**Project by:**

**1) BT2024134: Vansh Desai**

**2) BT2024158: Dev Taggar**

**3) BT2024121: Parth Kapoor**

**4) BT2024142: Jeswin Jose**

**5) BT2024046: Mitansh Shringi**

**Please refer the README first.**

**Micro:bit Implementation (TypeScript)**

**Data Structures and Variables**

* **Game States**: Uses an enum GameState with three states: Running, Paused, and GameOver.
* **Movement Direction**: Enum MoveDirection defines four directions: Up, Right, Down, and Left, with numeric values 0-3.
* **Snake Representation**: Stored as an array of segments, each with x/y coordinates and direction information.
* **Accelerometer Data**: Uses buffer arrays to store and smooth accelerometer readings for movement detection.
* **Game Variables**: Tracks score, treat position, movement counters, and timing information.

**Accelerometer-based Control**

function determineDirection(): void {

if (isFlat(avgZ)) return;

let xDir = 0, yDir = 0;

if (avgX < -HORIZONTAL\_THRESHOLD) xDir = -1;

else if (avgX > HORIZONTAL\_THRESHOLD) xDir = 1;

if (avgY < -VERTICAL\_THRESHOLD) yDir = -1;

else if (avgY > VERTICAL\_THRESHOLD) yDir = 1;}

* Uses a buffer system to collect and smooth accelerometer data
* Implements thresholds to filter out small movements
* Determines movement direction based on device tilt
* Prevents 180° turns (snake can't turn directly back on itself)

**Snake Movement Logic**

* Snake moves in steps based on a frame counter
* Head is added in the direction of movement
* If a treat is eaten, snake grows (tail remains); otherwise, tail is removed
* Handles screen edge wrapping (snake appears on opposite side when it reaches an edge)
* Game ends if snake collides with itself

**Treat Spawning**

* Random position generation with collision checking
* Ensures treats never spawn on snake body

**Game State Management**

* Tracks running, paused, and game over states
* Includes game timing with pause functionality
* Handles winning condition (reaching maximum score)
* Displays end-game information (score and time)

**Display and Communication**

**LED Matrix Display**

* Updates the 5×5 LED matrix to show snake segments and treat
* Implements blinking effect for the treat
* Shows game state visually (paused state flashes)

**Serial Communication**

function sendLEDState(): void {

const state = getLEDState();

const gs = gameState.toString();

if (hasLEDStateChanged(state) || gs != lastSentGameState) {

const head = { x: snake[0].x, y: snake[0].y };

const payload = {

type: "display",

matrix: state,

head: head,

gameState: gameState,

score: score,

time: formatTime(getGameTime())

};

serial.writeString(JSON.stringify(payload) + "\n");

lastSentLEDState = state;

lastSentGameState = gs;

}

}

* Sends game state data via serial connection (USB)
* Uses JSON format for structured data exchange
* Implements change detection to minimize unnecessary updates
* Sends different message types for different events (display updates, game over, reset)

**Input Handling**

* Button A: Toggle pause state
* Button B: Reset game
* Accelerometer: Control snake direction
* Includes "flat" detection to prevent unintended movement

**Game Loop and Timing**

* Main loop runs continuously to collect sensor data
* Movement occurs at regular intervals controlled by the moveThreshold
* Implements game timing system that accounts for paused states
* Formats time display with millisecond precision

**2. Web Interface Implementation (HTML/CSS/JavaScript)**

**User Interface**

* Responsive design with clean, modern styling
* 5×5 grid representation of the Micro:bit display
* Visual differentiation between snake head, body, and treat
* Animation effects (blinking treat)
* Game over overlay with results display

**Serial Communication**

async function connectToMicrobit() {

try {

const port = await navigator.serial.requestPort();

await port.open({ baudRate: 115200 });

const textDecoder = new TextDecoderStream();

port.readable.pipeTo(textDecoder.writable);

const reader = textDecoder.readable

.pipeThrough(new TransformStream(new LineBreakTransformer()))

.getReader();

while(true) {

const { value, done } = await reader.read();

if (done) break;

if (value) processSerialData(value);

}

} catch(err) {

console.error('Serial error:', err);

}

}

* Uses Web Serial API to communicate with the Micro:bit
* Implements line-based JSON parsing for incoming messages
* Connection status indication
* Error handling for connection issues

**Game State Visualization**

* Dynamically updates the game board based on received data
* Special styling for snake head, body segments, and treat
* Game over screen with score and time display
* Win/lose differentiation and messaging

**Connection Management**

* User-initiated connection via button
* Status indication for connection state
* Reconnection capability

**3. Integration Points Between Both Components**

**Data Exchange Protocol**

The Micro:bit and web interface communicate through a simple JSON-based protocol:

1. **Display Updates**:
2. {
3. "type": "display",
4. "matrix": [[0,0,0,0,0], ...],
5. "head": {"x": 2, "y": 2},
6. "gameState": 0,
7. "score": 3,
8. "time": "12.45"
9. }
10. **Game Over Notification**:
11. {
12. "type": "gameOver",
13. "reason": "COLLISION",
14. "score": 5,
15. "time": "23.78"
16. }
17. **Game Reset**:
18. {
19. "type": "reset",
20. "state": "running"
21. }

**Synchronization Strategy**

* The Micro:bit sends updates only when the display state changes
* Uses state caching to minimize serial traffic
* Web interface parses and processes messages to update the visual representation
* Both systems use the same 5×5 matrix representation

**4. Notable Technical Aspects**

**Accelerometer Data Smoothing**

The Micro:bit implementation uses a sophisticated approach to process accelerometer data:

function calculateSmoothedValue(buffer: number[], minRange: number, maxRange: number): number {

if (buffer.length === 0) return 0;

let minVal = 2000, maxVal = -2000;

for (let v of buffer) {

if (v < minVal) minVal = v;

if (v > maxVal) maxVal = v;

}

if (minVal < minRange \* 0.9 || maxVal > maxRange \* 0.9) {

let sum = 0, count = 0;

for (let i = Math.max(0, buffer.length - 3); i < buffer.length; i++) {

sum += buffer[i];

count++;

}

return count ? sum / count : 0;

} else {

let sum = 0;

for (let v of buffer) sum += v;

return buffer.length ? sum / buffer.length : 0;

}

}

* Maintains a rolling buffer of recent readings
* Adapts averaging based on detected range of motion
* Uses different smoothing strategies for rapid vs. gradual movements
* Implements thresholds to filter out small movements

**Efficient Display Updates**

Both systems implement techniques to minimize unnecessary processing:

* Micro:bit side checks if the display has actually changed before sending updates
* Web interface processes messages by type and updates only what's needed
* Treat blinking is handled separately on each side to minimize communication

**Edge Case Handling**

The implementation handles several edge cases gracefully:

* Screen wrapping when snake crosses edges
* Direction change restrictions to prevent illegal moves
* Collision detection including self-collision
* Treat spawning with collision avoidance
* Timing pauses during game state changes

**2. Challenges Faced and Solutions Implemented**

**Challenge 1: Accelerometer Data Processing**

Problem: Raw accelerometer data from the Microbit

proved too noisy and inconsistent for precise directional control. Small movements, device tilt, and sensor noise created erratic behavior.

Solution:

* Implemented a buffer-based smoothing algorithm that collects multiple readings before determining direction
* Created an adaptive averaging system that uses different smoothing approaches based on detected movement intensity
* Established appropriate thresholds for each axis to filter out minor movements
* Added specific detection for the "flat" position to prevent unintended movement when the device is laid down

**Challenge 2: Reliable Serial Communication**

Problem: Serial communication between the Microbit and web browser needed to efficient or else the lag would cause a serious impact on user experience.

Solution:

* Developed a lightweight JSON-based protocol for structured data exchange
* Implemented change detection to minimize unnecessary updates over the serial connection
* Created a line-based parsing system to handle data streaming correctly
* Added error handling and connection status feedback
* Designed the system to handle reconnection scenarios

**Future Improvements**

* **Difficulty Levels**: Implement adjustable game speeds or movement thresholds to create easy, medium, and hard modes
* **Multi-player Mode**: Develop a two-Microbit version where players compete or collaborate
* **No-warp** mode: The user is not allowed to warp the snake making the game more difficult.