FINAL PROJECT: Labyrinth Ball-Bearing Maze Game

ECSE 444 - Microprocessors

Tofic Esses
B. Eng. U3 Electrical Engineering
tofic.esses@mail.mcgill.ca
ID: 260947836

Clarissa Baciu
B. Eng. U3 Computer Engineering
clarissa.baciu@mail.mcgill.ca
ID: 260929976

Mario Bouzakhm
B. Eng. U2 Software Engineering
mario.bouzakhm@mail.mcgill.ca
ID: 260954086

Yazan Saleh
B. Eng. U3 Computer Engineering
yazan.saleh@mail.mcgill.ca
ID: 260892738

2023-04-16

McGill University
ECSE Department

***Abstract*—This lab was designed to replicate a labyrinth ball bearing maze game using an embedded system, the STM32L4+ microprocessor. The objective of the game was to navigate a ball through a maze and reach the destination point using gravitational force. First, data from the accelerometer and gyroscope, internal sensors of the microprocessor, were combined via sensor fusion and smoothed out using a Kalman filter. Afterwards, the ball kinematics were configured using the kinematic equations, multiple mazes with varying levels of difficulty were stored in the QSPI Flash memory, and both were displayed on an I2C LCD display. Finally, several game-play features were added including a speaker and a timer countdown to enhance the user experience. Each feature was implemented independently by each team members, and integrated together to create successful ball bearing maze game for the final demonstration.**

***Index Terms*—Gyroscope, Accelerometer, Sensor Fusion, Kalman Filtering, Ball Kinematics, QSPI Flash, Maze, LCD Display, Speaker, STM32L4S5**

I. Introduction

This project was developed for the final project as part of the coursework for ECSE 444 Microprocessors at McGill University. The microcontroller used is included in the STM32L4S5 Discovery Kit.

A. Inspiration & Motivation

The inspiration for this project is derived from a toy we used to play with when we were kids. See Fig. 1 and Fig. 2 for visual reference.

Fig. 1. Maze on a Bubble Blower

Fig. 2. Ball-bearing Maze

B. Idea

The idea was to build an embedded equivalent by making use of the built-in gyroscope and accelerometer to accurately measure the orientation of the board, some logic, an I2C LCD with an open-source corresponding library to display the maze, and a speaker for sound effects. All of these features are used to enhance the user experience.

C. Key Features

The project was required to make use of at least four key features that have been learned over the course. The following features were used:

1) I2C Sensors: To gather data relating to the current position of the board relative to gravitational acceleration, the gyroscope and accelerometer sensors were chosen due to them being the sensors of choice for acquiring acceleration and rotational information [1]. These sensors are internally connected to the STM32L4+ microprocessor via the I2C2 interface.

2) Kalman Filter: The Kalman filter is a state-based adaptive estimator of a physical process. The function works by producing estimates of its state variables which are progressively updated based on the previous state and current input measurement [2]. In this experiment, it is used to normalize the movement of the ball.

3) QSPI Flash: The QSPI flash memory is used to store different maze configurations to vary the level of difficulty of the game, thereby reducing the memory used in-game.

4) Speaker, DAC & DMA : The speaker, DAC and DMA are used to play varying sounds when a win or loss is detected.

II. Implementation

A. Sensor Fusion with Kalman Filtering

1) Theory: Since the real ball-in-a-maze game works by tilting the maze from side to side to get the ball rolling, a similar implementation was realized using the internal sensors. The gyroscope and accelerometer were used to predict the orientation of the board, and when used together, they produced very accurate measurements. The process of combining data from two separate sensor to reduce uncertainty is known as sensor fusion [1].

Note that the measurements from the gyroscope were very noisy, therefore a Kalman Filter was applied to predict the true orientation. Also, note that the Kalman Filter takes time for the predicted measurements to converge to the actual value of the orientation. By applying sensor fusion, the measurements obtained were the instantaneous values.

To ensure proper measurements of the gyroscope, the bias would first have to be calculated. To calculate the gyroscope bias, multiple measurements of the board at rest were taken and the average was found. Each gyroscope reading was then subtracted from the average. Additionally, the min and max values of the measurements needed to be found to calculate the GYRO_SENS:

$$GYRO\_SENS = 1/(max - min) \tag{1}$$

This GYRO_SENS are individually calculated for the x and y measurements. The filtered gyroscope measurements is multiplied by the GYRO_SENS value for proper scaling. Once the adjusted gyro measurements are obtained, integration is applied on the readings based on the sampling period of the sensor.

$$\theta_{\text{gryo}} = \theta_{\text{gyro}-1} + m_{\text{gyro}} \cdot \Delta t \tag{2}$$

The values of the accelerometer do not require any filtering, adjustments, or calibration. The angle is directly found by normalizing each vector (x, y, z) to $\bar{a} = \left(\hat{x}, \hat{y}, \hat{z}\right)$. Using the normalized vector, the angle is obtained between the vectors using trigonometry.

$$\theta_{x,\text{accel}} = \arctan\left(\frac{\hat{y}}{\hat{z}}\right) \cdot \frac{180}{\pi}, \ \theta_{y,\text{accel}} = \arctan\left(-\frac{\hat{x}}{|\bar{a}|}\right) \cdot \frac{180}{\pi} \tag{3}$$

With representative angle of the gyroscope and accelerometer, sensor fusion is applied. Where α is 0.9 and $\beta = 1 - \alpha = 0.1$.

$$\theta = \alpha \cdot \theta_{\text{gyro}} + (1 - \alpha) \cdot \theta_{\text{accel}} \tag{4}$$

This now gives an accurate reading of the angle.

2) Board Orientation Testing: Testing was performed with the SWV Timeline Trace Graph to validate if the θ to t curve matches the behavior of the board's motion as can be seen in Fig. 3. Of course, other test cases were performed, such as seeing if the correct measurements is expected when the board is at 0°, ±45°, and ±90°.

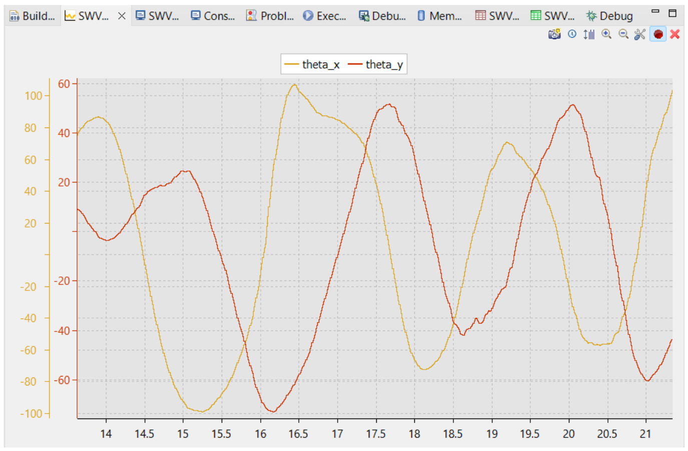


Fig. 3. SWV Timeline Trace Graph Displaying θ_x and θ_y due to a Swirling Motion

B. Ball Kinematics and Wall Collision

With accurate readings of the board's orientation, the model is simplified by approximating the ball-bearing as a box on a varying incline with no resistance.

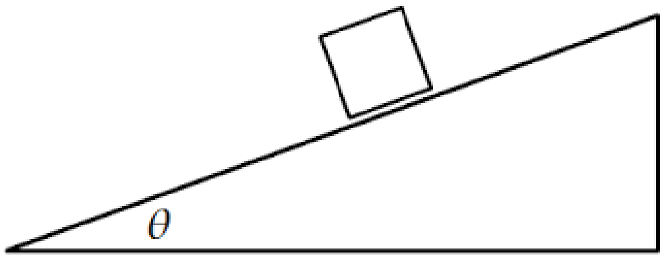


Fig. 4. Box on Incline Plane

Therefore, the instantaneous acceleration is found by:

$$\begin{aligned} a_x &= g \ \sin\left(\theta_x\right) \\ a_y &= g \ \sin\left(\theta_y\right) \end{aligned} \tag{5}$$

The new velocity is found by:

$$\begin{aligned} v_x &= v_{\text{x0}} + a_x \Delta t \\ v_y &= v_{\text{y0}} + a_y \Delta t \end{aligned} \tag{6}$$

Based on these velocities, check whether it is in the direction where a wall is present in the next pixel position. If there is a wall, then the new and previous velocity is set to zero so that no change in displacement occurs.

Otherwise, the change in displacement is found by using the trapezoidal rule:

$$\begin{aligned} d_x &= \tfrac{1}{2}\left(v_{\text{x0}} + v_x\right) \Delta t \\ d_y &= \tfrac{1}{2}\left(v_{\text{y0}} + v_y\right) \Delta t \end{aligned} \tag{7}$$

Using the change in velocity the accumulated velocity is found with:

$$\begin{aligned} D_x &= D_{\text{x0}} + d_x \\ D_y &= D_{\text{y0}} + d_y \end{aligned} \tag{8}$$

If the accumulated displacement reaches a certain scale (let's say 1/7000), then it is assumed that it has jumped to a new pixel. Therefore, the position of the ball is set to the new pixel location, and the accumulated displacement is reset to zero.

This ensures that the motion of the ball is realistically simulated on the LCD Display.

Note that there are a few more special cases that are also handled for the velocity.

- If the acceleration is in the opposite direction of the velocity, the velocity is scaled down so that there is no significant drift in the opposite direction to the incline.
- If the board is at rest, i.e. $\theta_x \approx 0$ and $\theta_y \approx 0$, the velocity is scaled by a factor of 0.98 at each time step to slow the ball down.

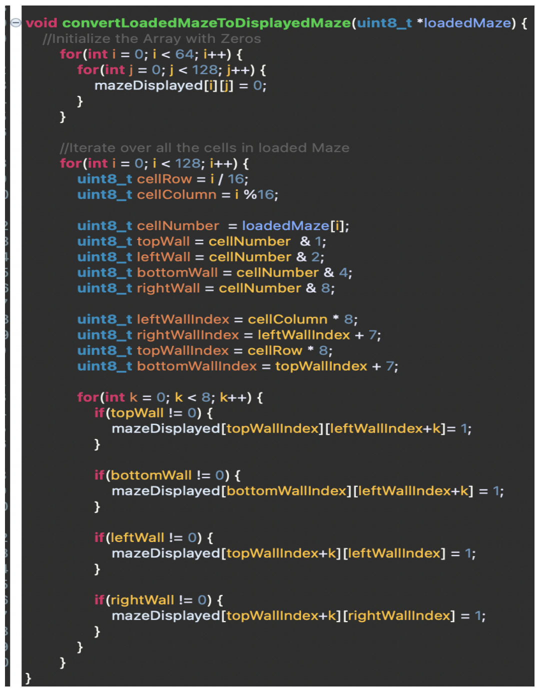
C. Maze Display with QSPI Flash

The different mazes are stored on the QSPI Flash that is available on the development board. The maze to be displayed to the user is loaded from the QSPI before the start of the game. For the game, the mazes are represented by an array of size 64x128, equivalent to the dimensions of the screen in pixels, where a 1 in the array represents a wall and a zero pixel represents a position that the ball can occupy. The advantage of saving the different mazes to the QSPI is that it reduces the amount saved on the chip itself and allows for more mazes to be added to the game. Currently, each maze occupies a sector of the flash memory, and the flash memory has 2047 sectors, meaning that the game can have at most 2047 different mazes.

Each maze contains 16 columns and 8 rows which means that there are 128 different cells in the maze. On the QSPI Flash memory, each maze is represented by 128 8-bit unsigned integers each representing a cell of the maze. Each integer is encoded in such a way that it represents the four walls around each cell:

1) The first bit represents the top wall, the second bit the left wall, the third bit, the bottom wall and the fourth bit the right wall
2) The fifth bit encodes whether this cell is the start of the maze and the sixth bit whether the cell is the end of the maze.

At the start of a a game, a random maze is loaded from the flash memory. The 128 8-bit unsigned integer array loaded is then used to produce the 64x128 array that is used to display the maze on the screen via the method shown in Fig. 5.



```c
void convertLoadedMazeToDisplayedMaze(uint8_t *loadedMaze) {
   //Initialize the Array with Zeros
      for(int i = 0; i < 64; i++) {
         for(int j = 0; j < 128; j++) {
            mazeDisplayed[i][j] = 0;
         }
      }

      //Iterate over all the cells in loaded Maze
      for(int i = 0; i < 128; i++) {
         uint8_t cellRow = i / 16;
         uint8_t cellColumn = i %16;

         uint8_t cellNumber  = loadedMaze[i];
         uint8_t topWall = cellNumber  & 1;
         uint8_t leftWall = cellNumber & 2;
         uint8_t bottomWall = cellNumber & 4;
         uint8_t rightWall = cellNumber & 8;

         uint8_t leftWallIndex = cellColumn * 8;
         uint8_t rightWallIndex = leftWallIndex + 7;
         uint8_t topWallIndex = cellRow * 8;
         uint8_t bottomWallIndex = topWallIndex + 7;

         for(int k = 0; k < 8; k++) {
            if(topWall != 0) {
               mazeDisplayed[topWallIndex][leftWallIndex+k]= 1;
            }

            if(bottomWall != 0) {
               mazeDisplayed[bottomWallIndex][leftWallIndex+k] = 1;
            }

            if(leftWall != 0) {
               mazeDisplayed[topWallIndex+k][leftWallIndex] = 1;
            }

            if(rightWall != 0) {
               mazeDisplayed[topWallIndex+k][rightWallIndex] = 1;
            }
         }
      }
}
```

Fig. 5. Convert Flash Maze to Display Maze

D. Speaker for UX

We have selected two different sounds:

- WIN/Victory sound
- GAME OVER/Defeat sound

These sounds have been searched for on YouTube and converted to an MP3 file using a converter [6]. A Python script has been written that uses the audio2numpy library to discretize the Mp3 file into a list. This Python list must be converted to an array in C with 12-bit precision. This has been done with the same script. The resulting array is stored in a header file. The sound can be played by passing the address of the array to the DMA so that it can be outputted to the DAC.

E. Additional Game-play Features

Additional features were added to the game to enhance the user experience and add competitive factors, therefore engaging the user's reward center within the brain and increasing the probability of multiple uses.

1) Push-Button Feature: A push-button element was integrated to randomly select between mazes. As explained in the previous section, the QSPI flash memory was used to store multiple mazes with varying levels of difficulty. To select between these, the push-button feature was added by enabling external hardware interrrupts and creating the HAL_GPIO_EXTI_Callback function.

This function starts by verifying that the GPIO_PIN_13, the pin corresponding to the blue button, calls the callback function, then calls the selectRandomMaze() function, and finally reinitializes the game with the appropriate functions. SelectRandomMaze(), from the maze.c file, selects a random maze number using rand() modulo MAZE_NUM, a constant containing the total number of mazes present in the QSPI. In addition, it calls to the readMazeFromSector() and convertLoadedMazeToDisplayedMaze() functions which, respectively, read the mazes from memory and display them to the screen.

2) Timer Feature: As there was not much of a competitive aspect to the game, a timer feature was added to the top right corner of the screen, giving the user only 30 seconds to complete the game. TIM2 was configured to enable an interrupt every second, which updates the timer on the screen using the updateTime(timeSec, MAXTIME) function from maze.c, where timeSec is the current elapsed time, initialized to 0 at the start of the game, and MAXTIME is the maximum time limit, set to 30 in this case. The function itself subtracts the elapsed time from the total time and updates it on the screen.

3) Winning & Losing Sequence: To create a reward and loss effect, two sequences were created with the winSequence() and gameOver() functions from ball_kinematics.c respectively. Both sequences are called in the main loop of the program and two booleans, winBool and lossBool, are used to keep track of the current state of the program. The loss sequence is activated when the timeSec exceeds MAXTIME, in other words, when the timer runs out. The loss string is then displayed onto the screen for a duration of 500ms as shown in Fig. 6, the appropriate losing sound is played, and the game is restarted with a random maze selected.

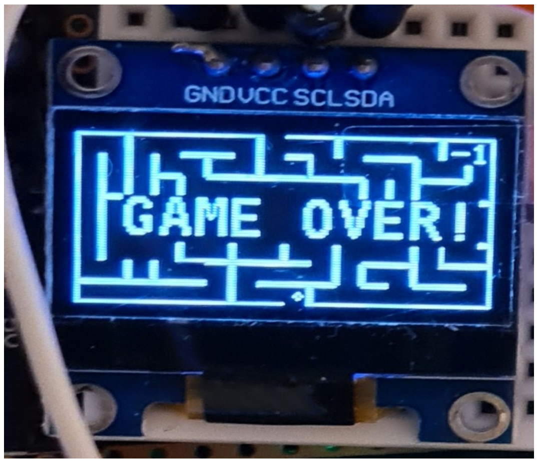


Fig. 6. Game Over Display

The winning sequence is called when the ball reaches the exit situated at the top of the maze before the timer elapses. To check for this situation, the Calculate_Ball_Velocity() function in ball_kinematics.c checks for a win every time there is a collision in the upwards direction. If the wall is absent at the point where the ball collided with the top of the screen, the global variable *win* is set to 1 and is returned by the Determine_Ball_Movement() function that is called in the main loop. This sets the win boolean to 1 which, in turn, enables winSequence(). Similar to the loss sequence, the win sequence displays a winning message on the screen shown in Fig. 7, plays the appropriate winning sounds and reinitializes the game.

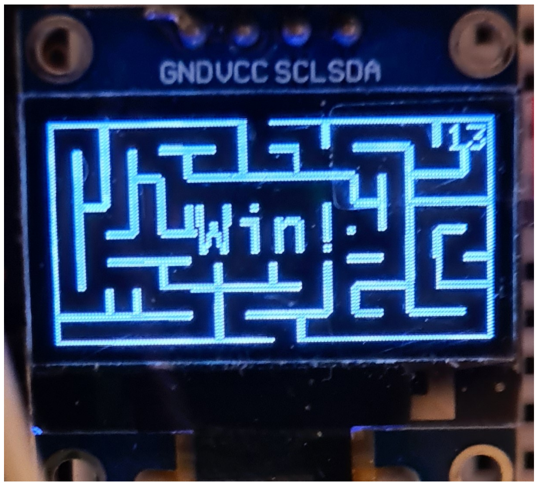


Fig. 7. Win Display

F. Pin-out

The pin-out for the LCD display is:

- GND ⇐ GND
- VCC ⇐ 3V
- SCL ⇐ D15
- SDA ⇐ D14

The pin-out for the speaker is:

- + ⇐ D7
- - ⇐ GND

III. Application

The game starts with a randomly configured maze that shows up on the screen. The user then has 30 seconds to complete the maze as shown in Fig. 8. If the user successfully completes the maze within that time frame, the winning sequence will take place and the display will change to the visual shown in Fig. 7, accompanied by the victory sound. Otherwise, if the user does not complete the maze in time, the display will change to the visual shown in Fig. 6, accompanied by the sound of the defeat.

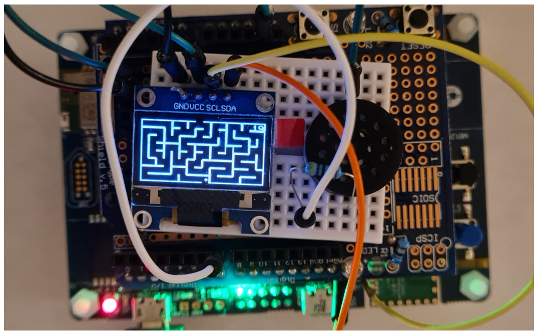


Fig. 8. Maze Display

IV. Conclusion & Future Plans

In conclusion, we were able to design and implement a user-friendly maze game that is very smooth to play. The game uses a the following peripherals: QSPI Flash Memory, on-board accelerometer and gyroscope, LCD Display, a kalman filter to smooth out the readings and DAC speaker to display various sounds. Moreover, ball kinematics were implemented to mimic real-world physics of the ball.

Here are some future ideas or improvements that can be applied to this existing project:

- High score leader board for the fastest time
- Sound of ball rolling
- Sound of ball-wall collision
- Use a bigger LCD display for more complex mazes
- Create a mobile application for this game

References

[1] ”Apply Sensor Fusion to Accelerometers and Gyroscopes,” Digi-Key Electronics, 05-Mar-2019. [Online]. Available: https://www.digikey.ca/en/articles/apply-sensor-fusion-to-accelerometers-and-gyroscopes. [Accessed: 10-Apr-2023].

[2] B. Alsadik,, “Chapter 10 - Kalman Filter” In Computational Geophysics,vol. 4, pp. 299–326, 2019. doi: https://doi.org/10.1016/B978-0-12-817588-0.00010-6.

[3] STMicroelectronics, ”Description of STM32L4/L4+ HAL and low-layer drivers,” STM32L4/L4+ Reference Manual, RM0394, Rev 7, Sept. 2017.

[4] STMicroelectronics, ”STM32L4+ Series advanced Arm-based 32-bit MCUs,” Datasheet, DS12302, Rev 8, Dec. 2020.

[5] MyCourses, ”Board Schematic Diagrams”, W2023.

[6] YouTube to MP3 Converter: YtMp3,” YtMp3. [Online]. Available: https://ytmp3.nu/. [Accessed: 10-Apr-2023].