CPRE 1850 – DS4Maze

LAB 8

SECTION CN

SUBMITTED BY:

MAXWELL MILLER

11/21/2025

Problem

This lab poses the objective of creating a random maze of characters that can be traversed through using the accelerometer built into the DualShock controller.

Analysis

In order to create a random maze, the program must be able to loop through all rows and columns and decide if they should be a wall or empty. To allow the user to traverse the maze with the controller we must read the accelerometer data with some kind of tolerance to determine the direction the user wishes to go. The controller's time input is the only available way of determining time between actions. And finally, the program must only allow user the user to go in a direction if the space if empty.

Design

The first function that was implemented was the ***generate_maze*** function. Essentially this works by looping through each row and column in the global **MAZE** 2D array and randomly filling indices with **WALL** and filling the rest of the entries with **EMPTY_SPACE**. If the density was set to zero, the function will just fill every space with **EMPTY_SPACE**. Once the maze is generated, it is then drawn by ***draw_maze***.

Then the program enters the main do-while loop, continuing after each iteration only if the avatar has not yet reached the last row. The first thing the program does is scan in the data from ds4rd.exe, saving the new data to the respective instantaneous variables. To earn the bonus points, the script then checks if the avatar can move, if it cannot, the game is ended and "GOT STUCK!" is printed to the terminal.

Next, the script checks if it's time to move, checking if the time delta between moves is greater than, or equal to **MOVE_DELAY**. If this condition is met, the **last_move_time** variable is set to the current time. Then the roll is calculated, using the roll scaling function: ***calc_roll***. Using this, the program checks if the controller has reached the **ROLL_THRESHOLD** in either the left or right directions. If it meets one of these conditions, the ***move_avatar*** function is called to move the avatar in the desired direction. When the **move_avatar** function is called, it erases the avatar from its current location in **MAZE** and then adds it to the new location in **MAZE**. Finally, we check if the avatar can move down. If it is able to, the program calls ***move_avatar*** to move the avatar down one index. This event loop is continued until either the avatar can't move, or it reaches the last row and wins.

In order to earn the all-bonus points, the script checks each iteration if the avatar has gotten stuck. In order to do that, it calls the ***canMove*** function. This function first checks if the avatar can move down in its currently location, if it can then it is not stuck. Next, it checks if it's in a small box where it surrounded by adjacent walls on the below, left, and right. If the avatar is directly surround by walls, then the function returns false. Then the script iterates through all indices to the right, then left until it reaches a wall. In this loop, it checks if in any of these indices can move down, if they can, then the avatar isn't stuck. If a way down cannot be found in either direction, the avatar is then determined to be stuck.

Testing

In order to test this program, a number of densities were tested, and movements tried. Ultimately it was determined to meet the requirements presented by the lab specifications. It was then demonstrated to the undergraduate TAs, and was determined to meet all requirements.

Comments

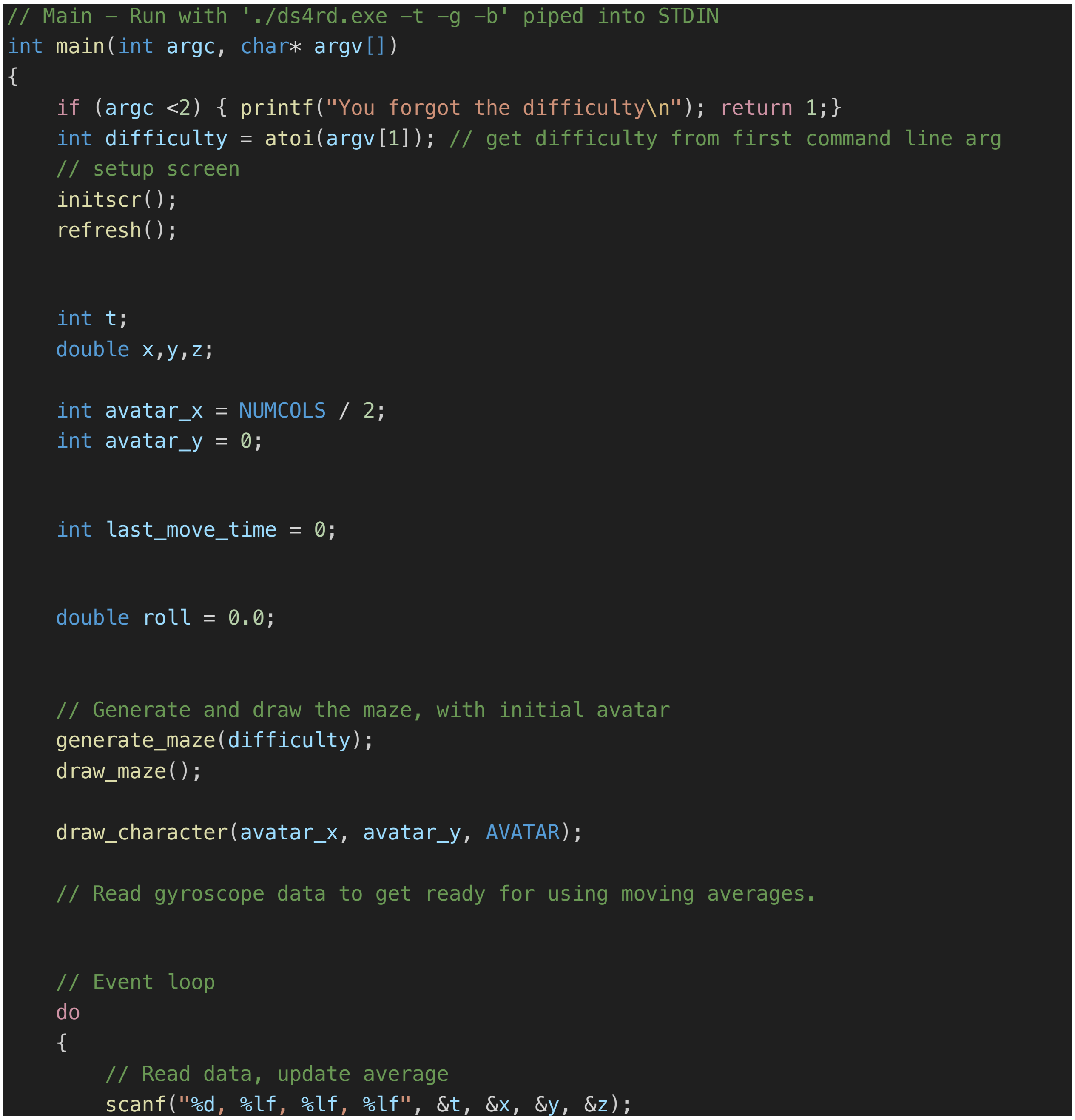
The given state machine in part 2 of the lab outline only allows the user to move left or right once before the avatar moves down. This isn't ideal and makes gameplay, especially in high density mazes, incredibly difficult. If this were to be made into an actual playable game, it would need to allow more flexibility in the avatar's motion in order to be more functional.

Additionally, it is possible for the program to create a maze that isn't traversable, especially with larger density inputs. A finished version of the game would need to determine if the maze generated is actually traversable. The odds are relatively low at reasonable density mazes, but not impossible, making the functionality mixed between runs.

Code implementation

```
// Lab 8 - Part 2

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ncurses/ncurses.h>
#include <unistd.h>

// Mathematical constants
#define PI 3.14159

// Screen geometry
// Use ROWS and COLS for the screen height and width (set by system)
// MAXIMUMS
#define NUMCOLS 100
#define NUMROWS 72

// Character definitions taken from the ASCII table
#define AVATAR 'A'
#define WALL '*'
#define EMPTY_SPACE ' '

#define MOVE_DELAY 100
#define ROLL_THRESHOLD 0.4


// 2D character array which the maze is mapped into
char MAZE[NUMROWS][NUMCOLS];


// Functional Prototypes
int isCellOpen(int x, int y);
int canMoveLeft(int x, int y);
int canMoveRight(int x, int y);
int canMoveDown(int x, int y);
int canMove(int x, int y);
void move_avatar(int x0, int y0, int x1, int y1);
void generate_maze(int difficulty);
void draw_maze(void);
void draw_character(int x, int y, char use);
float calc_roll(float x_mag);
```

Continued on the next page.



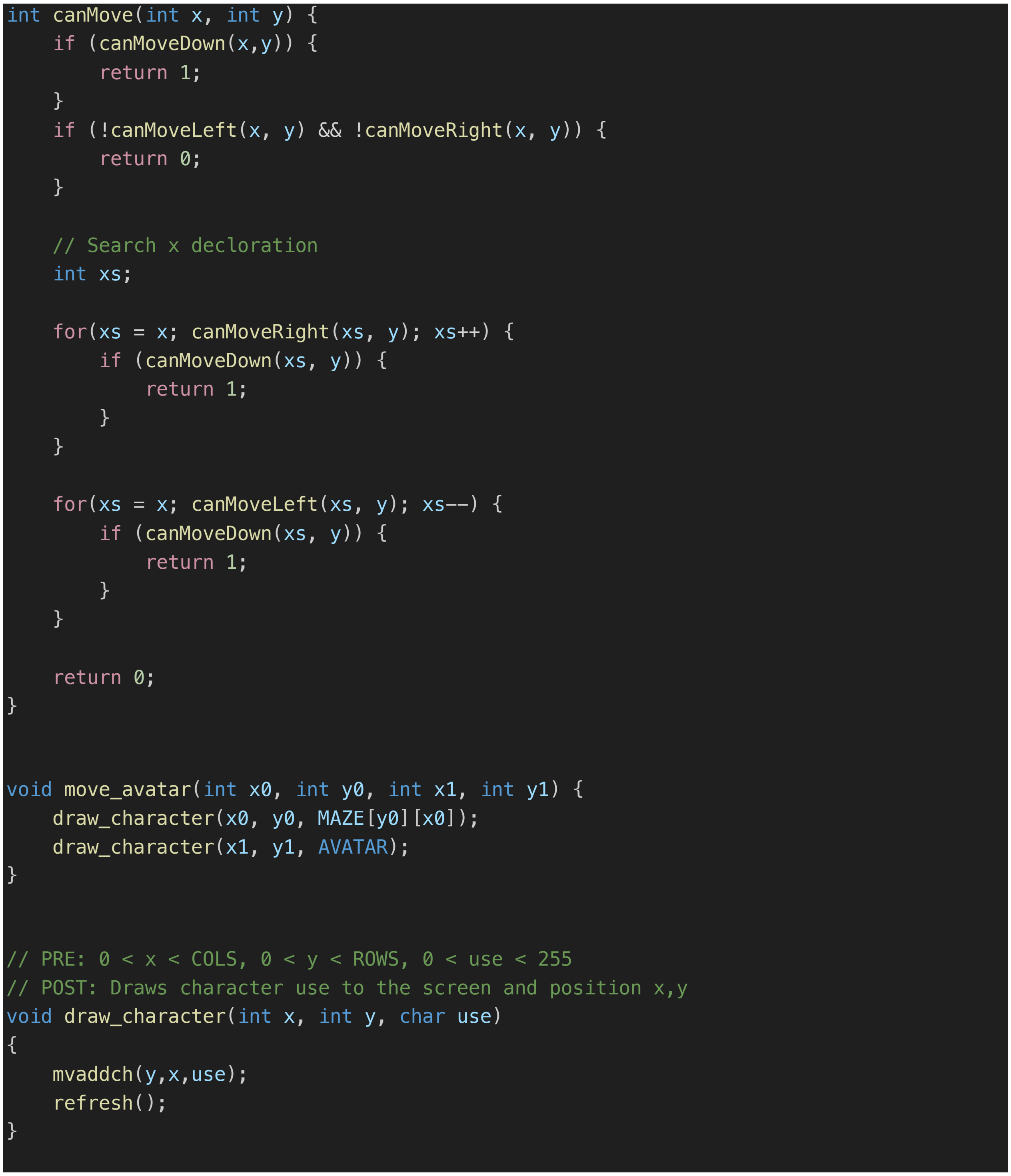
```c
// Main - Run with './ds4rd.exe -t -g -b' piped into STDIN
int main(int argc, char* argv[])
{
    if (argc <2) { printf("You forgot the difficulty\n"); return 1;}
    int difficulty = atoi(argv[1]); // get difficulty from first command line arg
    // setup screen
    initscr();
    refresh();


    int t;
    double x,y,z;

    int avatar_x = NUMCOLS / 2;
    int avatar_y = 0;


    int last_move_time = 0;


    double roll = 0.0;


    // Generate and draw the maze, with initial avatar
    generate_maze(difficulty);
    draw_maze();

    draw_character(avatar_x, avatar_y, AVATAR);

    // Read gyroscope data to get ready for using moving averages.


    // Event loop
    do
    {
        // Read data, update average
        scanf("%d, %lf, %lf, %lf", &t, &x, &y, &z);
```

Continued on the next page.

```
        // Check if we can't move
        // If not: End the game!
        // BONUS #1/#2
        if (!canMove(avatar_x, avatar_y)) {
            endwin();

            printf("GOT STUCK! GAME OVER!\n");

            return 0;
        }

        // Is it time to move?  if so, then move avatar
        if (t / MOVE_DELAY > last_move_time) {
            last_move_time = t / MOVE_DELAY;

            roll = calc_roll(x);

            // Left Move?
            if (roll > ROLL_THRESHOLD && canMoveLeft(avatar_x, avatar_y)) {
                move_avatar(avatar_x, avatar_y, avatar_x - 1, avatar_y);
                avatar_x--;
            }

            // Right Move?
            else if (roll < -ROLL_THRESHOLD && canMoveRight(avatar_x, avatar_y)) {
                move_avatar(avatar_x, avatar_y, avatar_x + 1, avatar_y);
                avatar_x++;
            }

            // Can we move down?
            if (canMoveDown(avatar_x, avatar_y)) {
                move_avatar(avatar_x, avatar_y, avatar_x, avatar_y + 1);
                avatar_y++;
            }

        }

    } while(avatar_y + 1 < NUMROWS); // Have we won?

    // Print the win message
    endwin();

    printf("YOU WIN!\n");
    return 0;
}
```
Continued on the next page.

```c
// PRE: 0 < x < COLS, 0 < y < ROWS
// POST: Returns 1 if cell is open, 0 if cell is a wall or out of bounds
int isCellOpen(int x, int y) {
    if (x < 0 || x >= NUMCOLS || y < 0 || y >= NUMROWS) {
        return 0;
    }
    if (MAZE[y][x] == WALL) {
        return 0;
    }
    return 1;
}


int canMoveLeft(int x, int y) {
    return isCellOpen(x - 1, y);;
}


int canMoveRight(int x, int y) {
    return isCellOpen(x + 1, y);
}


int canMoveDown(int x, int y) {
    return isCellOpen(x, y + 1);
}
```

Continued on the next page.



```c
int canMove(int x, int y) {
    if (canMoveDown(x,y)) {
        return 1;
    }
    if (!canMoveLeft(x, y) && !canMoveRight(x, y)) {
        return 0;
    }

    // Search x decloration
    int xs;

    for(xs = x; canMoveRight(xs, y); xs++) {
        if (canMoveDown(xs, y)) {
            return 1;
        }
    }

    for(xs = x; canMoveLeft(xs, y); xs--) {
        if (canMoveDown(xs, y)) {
            return 1;
        }
    }

    return 0;
}


void move_avatar(int x0, int y0, int x1, int y1) {
    draw_character(x0, y0, MAZE[y0][x0]);
    draw_character(x1, y1, AVATAR);
}


// PRE: 0 < x < COLS, 0 < y < ROWS, 0 < use < 255
// POST: Draws character use to the screen and position x,y
void draw_character(int x, int y, char use)
{
    mvaddch(y,x,use);
    refresh();
}
```

Continued on the next page.

```c
// POST: Generates a random maze structure into MAZE[][]
void generate_maze(int difficulty) {
    int i,j;
    int random = 0.0;

    for(i = 0; i < NUMROWS; i++) {
        for (j = 0; j < NUMCOLS; j++) {

            if (difficulty == 0) {
                MAZE[i][j] = EMPTY_SPACE;
                continue;
            }

            random = rand() % 100;

            if (random < difficulty) {
                MAZE[i][j] = WALL;
                continue;
            }

            MAZE[i][j] = EMPTY_SPACE;
        }
    }
}


// PRE: MAZE[][] has been initialized by generate_maze()
// POST: Draws the maze to the screen
void draw_maze(void) {
    int i,j;

    for(i = 0; i < NUMROWS; i++) {
        for (j = 0; j < NUMCOLS; j++) {
            mvaddch(i,j,MAZE[i][j]);
        }
    }
}


// PRE: -1.0 < x_mag < 1.0
// POST: Returns tilt magnitude scaled to -1.0 -> 1.0
float calc_roll(float x_mag) {
    if (x_mag > 1.0)    {   return 1.0;     }
    if (x_mag < -1.0)   {   return -1.0;    }

    return x_mag;
}
```