****

**Mini Space Shooter Game Microcontroller**

Prepared by: Abdulmohaimin Bashir (B2205.010021)

Abdulrahman Mekkaoui (B2105.020035)

Muhammad Bilal (B2105.010028)

Nusaibah Mekkaoui (B2205.010015)

Shirina Maci (B2105.020104)

Istanbul Aydin University

COM382: Microprocessors

Professor: MHD WASIM RAED

June 6, 2024

**Abstract**

In this project, we have designed a retro space shooter game to be played and enjoyed by users of most ages and all ranges of gaming experience. The game features a Start Screen and a Game Over screen both to be displayed on the OLED screen. Using Arduino Nano, the game keeps count of the remaining lives that a ship has, the level, elapsed time to finish the game in seconds and, finally, the overall score. The final score accumulated throughout the game is displayed in the Game Over screen for the user to see on the OLED screen. Also, implemented buttons will be used to control the game.

*Keywords:* Arduino, Arduino Nano, microcontroller, OLED, game

**Contents**

Abstract…………………………………………………………………………………………………………….……………………………….2

Introduction………………………………..…………..……………………………………………………….……………………………….5

Project Components ………………………………..…………..………………………………………….……………………………….5

Arduino Overview………………………………………………………………………………………………………………………………6

OLED Screen Overview………………..……………………………………………………………………………………………………..7

Theory of Operation…………………………………………………………………………………………………………………………..8

Components Wiring……………..…………………….……………………………………………………………………………………..9

The Code……………………..…………………………………………………………………………………….…………………………….10

Algorithm Explanation...…………………………………………………………………………………….…………………………….18

Project Challenges……………………………………………………………………………………………………………………………24

Result…………………..………………………………………………………………………………………………………………………….27

Discussion…………...………………………………………………………………………………………………………………………….29

References……………………………………………………………………………………………………………………………………….30

Appendix………………………………………………………………………………………………………………………………………….31

**Table of Figures**

Figure 1: Project Components…………………………………………………………………………………………………………….5

Figure 2 (a): Arduino Nano (front)……………………………………………………………………………………………………….6

Figure 2 (b): Arduino Nano (back)……………………………………………………………………………………………………….6

Figure 3: SSD1306 OLED display………………………………………………………………………………………………………….7

Figure 4: OLED display layers………………………………………………………………………………………………………………7

Figure 5: Project Schematic Diagram…………………………………………………………………………………………………..8

Figure 6: Assembled Project……………………………………………………………………….………………………………………9

Figure 7: Running Project (Start Screen)…………………………………………………….……………………………………..27

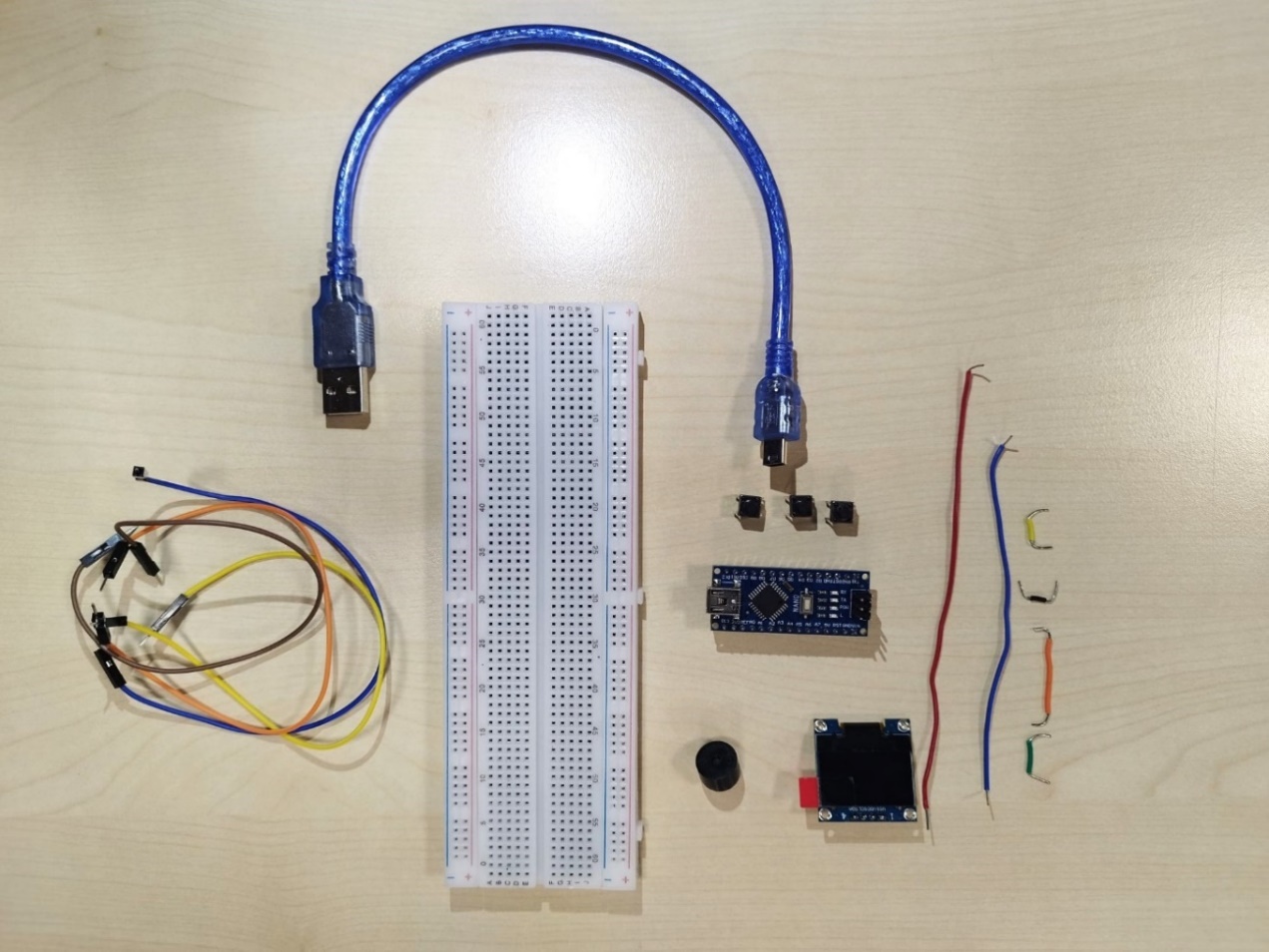
Figure 8: Running Project (Game Screen)………………………………………………….………………………………………28

Figure 9: Running Project (Game Over Screen)**..**……………………………………….……………………….……………..28

**Mini Space Shooter Game Microcontroller**

The Mini Space Shooter Game Microcontroller is an easy and entertaining mini game aimed at audiences of most ages whether they have plenty of gaming experience or are complete beginners. In this game microcontroller, the player gains score, loses lives, and advances in levels. This microcontroller project can be further enhanced and modified to accommodate to a variety of similar games.

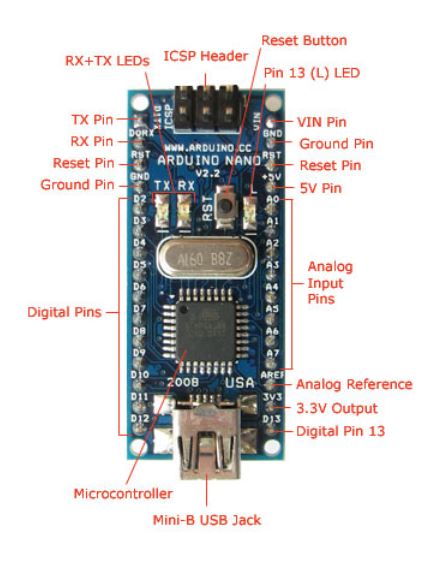
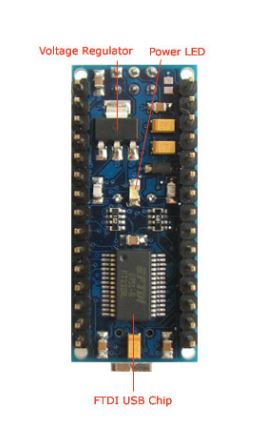
**Project Components**

1. Arduino Nano
2. 128x64 I2C OLED Display
3. Push Buttons x3
4. Buzzer
5. Wires & Breadboard

**Figure 1: Project Components**

**Arduino Overview**

Introduced in 2005, Arduino is a programmable microcontroller for projects involving sensors of all kinds. This microcontroller was created and was meant to be used by experts, who have the experience, or students, who are just starting out and experimenting. The way an Arduino microcontroller works is that it receives input, that being the information from a sensor, and based on that input, Arduino sends back the output. Certain types of Arduino microcontrollers, like the ESP32 for example, are also capable of connecting to the internet to send and receive data using HTTP requests.

The Arduino platform consists of both hardware and software; the hardware being the Arduino development board, and the software being the Arduino IDE (integrated development environment). This IDE is where the code is written for the microcontroller to be programmed using the C or C++ language. The code written in the Arduino IDE is then uploaded to the Arduino board using a USB cable which can be plugged into almost any computer/laptop. For each project, one must research the type of Arduino microcontroller the project requires as Arduino offers a variety of microcontrollers in the market (Ismailov & Jo‘Rayev, 2022). The sake of our mini space shooter game, we will be using the Arduino Nano microcontroller.

**Figure 2 (b): Arduino Nano (back)**

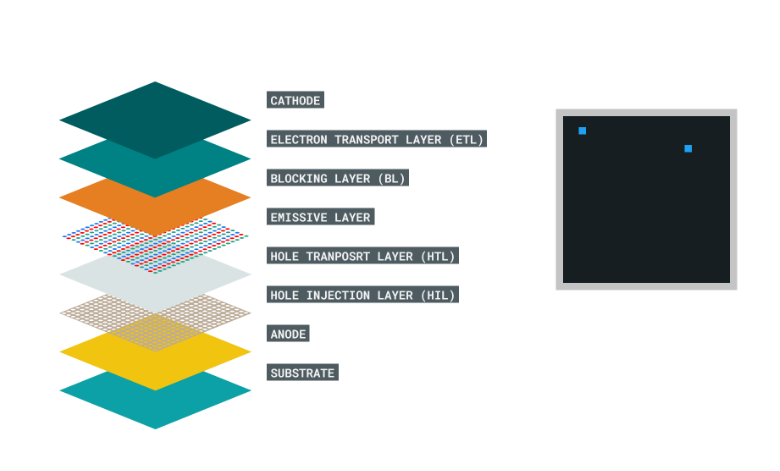
**Figure 2 (a): Arduino Nano (front)**

Arduino Nano is a type of Arduino microcontroller that is breadboard-friendly and small and is based on the ATmega328 (Arduino Nano 3.0) or ATmega168 (Arduino Nano 2.x). This microcontroller has a similar functionality to the Arduino Duemilanove; however, it works with a Mini-B USB cable and does not have a DC power jack (Nano, 2018).

**OLED Screen Overview**

According to the Arduino website (n.d.), the Organic Light-Emitting Diode or better known as OLED, is what the typical Arduino screen uses. Using organic, carbon-based material, the OLED screen emits light when electricity is applied.

**Figure 3: SSD1306 OLED display**

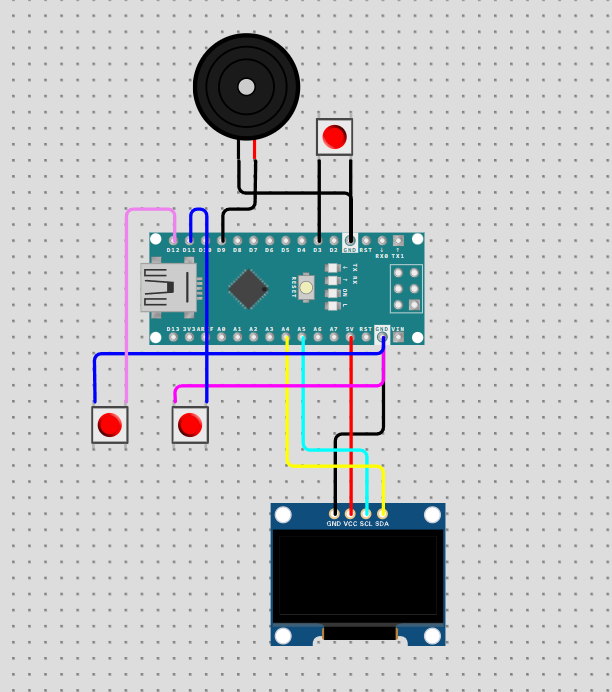
Understanding how the screen works is essential to knowing how to program it as the emitter comprises several layers situated between an anode and a cathode layer, and unlike many other displays, it uses no backlight. What this means is that in order to display anything on the screen, each individual pixel needs to be turned on (Arduino, n.d.).

**Figure 4: OLED display layers**

**Theory of Operation**

Generally speaking, the game is to be displayed on the OLED screen connected to the Arduino Nano microcontroller, which allows us to manipulate the game. The buttons allow us to control the ship’s movement and fire the bullets. The buzzer outputs the music and the sounds of the bullets being shot including their interactions with the objects they’re hitting.

While the schematic diagram in Figure 5 doesn’t show the exact positioning of the components on the breadboard, it does show how the components are precisely wired to each other.

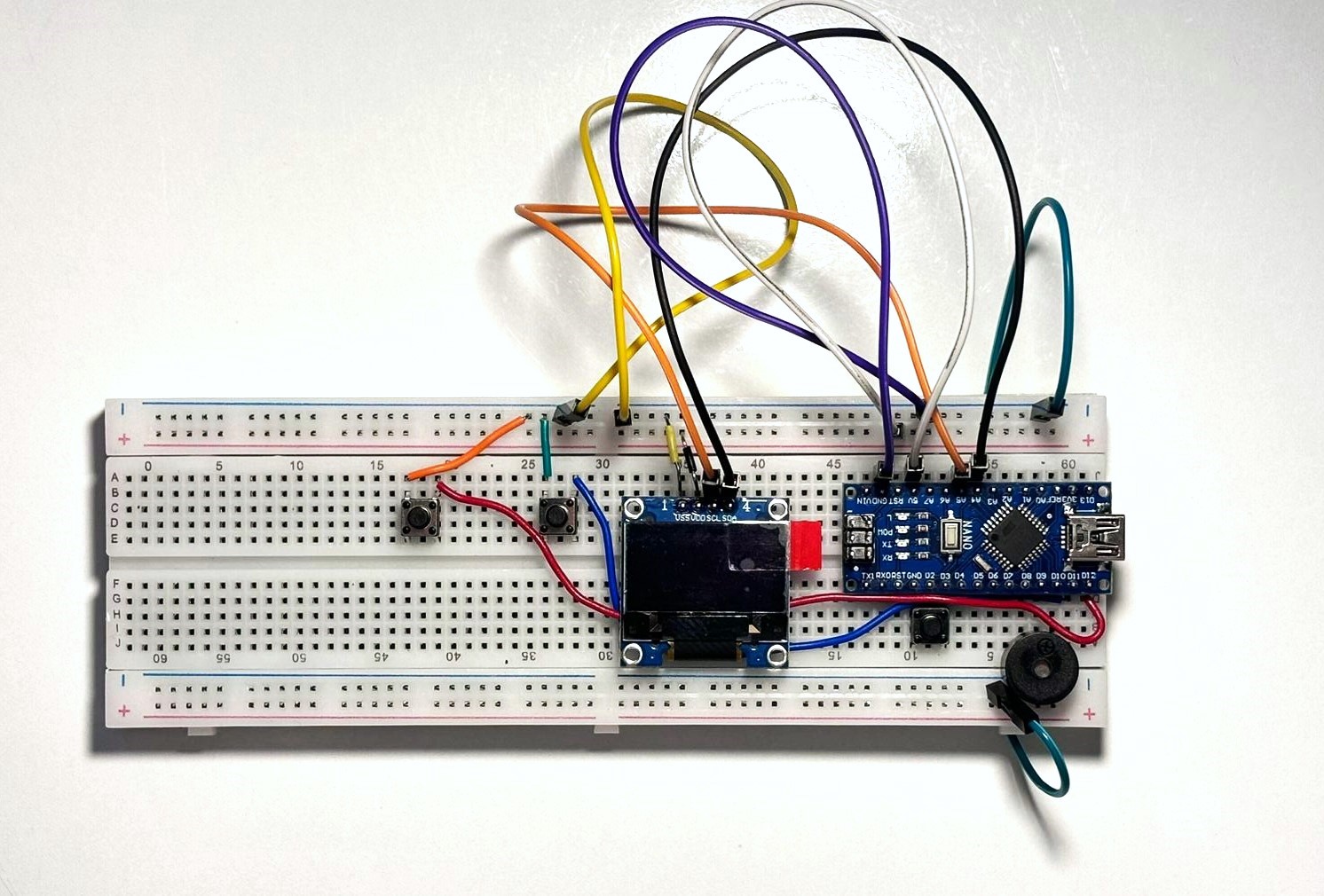


**Figure 5: Project Schematic Diagram**

**Components wiring**

As can be seen in Figure 6, the Arduino Nano microcontroller is connected to a 0.96-inch I2C OLED screen, 3 buttons and a buzzer. Button 1 is connected to D3 and ground (GND), Button 2 is connected to D12 and ground (GND), and, lastly, Button 3 is connected to D13 and ground (GND).

Button 1 is programmed to shoot the bullets. Buttons 2 and 3 are used to move the ship up and down on the OLED display. Note that that pins connected to each button are susceptible to change in both the circuit and the code should we want to.

****The default I2C address of the SSD1306 module is 0x3c (60). This OLED has 4 pins connected as follows: GND is the ground and is connected to the Arduino Nano’s ground. VDD is the supply voltage and is connected to the Arduino Nano’s 5V pin. SCL is the I2C clock line and is connected to the 5th Arduino Nano Analog pin. SDA is the I2C data line and is connected to the 4th Arduino Nano Analog pin. The buzzer is connected to the 9th pin in the Arduino Nano and the ground.

**Figure 6: Assembled Project**

**The Code**

// Includes the necessary libraries for OLED display and graphics

#include <Adafruit\_GFX.h>

#include <Adafruit\_SSD1306.h>

// Pin Definitions

const int buttonUpPin = 12;

const int buttonDownPin = 11;

const int buttonFirePin = 3;

const int buzzerPin = 9;

// OLED Display Configuration

#define SCREEN\_WIDTH 128

#define SCREEN\_HEIGHT 64

#define OLED\_RESET -1

Adafruit\_SSD1306 display(SCREEN\_WIDTH, SCREEN\_HEIGHT, &Wire, OLED\_RESET);

// I2C address for the display

#define SCREEN\_ADDRESS 0x3C

// Game Constants

const int playerWidth = 16;

const int playerHeight = 16;

const int playerStartPos = 24;

const int playerBulletSpeed = 8;

const int enemyBulletSpeedStart = 2;

const int enemyStartDiameter = 12;

const int playerStartLives = 5;

const int levelUpInterval = 10; // 10 points

// Game State Variables

int playerPos = playerStartPos;

int playerBulletX = 0;

int playerBulletY = 0;

bool playerBulletFired = false;

int enemyPosY = 8;

int enemyDirection = 0;

int enemyDiameter = enemyStartDiameter;

int enemyBulletSpeed = enemyBulletSpeedStart;

int enemySpeedAttribute = 6;

int numEnemyBullets = 0;

int score = 0;

int level = 1;

int lives = playerStartLives;

unsigned long inGameTime = 0;

unsigned long levelUpScore = 0;

unsigned long lastBulletLaunchTime = 0;

unsigned long gameStartTime = 0;

bool gameRunning = true;

bool gameOver = false;

// Enemy Bullets

struct Bullet {

int x, y;

int diameter;

};

Bullet enemyBullets[4];

// Define musical notes frequencies

const int f = 1396; // F

const int a = 1865; // A

const int cH = 2093; // High C

// Player sprite (bitmap)

const uint8\_t dioda16[] PROGMEM = {

0x00, 0x00, 0x00, 0x00, 0x1C, 0x00, 0x3F, 0xF0, 0x3C, 0x00, 0x3C, 0x00, 0xFF, 0x00, 0x7F, 0xFF,

0x7F, 0xFF, 0xFF, 0x00, 0x3C, 0x00, 0x3C, 0x00, 0x1F, 0xF0, 0x1C, 0x00, 0x00, 0x00, 0x00, 0x00

};

// Bitmap loading screen

const unsigned char storm [] PROGMEM = {

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x7F, 0xFE, 0x00, 0x00, 0x00, 0x07, 0x80, 0x01, 0xE0, 0x00, 0x00, 0x0C,

0x00, 0x00, 0x20, 0x00, 0x00, 0x18, 0x00, 0x00, 0x18, 0x00, 0x00, 0x30, 0x00, 0x00, 0x04, 0x00,

0x00, 0x20, 0x00, 0x00, 0x04, 0x00, 0x00, 0x20, 0x00, 0x00, 0x04, 0x00, 0x00, 0x60, 0x00, 0x00,

0x02, 0x00, 0x00, 0x40, 0x00, 0x00, 0x02, 0x00, 0x00, 0x40, 0x00, 0x00, 0x01, 0x00, 0x00, 0x40,

0x00, 0x00, 0x01, 0x00, 0x00, 0x40, 0x00, 0x00, 0x01, 0x00, 0x00, 0x7F, 0xE0, 0x00, 0x01, 0x00,

0x00, 0x7F, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x7F, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0xD7, 0xFF, 0xFF,

0xE1, 0x00, 0x01, 0xBF, 0xFC, 0x1F, 0xFA, 0x80, 0x01, 0xBF, 0xF1, 0xCF, 0xFA, 0x80, 0x01, 0x3F,

0xC2, 0x37, 0xF7, 0x80, 0x01, 0xEF, 0x9C, 0x01, 0xE7, 0xC0, 0x01, 0xE0, 0x70, 0x06, 0x06, 0x80,

0x01, 0xE0, 0xC0, 0x03, 0x06, 0x80, 0x01, 0xFF, 0x80, 0x01, 0xFF, 0x80, 0x01, 0xF8, 0x00, 0x00,

0x1D, 0xC0, 0x03, 0x70, 0x00, 0x80, 0x0C, 0x60, 0x05, 0xB0, 0x07, 0xF0, 0x08, 0x90, 0x09, 0x10,

0x1F, 0xF8, 0x09, 0xD0, 0x0B, 0x90, 0x1F, 0x7C, 0x03, 0xF0, 0x0F, 0xC0, 0xFC, 0x0F, 0x07, 0x90,

0x0D, 0x43, 0xC0, 0x03, 0x07, 0x90, 0x05, 0x64, 0x00, 0x00, 0xCF, 0x10, 0x07, 0xFC, 0x00, 0x00,

0x26, 0x10, 0x01, 0x80, 0x00, 0x00, 0x10, 0x20, 0x01, 0x00, 0x00, 0x00, 0x0E, 0x40, 0x01, 0x80,

0x07, 0xF0, 0x01, 0x80, 0x00, 0x80, 0x07, 0xC8, 0x00, 0x80, 0x00, 0x80, 0x0B, 0xE8, 0x00, 0x80,

0x00, 0x87, 0x97, 0xE9, 0xE0, 0x80, 0x00, 0x87, 0xDF, 0xEF, 0xA0, 0x80, 0x00, 0x4B, 0xFF, 0xFF,

0xA0, 0x80, 0x00, 0x6B, 0xDF, 0xFB, 0xA3, 0x00, 0x00, 0x24, 0x97, 0xE8, 0x24, 0x00, 0x00, 0x1E,

0x1F, 0xC0, 0x2C, 0x00, 0x00, 0x07, 0xF8, 0x1F, 0xF0, 0x00, 0x00, 0x00, 0x0F, 0xF8, 0x00, 0x00

};

// Main setup function

void setup() {

// Initialize the OLED display

if (!display.begin(SSD1306\_SWITCHCAPVCC, SCREEN\_ADDRESS)) {

for (;;); // Don't proceed, loop forever

}

display.display();

// Clear the buffer.

display.clearDisplay();

// Display bitmap image

display.drawBitmap(40, 9, storm, 48, 48, WHITE);

display.display();

// Play startup melody

beep(a, 500);

beep(a, 500);

beep(f, 350);

beep(cH, 150);

beep(a, 500);

beep(f, 350);

beep(cH, 150);

beep(a, 650);

delay(1500);

display.clearDisplay();

// Initialize pins for buttons and buzzer

pinMode(buttonUpPin, INPUT\_PULLUP);

pinMode(buttonDownPin, INPUT\_PULLUP);

pinMode(buttonFirePin, INPUT\_PULLUP);

pinMode(buzzerPin, OUTPUT);

// Initialize game

resetGame();

}

// Main loop function

void loop() {

if (gameRunning) {

inGameTime = millis() / 1000;

updateGame();

drawGame();

} else if (gameOver) {

displayGameOver();

}

}

// Function to update game state

void updateGame() {

updatePlayer();

updateBullets();

updateEnemy();

checkCollisions();

checkLevelUp();

}

// Function to draw game on the OLED display

void drawGame() {

display.clearDisplay();

drawStars();

drawPlayer();

drawEnemy();

drawBullets();

drawHUD();

display.display();

}

// Function to update player's position and bullet firing

void updatePlayer() {

// Handle player movement and firing

if (digitalRead(buttonUpPin) == LOW && playerPos >= 2) {

playerPos -= 2;

}

if (digitalRead(buttonDownPin) == LOW && playerPos <= SCREEN\_HEIGHT - playerHeight - 2) {

playerPos += 2;

}

if (digitalRead(buttonFirePin) == LOW && !playerBulletFired) {

playerBulletFired = true;

playerBulletX = playerWidth;

playerBulletY = playerPos + 8;

beep(1200, 20); // Play firing sound

}

if (playerBulletFired) {

playerBulletX += playerBulletSpeed;

if (playerBulletX > SCREEN\_WIDTH) {

playerBulletFired = false;

}

}

}

// Function to update enemy bullets and launch new bullets

void updateBullets() {

for (int i = 0; i < numEnemyBullets; ++i) {

enemyBullets[i].x -= enemyBulletSpeed;

if (enemyBullets[i].x < 0) {

enemyBullets[i] = enemyBullets[--numEnemyBullets];

}

}

if (millis() - lastBulletLaunchTime > random(400, 1200)) {

launchEnemyBullet();

}

}

// Function to update enemy's position and behavior

void updateEnemy() {

if (enemyDirection == 0) {

enemyPosY += enemyDiameter / enemySpeedAttribute;

} else {

enemyPosY -= enemyDiameter / enemySpeedAttribute;

}

// Bottom of the screen

if (enemyPosY >= SCREEN\_HEIGHT - enemyDiameter) {

enemyDirection = 1;

}

// Top of the screen

if (enemyPosY <= 13) {

enemyDirection = 0;

}

}

// Function to check if player leveled up and adjust game parameters

void checkLevelUp() {

if ((levelUpScore + 1) % 10 == 0) {

levelUpScore = 0;

level++;

enemyBulletSpeed++;

if (level % 2 == 0) {

enemyDiameter = max(1, enemyDiameter - 1);

enemySpeedAttribute = max(1, enemySpeedAttribute - 1);

}

}

}

// Function to draw background stars

void drawStars() {

int starCoords[18][2] = {

{50, 30}, {30, 17}, {60, 18}, {55, 16}, {25, 43},

{100, 43}, {117, 52}, {14, 49}, {24, 24}, {78, 36},

{80, 57}, {107, 11}, {150, 11}, {5, 5}, {8, 7},

{70, 12}, {10, 56}, {70, 25}

};

for (int i = 0; i < 18; ++i) {

display.drawPixel(starCoords[i][0], starCoords[i][1], WHITE);

}

}

// Function to draw player's spaceship

void drawPlayer() {

display.drawBitmap(4, playerPos, dioda16, playerWidth, playerHeight, WHITE);

if (playerBulletFired) {

display.drawLine(playerBulletX, playerBulletY, playerBulletX + 4, playerBulletY, WHITE);

}

}

// Function to draw enemy's spaceship

void drawEnemy() {

display.fillCircle(SCREEN\_WIDTH - 15, enemyPosY, enemyDiameter, WHITE);

display.fillCircle(SCREEN\_WIDTH - 13, enemyPosY + 3, enemyDiameter / 3, BLACK);

}

// Function to draw bullets on the display

void drawBullets() {

for (int i = 0; i < numEnemyBullets; ++i) {

display.drawCircle(enemyBullets[i].x, enemyBullets[i].y, enemyBullets[i].diameter, WHITE);

}

}

// Function to draw HUD (score, lives, level, time) on the display

void drawHUD() {

display.setTextSize(1);

display.setTextColor(WHITE);

display.setCursor(33, 57);

display.println("Score:");

display.setCursor(68, 57);

display.println(score);

display.setCursor(33, 0);

display.println("Lives:");

display.setCursor(68, 0);

display.println(lives);

display.setCursor(110, 0);

display.println("L:");

display.setCursor(122, 0);

display.println(level);

display.setCursor(108, 57);

display.println(inGameTime);

}

// Function to check collisions (player with enemy bullets and vice versa)

void checkCollisions() {

// Check player bullet collision with enemy

if (playerBulletFired &&

playerBulletY >= enemyPosY - enemyDiameter && playerBulletY <= enemyPosY + enemyDiameter &&

playerBulletX >= SCREEN\_WIDTH - 10 - enemyDiameter && playerBulletX <= SCREEN\_WIDTH - 10 + enemyDiameter) {

playerBulletFired = false;

score++;

levelUpScore++;

beep(500, 20);

}

// Check enemy bullets collision with player

for (int i = 0; i < numEnemyBullets; ++i) {

int bulletY = playerPos + 8;

if (enemyBullets[i].y >= bulletY - 8 && enemyBullets[i].y <= bulletY + 8 &&

enemyBullets[i].x >= 4 && enemyBullets[i].x <= 12) {

lives--;

if (lives <= 0) {

gameOver = true;

gameRunning = false;

}

enemyBullets[i] = enemyBullets[--numEnemyBullets];

beep(100, 20);

}

}

}

// Function to display Game Over and score

void displayGameOver() {

// Play game over melody

if (lives == 0) {

tone(9, 200, 300);

delay(300);

tone(9, 250, 200);

delay(200);

tone(9, 300, 300);

delay(300);

lives = playerStartLives; // Reset lives to initial value

}

display.clearDisplay(); // Clear the OLED display

display.setTextSize(2); // Set text size to 2x

display.setCursor(14, 10); // Set cursor position for "Game Over"

display.println("Game Over"); // Display "Game Over"

display.setTextSize(1); // Set text size back to 1x

display.setCursor(30, 34); // Set cursor position for "Final Score:"

display.println("Final Score:"); // Display "Final Score:"

display.setCursor(70, 46); // Set cursor position for the actual score value

display.println(score); // Display the current score

display.display(); // Update the display

// Restart game if button is pressed

if (digitalRead(buttonFirePin) == LOW) {

tone(9, 280, 300);

delay(300);

tone(9, 250, 200);

delay(200);

tone(9, 370, 300);

delay(300);

resetGame(); // Reset the game state

}

}

void resetGame() {

// Reset player position and bullet state

playerPos = playerStartPos;

playerBulletX = 0;

playerBulletY = 0;

playerBulletFired = false;

// Reset enemy parameters

enemyPosY = 8;

enemyDirection = 0;

enemyDiameter = enemyStartDiameter;

enemyBulletSpeed = enemyBulletSpeedStart;

numEnemyBullets = 0;

// Reset game score, level, and time

score = 0;

level = 1;

levelUpScore = 0;

lastBulletLaunchTime = 0;

gameStartTime = 0;

inGameTime = 0;

// Reset game state flags

gameRunning = true;

gameOver = false;

}

void launchEnemyBullet() {

int diameter = random(2, 7); // Generates a random size between 7 and 12 for bullet size

if (numEnemyBullets < 3) { // Only launch bullet if there are less than 3 bullets already

enemyBullets[numEnemyBullets++] = {SCREEN\_WIDTH - 10, enemyPosY, diameter}; // Add bullet at enemy position

lastBulletLaunchTime = millis(); // Record the launch time

}

}

void beep(int note, int duration) {

tone(buzzerPin, note, duration); // Generate a tone for specified duration

delay(duration); // Delay for the specified duration

noTone(buzzerPin); // Stop the tone

delay(50); // Small delay after the tone

}

long readVcc() {

// Determine the appropriate ADC settings for reading Vcc based on the microcontroller type

#if defined(\_AVR\_ATmega32U4) || defined(AVR\_ATmega1280) || defined(AVR\_ATmega2560\_)

ADMUX = \_BV(REFS0) | \_BV(MUX4) | \_BV(MUX3) | \_BV(MUX2) | \_BV(MUX1);

#elif defined (\_AVR\_ATtiny24) || defined(AVR\_ATtiny44) || defined(AVR\_ATtiny84\_)

ADMUX = \_BV(MUX5) | \_BV(MUX0);

#elif defined (\_AVR\_ATtiny25) || defined(AVR\_ATtiny45) || defined(AVR\_ATtiny85\_)

ADMUX = \_BV(MUX3) | \_BV(MUX2);

#else

ADMUX = \_BV(REFS0) | \_BV(MUX3) | \_BV(MUX2) | \_BV(MUX1);

#endif

delay(2); // Wait for Vref to settle

ADCSRA |= \_BV(ADSC); // Start conversion

while (bit\_is\_set(ADCSRA, ADSC)); // Wait for measurement

uint8\_t low = ADCL; // Read ADCL first to lock ADCH

uint8\_t high = ADCH; // Read ADCH

long result = (high << 8) | low; // Combine high and low bytes

result = 1125300L / result; // Calculate Vcc (in mV)

return result; // Return Vcc in millivolts

}

While this project was an open-source project which we found online, we found that the code provided was difficult to work with and hard to maintain. And so, in this project, we have decided to examine, take apart and ultimately revamp the code; making radical changes to it such that it is scalable, easy to understand, and manipulate.

The code is structured into several functions to handle specific tasks. Firstly, in the Setup section, the frame display, buttons, and any game related variables that can be set at the beginning are initialized.

Secondly, the game Loop has an overall control of how the game progresses and time, main game loop that handles updating the game state and drawing to the screen.

Lastly, the Game State Updates includes different functions that alter player position, bullets fired, enemies, and collisions.

On another note, the functions displayGameOver and restartGame deal with the different ways the game could end, the final results of the game, and the relaunching of the game should the player decide to play again.

As mentioned previously, this game was an open-source project that we had to reconstruct due to its poor scalability and maintainability. Thus, the significant modifications we’ve made to the code include enhancing its readability by structuring the code in such a way that we use subfunctions or, rather, user-defined functions called in the main loop function.

Blocking delays were minimized while timing management for non-blocking operations were enhanced by the millis() method; this makes for increased efficiency in our projects improved code.

We have also used bitmap graphics for both the player and the enemies, increasing the OLED display’s potential by using optimized methods.

By using an array of structs for the enemy bullets, we made it possible to further expand the enemies’ behavior making it easy to add more to the game in the future; after all, it is generally agreed upon that a good code is a scalable one.

**Algorithm Explanation**

***Global Variables***

The game uses several global variables to manage the state:

* **playerPos**: Current position of the player.
* **playerBulletX**, **playerBulletY**: Coordinates of the player's bullet.
* **playerBulletFired**: Boolean indicating if the player's bullet is active.
* **enemyPosY**: Vertical position of the enemy.
* **enemyDirection**: Direction of enemy movement.
* **enemyDiameter**: Size of the enemy.
* **enemyBulletSpeed**: Speed of enemy bullets.
* **numEnemyBullets**: Number of active enemy bullets.
* **score**, **level**, **levelUpScore**: Game score, current level, and score needed to level up.
* **lastBulletLaunchTime**, **gameStartTime**, **inGameTime**: Timing variables.
* **gameRunning**, **gameOver**: Game state flags.
* **lives**: Player's remaining lives.

***Setup and Initialization***

void setup() {

// Initialize display, buttons, and sound pins

display.begin(SSD1306\_SWITCHCAPVCC, 0x3C);

pinMode(buttonLeftPin, INPUT\_PULLUP);

pinMode(buttonRightPin, INPUT\_PULLUP);

pinMode(buttonFirePin, INPUT\_PULLUP);

pinMode(buzzerPin, OUTPUT);

randomSeed(analogRead(0));

resetGame();

}

The **setup()** function initializes the OLED display, buttons, and sound pin.

**resetGame()** initializes or resets game variables to their starting values.

***Main Game Loop***

void loop() {

if (gameRunning) {

// Player actions

movePlayer();

firePlayerBullet();

updatePlayerBullet();

// Enemy actions

moveEnemy();

launchEnemyBullet();

updateEnemyBullets();

// Collision detection

checkCollisions();

// Update display

displayGame();

// Check for level up

if (score >= levelUpScore) {

levelUp();

}

} else if (gameOver) {

displayGameOver();

}

}

The **loop()** function is the main game loop that continuously executes while the game is running.

* Player actions include **movePlayer()**, **firePlayerBullet()**, **updatePlayerBullet()**.
* Enemy actions are **moveEnemy()**, **launchEnemyBullet()**, **updateEnemyBullets()**.
* Collision detection make use of the **checkCollisions()** function.
* Display update uses the **displayGame()** function.
* Level up checks if the score meets the **levelUpScore**, then the **levelUp()** function is called.
* If the game is over, **displayGameOver()** is called.

***Player Controls***

void movePlayer() {

if (digitalRead(buttonLeftPin) == LOW && playerPos > 0) {

playerPos--;

}

if (digitalRead(buttonRightPin) == LOW && playerPos < SCREEN\_WIDTH - playerWidth) {

playerPos++;

}

}

void firePlayerBullet() {

if (digitalRead(buttonFirePin) == LOW && !playerBulletFired) {

playerBulletX = playerPos + playerWidth / 2;

playerBulletY = SCREEN\_HEIGHT - playerHeight;

playerBulletFired = true;

}

}

void updatePlayerBullet() {

if (playerBulletFired) {

playerBulletY -= 2;

if (playerBulletY < 0) {

playerBulletFired = false;

}

}

}

* **movePlayer()** reads left and right button inputs to move the player within screen boundaries.
* **firePlayerBullet()** fires a bullet if the fire button is pressed and no bullet is currently active.
* **updatePlayerBullet()** updates the bullet position and deactivates it if it moves off-screen.

***Enemy Behaviour***

void moveEnemy() {

enemyPosY += enemyDirection ? 1 : -1;

if (enemyPosY <= 0 || enemyPosY >= SCREEN\_HEIGHT - enemyDiameter) {

enemyDirection = !enemyDirection;

}

}

void launchEnemyBullet() {

int diameter = random(2, 7);

if (numEnemyBullets < 3) {

enemyBullets[numEnemyBullets++] = {SCREEN\_WIDTH - 10, enemyPosY, diameter};

lastBulletLaunchTime = millis();

}

}

void updateEnemyBullets() {

for (int i = 0; i < numEnemyBullets; i++) {

enemyBullets[i].x -= enemyBulletSpeed;

if (enemyBullets[i].x < 0) {

for (int j = i; j < numEnemyBullets - 1; j++) {

enemyBullets[j] = enemyBullets[j + 1];

}

numEnemyBullets--;

i--;

}

}

}

* **moveEnemy()** moves the enemy up and down the screen, reversing direction at boundaries.
* **launchEnemyBullet()** launches a bullet from the enemy position if fewer than 3 bullets are active.
* **updateEnemyBullets()** updates the position of active enemy bullets and removes them if they move off-screen.

***Collision Detection***

void checkCollisions() {

// Player bullet and enemy collision

if (playerBulletFired) {

if (playerBulletX > enemyPosX && playerBulletX < enemyPosX + enemyDiameter &&

playerBulletY > enemyPosY && playerBulletY < enemyPosY + enemyDiameter) {

playerBulletFired = false;

score += 10;

enemyDiameter -= 2;

if (enemyDiameter < 5) {

enemyDiameter = enemyStartDiameter;

levelUpScore += 50;

level++;

}

}

}

// Enemy bullets and player collision

for (int i = 0; i < numEnemyBullets; i++) {

if (enemyBullets[i].x > playerPos && enemyBullets[i].x < playerPos + playerWidth &&

enemyBullets[i].y > SCREEN\_HEIGHT - playerHeight) {

for (int j = i; j < numEnemyBullets - 1; j++) {

enemyBullets[j] = enemyBullets[j + 1];

}

numEnemyBullets--;

i--;

lives--;

if (lives == 0) {

gameRunning = false;

gameOver = true;

}

}

}

}

Checks for collisions between the player's bullet and the enemy. If a collision occurs, the enemy shrinks, and the score increases.

Checks for collisions between enemy bullets and the player. If a collision occurs, the player loses a life, and the game ends if all lives are lost.

***Display Functions***

void displayGame() {

display.clearDisplay();

display.setCursor(0, 0);

display.print("Score: ");

display.println(score);

display.print("Lives: ");

display.println(lives);

display.print("Level: ");

display.println(level);

// Display player

display.fillRect(playerPos, SCREEN\_HEIGHT - playerHeight, playerWidth, playerHeight, WHITE);

// Display player bullet

if (playerBulletFired) {

display.fillRect(playerBulletX, playerBulletY, bulletWidth, bulletHeight, WHITE);

}

// Display enemy

display.fillRect(enemyPosX, enemyPosY, enemyDiameter, enemyDiameter, WHITE);

// Display enemy bullets

for (int i = 0; i < numEnemyBullets; i++) {

display.fillRect(enemyBullets[i].x, enemyBullets[i].y, enemyBullets[i].size, enemyBullets[i].size, WHITE);

}

display.display();

}

void displayGameOver() {

// Play game over melody

if (lives == 0) {

tone(9, 200, 300);

delay(300);

tone(9, 250, 200);

delay(200);

tone(9, 300, 300);

delay(300);

lives = playerStartLives;

}

display.clearDisplay();

display.setTextSize(2);

display.setCursor(14, 10);

display.println("Game Over");

display.setTextSize(1);

display.setCursor(30, 34);

display.println("Final Score:");

display.setCursor(70, 46);

display.println(score);

display.display();

// Restart game if button is pressed

if (digitalRead(buttonFirePin) == LOW) {

tone(9, 280, 300);

delay(300);

tone(9, 250, 200);

delay(200);

tone(9, 370, 300);

delay(300);

resetGame();

}

}

* **displayGame()** clears the display and updates it with the current game state, including the player, bullets, and enemy positions.
* **displayGameOver()** displays the game over screen with the final score and allows restarting the game upon button press.

***Sound Effects***

void beep(int note, int duration) {

tone(buzzerPin, note, duration);

delay(duration);

noTone(buzzerPin);

delay(50);

}

* **beep(int note, int duration)** generates a tone of a specified note and duration, followed by a short delay.

***Utility Functions***

long readVcc() {

#if defined(\_AVR\_ATmega32U4) || defined(AVR\_ATmega1280) || defined(AVR\_ATmega2560\_)

ADMUX = \_BV(REFS0) | \_BV(MUX4) | \_BV(MUX3) | \_BV(MUX2) | \_BV(MUX1);

#elif defined (\_AVR\_ATtiny24) || defined(AVR\_ATtiny44) || defined(AVR\_ATtiny84\_)

ADMUX = \_BV(MUX5) | \_BV(MUX0);

#elif defined (\_AVR\_ATtiny25) || defined(AVR\_ATtiny45) || defined(AVR\_ATtiny85\_)

ADMUX = \_BV(MUX3) | \_BV(MUX2);

#else

ADMUX = \_BV(REFS0) | \_BV(MUX3) | \_BV(MUX2) | \_BV(MUX1);

#endif

delay(2);

ADCSRA |= \_BV(ADSC);

while (bit\_is\_set(ADCSRA, ADSC));

uint8\_t low = ADCL;

uint8\_t high = ADCH;

long result = (high << 8) | low;

result = 1125300L / result;

return result;

}

**readVcc()** reads the supply voltage (Vcc) using the ADC and returns it in millivolts. The function adjusts ADC settings based on the microcontroller type.

***Game Reset***

void resetGame() {

playerPos = playerStartPos;

playerBulletX = 0;

playerBulletY = 0;

playerBulletFired = false;

enemyPosY = 8;

enemyDirection = 0;

enemyDiameter = enemyStartDiameter;

enemyBulletSpeed = enemyBulletSpeedStart;

numEnemyBullets = 0;

score = 0;

level = 1;

levelUpScore = 0;

lastBulletLaunchTime = 0;

gameStartTime = 0;

inGameTime = 0;

gameRunning = true;

gameOver = false;

}

* **resetGame()** resets all game variables to their initial states, effectively restarting the game.

**Project Challenges**

***Global Variable Management***

The original code was structured with a heavy reliance on global variables for handling states in the game. As with any approach that involves creating multiple copies of a single template, this can complicate the tracking of changes and bug fixing. Making state and data self-contained and independent from other components would improve scalability but would involve making many changes to the system’s architecture.

***Function Complexity***

The major issues identified in the original code included several functions performing more than one operation, such as the checkCollisions() function. Dividing these functions into a set of smaller functions that each perform a single task is important to produce a highly scalable and maintainable code, however it entails a more intricate refactoring of the overall architectural design of the system to make sure all the interactions are correctly managed.

***Hardcoded Values***

A lack of flexibility is another drawback that results from having hardcoded values for such aspects as screen size, players’ locations, and other characteristics of the game. However, translating these values to constants or a configuration file can improve the code’s maintainability at the cost of requiring a significant review of the new values and testing to guarantee their equivalence to the previous values.

***Limited Use of Object-Oriented Principles: Limited Use of Object-Oriented Principles***

The said code specifically follows procedural programming, and this makes it hard to make further changes with the game elements. To demonstrate this point, let us consider how the player and enemy behaviours are defined and organized in code. While organizing the code in a more OOP manner, such as encapsulating the player and enemy classes, can make the code easier to organize and reuse, it would require a lot of refactoring.

***Error Handling***

Originally, the code that was written did not have adequate error checking measures in place. To address this, we incorporate error handling and validation checks which make the code more reliable but also complex.

***Scalability Issues with Game Elements: Scalability Issues with Game Elements***

If more elements are added to the structure now, it will be complicated to incorporate things like different types of enemies, power-ups and so on. To bring in several improvements, it is necessary to design a more flexible system that would allow adding new elements easily, though, it is an important change in the architecture of the game.

***Performance Considerations***

To achieve high efficiency, coupled with legibility and variability where necessary, is not an easy task especially when working under limited constrains on microcontroller environments. Managing these aspects involve profiling and optimization skills and strategies that must be used appropriately.

***Display and Input Handling***

The original code has many more lines that directly manipulate the display and the input handling which are intrinsically tied in with the game logic. While this makes for better maintainability and testability, it requires a major reorganization of the code base as the aspects are factored into separate modules or classes.

**Buzzer Adjustments**

We faced some difficulties with the buzzer playing music. Initially, the music sounded like noise whenever the player starts the game due to low frequency. We tried different methods to fix this including adjusting all the notes’ frequencies together but this led the music to sound slightly disoriented and essentially different than what we needed it to be, and, so, we came up with the solution of trying to increase the frequency for each note included in the code separately in order for it to sound like actual music. We thought we’d face a delay between the player pressing an input button and the music coming out of the buzzer; however, the way we structured the code made it possible for us to avoid such a petty problem.

**Screen Size**

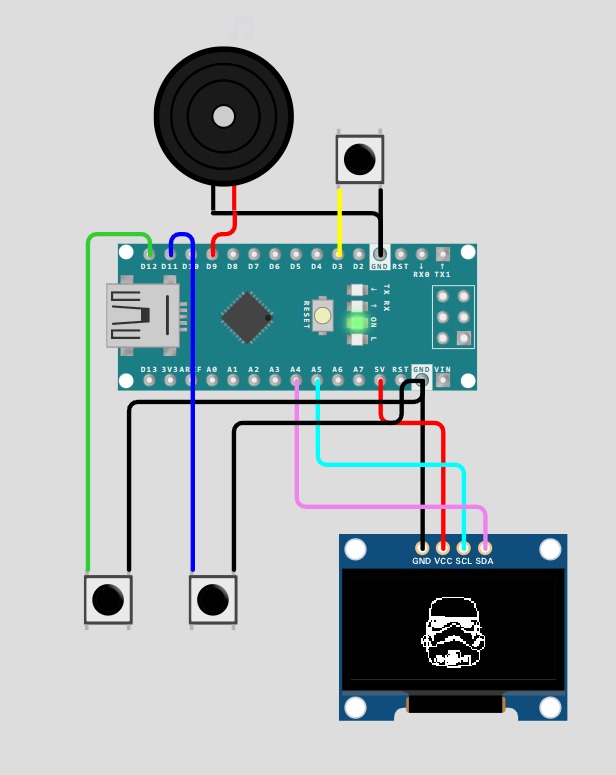
The OLED display has a very small screen and fitting such a game with many components posed some difficulty for us as we had to fix the dimensions of the game to fit the borders quite often. Some solutions discussed and methods implemented were changing the direction of the components in the game and their positions. We ultimately had to stick with changing the resolution of the game itself in order to fit the borders of the screen and for this we had to go through much trial-and-error.

**Wiring The Components to The Board**

With such a small game, using small components so closely related on the bread, we had the issue of wiring the components together. The wires were messy and all over the place until we used small wired to directly imbed them into the board and between the components, even underneath them. Some of the wires we used were better than your average wires such that they stayed in place once we bent them. This may seem like a trivial issue for most people; however, as with most devices, and, especially, compact ones, we found that the wires used best suited for the task to be accomplished.

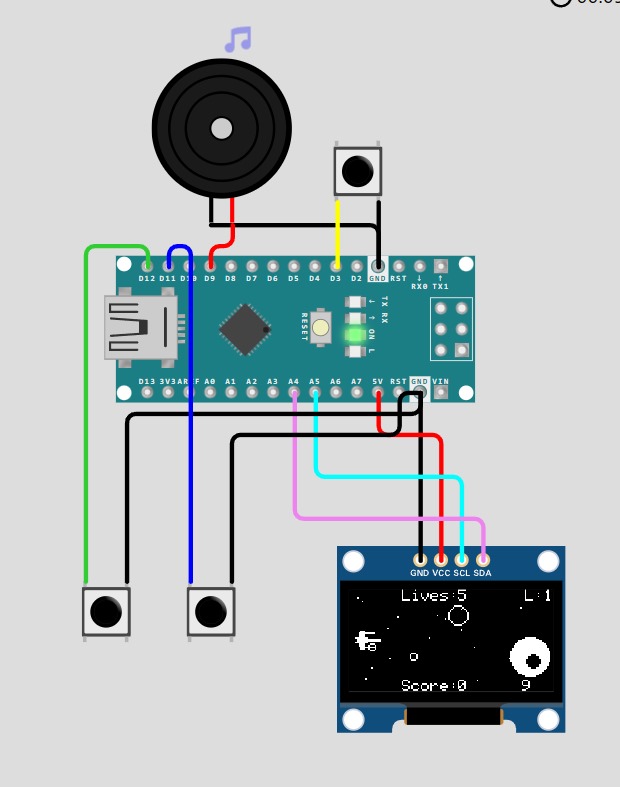
**Results**

Using the online Electronics Simulator Tool *Wokwi*, we have tested out the assembled project along with the code to end up with a running game as shown in Figure 7. As designed, the player is faced with a Start Screen and music.

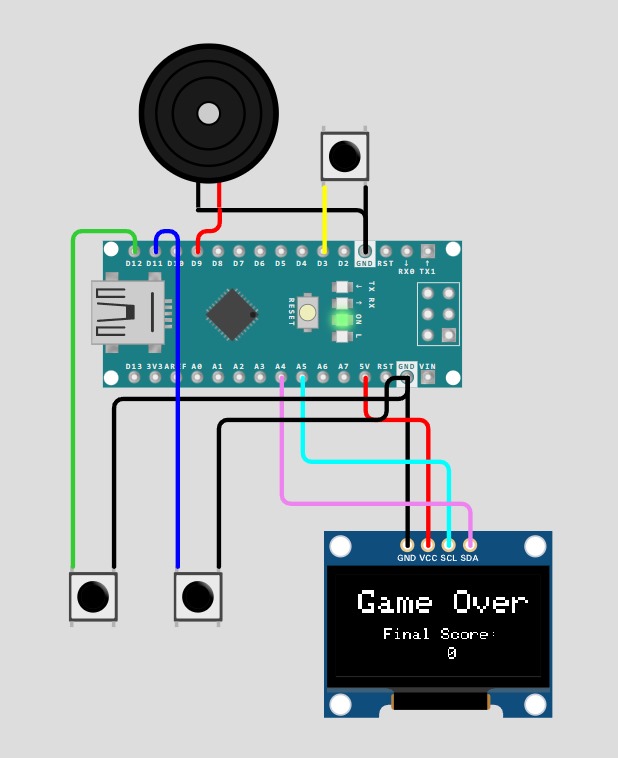
****

**Figure 7: Running Project (Start Screen)**

The player is, then, able to play the game and gain score upon pushing the allocated buttons. As expected, the player is also able to level up and, thus, increase the difficulty of the game, as well as lose lives, as can be seen in Figure 8.



**Figure 8: Running Project (Game Screen)**

Upon losing all lives, a Game Over Screen is then displayed on the OLED screen. The player can then restart the game by pushing the button closest to the buzzer as shown in Figure 9.

**Figure 9: Running Project (Game Over Screen)**

**Discussion**

As seen in this Arduino project, we made use of the microcontroller’s hardware and software capabilities to create a simple yet entertaining game for people of most ages and experience. Using an Arduino Nano, OLED display, buttons, and a buzzer, and a structured code, we brought forth an engaging game that is also nostalgic to many players. After successful testing using the online simulator, we have concluded that this project highlights Arduino's versatility in creating interactive gaming experiences.

**References**

Ismailov, A. S., & Jo‘Rayev, Z. B. (2022). Study of arduino microcontroller board. *Science and Education*, *3*(3), 172-179.

Arduino Nano. (n.d.). <https://www.mantech.co.za/datasheets/products/a000005-6s.pdf>

*THE OLED SCREEN*. Sensorkit.arduino.cc. (n.d.). <https://sensorkit.arduino.cc/sensorkit/module/lessons/lesson/10-the-oled-screen>

**Appendix**

**Tools and Resources**

This appendix contains resources and tools which have been used for the sake of this project or have helped in some way or form.

Wokwi is an online Electronics simulator used to simulate Arduino and other microcontrollers. It includes boards, parts and sensors which the user can test out without worrying about mistakes and easily get feedback.

To understand this project better, please refer to the following YouTube video: <https://www.youtube.com/watch?v=lOz_GuME63E>