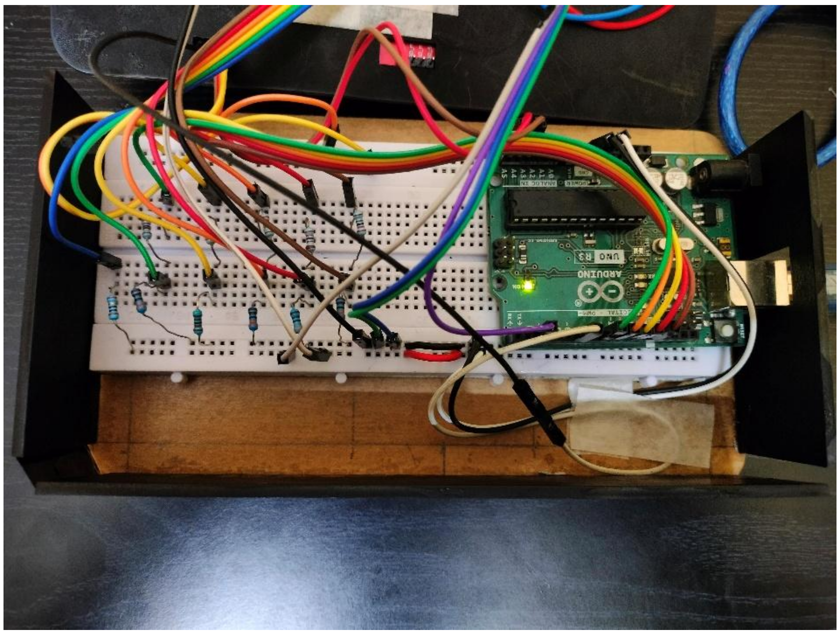


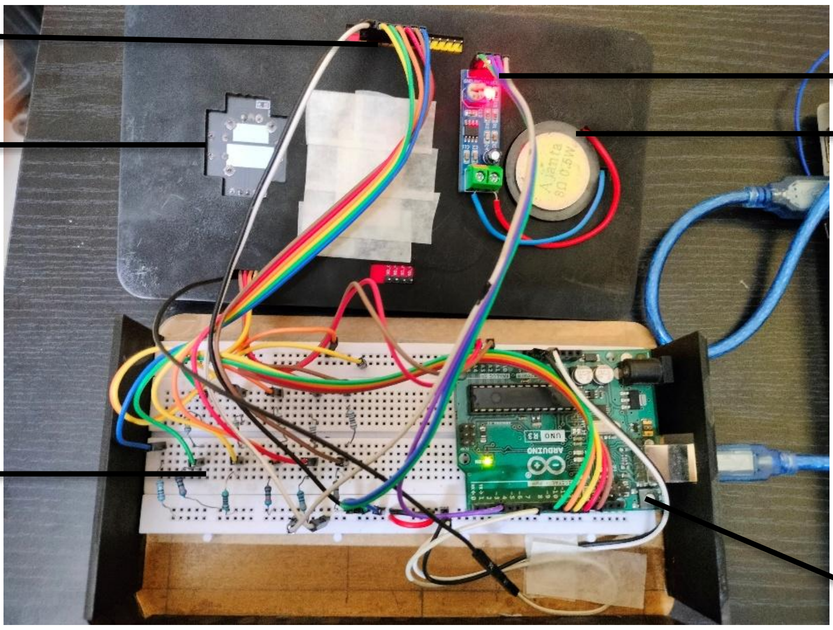
Figure 2: Circuit Diagram - Arduino board, Potential Divider Circuit and Wires connecting the joystick, TFT display and the speaker through amplifier to the UNO

The 8 pins of the TFT display (being used):

VCC, GND, CS, RST, SCK, MOSI and LED

Joystick, 5 pins: 5V VCC, GND, VRX, VRY, SW

The potential divider circuit, with 5 x 2.2k ohm and 5 x 3.3k ohm resistors, corresponding to pins CS, DC, RESET, MOSI, SCK of the TFT



LM386 amplifier, 4 pins: 2 GNDs, 5V VCC, IN

8 Ohm, 0.5 W speaker to LM386

Figure 3: Expanded Circuit Diagram, showing the back side of the console's top, having joystick, display, and speaker attached.

Arduino Uno, with serial connection to the PC for power

- List of external libraries used:
  1. SPI -- lib to enable SPI connection between the Uno and TFT
  2. Adafruit_GFX -- this is the core graphics library - all shapes and graphics
  3. Adafruit_ILI9341 -- display driver

- The internal libraries used:
  1. EEPROM -- electrically erasable programmable read-only memory; used it to store my high scores - Uno allocates 1 kb of ram to EEPROM. I am just using up 3 * 2 bytes = 6 bytes for EEPROM to store 3 integer values for easy, medium and hard level scores .

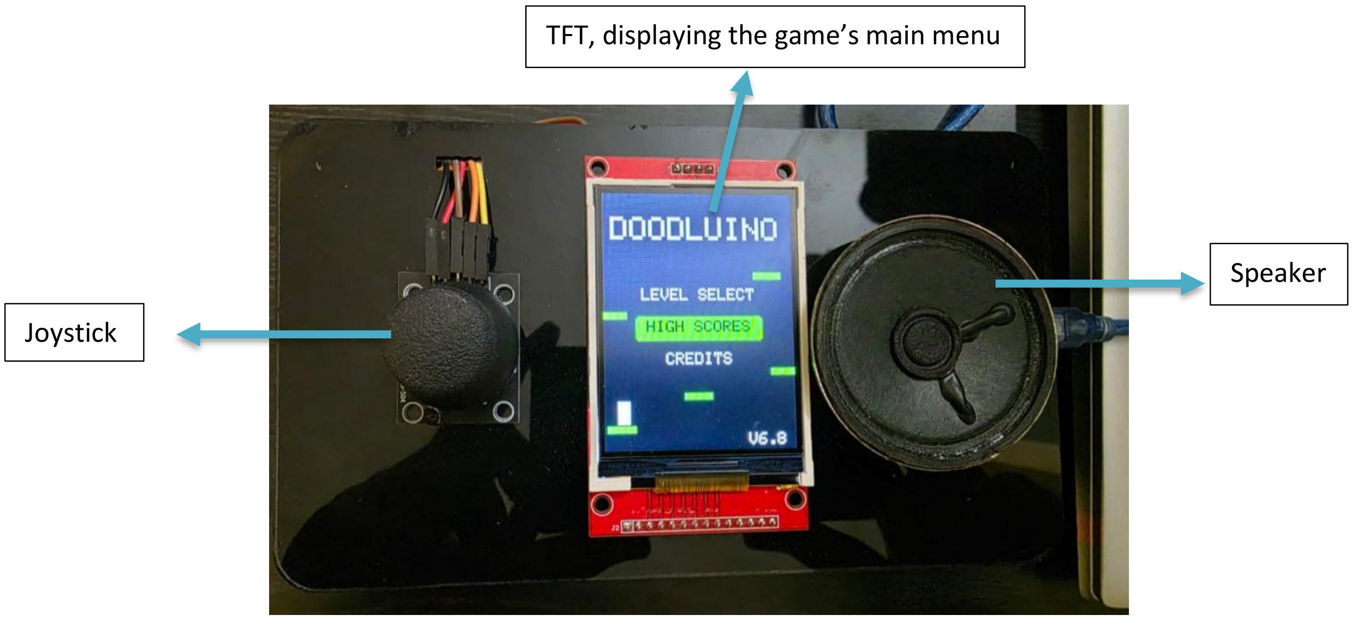
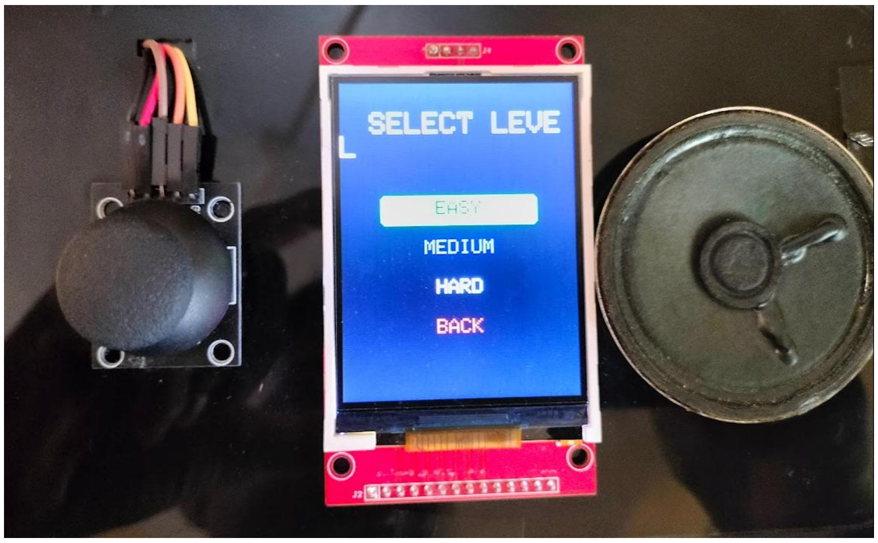
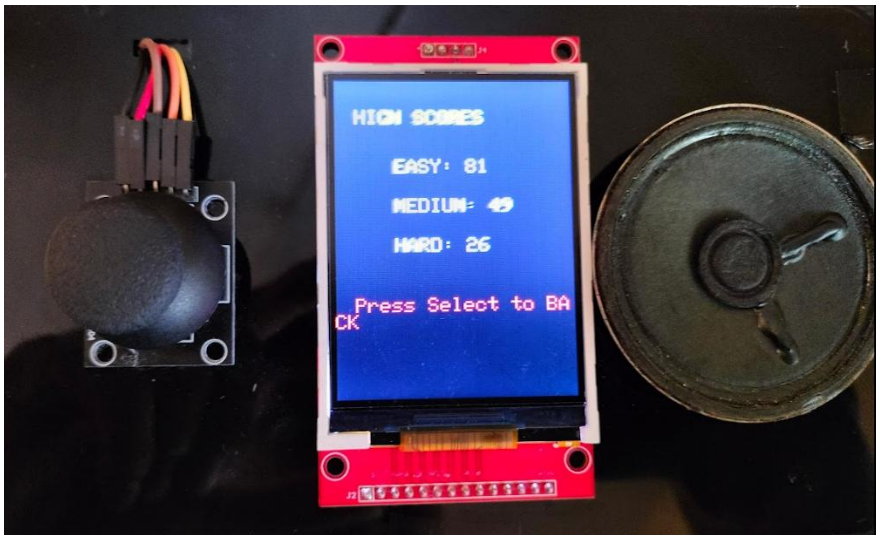


Figure 4: Top of the Console



Select level Menu



High Scores displayed; the values are fetched from EEPROM, so even if we power off and power it back on, the values still remain

- Video Demonstration –

  1. With Gameplay music
     https://www.youtube.com/watch?v=0u_XmoVJk0Y
  2. Without gameplay music
     https://www.youtube.com/watch?v=of0zH08SPe8

Milestones achieved in each week –

1. **Note:** Since my game involved a lot of components - functions, structures, etc., each of which formed an integral part of my code, I maintained a separate log sheet of my updates every week, what I intended to complete, and the glitches and problems that I faced and needed to rectify along the way: https://docs.google.com/spreadsheets/d/124PygBUTDnI5tR8Jvsb0EWS77CMUWYxi1bj8dvN-BO4/edit?usp=sharing
   This turned out to be very useful at the end - I could just check whether I had fixed something or not, since it was very easy to forget very minor, yet hard to fix glitches.

2. **Note:** I also created a GitHub Repository where I have posted all versions of my code, ranging from Version 1.00 to 7.xx: https://github.com/ramanan849/Doodluino

3. **Overview of my progress throughout the project phase, weekly:**

   WEEK 1 –
   A. Formulated a plan of how I am going to build my project – a basic version of what kind of features I wanted to build, how I am going to get going with it.
   B. Familiarized myself with the various Arduino components that would be required.
   C. Explored various Arduino libraries that were capable of displaying animated objects on a TFT display. Chose to use the TFT library provided by Adafruit. Also, went through others, such as the "TFT-eSPI library" (sadly, incompatible with the UNO due to low clock frequency).
   D. Then, wrote a simple program to animate bouncing balls (my video link: https://youtu.be/ppXeX2nozWA)

   WEEK 2 –
   A. Implementing the moving platforms, warp left and right mechanisms, the skeletal features of the game, while optimizing game dynamics to improve animation and FPS

   WEEK 3 –
   A. Designing the in-game graphics - characters and obstacles
   B. Adding logic to store previous data – EEPROM , such as high scores, and obtaining velocity and position data

   WEEK 4 –
   A. Worked on game GUI
   B. Tried fixing the SD card and SPI issues

   WEEK 5 –
   A. Fixed Glitches in the game – for Level Easy, Medium and Hard

B. Worked on using interrupts to play music and play the game simultaneously

WEEK 6 –

A. Fixed glitches
B. Worked on creating a console from acrylic, used CAD to design the model
C. Report

Code –
(As of Tuesday, 08/04/2025, Subject To Change By Demo Day)

```
// v7.0 - Doodle Jump - As of 08/04/2025
// Gokularamanan RS
// https://github.com/ramanan849/doodluino


/*
List of all the functions in the code, with a one-line explanation :
  1. initializeEEPROM()-
  2. setup()
  3. loop()
  4. resetDrawFlags()
  5. drawMainMenu()
  6. drawLevelMenu()
  7. drawHighScores()
  8. drawCredits()
  9. readMenuInput()
  10. handleMenuSelection()
  11. readMenuInput_GameOver()
  12. updatePlatforms()
  13. updateObstacles()
  14. drawGame()
  15. checkCollisions()
  16. initGame()
  17. updateMusic()
  18. handleGameOver()
  19. checkGameOver()
  20. handleScrolling()
*/

#include <SPI.h> // lib to enable SPI connection between the Uno and TFT
#include <Adafruit_GFX.h> // this is the core graphics library - all shapes and
graphics
#include <Adafruit_ILI9341.h> // display driver
#include <EEPROM.h> // EEPROM - electrically erasable programmable read-only memory
//- to store my high scores - Uno, i belive allocates 1 kb of ram to eeprom. I am
just using up 3 * 2 bytes = 6 bytes for eeprom to store 3 integer values for easy,
medium and hard level scores

// --- Pin Definitions ---
// i am using 8 pins on my tft - cs, dc, rst, sck (serial clock), mosi (master out
slave in)
#define TFT_CS 10
#define TFT_DC 9
#define TFT_RST 8
#define JOY_X A5
#define JOY_Y A3
#define JOY_SW 6
#define BUZZER_PIN 2

// --- Game Constants ---
#define SCREEN_WIDTH 240
#define SCREEN_HEIGHT 320
// the display is a 240x320 unit
#define DOODLER_WIDTH 15
#define DOODLER_HEIGHT 28
#define PLATFORM_WIDTH 40
#define PLATFORM_HEIGHT 8
#define NUM_PLATFORMS 7
```

```
#define MAX_SCROLL 5    // Smoother scrolling
#define JUMP_FORCE -12
#define GRAVITY 0.4 // dynamic gravity in updateGame
#define TOP_OFFSET 30   // Height of score panel - the separate one
# define BOTTOM_OFFSET 40
#define PLAY_AREA_HEIGHT (SCREEN_HEIGHT - TOP_OFFSET)
#define VISIBLE_PLATFORMS 7 // Start with 7 platforms
#define BASE_SCROLL 5       // Base scroll speed
#define SCROLL_INCREASE 0.1 // Scroll speed increase per score
#define BASE_GRAVITY 0.4    // Base gravity
#define GRAVITY_INCREASE 0.01 // Gravity increase per score
#define OBSTACLE_SPEED 2


// <<< NEW: Obstacle Constants >>>
#define MAX_OBSTACLES 2
#define OBSTACLE_RADIUS 8
#define OBSTACLE_COLOR RED
#define OBSTACLE_START_SCORE 5
#define OBSTACLE_SPAWN_CHANCE 3 // 3% chance per frame
#define OBSTACLE_SPEED 2
#define MIN_OBSTACLE_DISTANCE 40

// --- Game States ---
#define MAIN_MENU 0
#define LEVEL_MENU 1
#define HIGHSCORES_MENU 2
#define CREDITS_MENU 3
#define GAME_PLAYING 4
#define GAME_OVER_STATE 5



// --- Color Definitions ---
#define BLACK 0x0000
#define WHITE 0xFFFF
#define GREEN 0x07E0
#define BLUE 0x001F
#define RED 0xF800
#define YELLOW 0xFFE0
#define MAGENTA 0xF81F
#define CYAN 0x07FF


#define LEVEL_EASY 0
#define LEVEL_MEDIUM 1
#define LEVEL_HARD 2

// Musical note definitions
#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
```

```
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
#define NOTE_A3  220
#define NOTE_AS3 233
#define NOTE_B3  247
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
#define NOTE_F4  349
#define NOTE_FS4 370
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494
#define NOTE_C5  523
#define NOTE_CS5 554
#define NOTE_D5  587
#define NOTE_DS5 622
#define NOTE_E5  659
#define NOTE_F5  698
#define NOTE_FS5 740
#define NOTE_G5  784
#define NOTE_GS5 831
#define NOTE_A5  880
#define NOTE_AS5 932
#define NOTE_B5  988
#define NOTE_C6  1047
#define NOTE_CS6 1109
#define NOTE_D6  1175
#define NOTE_DS6 1245
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_FS6 1480
#define NOTE_G6  1568
#define NOTE_GS6 1661
#define NOTE_A6  1760
#define NOTE_AS6 1865
#define NOTE_B6  1976
#define NOTE_C7  2093
#define NOTE_CS7 2217
```

```
#define NOTE_D7  2349
#define NOTE_DS7 2489
#define NOTE_E7  2637
#define NOTE_F7  2794
#define NOTE_FS7 2960
#define NOTE_G7  3136
#define NOTE_GS7 3322
#define NOTE_A7  3520
#define NOTE_AS7 3729
#define NOTE_B7  3951
#define NOTE_C8  4186
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978
#define REST      0


enum MusicState { MUSIC_OFF, MUSIC_MAIN, MUSIC_GAME_OVER, MUSIC_CREDITS, MUSIC_HARD
}; // basically like a list/ set to store music realted constants

struct GameState { // the most important part of my game - this structure contains
most of the variables for spawning platforms, obstacles, making animation and
player - arduino interactions possible
  int doodlerX, doodlerY;
  float doodlerVelocityY;
  int platformX[NUM_PLATFORMS];
  int platformY[NUM_PLATFORMS];
  int platformDirection[NUM_PLATFORMS];
  int score = 0;
  bool gameOver = true;
  bool platformUsed[NUM_PLATFORMS];
  int prevDoodlerX, prevDoodlerY;
  int prevPlatformX[NUM_PLATFORMS];
  int prevPlatformY[NUM_PLATFORMS];
  int gameLevel = 0;
  int selectedOption = 0; // Used by original readMenuInput, now new menu too
  bool levelSelected = false;
  int platformX_start;
  int platformY_start;
  bool plat_start_used;
  bool gameStartedByUser;
  int visiblePlatforms;
  float displayGravity;
  // <<< NEW: Menu state variables >>>
  int currentMenu; // Current state (menu, game, etc.)
  int mainMenuSelection = 0; // For main menu navigation
  bool gameIsActive = false; // v6.3
  // <<< NEW: Obstacle variables >>>
  int obstacleX[MAX_OBSTACLES];
  int obstacleY[MAX_OBSTACLES];
  int obstacleDir[MAX_OBSTACLES]; // 1 = right, -1 = left
  bool obstacleActive[MAX_OBSTACLES];
  int prevObstacleX[MAX_OBSTACLES];
  int prevObstacleY[MAX_OBSTACLES];
  // the variables corresponding to music
  MusicState musicState;
  int currentMelodyNote;
  unsigned long previousNoteTime;
  int melodyNoteDuration;
  const int* currentMelody;
  int currentTempo;
  int currentNotesCount;
```

```
};

const int melody[] PROGMEM = {
  // I took the code from https://github.com/robsoncouto/arduino-songs -- TRUE GOAT
  // Super Mario Bros theme
  // Score available at https://musescore.com/user/2123/scores/2145
  // Theme by Koji Kondo
    NOTE_E5,8, NOTE_E5,8, REST,8, NOTE_E5,8, REST,8, NOTE_C5,8, NOTE_E5,8, //1
  NOTE_G5,4, REST,4, NOTE_G4,8, REST,4,
  NOTE_C5,-4, NOTE_G4,8, REST,4, NOTE_E4,-4, // 3
  NOTE_A4,4, NOTE_B4,4, NOTE_AS4,8, NOTE_A4,4,
  NOTE_G4,-8, NOTE_E5,-8, NOTE_G5,-8, NOTE_A5,4, NOTE_F5,8, NOTE_G5,8,
  REST,8, NOTE_E5,4,NOTE_C5,8, NOTE_D5,8, NOTE_B4,-4,
  NOTE_C5,-4, NOTE_G4,8, REST,4, NOTE_E4,-4, // repeats from 3
  NOTE_A4,4, NOTE_B4,4, NOTE_AS4,8, NOTE_A4,4,
  NOTE_G4,-8, NOTE_E5,-8, NOTE_G5,-8, NOTE_A5,4, NOTE_F5,8, NOTE_G5,8,
  REST,8, NOTE_E5,4,NOTE_C5,8, NOTE_D5,8, NOTE_B4,-4,


  REST,4, NOTE_G5,8, NOTE_FS5,8, NOTE_F5,8, NOTE_DS5,4, NOTE_E5,8,//7
  REST,8, NOTE_GS4,8, NOTE_A4,8, NOTE_C4,8, REST,8, NOTE_A4,8, NOTE_C5,8,
NOTE_D5,8,
  REST,4, NOTE_DS5,4, REST,8, NOTE_D5,-4,
  NOTE_C5,2, REST,2,

  REST,4, NOTE_G5,8, NOTE_FS5,8, NOTE_F5,8, NOTE_DS5,4, NOTE_E5,8,//repeats from 7
  REST,8, NOTE_GS4,8, NOTE_A4,8, NOTE_C4,8, REST,8, NOTE_A4,8, NOTE_C5,8,
NOTE_D5,8,
  REST,4, NOTE_DS5,4, REST,8, NOTE_D5,-4,
  NOTE_C5,2, REST,2,

  NOTE_C5,8, NOTE_C5,4, NOTE_C5,8, REST,8, NOTE_C5,8, NOTE_D5,4,//11
  NOTE_E5,8, NOTE_C5,4, NOTE_A4,8, NOTE_G4,2,

  NOTE_C5,8, NOTE_C5,4, NOTE_C5,8, REST,8, NOTE_C5,8, NOTE_D5,8, NOTE_E5,8,//13
  REST,1,
  NOTE_C5,8, NOTE_C5,4, NOTE_C5,8, REST,8, NOTE_C5,8, NOTE_D5,4,
  NOTE_E5,8, NOTE_C5,4, NOTE_A4,8, NOTE_G4,2,
  NOTE_E5,8, NOTE_E5,8, REST,8, NOTE_E5,8, REST,8, NOTE_C5,8, NOTE_E5,4,
  NOTE_G5,4, REST,4, NOTE_G4,4, REST,4,
  NOTE_C5,-4, NOTE_G4,8, REST,4, NOTE_E4,-4, // 19

  NOTE_A4,4, NOTE_B4,4, NOTE_AS4,8, NOTE_A4,4,
  NOTE_G4,-8, NOTE_E5,-8, NOTE_G5,-8, NOTE_A5,4, NOTE_F5,8, NOTE_G5,8,
  REST,8, NOTE_E5,4, NOTE_C5,8, NOTE_D5,8, NOTE_B4,-4,

  NOTE_C5,-4, NOTE_G4,8, REST,4, NOTE_E4,-4, // repeats from 19
  NOTE_A4,4, NOTE_B4,4, NOTE_AS4,8, NOTE_A4,4,
  NOTE_G4,-8, NOTE_E5,-8, NOTE_G5,-8, NOTE_A5,4, NOTE_F5,8, NOTE_G5,8,
  REST,8, NOTE_E5,4, NOTE_C5,8, NOTE_D5,8, NOTE_B4,-4,

  NOTE_E5,8, NOTE_C5,4, NOTE_G4,8, REST,4, NOTE_GS4,4,//23
  NOTE_A4,8, NOTE_F5,4, NOTE_F5,8, NOTE_A4,2,
  NOTE_D5,-8, NOTE_A5,-8, NOTE_A5,-8, NOTE_A5,-8, NOTE_G5,-8, NOTE_F5,-8,

  NOTE_E5,8, NOTE_C5,4, NOTE_A4,8, NOTE_G4,2, //26
  NOTE_E5,8, NOTE_C5,4, NOTE_G4,8, REST,4, NOTE_GS4,4,
  NOTE_A4,8, NOTE_F5,4, NOTE_F5,8, NOTE_A4,2,
  NOTE_B4,8, NOTE_F5,4, NOTE_F5,8, NOTE_F5,-8, NOTE_E5,-8, NOTE_D5,-8,
  NOTE_C5,8, NOTE_E4,4, NOTE_E4,8, NOTE_C4,2,

  NOTE_E5,8, NOTE_C5,4, NOTE_G4,8, REST,4, NOTE_GS4,4,//repeats from 23
  NOTE_A4,8, NOTE_F5,4, NOTE_F5,8, NOTE_A4,2,
  NOTE_D5,-8, NOTE_A5,-8, NOTE_A5,-8, NOTE_A5,-8, NOTE_G5,-8, NOTE_F5,-8,

  NOTE_E5,8, NOTE_C5,4, NOTE_A4,8, NOTE_G4,2, //26
  NOTE_E5,8, NOTE_C5,4, NOTE_G4,8, REST,4, NOTE_GS4,4,
  NOTE_A4,8, NOTE_F5,4, NOTE_F5,8, NOTE_A4,2,
  NOTE_B4,8, NOTE_F5,4, NOTE_F5,8, NOTE_F5,-8, NOTE_E5,-8, NOTE_D5,-8,
  NOTE_C5,8, NOTE_E4,4, NOTE_E4,8, NOTE_C4,2,
  NOTE_C5,8, NOTE_C5,4, NOTE_C5,8, REST,8, NOTE_C5,8, NOTE_D5,8, NOTE_E5,8,
  REST,1,

  NOTE_C5,8, NOTE_C5,4, NOTE_C5,8, REST,8, NOTE_C5,8, NOTE_D5,4, //33
  NOTE_E5,8, NOTE_C5,4, NOTE_A4,8, NOTE_G4,2,
  NOTE_E5,8, NOTE_E5,8, REST,8, NOTE_E5,8, REST,8, NOTE_C5,8, NOTE_E5,4,
  NOTE_G5,4, REST,4, NOTE_G4,4, REST,4,
  NOTE_E5,8, NOTE_C5,4, NOTE_G4,8, REST,4, NOTE_GS4,4,
  NOTE_A4,8, NOTE_F5,4, NOTE_F5,8, NOTE_A4,2,
  NOTE_D5,-8, NOTE_A5,-8, NOTE_A5,-8, NOTE_A5,-8, NOTE_G5,-8, NOTE_F5,-8,

  NOTE_E5,8, NOTE_C5,4, NOTE_A4,8, NOTE_G4,2, //40
  NOTE_E5,8, NOTE_C5,4, NOTE_G4,8, REST,4, NOTE_GS4,4,
  NOTE_A4,8, NOTE_F5,4, NOTE_F5,8, NOTE_A4,2,
  NOTE_B4,8, NOTE_F5,4, NOTE_F5,8, NOTE_F5,-8, NOTE_E5,-8, NOTE_D5,-8,
  NOTE_C5,8, NOTE_E4,4, NOTE_E4,8, NOTE_C4,2,
};

const int gameOver[] PROGMEM = {
  // Super Mario's gameOver
  // I took the code from https://github.com/robsoncouto/arduino-songs -- TRUE GOAT
  NOTE_C5,-4, NOTE_G4,-4, NOTE_E4,4, //45
  NOTE_A4,-8, NOTE_B4,-8, NOTE_A4,-8, NOTE_GS4,-8, NOTE_AS4,-8, NOTE_GS4,-8,
  NOTE_G4,8, NOTE_D4,8, NOTE_E4,-2,
};

const int hardMelody[] PROGMEM = {
  // credits: Beethoven
  // I took the code from https://github.com/robsoncouto/arduino-songs -- TRUE GOAT
  NOTE_E5, 16, NOTE_DS5, 16, //1
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, -8, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, -8, NOTE_E4, 16, NOTE_GS4, 16, NOTE_B4, 16,
  NOTE_C5, 8,  REST, 16, NOTE_E4, 16, NOTE_E5, 16,  NOTE_DS5, 16,

  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,//6
  NOTE_A4, -8, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, -8, NOTE_E4, 16, NOTE_C5, 16, NOTE_B4, 16,
  NOTE_A4 , 4, REST, 8, //9 - 1st ending

  //repaets from 1 ending on 10
  NOTE_E5, 16, NOTE_DS5, 16, //1
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, -8, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, -8, NOTE_E4, 16, NOTE_GS4, 16, NOTE_B4, 16,
  NOTE_C5, 8,  REST, 16, NOTE_E4, 16, NOTE_E5, 16,  NOTE_DS5, 16,

  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,//6
  NOTE_A4, -8, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, -8, NOTE_E4, 16, NOTE_C5, 16, NOTE_B4, 16,
  NOTE_A4, 8, REST, 16, NOTE_B4, 16, NOTE_C5, 16, NOTE_D5, 16, //10 - 2nd ending
  //continues from 11
  NOTE_E5, -8, NOTE_G4, 16, NOTE_F5, 16, NOTE_E5, 16,
  NOTE_D5, -8, NOTE_F4, 16, NOTE_E5, 16, NOTE_D5, 16, //12
  NOTE_C5, -8, NOTE_E4, 16, NOTE_D5, 16, NOTE_C5, 16, //13
  NOTE_B4, 8, REST, 16, NOTE_E4, 16, NOTE_E5, 16, REST, 16,
  REST, 16, NOTE_E5, 16, NOTE_E6, 16, REST, 16, REST, 16, NOTE_DS5, 16,
  NOTE_E5, 16, REST, 16, REST, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_DS5, 16,
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, 8, REST, 16, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,

  NOTE_B4, 8, REST, 16, NOTE_E4, 16, NOTE_GS4, 16, NOTE_B4, 16, //19
  NOTE_C5, 8, REST, 16, NOTE_E4, 16, NOTE_E5, 16,  NOTE_DS5, 16,
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, 8, REST, 16, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, 8, REST, 16, NOTE_E4, 16, NOTE_C5, 16, NOTE_B4, 16,
  NOTE_A4, 8, REST, 16, NOTE_B4, 16, NOTE_C5, 16, NOTE_D5, 16, //24 (1st ending)

  //repeats from 11
  NOTE_E5, -8, NOTE_G4, 16, NOTE_F5, 16, NOTE_E5, 16,
  NOTE_D5, -8, NOTE_F4, 16, NOTE_E5, 16, NOTE_D5, 16, //12

  NOTE_C5, -8, NOTE_E4, 16, NOTE_D5, 16, NOTE_C5, 16, //13
  NOTE_B4, 8, REST, 16, NOTE_E4, 16, NOTE_E5, 16, REST, 16,
  REST, 16, NOTE_E5, 16, NOTE_E6, 16, REST, 16, REST, 16, NOTE_DS5, 16,
  NOTE_E5, 16, REST, 16, REST, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_DS5, 16,
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, 8, REST, 16, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,

  NOTE_B4, 8, REST, 16, NOTE_E4, 16, NOTE_GS4, 16, NOTE_B4, 16, //19
  NOTE_C5, 8, REST, 16, NOTE_E4, 16, NOTE_E5, 16,  NOTE_DS5, 16,
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, 8, REST, 16, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, 8, REST, 16, NOTE_E4, 16, NOTE_C5, 16, NOTE_B4, 16,
  NOTE_A4, 8, REST, 16, NOTE_C5, 16, NOTE_C5, 16, NOTE_C5, 16, //25 - 2nd ending

  //continues from 26
  NOTE_C5 , 4, NOTE_F5, -16, NOTE_E5, 32, //26
  NOTE_E5, 8, NOTE_D5, 8, NOTE_AS5, -16, NOTE_A5, 32,
  NOTE_A5, 16, NOTE_G5, 16, NOTE_F5, 16, NOTE_E5, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_AS4, 8, NOTE_A4, 8, NOTE_A4, 32, NOTE_G4, 32, NOTE_A4, 32, NOTE_B4, 32,
  NOTE_C5 , 4, NOTE_D5, 16, NOTE_DS5, 16,
  NOTE_E5, -8, NOTE_E5, 16, NOTE_F5, 16, NOTE_A4, 16,
  NOTE_C5 , 4,  NOTE_D5, -16, NOTE_B4, 32,


  NOTE_C5, 32, NOTE_G5, 32, NOTE_G4, 32, NOTE_G5, 32, NOTE_A4, 32, NOTE_G5, 32,
NOTE_B4, 32, NOTE_G5, 32, NOTE_C5, 32, NOTE_G5, 32, NOTE_D5, 32, NOTE_G5, 32, //33
  NOTE_E5, 32, NOTE_G5, 32, NOTE_C6, 32, NOTE_B5, 32, NOTE_A5, 32, NOTE_G5, 32,
NOTE_F5, 32, NOTE_E5, 32, NOTE_D5, 32, NOTE_G5, 32, NOTE_F5, 32, NOTE_D5, 32,
  NOTE_C5, 32, NOTE_G5, 32, NOTE_G4, 32, NOTE_G5, 32, NOTE_A4, 32, NOTE_G5, 32,
NOTE_B4, 32, NOTE_G5, 32, NOTE_C5, 32, NOTE_G5, 32, NOTE_D5, 32, NOTE_G5, 32,

  NOTE_E5, 32, NOTE_G5, 32, NOTE_C6, 32, NOTE_B5, 32, NOTE_A5, 32, NOTE_G5, 32,
NOTE_F5, 32, NOTE_E5, 32, NOTE_D5, 32, NOTE_G5, 32, NOTE_F5, 32, NOTE_D5, 32, //36
  NOTE_E5, 32, NOTE_F5, 32, NOTE_E5, 32, NOTE_DS5, 32, NOTE_E5, 32, NOTE_B4, 32,
NOTE_E5, 32, NOTE_DS5, 32, NOTE_E5, 32, NOTE_B4, 32, NOTE_E5, 32, NOTE_DS5, 32,
  NOTE_E5, -8, NOTE_B4, 16, NOTE_E5, 16, NOTE_DS5, 16,
  NOTE_E5, -8, NOTE_B4, 16, NOTE_E5, 16, REST, 16,

  REST, 16, NOTE_DS5, 16, NOTE_E5, 16, REST, 16, REST, 16, NOTE_DS5, 16, //40
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, 8, REST, 16, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, 8, REST, 16, NOTE_E4, 16, NOTE_GS4, 16, NOTE_B4, 16,
  NOTE_C5, 8, REST, 16, NOTE_E4, 16, NOTE_E5, 16, NOTE_DS5, 16,
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,

  NOTE_A4, 8, REST, 16, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16, //46
  NOTE_B4, 8, REST, 16, NOTE_E4, 16, NOTE_C5, 16, NOTE_B4, 16,
  NOTE_A4, 8, REST, 16, NOTE_B4, 16, NOTE_C5, 16, NOTE_D5, 16,
  NOTE_E5, -8, NOTE_G4, 16, NOTE_F5, 16, NOTE_E5, 16,
  NOTE_D5, -8, NOTE_F4, 16, NOTE_E5, 16, NOTE_D5, 16,
  NOTE_C5, -8, NOTE_E4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_B4, 8, REST, 16, NOTE_E4, 16, NOTE_E5, 16, REST, 16,
  REST, 16, NOTE_E5, 16, NOTE_E6, 16, REST, 16, REST, 16, NOTE_DS5, 16,

  NOTE_E5, 16, REST, 16, REST, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_D5, 16, //54
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, 8, REST, 16, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, 8, REST, 16, NOTE_E4, 16, NOTE_GS4, 16, NOTE_B4, 16,
  NOTE_C5, 8, REST, 16, NOTE_E4, 16, NOTE_E5, 16, NOTE_DS5, 16,
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,

  NOTE_A4, 8, REST, 16, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16, //60
  NOTE_B4, 8, REST, 16, NOTE_E4, 16, NOTE_C5, 16, NOTE_B4, 16,
  NOTE_A4, 8, REST, 16, REST, 16, REST, 8,
  NOTE_CS5 , -4,
  NOTE_D5 , 4, NOTE_E5, 16, NOTE_F5, 16,
  NOTE_F5 , 4, NOTE_F5, 8,
  NOTE_E5 , -4,
  NOTE_D5 , 4, NOTE_C5, 16, NOTE_B4, 16,
  NOTE_A4 , 4, NOTE_A4, 8,
  NOTE_A4, 8, NOTE_C5, 8, NOTE_B4, 8,
  NOTE_A4 , -4,
  NOTE_CS5 , -4,

  NOTE_D5 , 4, NOTE_E5, 16, NOTE_F5, 16, //72
  NOTE_F5 , 4, NOTE_F5, 8,
  NOTE_F5 , -4,
  NOTE_DS5 , 4, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_AS4 , 4, NOTE_A4, 8,
  NOTE_GS4 , 4, NOTE_G4, 8,
  NOTE_A4 , -4,
  NOTE_B4 , 4, REST, 8,
  NOTE_A3, -32, NOTE_C4, -32, NOTE_E4, -32, NOTE_A4, -32, NOTE_C5, -32, NOTE_E5, -
32, NOTE_D5, -32, NOTE_C5, -32, NOTE_B4, -32,

  NOTE_A4, -32, NOTE_C5, -32, NOTE_E5, -32, NOTE_A5, -32, NOTE_C6, -32, NOTE_E6, -
32, NOTE_D6, -32, NOTE_C6, -32, NOTE_B5, -32, //80
  NOTE_A4, -32, NOTE_C5, -32, NOTE_E5, -32, NOTE_A5, -32, NOTE_C6, -32, NOTE_E6, -
32, NOTE_D6, -32, NOTE_C6, -32, NOTE_B5, -32,
  NOTE_AS5, -32, NOTE_A5, -32, NOTE_GS5, -32, NOTE_G5, -32, NOTE_FS5, -32, NOTE_F5,
-32, NOTE_E5, -32, NOTE_DS5, -32, NOTE_D5, -32,

  NOTE_CS5, -32, NOTE_C5, -32, NOTE_B4, -32, NOTE_AS4, -32, NOTE_A4, -32, NOTE_GS4,
-32, NOTE_G4, -32, NOTE_FS4, -32, NOTE_F4, -32, //84
  NOTE_E4, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, -8, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, -8, NOTE_E4, 16, NOTE_GS4, 16, NOTE_B4, 16,

  NOTE_C5, 8, REST, 16, NOTE_E4, 16, NOTE_E5, 16, NOTE_DS5, 16, //88
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, -8, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, -8, NOTE_E4, 16, NOTE_C5, 16, NOTE_B4, 16,
  NOTE_A4, -8, REST, -8,
  REST, -8, NOTE_G4, 16, NOTE_F5, 16, NOTE_E5, 16,
  NOTE_D5 , 4, REST, 8,
  REST, -8, NOTE_E4, 16, NOTE_D5, 16, NOTE_C5, 16,

  NOTE_B4, -8, NOTE_E4, 16, NOTE_E5, 8, //96
  NOTE_E5, 8, NOTE_E6, -8, NOTE_DS5, 16,
  NOTE_E5, 16, REST, 16, REST, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_DS5, 16,
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, -8, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, -8, NOTE_E4, 16, NOTE_GS4, 16, NOTE_B4, 16,

  NOTE_C5, 8, REST, 16, NOTE_E4, 16, NOTE_E5, 16, NOTE_DS5, 16, //102
  NOTE_E5, 16, NOTE_DS5, 16, NOTE_E5, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_C5, 16,
  NOTE_A4, -8, NOTE_C4, 16, NOTE_E4, 16, NOTE_A4, 16,
  NOTE_B4, -8, NOTE_E4, 16, NOTE_C5, 16, NOTE_B4, 16,
  NOTE_A4 , -4,
};

const int creditMelody[] PROGMEM = {
  // credits : Never Gonna Give You Up, Rick AStley
  // I took the code from https://github.com/robsoncouto/arduino-songs -- TRUE GOAT
  REST,8, NOTE_B4,8, NOTE_B4,8, NOTE_CS5,8, NOTE_D5,8, NOTE_B4,4, NOTE_A4,8, //7
  NOTE_A5,8, REST,8, NOTE_A5,8, NOTE_E5,-4, REST,4,
  NOTE_B4,8, NOTE_B4,8, NOTE_CS5,8, NOTE_D5,8, NOTE_B4,8, NOTE_D5,8, NOTE_E5,8,
REST,8,
  REST,8, NOTE_CS5,8, NOTE_B4,8, NOTE_A4,-4, REST,4,
  REST,8, NOTE_B4,8, NOTE_B4,8, NOTE_CS5,8, NOTE_D5,8, NOTE_B4,8, NOTE_A4,4,
  NOTE_E5,8, NOTE_E5,8, NOTE_E5,8, NOTE_FS5,8, NOTE_E5,4, REST,4,

};


int hard_tempo = 80;
int tempo = 200; // for both mario and game over
int credit_tempo = 114;

int wholenote_mario = (60000 * 4) / tempo;
int wholenote_hard = (60000 * 4) / hard_tempo;
int wholenote_credit = (60000 * 4) / credit_tempo;
int wholenote_gameover = (60000 * 4) / tempo;
int notes = sizeof(melody) / sizeof(melody[0]) / 2; // for mario
int notes_hard = sizeof(hardMelody) / sizeof(hardMelody[0]) / 2;
int notes_gameover = sizeof(gameOver) / sizeof(gameOver[0]) / 2;
int notes_credit = sizeof(creditMelody) / sizeof(hardMelody[0]) / 2;



const int easyAddress = 0;
const int medAddress = 2; // Assuming sizeof(int) == 2 on target
const int highAddress = 4;
const int INIT_MARKER_ADDR = 6;
const float HOMING_FACTOR = 0.8;

// --- Global Variables () ---
bool beat_easy = 0; // Used in original handleGameOver - ref v5
bool beat_med = 0;  // Used in original handleGameOver - ref v5
bool beat_hard = 0; // Used in original handleGameOver - ref v5

Adafruit_ILI9341 tft(TFT_CS, TFT_DC, TFT_RST); // this declr initalizes the tft
display -
//basically, the white startup and a then flashdown animation that happens at the
start is due to this
```

```
// --- Moving Platform Constants () ---
const int PLATFORM_MOVE_SPEED = 1;
const int PLATFORM_MOVE_RANGE = 50; // movement range

// <<< NEW: Flags for menu drawing state >>>
static bool mainMenuFirstDraw = true;
static bool levelMenuFirstDraw = true;
static bool highScoresDrawn = false;
static bool creditsDrawn = false;
static bool gameOverMsgDrawn = false;


// === EEPROM Function (Original ) ===
void initializeEEPROM() {
  //Serial.println("Initializing EEPROM check...");
  if (EEPROM.read(INIT_MARKER_ADDR) != 0x33) { // Check new marker address
    //Serial.println("Initializing EEPROM with default high score values");
    EEPROM.put(easyAddress, highScore.easy_score);
    EEPROM.put(medAddress, highScore.med_score);
    EEPROM.put(highAddress, highScore.hard_score);
    EEPROM.write(INIT_MARKER_ADDR, 0x33); // Write marker to new address
  } else {
    //Serial.println("EEPROM already initialized.");
    EEPROM.get(easyAddress, highScore.easy_score);
    //Serial.print("Easy Level High Score = ");
Serial.println(highScore.easy_score);
    EEPROM.get(medAddress, highScore.med_score);
    //Serial.print("Med Level High Score = "); Serial.println(highScore.med_score);
    EEPROM.get(highAddress, highScore.hard_score);
    //Serial.print("Hard Level High Score = ");
Serial.println(highScore.hard_score);
  }
}

// === Setup Function (MODIFIED for State Machine) ===
void setup() {
  //Serial.begin(115200);
  //Serial.println("v5 Base Code + New Menu Setup..."); // Identify version

  // Hardware Init ()
  pinMode(TFT_RST, OUTPUT);
  pinMode(JOY_SW, INPUT_PULLUP);
  digitalWrite(TFT_RST, LOW); delay(10); digitalWrite(TFT_RST, HIGH); delay(10);

  tft.begin();
  tft.setRotation(0);
  SPI.setClockDivider(SPI_CLOCK_DIV2); // Keep as per  code

  // EEPROM Init ()
  initializeEEPROM();
  // EEPROM gets  (redundant if initializeEEPROM does it, but keep if intended)
  EEPROM.get(easyAddress, highScore.easy_score);
  EEPROM.get(medAddress, highScore.med_score);
  EEPROM.get(highAddress, highScore.hard_score);




  // <<< NEW: Initialize state machine >>>
  state.currentMenu = MAIN_MENU; // Start at main menu
  state.mainMenuSelection = 0;
  state.gameOver = true; // Ensure game isn't running initially
  state.levelSelected = false; // Reset flag

  randomSeed(analogRead(A0)); // Seed random generator (use a floating pin like A0)
}

// === Main Loop (REPLACED with State Machine) ===
void loop() {
  updateMusic();
  switch (state.currentMenu) {
    case MAIN_MENU:
      drawMainMenu();
      readMenuInput(); // New non-blocking input handler
      break;
    case LEVEL_MENU:
      drawLevelMenu(); // New non-blocking draw handler
      readMenuInput(); // New non-blocking input handler
      break;
    case HIGHSCORES_MENU:
      drawHighScores();
      readMenuInput();
      break;
    case CREDITS_MENU:
      drawCredits();
      readMenuInput();
      break;
    case GAME_PLAYING:
      // Use original game loop structure
      if (!state.gameOver) {
          // Original throttle logic
          static uint32_t lastUpdate = 0;
          if (millis() - lastUpdate < 33) return; // Original throttle
          lastUpdate = millis();

          updateGame(); // Calls ORIGINAL updateGame
          drawGame();   // Calls ORIGINAL drawGame
      } else {
          // If gameOver becomes true, transition state
          // updateHighScore logic moved into handleGameOver
          state.currentMenu = GAME_OVER_STATE;
          resetDrawFlags();
      }
      break;
    case GAME_OVER_STATE:
      handleGameOver();         // Display score, update EEPROM (Adapted from
original)
      readMenuInput_GameOver(); // Wait for input to return to menu
      break;
  }
}


// === NEW Menu Drawing Functions ===
void resetDrawFlags() {
    mainMenuFirstDraw = true; levelMenuFirstDraw = true;
    highScoresDrawn = false; creditsDrawn = false; gameOverMsgDrawn = false;
}

void drawMainMenu() {
  static int lastSelection = -1;
  if (state.mainMenuSelection != lastSelection || mainMenuFirstDraw) {
    if (mainMenuFirstDraw) {
      tft.fillScreen(BLACK); tft.setTextColor(WHITE); tft.setTextSize(4);
      tft.setCursor(10, 30); tft.print("DOODLUINO"); tft.setTextSize(2);
      mainMenuFirstDraw = false;
    } else { tft.fillRect(35, 110, 170, 130, BLACK); }

    const char* menuItems[] = {"LEVEL SELECT", "HIGH SCORES", "CREDITS"};
    for (int i = 0; i < 3; i++) {
      int yPos = 120 + (i * 40);
      uint16_t fgColor = WHITE, bgColor = GREEN;
      if (i == state.mainMenuSelection) { tft.fillRoundRect(40, yPos - 5, 160, 30,
5, bgColor); fgColor = BLACK; }
      tft.setTextColor(fgColor);
      int textWidth = strlen(menuItems[i]) * 6 * 2;
      tft.setCursor(SCREEN_WIDTH / 2 - textWidth / 2, yPos);
tft.print(menuItems[i]);
    }
    lastSelection = state.mainMenuSelection;
  }
  tft.fillRect(-3,150,PLATFORM_WIDTH-8, PLATFORM_HEIGHT, GREEN);
  tft.fillRect(190,100,PLATFORM_WIDTH-5, PLATFORM_HEIGHT, GREEN);
  tft.fillRect(100,250,PLATFORM_WIDTH-5, PLATFORM_HEIGHT, GREEN); // dont - oh
no!!!
  tft.fillRect(4,290,PLATFORM_WIDTH-5, PLATFORM_HEIGHT, GREEN);
  tft.fillRect(150,340,PLATFORM_WIDTH-5, PLATFORM_HEIGHT, GREEN);
  tft.fillRect(210,220,PLATFORM_WIDTH-5, PLATFORM_HEIGHT, GREEN);
  tft.fillRect(17, 260 ,DOODLER_WIDTH, DOODLER_HEIGHT, WHITE);



  tft.setCursor(180,300);
  tft.print("V7.0");
}

void drawLevelMenu() {
  static int lastSelection = -1;
  const int numOptions = 4; // Easy, Med, Hard, Back
  if (state.selectedOption != lastSelection || levelMenuFirstDraw) {
    if (levelMenuFirstDraw) {
       tft.fillScreen(BLACK); tft.setTextColor(WHITE); tft.setTextSize(3);
       tft.setCursor(30, 30); tft.print("SELECT LEVEL"); tft.setTextSize(2);
       levelMenuFirstDraw = false;
    } else { tft.fillRect(35, 110, 170, 170, BLACK); } // Clear menu area + back

    for (int i = 0; i < numOptions; i++) { // Loop includes Back option
      int yPos = 120 + (i * 40);
      bool isSelected = (i == state.selectedOption);
      uint16_t color = BLUE; // Default highlight for Back
      uint16_t textColor = WHITE;
      const char* optionText = "";

      if (isSelected) {
         if (i == LEVEL_EASY) color = GREEN; else if (i == LEVEL_MEDIUM) color =
YELLOW; else if (i == LEVEL_HARD) color = RED;
         tft.fillRoundRect(40, yPos - 5, 160, 30, 5, color);
      }

      switch (i) {
        case LEVEL_EASY: optionText = "EASY"; textColor = (isSelected ? BLACK :
WHITE); break;
        case LEVEL_MEDIUM: optionText = "MEDIUM"; textColor = (isSelected ? BLACK :
BLUE); break;
```

```
        case LEVEL_HARD: optionText = "HARD"; textColor = (isSelected ? BLACK :
WHITE); break;
        case 3: optionText = "BACK"; textColor = (isSelected ? WHITE : RED); break;
      }
      tft.setTextColor(textColor);
      int textWidth = strlen(optionText) * 6 * 2;
      tft.setCursor(SCREEN_WIDTH / 2 - textWidth / 2, yPos); tft.print(optionText);
    }
    lastSelection = state.selectedOption;
  }
}

void drawHighScores() {
  if (!highScoresDrawn) {
      tft.fillScreen(BLACK); tft.setTextColor(WHITE); tft.setTextSize(2);
      tft.setCursor(20, 30); tft.print("HIGH SCORES");
      tft.setCursor(60, 80); tft.print("EASY: "); tft.print(highScore.easy_score);
      tft.setCursor(60, 120); tft.print("MEDIUM: ");
tft.print(highScore.med_score);
      tft.setCursor(60, 160); tft.print("HARD: "); tft.print(highScore.hard_score);
      tft.setTextColor(RED); tft.setCursor(20, 220); tft.print("Press Select to
BACK");
      highScoresDrawn = true;
  }
}

void drawCredits() {
  if (!creditsDrawn) {
      tft.fillScreen(BLACK); tft.setTextColor(WHITE); tft.setTextSize(3);
      tft.setCursor(60, 50); tft.print("CREDITS"); tft.setTextSize(2);
      tft.setCursor(30, 120); tft.print("Game By: LUKOG"); //  name

      tft.setTextColor(RED); tft.setCursor(20, 220); tft.print("Press Select to
BACK");
      creditsDrawn = true;
   }
}

// === NEW Input Handling Functions ===
void readMenuInput() {
  static uint32_t lastInputTime = 0; uint32_t now = millis(); if (now -
lastInputTime < 180) return;
  int yValue = analogRead(JOY_Y); int swValue = digitalRead(JOY_SW); bool
inputProcessed = false; int currentSelection = 0; int maxOption = 0;

  switch(state.currentMenu) {
      case MAIN_MENU: currentSelection = state.mainMenuSelection; maxOption = 2;
break;
      case LEVEL_MENU: currentSelection = state.selectedOption; maxOption = 3;
break;
      case HIGHSCORES_MENU: case CREDITS_MENU: break; default: return;
  }
  if (state.currentMenu == MAIN_MENU || state.currentMenu == LEVEL_MENU) {
      // Use thresholds consistent with original readMenuInput if different
      if (yValue > 600) { currentSelection++; if (currentSelection > maxOption)
currentSelection = 0; inputProcessed = true; }
      else if (yValue < 400) { currentSelection--; if (currentSelection < 0)
currentSelection = maxOption; inputProcessed = true; }
      if (inputProcessed) { if(state.currentMenu == MAIN_MENU)
state.mainMenuSelection = currentSelection; else if (state.currentMenu ==
LEVEL_MENU) state.selectedOption = currentSelection; lastInputTime = now; }
  }
  if (swValue == LOW) { handleMenuSelection(); lastInputTime = now + 250; } //
Debounce after select
}

void handleMenuSelection() {
  int previousMenu = state.currentMenu;
  switch(state.currentMenu) {
      case MAIN_MENU:
          switch(state.mainMenuSelection) {
              case 0: state.currentMenu = LEVEL_MENU; state.selectedOption = 0;
break;
              case 1: state.currentMenu = HIGHSCORES_MENU; break;
              case 2: state.currentMenu = CREDITS_MENU;
                      state.currentMenu = CREDITS_MENU;
                                          state.musicState = MUSIC_CREDITS;
                                          state.currentMelody = creditMelody;
                                          state.currentTempo = credit_tempo;
                                          state.currentNotesCount =
sizeof(creditMelody)/sizeof(creditMelody[0])/2;
                                          state.currentMelodyNote = 0;
                                          break;
          } break;
      case LEVEL_MENU:
          if (state.selectedOption == 3) { state.currentMenu = MAIN_MENU;
state.mainMenuSelection = 0; }
          else { state.gameLevel = state.selectedOption; state.levelSelected =
true; state.currentMenu = GAME_PLAYING; tft.fillScreen(RED);
                  delay(2);
                  tft.fillScreen(YELLOW);
                  delay(2);
                  tft.fillScreen(GREEN);
                  delay(2);
                  tft.fillScreen(BLACK);initGame(); } // Calls ORIGINAL initGame
          break;
      case HIGHSCORES_MENU: case CREDITS_MENU: state.currentMenu = MAIN_MENU;
state.musicState = MUSIC_OFF; noTone(BUZZER_PIN); state.mainMenuSelection =
(previousMenu == HIGHSCORES_MENU) ? 1 : 2; break;
  }
  if (state.currentMenu != previousMenu) { resetDrawFlags(); }
}

void readMenuInput_GameOver() {
    static uint32_t lastInputTime_GO = 0; uint32_t now = millis(); if (now -
lastInputTime_GO < 400) return;
    if (digitalRead(JOY_SW) == LOW) { state.currentMenu = MAIN_MENU;
state.mainMenuSelection = 0; state.gameOver = true; lastInputTime_GO = now;
resetDrawFlags(); }
}



void updateGame() { // Modify existing updateGame

  // Store previous positions first
  state.prevDoodlerX = state.doodlerX;
  state.prevDoodlerY = state.doodlerY;

  // <<< NEW: Check if the game is active >>>
  if (!state.gameIsActive) {
    // --- PRE-GAME IDLE STATE ---

    // 1. Check for Start Trigger (Joystick Button Press)
    if (digitalRead(JOY_SW) == LOW) {
        state.gameIsActive = true; // Start the game!
        state.doodlerVelocityY = JUMP_FORCE; // Apply initial full jump force
        // Serial.println("Game Started!"); // Debug message

        return; // Skip rest of update for this frame
    }

    // 2. Idle Bounce Physics (simple bounce on platform 0)
    const float IDLE_GRAVITY = 0.1f; // Very weak gravity for idle
    const float IDLE_JUMP = -1.5f;   // Small upward bounce velocity

    state.doodlerVelocityY += IDLE_GRAVITY; // Apply weak gravity
    state.doodlerY += state.doodlerVelocityY; // Update position

    // Collision check ONLY with platform 0 (the starting one)
    int i = 0; // Index of the starting platform
    bool xOverlap = (state.doodlerX + DOODLER_WIDTH > state.platformX[i]) &&
                    (state.doodlerX < state.platformX[i] + PLATFORM_WIDTH);
    // Simplified Y check for idle bounce (are feet at or below platform top?)
    bool yLanded = (state.doodlerY + DOODLER_HEIGHT >= state.platformY[i]);

    // Apply bounce only if falling and landed
    if (xOverlap && yLanded && state.doodlerVelocityY >= 0) {
        state.doodlerVelocityY = IDLE_JUMP; // Apply small bounce
        state.doodlerY = state.platformY[i] - DOODLER_HEIGHT; // Snap to top
    }

    // Make sure regular platforms update their prev positions but don't move
    // Call original updatePlatforms but it should ideally do nothing if
level==easy
    // or if platformDirection is 0 for the relevant platforms
    updatePlatforms(); // Keep original call - needed to update prevX/Y for drawing

  } else {
    // --- ACTIVE GAME STATE

    // Input handling (Horizontal)
    int joy = analogRead(JOY_X);
    // Use the horizontal control logic from  v5 code:
    state.doodlerX += (joy < 400) ? +5 : (joy > 600) ? -5 : 0; // Assuming this is
v5 logic
    // Screen wrap
    if (state.doodlerX < 0) state.doodlerX = SCREEN_WIDTH - DOODLER_WIDTH;
    if (state.doodlerX > SCREEN_WIDTH - DOODLER_WIDTH) state.doodlerX = 0;

    // Physics (Normal Gravity - using  original dynamic gravity)
    float dynamicGravity = BASE_GRAVITY + (state.score * GRAVITY_INCREASE);
    state.displayGravity = dynamicGravity;
    state.doodlerVelocityY += dynamicGravity;
    // Apply terminal velocity (optional)
    // state.doodlerVelocityY = min(state.doodlerVelocityY, 15.0f);
    state.doodlerY += state.doodlerVelocityY;

    // Standard Game Logic Calls ( Original Functions)
    updateObstacles();
    checkCollisions();
    handleScrolling();
    updatePlatforms(); // Platforms will move now based on level/direction
    checkGameOver();
  }
```

```
}

void updatePlatforms() {
  int moveSpeed = PLATFORM_MOVE_SPEED;

  // Adjust speed based on level and score
  if (state.gameLevel == LEVEL_HARD) {
    moveSpeed += state.score * 0.25;
  }

  for (int i = 0; i < NUM_PLATFORMS; i++) {
    if (state.gameLevel != LEVEL_EASY) {
      state.platformX[i] += moveSpeed * state.platformDirection[i];
      // Boundary check
      if (state.platformX[i] <= 0) {
        state.platformDirection[i] = 1;
        state.platformX[i] = 0;
      } else if (state.platformX[i] >= SCREEN_WIDTH - PLATFORM_WIDTH) {
        state.platformDirection[i] = -1;
        state.platformX[i] = SCREEN_WIDTH - PLATFORM_WIDTH;
      }
    }
  }
}



void updateObstacles() {
  // Calculate current max obstacles based on score (1 + 1 per 10 points)
  int currentMaxObstacles = 1 + (state.score / 10);
  currentMaxObstacles = min(currentMaxObstacles, MAX_OBSTACLES);

  // Spawn new obstacles
  if (state.score >= OBSTACLE_START_SCORE && random(100) < OBSTACLE_SPAWN_CHANCE) {
    for (int i = 0; i < currentMaxObstacles; i++) { // Only check allowed slots
      if (!state.obstacleActive[i]) {
        state.obstacleActive[i] = true;
        state.obstacleX[i] = random(OBSTACLE_RADIUS, SCREEN_WIDTH -
OBSTACLE_RADIUS);

        // Homing effect only in Hard mode
        if (state.gameLevel == LEVEL_HARD) {
          float horizontalDifference = state.doodlerX - state.obstacleX[i];
          if (horizontalDifference > 5) {
            state.obstacleX[i] += HOMING_FACTOR;
          }
          else if (horizontalDifference < -5) {
            state.obstacleX[i] -= HOMING_FACTOR;
          }
          // Keep within bounds after adjustment
          state.obstacleX[i] = constrain(state.obstacleX[i],
            OBSTACLE_RADIUS,
            SCREEN_WIDTH - OBSTACLE_RADIUS);
        }

        state.obstacleY[i] = random(-PLATFORM_HEIGHT+20, 0);
        state.obstacleDir[i] = (random(2) ? 1 : -1);
        state.prevObstacleX[i] = state.obstacleX[i];
        state.prevObstacleY[i] = state.obstacleY[i];
        break;
      }
    }
  }

  // Move active obstacles - only in Hard mode
  for (int i = 0; i < MAX_OBSTACLES; i++) {
    if (state.obstacleActive[i]) {
      state.prevObstacleX[i] = state.obstacleX[i];
      state.prevObstacleY[i] = state.obstacleY[i];

      // Only move obstacles in Hard difficulty
      if (state.gameLevel == LEVEL_HARD) {
        state.obstacleX[i] += OBSTACLE_SPEED * state.obstacleDir[i];

        // Bounce logic
        if (state.obstacleX[i] <= OBSTACLE_RADIUS ||
            state.obstacleX[i] >= SCREEN_WIDTH - OBSTACLE_RADIUS) {
          state.obstacleDir[i] *= -1;
          // Prevent sticking at edges
          state.obstacleX[i] = constrain(state.obstacleX[i],
            OBSTACLE_RADIUS,
            SCREEN_WIDTH - OBSTACLE_RADIUS);
        }
      }
    }
  }
}
void drawGame() {
    static int lastScore = -1;
  if (state.score != lastScore) {
    if (state.score <= 10) {
      tft.fillRect(0, 0, SCREEN_WIDTH, TOP_OFFSET - 10, RED);
    } else if (state.score > 10 && state.score <= 30) {
      tft.fillRect(0, 0, SCREEN_WIDTH, TOP_OFFSET - 10, BLUE);
    } else if (state.score > 30 && state.score < 50) {
      tft.fillRect(0, 0, SCREEN_WIDTH, TOP_OFFSET - 10, CYAN);
    } else if (state.score >= 50 && state.score <= 89) {
      tft.fillRect(0, 0, SCREEN_WIDTH, TOP_OFFSET - 10, GREEN);
    }
    tft.setCursor(5, 5);
    tft.setTextColor(WHITE);
    tft.print("Score: ");

    tft.print(state.score);
    lastScore = state.score;
    // tft.setf
    tft.setCursor(150,5);
    tft.print("g: ");
    tft.print(state.displayGravity);

  }

  // Clear previous doodler position
  tft.fillRect(state.prevDoodlerX, state.prevDoodlerY + TOP_OFFSET,
               DOODLER_WIDTH, DOODLER_HEIGHT, BLACK);
  // Draw new doodler position
  tft.fillRect(state.doodlerX, state.doodlerY + TOP_OFFSET,
               DOODLER_WIDTH, DOODLER_HEIGHT, WHITE);

  // Update platforms
  int visiblePlatforms = NUM_PLATFORMS; // Default to NUM_PLATFORMS

  if (state.gameLevel != LEVEL_EASY) {
    if (state.score > 10 && state.score < 20) {
      // Reduce visible platforms gradually
      visiblePlatforms = NUM_PLATFORMS - (int)((state.score - 10) * 0.3); // Reduce
by 3 between 10-20
      visiblePlatforms = max(4, visiblePlatforms); // Ensure at least 4 are visible
    }
  }

  for (int i = 0; i < NUM_PLATFORMS; i++) {
    if (i < visiblePlatforms && // Only draw visible platforms
        state.platformY[i] + TOP_OFFSET >= 0 &&
        state.platformY[i] + TOP_OFFSET < SCREEN_HEIGHT) {
      if (state.platformX[i] != state.prevPlatformX[i] ||
          state.platformY[i] != state.prevPlatformY[i]) {
        tft.fillRect(state.prevPlatformX[i], state.prevPlatformY[i] + TOP_OFFSET,
                     PLATFORM_WIDTH, PLATFORM_HEIGHT, BLACK);
      }

      tft.fillRect(state.platformX[i], state.platformY[i] + TOP_OFFSET,
                   PLATFORM_WIDTH, PLATFORM_HEIGHT, GREEN);
      state.prevPlatformX[i] = state.platformX[i];
      state.prevPlatformY[i] = state.platformY[i];
    }
  }
  if (!state.gameIsActive) {
        tft.setTextColor(YELLOW);
        tft.setTextSize(2);
        const char* startText = "Press Start!";
        int16_t x1, y1;
        uint16_t w, h;
        tft.getTextBounds(startText, 0, 0, &x1, &y1, &w, &h); // Get text bounds
        // Position near bottom center
        tft.setCursor(SCREEN_WIDTH / 2 - w / 2, SCREEN_HEIGHT - h - 10);
        tft.print(startText);
    }

  // <<< NEW: Draw obstacles >>>
  for (int i = 0; i < MAX_OBSTACLES; i++) {
    if (state.obstacleActive[i]) {
      // Clear old position
      tft.fillCircle(state.prevObstacleX[i],
                     state.prevObstacleY[i] + TOP_OFFSET,
                     OBSTACLE_RADIUS, BLACK);

      // Draw new position
      tft.fillCircle(state.obstacleX[i],
                     state.obstacleY[i] + TOP_OFFSET,
                     OBSTACLE_RADIUS, OBSTACLE_COLOR);
    }
  }
}

void checkCollisions() {
  for (int i = 0; i < NUM_PLATFORMS; i++) {
    if (state.platformY[i] + TOP_OFFSET >= 0 && state.platformY[i] + TOP_OFFSET <
SCREEN_HEIGHT) { // Visibility check
      // Collision check logic
      if (state.doodlerVelocityY > 0 &&
          state.doodlerX + DOODLER_WIDTH > state.platformX[i] &&
          state.doodlerX < state.platformX[i] + PLATFORM_WIDTH &&
          state.prevDoodlerY + DOODLER_HEIGHT <= state.platformY[i] &&
          state.doodlerY + DOODLER_HEIGHT >= state.platformY[i]) {
```

```
        // Dynamic jump force (as per  original code)
        float dynamicJump = JUMP_FORCE - (state.score * 0.1);
        state.doodlerVelocityY = dynamicJump;
        state.doodlerY = state.platformY[i] - DOODLER_HEIGHT; // Snap to top

        if (!state.platformUsed[i]) { state.score++; state.platformUsed[i] = true;
}
        // return; // Original code didn't have return, check implications
      }
    }
  }

  // <<< NEW: Obstacle collisions >>>
  for (int i = 0; i < MAX_OBSTACLES; i++) {
    if (state.obstacleActive[i]) {
      // Simple Axis-Aligned Bounding Box (AABB) check for Rect-Circle
      // More robust and common than the previous distance check.

      // Find closest point on doodler rectangle to circle center
      float closestX = max((float)state.doodlerX, min((float)state.obstacleX[i],
(float)(state.doodlerX + DOODLER_WIDTH)));
      float closestY = max((float)state.doodlerY, min((float)state.obstacleY[i],
(float)(state.doodlerY + DOODLER_HEIGHT)));

      // Calculate distance squared between circle center and closest point
      float dx = state.obstacleX[i] - closestX;
      float dy = state.obstacleY[i] - closestY;
      float distanceSquared = (dx * dx) + (dy * dy);

      // If distance is less than radius squared, collision!
      if (distanceSquared < (OBSTACLE_RADIUS * OBSTACLE_RADIUS)) {
        state.gameOver = true;
        // You might want a sound effect here
        // tone(BUZZER_PIN, NOTE_C4, 100); // Example collision sound
        return; // Exit collision check early
      }
    }
  }

}

// ========== INITIALIZATION ========== //
void initGame() {
  state.score = 0;
  state.gameOver = false;
  state.gameIsActive = false; // <<< Game doesn't start immediately
  state.levelSelected = true; // Mark level as selected if needed

  // --- Platform Initialization (Keep  original logic) ---
  state.plat_start_used = false; // From original v5 initGame
  state.platformX_start = state.doodlerX; // From original v5 initGame
  state.platformY_start = state.doodlerY - PLAY_AREA_HEIGHT / 2 + 5; // From
original v5 initGame
  state.visiblePlatforms = VISIBLE_PLATFORMS; // From original v5 initGame

  if (state.gameLevel == LEVEL_HARD) {
    state.musicState = MUSIC_HARD;
    state.currentMelody = hardMelody;
    state.currentTempo = hard_tempo;
    state.currentNotesCount = sizeof(hardMelody)/sizeof(hardMelody[0])/2;
  } else {
    state.musicState = MUSIC_MAIN;
    state.currentMelody = melody;
    state.currentTempo = tempo;
    state.currentNotesCount = sizeof(melody)/sizeof(melody[0])/2;
  }
  state.currentMelodyNote = 0;
  state.previousNoteTime = 0;
  state.melodyNoteDuration = 0;

  for (int i = 0; i < NUM_PLATFORMS; i++) {
    state.plat_start_used = true; // From original v5 initGame
    state.platformUsed[i] = false;
    state.platformX[i] = random(SCREEN_WIDTH - PLATFORM_WIDTH);
    state.platformY[i] = PLAY_AREA_HEIGHT - (i * (PLAY_AREA_HEIGHT /
NUM_PLATFORMS));
    if (state.gameLevel == LEVEL_EASY) { state.platformDirection[i] = 0; }
    else { state.platformDirection[i] = (random(2) == 0) ? 1 : -1; }
    state.prevPlatformX[i] = state.platformX[i];
    state.prevPlatformY[i] = state.platformY[i];
  }
   // --- END Platform Initialization ---


  // <<< NEW: Place Doodler ON the starting platform (platform[0]) >>>
  // Ensure platform 0 is positioned reasonably for start
  state.platformX[0] = SCREEN_WIDTH / 2 - PLATFORM_WIDTH / 2; // Center first
platform
  state.platformY[0] = PLAY_AREA_HEIGHT - 60; // Place it relatively low
  state.prevPlatformX[0] = state.platformX[0]; // Update its prev position too
  state.prevPlatformY[0] = state.platformY[0];

  // Set doodler position based on platform 0
  state.doodlerX = state.platformX[0] + (PLATFORM_WIDTH / 2) - (DOODLER_WIDTH / 2);
  state.doodlerY = state.platformY[0] - DOODLER_HEIGHT; // Place doodler feet on
platform 0

  state.doodlerVelocityY = -1.5; // <<< Small initial upward velocity for idle
bounce

  state.prevDoodlerX = state.doodlerX;
  state.prevDoodlerY = state.doodlerY;

  resetDrawFlags();

  // Initialize obstacles


  for (int i = 0; i < MAX_OBSTACLES; i++) {
    state.obstacleActive[i] = false;
    state.obstacleX[i] = -100; // Off-screen
    state.obstacleY[i] = -100;
    state.obstacleDir[i] = (random(2) ? 1 : -1); // Random initial direction
  }
}


void updateMusic() {
  if (state.musicState == MUSIC_OFF) return;

  unsigned long currentTime = millis();
  // Check if it's time for the next note based on the *previous* note's duration
  if (currentTime - state.previousNoteTime >= state.melodyNoteDuration) {
    // Check if we are still within the melody bounds
    if (state.currentMelodyNote < state.currentNotesCount * 2) {

      // --- CORRECT: Access PROGMEM data using pgm_read_word_near() ---
      // Read the note frequency from PROGMEM
      int note = pgm_read_word_near(state.currentMelody + state.currentMelodyNote);
      // Read the note duration specifier from PROGMEM
      int duration = pgm_read_word_near(state.currentMelody +
state.currentMelodyNote + 1);
      // -----------------------------------------------------------------

      // Calculate the duration for THIS note (will be used for the NEXT check)
      int wholenote = (60000 * 4) / state.currentTempo;
      int divider = duration;
      if (divider > 0) {
         state.melodyNoteDuration = wholenote / divider;
      } else {
         // Handle dotted notes (negative divider)
         state.melodyNoteDuration = (wholenote / abs(divider)) * 1.5;
      }

      // Play the note (if it's not a rest)
      if (note == REST) {
         noTone(BUZZER_PIN);
      } else {
         // Play note for 90% of its calculated duration
         tone(BUZZER_PIN, note, state.melodyNoteDuration * 0.9);
      }

      // Update the time the last note *started* playing
      state.previousNoteTime = currentTime;

      // Increment index AFTER reading and calculating duration for the current
note pair
      state.currentMelodyNote += 2;

    } else {
      // Reached the end of the melody
      switch(state.musicState) {
        case MUSIC_MAIN:
        case MUSIC_HARD:
        case MUSIC_CREDITS:
          state.currentMelodyNote = 0; // Loop back to the beginning
          // Optionally reset previousNoteTime and melodyNoteDuration to avoid
initial delay?
          // state.previousNoteTime = currentTime; // Start next loop check
immediately
          // state.melodyNoteDuration = 0; // Ensure first note plays right away
          break;
        case MUSIC_GAME_OVER:
          state.musicState = MUSIC_OFF; // Play only once
          noTone(BUZZER_PIN);
          break;
      }
    }
  }
}

void handleGameOver() {
    // Update High Score (using exact logic from original handleGameOver)

    state.musicState = MUSIC_OFF;
    noTone(BUZZER_PIN);;

    state.musicState = MUSIC_GAME_OVER;
    state.currentMelody = gameOver;
    state.currentTempo = tempo;
    state.currentNotesCount = sizeof(gameOver)/sizeof(gameOver[0])/2;
    state.currentMelodyNote = 0;


    bool highScoreUpdated = false;
    // Reset local flags each time entering this state
    beat_easy = 0; beat_med = 0; beat_hard = 0;

    if (state.gameLevel == LEVEL_EASY) {
        if (state.score > highScore.easy_score) { highScore.easy_score =
state.score; beat_easy = 1; highScoreUpdated = true; }
    } else if (state.gameLevel == LEVEL_MEDIUM) {
        if (state.score > highScore.med_score) { highScore.med_score = state.score;
beat_med = 1; highScoreUpdated = true; }
    } else if (state.gameLevel == LEVEL_HARD) {
        if (state.score > highScore.hard_score) { highScore.hard_score =
state.score; beat_hard = 1; highScoreUpdated = true; }
    }

    // EEPROM update
    if (highScoreUpdated) {
        int addressToUpdate = 0; int scoreToSave = 0;
        if (state.gameLevel == LEVEL_EASY) { addressToUpdate = easyAddress;
scoreToSave = highScore.easy_score; }
        else if (state.gameLevel == LEVEL_MEDIUM) { addressToUpdate = medAddress;
scoreToSave = highScore.med_score; }
        else { addressToUpdate = highAddress; scoreToSave = highScore.hard_score; }
        EEPROM.update(addressToUpdate, lowByte(scoreToSave));
        EEPROM.update(addressToUpdate + 1, highByte(scoreToSave));
        // Verification (as per original code)
        int readBackValue; EEPROM.get(addressToUpdate, readBackValue); // Use get
for verification
        if (readBackValue != scoreToSave) { //Serial.print("EEPROM Write Error!
Addr: "); /* ... */ }
        else { //Serial.println("EEPROM Save Verified."); }
    }

    // Draw Game Over Screen (only once per entry using global flag)
    if (!gameOverMsgDrawn) {
        tft.fillScreen(BLACK);
        tft.setTextSize(2);
        tft.setCursor(50, 50); tft.setTextColor(RED); tft.print("GAME OVER :("); //
Original text
        tft.setCursor(50, 100); tft.setTextColor(WHITE); tft.print("You scored: ");
tft.print(state.score);
        if (highScoreUpdated) { tft.setTextColor(YELLOW); tft.setCursor(40, 130);
tft.print("New High Score!"); } // Optional msg
        tft.setTextColor(WHITE); tft.setCursor(40, 150); tft.print("Click joystick
"); tft.setCursor(60, 180); tft.print("to restart"); // Original text
        gameOverMsgDrawn = true;
    }
}

void checkGameOver() { // Original
  if (state.doodlerY > SCREEN_HEIGHT)
    state.gameOver = true;
}
```

```
void handleScrolling() {
  float dynamicScroll = BASE_SCROLL + (state.score * SCROLL_INCREASE);
  if (state.doodlerY < PLAY_AREA_HEIGHT / 3) {
    int scroll = min(PLAY_AREA_HEIGHT / 3 - state.doodlerY, (int)dynamicScroll);
    state.doodlerY += scroll;

    // Scroll platforms
    for (int i = 0; i < NUM_PLATFORMS; i++) {
      state.platformY[i] += scroll;

      // Regenerate platforms that scroll off bottom
      if (state.platformY[i] > PLAY_AREA_HEIGHT) {
        state.platformX[i] = random(SCREEN_WIDTH - PLATFORM_WIDTH);
        state.platformY[i] = random(-PLATFORM_HEIGHT, 0); // Spawn above screen
        state.platformUsed[i] = false;
        if (state.gameLevel != LEVEL_EASY) {
          state.platformDirection[i] = (random(2) ? 1 : -1);
        }
      }
    }

    // Scroll obstacles
    for (int i = 0; i < MAX_OBSTACLES; i++) {
      if (state.obstacleActive[i]) {
        state.obstacleY[i] += scroll;

        // Remove obstacles that scroll off bottom
        if (state.obstacleY[i] > PLAY_AREA_HEIGHT) {
          state.obstacleActive[i] = false;
        }
      }
    }
  }
}
```

Things That Were Dropped–

List of features that were included in my project proposal, but have been dropped in the final version due to better alternatives/ hardware limitations/ non-feasibility/ similar reasons –

1. A secondary LCD to display real-time position and velocity data - Found it redundant to have a 2nd screen. I came up with a better alternative to create sections on my 2.8” TFT display, display details such as score and gravity with real-time updates.

2. An SD card reader to read and display/ play images and music – It seems that the SPI driver on Uno can drive only one component at time – either work as a MOSI (Master Out Slave In – Arduino acting as the master and the TFT being the slave) or MISO (Master-in Slave-Out, just the opposite), but not both simultaneously. Since, I had to play the game (MOSI) and play music (MISO), this was simply not possible on an Uno.

3. Game Features:

- Shrinking and elongating platforms – felt it to be unnecessary
- Black hole as obstacles – too complex and memory demanding, also didn’t go well with SPI framerate – made the game too slow (ref: my GitHub repo, code V5.5)
- Display of real-time position and velocity – again, the same issue of insufficient SPI rate.

Bibliography –

- https://github.com/robsoncouto/arduino-songs
- https://docs.arduino.cc/retired/getting-started-guides/TFT/
- https://docs.arduino.cc/libraries/tft_espi/
- https://docs.arduino.cc/learn/programming/eeprom-guide/

Credits –

Huge thanks to

- The Professor, and all the Digital lab staff for their immense help throughout
- My friends, who provided me with valuable suggestions for various game features
- My Beta Tester (aka, my roommate)

***