








## Tutorial para implementar el juego TETRIS minimalista en Arduino

[labtec@umce.cl](mailto:labtec@umce.cl)  
[jonnhatan.garcia@umce.cl](mailto:jonnhatan.garcia@umce.cl)

### Materiales necesarios

<ul style="list-style-type: none"><li>1 Arduino UNO</li></ul>	
<ul style="list-style-type: none"><li>1 Display 4 Matriz De Puntos Led 8x8 Max7219</li></ul>	
<ul style="list-style-type: none"><li>1 Buzzer</li></ul>	<p>KY-006</p> 
<ul style="list-style-type: none"><li>1 Módulo de joystick analógico de 3 ejes para Arduino</li></ul>	
<ul style="list-style-type: none"><li>Cables varios</li></ul>	



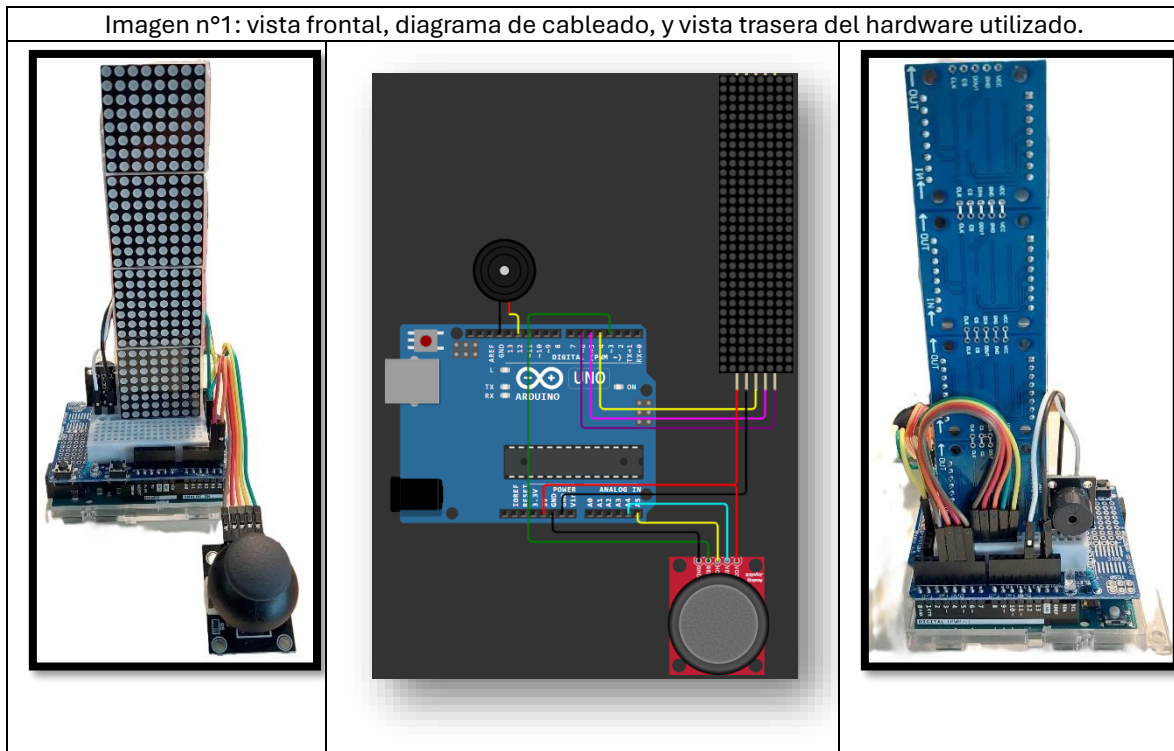
Este es el famoso juego de Tetris adaptado para 4 matrices MAX7219 (32x8) dispuesto de forma que las piezas caigan de manera vertical. El juego es controlado con joystick analógico y botón de rotación. Utiliza la librería LedControl para manejar las matrices LED. El código incluye melodía de Tetris y efectos de sonido con el buzzer. El código también gestiona la lógica de piezas, tablero, colisiones, rotaciones y puntuación. Además, muestra cuenta regresiva y pantalla de “Game Over” con animaciones y puntuación.

### Características generales

Todos los juegos están diseñados para ser minimalistas y funcionar con el mismo hardware sencillo y de bajo costo. El documento incluye el código fuente completo en Arduino para cada juego, con comentarios explicativos en español. Se detallan las funciones principales, la inicialización de hardware, la gestión de entradas y salidas, y la visualización en matrices LED. Se utilizan animaciones y sonidos para mejorar la experiencia de juego.

### Cómo implementar estos juegos minimalistas en Arduino

Imagen n°1: vista frontal, diagrama de cableado, y vista trasera del hardware utilizado.



### Hardware necesario

Construye el circuito de acuerdo con el diagrama de cableado mostrado en la imagen n°1. Los pines que controlan la matriz led configurada de forma vertical son: DIN\_PIN 4, CS\_PIN 5, CLK\_PIN 6. La matriz fue configurada para que las piezas se muevan de forma vertical, por lo tanto la matriz dispone de 8 columnas y 32 filas para el movimiento de las piezas del juego. Los pines del joystick se conectan a:



JOY\_X (salida horizontal) al pin analógico A5; y JOY\_Y (salida vertical) al pin analógico A4, y el botón de rotación (botón integrado SW del joystick) se conecta al pin digital 3 del Arduino UNO. El Buzzer se conecta al pin digital 12 de la placa y a GND.

## Estructura básica del código TETRIS

### Inicialización

- Se utiliza la librería LedControl para controlar las matrices LED.
- Se definen los pines para el joystick, el botón de rotación y el buzzer.
- Se inicializan las matrices LED y se prepara la primera pieza.

### Variables principales

- **currentPiece[3][3]**: Matriz que representa la pieza actual.
- **board[32][8]**: Tablero lógico (32 columnas x 8 filas).
- **currentX**, **currentY**: Posición de la pieza en el tablero.
- **score**: Puntuación acumulada.
- **gameOver**: Estado del juego.
- **shapes[7][3][3]**: Matrices que definen las 7 formas de piezas de Tetris.

## Lógica del juego

### Bucle principal (loop)

- Si el juego terminó, muestra la pantalla de “Game Over”, la puntuación y reinicia el juego.
- Si no, lee el joystick para mover la pieza a izquierda/derecha y el botón para rotar la pieza.
- La pieza cae automáticamente cada cierto intervalo de tiempo.
- Si la pieza no puede bajar más, se fija al tablero, se comprueban columnas completas y se genera una nueva pieza.
- Si no hay espacio para la nueva pieza, el juego termina.

### Lógica principal

- **Lectura del joystick**: Detecta el movimiento horizontal y la rotación de la pieza.
- **Movimiento y colisiones**: Comprueba si la pieza puede moverse o rotar sin colisionar con otras piezas o los límites.
- **Fijar pieza**: Cuando la pieza no puede bajar más, se añade al tablero.
- **Eliminar columnas**: Si una columna está completa, se elimina y se desplazan las demás.
- **Puntuación**: Se incrementa por cada columna eliminada.
- **Animaciones y sonidos**: Se muestran animaciones y se reproducen sonidos con el buzzer.

### Funciones clave

- **spawnPiece()**: Inicializa una nueva pieza en la parte superior y centrada.
- **movePiece(dy)**: Mueve la pieza horizontalmente si es posible.
- **dropPiece()**: Baja la pieza; si no puede bajar, la fija al tablero y comprueba columnas completas.
- **rotatePiece()**: Rota la pieza 90° si es posible.



- **canMove(x, y):** Comprueba si la pieza puede moverse a la posición indicada.
- **addPieceToBoard():** Añade la pieza actual al tablero.
- **checkForFullColumns():** Elimina columnas completas y actualiza la puntuación.
- **drawBoard()** y **drawPiece():** Dibuja el tablero y la pieza actual en las matrices LED.
- **displayGameOver():** Muestra la puntuación y animación de fin de juego.
- **resetGame():** Reinicia el tablero y las variables del juego.
- **sing():** Reproduce la melodía de Tetris con el buzzer.

#### Sugerencias para adaptar el código

- Puedes ajustar la velocidad de caída modificando la variable dropInterval.
- El brillo de las matrices LED se puede ajustar con setIntensity.
- El código está comentado en español para facilitar la comprensión y modificación.

#### Código fuente TETRIS minimalista

```
/* Tetris minimalista para 4 matrices MAX7219 (32x8)
 * Controlado con joystick analógico y botón de rotación.
 * Utiliza la librería LedControl para manejar las matrices LED.
 * Adaptado de varios ejemplos de Tetris en Arduino.
 * by LabTec
 * octubre/2025
 */

#include <LedControl.h>

// Pines para el MAX7219 (LedControl)
#define DIN_PIN 4
#define CS_PIN 5
#define CLK_PIN 6
#define NUM_OF_MATRIX 4

// Joystick, botón de rotación y Buzzer
#define JOY_X A5
#define JOY_Y A4
#define JOY_SW 3 // Rotate button
#define BUZZER 12

LedControl lc = LedControl(DIN_PIN, CLK_PIN, CS_PIN, NUM_OF_MATRIX);

// Tetris melody
#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
```



**UMCE**  
el poder transformador de la educación

Laboratorio de Tecnología  
Departamento de Física  
labtec@umce.cl



```
#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
```

**Si tienes dudas, escríbenos a [labtec@umce.cl](mailto:labtec@umce.cl)**



```
#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

int tempo=144;

// Musica Tetris
int melody[] = {
  NOTE_E5, 4, NOTE_B4, 8, NOTE_C5, 8, NOTE_D5, 4, NOTE_C5, 8, NOTE_B4, 8,
  NOTE_A4, 4, NOTE_A4, 8, NOTE_C5, 8, NOTE_E5, 4, NOTE_D5, 8, NOTE_C5, 8,
  NOTE_B4, -4, NOTE_C5, 8, NOTE_D5, 4, NOTE_E5, 4,
  NOTE_C5, 4, NOTE_A4, 4, NOTE_A4, 4, REST, 4,

  NOTE_D5, -4, NOTE_F5, 8, NOTE_A5, 4, NOTE_G5, 8, NOTE_F5, 8,
  NOTE_E5, -4, NOTE_C5, 8, NOTE_E5, 4, NOTE_D5, 8, NOTE_C5, 8,
  NOTE_B4, 4, NOTE_B4, 8, NOTE_C5, 8, NOTE_D5, 4, NOTE_E5, 4,
  NOTE_C5, 4, NOTE_A4, 4, NOTE_A4, 4, REST, 4,
};

// sizeof indica el número de bytes; cada valor int se compone de dos bytes (16 bits).
// Hay dos valores por nota (tono y duración), por lo que cada nota tiene cuatro bytes.
int notes=sizeof(melody)/sizeof(melody[0])/2;

// Calcula la duracion delas notas en ms (60s/tempo)*4 beats
int wholenote = (60000 * 4) / tempo;
int divider = 0, noteDuration = 0;

// Dimensiones lógicas del tablero (4 matrices x 8 columnas = 32x8)
const int MATRIX_WIDTH = 32; // 4 matrices de 8 columnas
const int MATRIX_HEIGHT = 8; // Altura fija

// Pieza actual (3x3), posición en el tablero y tiempos
uint8_t currentPiece[3][3];
unsigned long lastDropTime;
unsigned long lastRotateTime;
const int rotateDebounceTime = 250;
const int dropInterval = 250; // tiempo de caída (ms)
int currentX, currentY;
int score = 0;
int velocidad = 100; //ajusta la velocidad de muestreo
bool gameOver = false;
int centrado = 0;

// Estado del tablero: true = LED encendido (ocupado)
bool board[MATRIX_WIDTH][MATRIX_HEIGHT] = {false};

// Definición de 7 "shapes" (cada una en una matriz 3x3)
// Nota: orientaciones y anclas dentro de 3x3
const uint8_t shapes[7][3][3] = {
  {{1, 1, 1}, {0, 0, 0}, {0, 0, 0}}, // línea de 3 bits
  {{1, 1, 1}, {0, 1, 0}, {0, 0, 0}}, // T
  {{1, 1, 0}, {0, 0, 1}, {0, 0, 0}}, // Z
  {{1, 0, 0}, {1, 1, 1}, {0, 0, 0}}, // J (L invertida)
  {{1, 1, 0}, {1, 1, 0}, {0, 0, 0}}, // bloque de 2x2 bits
  {{0, 1, 1}, {1, 1, 0}, {0, 0, 0}}, // S
```



```
{0, 0, 1}, {1, 1, 1}, {0, 0, 0}}, // L
};

void setup() {
  //Serial.begin(9600);
  pinMode(JOY_SW, INPUT_PULLUP);

  // Inicializar las 4 matriz de led
  for (int i = 0; i < NUM_OF_MATRIX; i++) {
    lc.shutdown(i, false);
    lc.setIntensity(i, 8); // brillo medio
    lc.clearDisplay(i);
  }

  displayCountdown(); //Cuenta regresiva antes de iniciar
  sing();
  spawnPiece();      // crear la primera pieza
  lastDropTime = millis();
  lastRotateTime = 0;
  tone(BUZZER, 1000);
  delay(100);
  noTone(BUZZER);
}

void loop() {
  // Si el juego terminó, mostrar pantalla final y no continuar lógica
  if (gameOver) {
    displayGameOver();
    displayCountdown();
    sing();
    tone(BUZZER, 1000);
    delay(100);
    noTone(BUZZER);
    return;
  }

  // Lectura del joystick para mover a izquierda/derecha (en Y se define movimiento)
  if (analogRead(JOY_Y) < 400) {
    movePiece(-1); // mover a la izquierda (dy = -1 sobre currentY)
    delay(1);
  }
  if (analogRead(JOY_Y) > 600) {
    movePiece(1); // mover a la derecha (dy = +1 sobre currentY)
    delay(1);
  }

  // Botón de rotación con anti-rebote simple
  if (digitalRead(JOY_SW) == LOW && (millis() - lastRotateTime > rotateDebounceTime)) {
    rotatePiece();
    lastRotateTime = millis();
  }

  // Intervallo dinámico de caída: si JOY_X > threshold, cae más rápido
  unsigned long effectiveDropInterval = dropInterval;
  const int JOY_X_FAST_THRESHOLD = 600;
  const float FAST_MULTIPLIER = 0.01; // 1% del intervalo normal (cayendo muy rápido)

  if ((analogRead(JOY_X)) > JOY_X_FAST_THRESHOLD) {
    // Evitar un intervalo demasiado corto con max(50,...)
    effectiveDropInterval = max(50, (int)(dropInterval * FAST_MULTIPLIER));
  }
}
```



```
// Comprobar si toca hacer una caída automática
if (millis() - lastDropTime >= effectiveDropInterval) {
    dropPiece();
    lastDropTime = millis();
}

drawBoard(); // dibujar tablero y pieza actual
delay(velocidad); // retraso para reducir la velocidad de muestreo
}

/* spawnPiece()
 * Inicializa currentPiece con una forma aleatoria y coloca la pieza
 * en la fila superior (currentX = 0) y centro vertical.
 */
void spawnPiece() {
    currentX = 0;
    currentY = MATRIX_HEIGHT / 2 - 1;
    int randShape = random(0, 7);
    memcpy(currentPiece, shapes[randShape], sizeof(currentPiece));
}

/* movePiece(dy)
 * mueve la pieza horizontalmente (en el código dy representa cambio en Y lógico).
 * Comprueba colisión con canMove.
 */
void movePiece(int dy) {
    if (canMove(currentX, currentY + dy)) {
        currentY += dy;
    }
}

/* dropPiece()
 * Intenta bajar la pieza (aumenta currentX).
 * Si no puede bajar, fija la pieza al tablero, comprueba columnas completas,
 * genera una nueva pieza y evalúa game over.
 */
void dropPiece() {
    if (canMove(currentX + 1, currentY)) {
        currentX++;
    } else {
        addPieceToBoard();
        checkForFullColumns();
        spawnPiece();
        if (!canMove(currentX, currentY)) {
            gameOver = true;
        }
    }
}

/* drawBoard()
 * Limpia las matrices y dibuja el estado del tablero y la pieza actual.
 * Atención: la conversión de coordenadas a matrices requiere ajuste
 * dependiendo de cómo estén cableadas las matrices.
 */
void drawBoard() {
    for (int m = 0; m < NUM_OF_MATRIX; m++) {
        lc.clearDisplay(m);
    }

    // Dibujar cada LED del tablero
    for (int x = 0; x < MATRIX_WIDTH; x++) {
        for (int y = 0; y < MATRIX_HEIGHT; y++) {
```





```
if (board[x][y]) {
    // matrizIndex y localX calculados para la disposición física
    int matrixIndex = NUM_OF_MATRIX - 1 - (x / 8);
    int localX = x % 8;
    lc.setLed(matrixIndex, y, localX, true);
}
}
}

drawPiece(); // superponer la pieza actual
}

/* drawPiece
 * Dibuja la pieza en su posición actual (sin modificar el tablero).
 * Comprueba límites para no escribir fuera de rango.
 */
void drawPiece() {
    for (int y = 0; y < 3; y++) {
        for (int x = 0; x < 3; x++) {
            if (currentPiece[y][x]) {
                int globalX = currentX + x;
                int globalY = currentY + y;
                if (globalX >= 0 && globalX < MATRIX_WIDTH && globalY >= 0 && globalY < MATRIX_HEIGHT) {
                    int matrixIndex = NUM_OF_MATRIX - 1 - (globalX / 8);
                    int localX = globalX % 8;
                    lc.setLed(matrixIndex, globalY, localX, true);
                }
            }
        }
    }
}

/* addPieceToBoard()
 * Copia los bloques de currentPiece al array board (fija la pieza).
 */
void addPieceToBoard() {
    for (int y = 0; y < 3; y++) {
        for (int x = 0; x < 3; x++) {
            if (currentPiece[y][x]) {
                int boardX = currentX + x;
                int boardY = currentY + y;
                if (boardX >= 0 && boardX < MATRIX_WIDTH && boardY >= 0 && boardY < MATRIX_HEIGHT) {
                    board[boardX][boardY] = true;
                }
            }
        }
    }
}

/* canMove(x,y)
 * Comprueba si la pieza actual puede colocarse en (x,y) sin colisiones ni salirse.
 * newX/newY calculados en base a la posición de la celda dentro de currentPiece.
 */
bool canMove(int x, int y) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (currentPiece[i][j]) {
                int newX = x + j;
                int newY = y + i;
                if (newX < 0 || newX >= MATRIX_WIDTH || newY < 0 || newY >= MATRIX_HEIGHT) {
                    return false;
                }
            }
        }
    }
}
```



```
if (board[newX][newY]) {
    return false;
}
}
}
}
return true;
}

/* rotatePiece()
 * Rota currentPiece 90 grados en sentido horario (usa tempPiece para la rotación).
 * Intenta aplicar la rotación en la posición actual; si falla, intenta pequeños desplazamientos
 * en Y para "ajustar" la pieza (simple kick).
 */
void rotatePiece() {
    uint8_t tempPiece[3][3];
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            tempPiece[j][2 - i] = currentPiece[i][j];
        }
    }

    // Probar rotación en la posición actual y con pequeños desplazamientos verticales
    if (canMoveWithShape(tempPiece, currentX, currentY)) {
        memcpy(currentPiece, tempPiece, sizeof(currentPiece));
    } else if (canMoveWithShape(tempPiece, currentX, currentY - 1)) {
        currentY -= 1;
        memcpy(currentPiece, tempPiece, sizeof(currentPiece));
    } else if (canMoveWithShape(tempPiece, currentX, currentY + 1)) {
        currentY += 1;
        memcpy(currentPiece, tempPiece, sizeof(currentPiece));
    }
}

/* canMoveWithShape(shape, x, y)
 * Igual que canMove pero para una forma dada (utilizada al rotar para probar la nueva orientación).
 */
bool canMoveWithShape(uint8_t shape[3][3], int x, int y) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (shape[i][j]) {
                int newX = x + j;
                int newY = y + i;
                if (newX < 0 || newX >= MATRIX_WIDTH || newY < 0 || newY >= MATRIX_HEIGHT || board[newX][newY]) {
                    return false;
                }
            }
        }
    }
    return true;
}

/* checkForFullColumns
 * Comprueba columnas completas (todas las filas a 'true') a partir del lado derecho hacia la izquierda.
 * Si encuentra una columna llena, desplaza todas las columnas a la derecha de esa columna hacia la derecha
 * (efecto "borrar columna" y desplazar).
 * Incrementa score por cada columna eliminada y re-evalúa la misma x tras el desplazamiento (x++).
 */
void checkForFullColumns() {
    for (int x = MATRIX_WIDTH - 1; x >= 0; x--) {
        bool fullColumn = true;
        for (int y = 0; y < MATRIX_HEIGHT; y++) {
```



```
if (!board[x][y]) {
    fullColumn = false;
    break;
}
}
if (fullColumn) {
    // Desplazar columnas hacia la derecha (sustrayendo la columna encontrada)
    for (int i = x; i > 0; i--) {
        for (int j = 0; j < MATRIX_HEIGHT; j++) {
            board[i][j] = board[i - 1][j];
        }
    }
    // Vaciar la columna 0
    for (int j = 0; j < MATRIX_HEIGHT; j++) {
        board[0][j] = false;
    }
    x++; // Re-evaluar la misma posición x después del desplazamiento
    score++;
    tone(BUZZER, 1000);
    delay(50);
    noTone(BUZZER);
    delay(50);
}
}
}

/* displayGameOver()
 * Muestra la puntuación en pantalla por 2 segundos y reinicia el juego.
 * La función displayDigit coloca dígitos en la matriz 2 usando un patrón 3x5 por dígito.
 */
void displayGameOver() {
    clearAllMatrices();
    // Convertir score a String y limitar a 2 dígitos para mostrar
    String scoreStr = String(score);
    if (scoreStr.length() > 2) {
        scoreStr = scoreStr.substring(0, 2); // Limitar a 2 dígitos
    }

    // Mostrar los dígitos en orden (ajustando posición)
    for (int i = 0; i < scoreStr.length(); i++) {
        int digit = scoreStr.charAt(i) - '0'; // Convertir char a int
        if (digit >= 0 && digit <= 9) {
            int pos = scoreStr.length() - 1 - i; // Mapear índice 0 -> izquierda, 1 -> derecha
            displayDigit(digit, pos); // Mostrar dígito en la posición calculada
        }
    }
    delay(2000); // mostrar 2 segundos
    fin();
    resetGame(); // reiniciar juego
    delay(500);
}

/* displayDigit(digit, pos)
 * Dibuja un dígito (0-9) usando un patrón 3x5. Coloca los bloques en la matriz con índice 2.
 * pos controla el desplazamiento horizontal (multiplicado por 4 para separarlo).
 */
void displayDigit(int digit, int pos) {
    const int digits[10][3][5] = {
        {{1,1,1,1,1},{1,0,0,0,1},{1,1,1,1,1}}, // 0 ok
        {{0,0,0,0,1},{1,1,1,1,1},{0,1,0,0,1}}, // 1 ok
        {{1,1,1,0,1},{1,0,1,0,1},{1,0,1,1,1}}, // 2 ok
        {{1,1,1,1,1},{1,0,1,0,1},{1,0,1,0,1}}, // 3 ok
```



```
{{1,1,1,1,1},{0,0,1,0,0},{1,1,1,0,0}}, // 4 ok
{{1,0,1,1,1},{1,0,1,0,1},{1,1,1,0,1}}, // 5 ok
{{1,0,1,1,1},{1,0,1,0,1},{1,1,1,1,1}}, // 6 ok
{{1,1,1,1,1},{1,0,0,0,0},{1,0,0,0,0}}, // 7 ok
{{1,1,1,1,1},{1,0,1,0,1},{1,1,1,1,1}}, // 8 ok
{{1,1,1,1,1},{1,0,1,0,1},{1,1,1,0,1}} // 9 ok
};

// Ensure the digit is valid
if (digit < 0 || digit > 9) return; // Invalid digit, exit

if (centrado == 1){
    int digitWidth = 5;
    int digitHeight = 3;
    int startCol = (MATRIX_WIDTH - digitWidth) / 2; // Centrado horizontal
    int startRow = (MATRIX_HEIGHT - digitHeight) / 2; // Centrado vertical
    for (int row = 0; row < digitHeight; row++) {
        for (int col = 0; col < digitWidth; col++) {
            bool ledState = digits[digit][row][col];
            int globalX = startCol + col;
            int globalY = startRow + row;
            if (globalX >= 0 && globalX < MATRIX_WIDTH && globalY >= 0 && globalY < MATRIX_HEIGHT) {
                int matrixIndex = NUM_OF_MATRIX - 1 - (globalX / 8);
                int localX = globalX % 8;
                lc.setLed(matrixIndex, globalY, localX, ledState);
            }
        }
    }
} else {
    // Dibujar patrón: filas = 3, columnas = 5
    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 5; col++) {
            bool ledState = digits[digit][row][col];
            lc.setLed(2, pos*4 + row, col, ledState);
        }
    }
}

/* resetGame()
 * Vuelve a inicializar el tablero y estado para empezar de nuevo.
 */
void resetGame() {
    memset(board, false, sizeof(board));
    clearAllMatrices();
    spawnPiece();
    currentX = 0;
    gameOver = false;
    score = 0;
}

/* sing()
 * interpreta la melodía del juego
 */
void sing() {
    // itera sobre las notas de la melodía.
    for (int thisNote = 0; thisNote < notes * 2; thisNote = thisNote + 2) {
        // Calcula la duración de cada nota
        divider = melody[thisNote + 1];
        if (divider > 0) {
            // Nota regular, simplemente la ejecuta
            noteDuration = (wholenote) / divider;
        } else if (divider < 0) {
```



```
// ¡Las notas punteadas se representan con duraciones negativas!
noteDuration = (wholenote) / abs(divider);
noteDuration *= 1.5; // increases the duration in half for dotted notes
}
// Solo tocamos la nota durante el 90% de la duración, dejando el 10% como pausa
tone(BUZZER, melody[thisNote], noteDuration*0.9);

// Espera la duración específica antes de reproducir la siguiente nota.
delay(noteDuration);

// Detiene la generación de forma de onda antes de la siguiente nota.
noTone(BUZZER);
}
}

// Enciende un LED en la posición global (x, y) mapeando a la matriz correspondiente
void showLed(int x, int y) {
  int matrixIndex = 3 - (y / 8);
  int localY = y % 8;
  lc.setLed(matrixIndex, x, localY, true);
}

void clearAllMatrices() {
  for (int i = 0; i < NUM_OF_MATRIX; i++) {
    lc.clearDisplay(i);
  }
}

void displayCountdown() {
  for (int i = 3; i >= 1; i--) {
    clearAllMatrices();
    centrado = 1;
    displayDigit(i, 0); // Mostrar el número en la posición izquierda
    delay(1000); // Esperar 1 segundo
  }
  clearAllMatrices();
  centrado = 0;
}

const byte C[8] = { //icono
  B01111110,
  B10000001,
  B10100101,
  B10000101,
  B10000101,
  B10100101,
  B10000001,
  B01111110
};

// Mostrar icono en su matriz correspondiente
for (int row = 0; row < 8; row++) {
  lc.setRow(2, row, C[row]); // Matriz 1: Nota musical
}

void fin() {
  for (int i = 0; i < MATRIX_WIDTH; i++) { // Recorrer todas las filas globales
    for (int j = 0; j < MATRIX_HEIGHT; j++) { // Recorrer todas las columnas
      showLed(j, i); // Encender cada LED (x = columna, y = fila)
      delay(1); // Pequeña pausa entre LEDs para animación
    }
  }
  clearAllMatrices();
}
```



```
const byte F[8] = { //F
  B00000000,
  B01000000,
  B01001000,
  B01001000,
  B01111110,
  B00000000,
  B00000000,
  B00000000
};
const byte I[8] = { //I
  B00000000,
  B00000000,
  B01000010,
  B01111110,
  B01000010,
  B00000000,
  B00000000,
  B00000000
};
const byte N[8] = { //N
  B00000000,
  B01111110,
  B00001100,
  B00011000,
  B00110000,
  B01111110,
  B00000000,
  B00000000
};

// Mostrar cada letra en su matriz correspondiente
for (int row = 0; row < 8; row++) {
  lc.setRow(3, row, F[row]); // Matriz 3: F
  lc.setRow(2, row, I[row]); // Matriz 2: I
  lc.setRow(1, row, N[row]); // Matriz 1: N
}

// Efecto de sonido de game over
for (int i = 0; i < 3; i++) {
  tone(BUZZER, 300 - (i * 50), 300);
  delay(400);
}
noTone(BUZZER);
}
```

**Espero que todo funcione.**