DS4 MAZE

LAB 8

SECTION C

SUBMITTED BY:

WESLEY REYNOLDS

SUBMISSION DATE:

11/17/25

Problem

In this lab, we were tasked with creating a simple, real-time game all controlled by a DualShock 4 controller. The game is going to be a randomly generated maze. The player must navigate this maze, without getting stuck, while the avatar descends towards the bottom. The avatar starts at the top-center of the maze. If the avatar gets stuck in a position where it can no longer descend, the player loses; if the avatar reaches the bottom, the player wins.

Analysis

There are a few requirements that our maze game must meet. We need to randomly generate a maze for the player to navigate. We need to then start the avatar at the top-center of the maze and, after waiting for initial player input, have it slowly fall down at any given interval. The player must have controller of this avatar by rolling the controller left and right. This should make the avatar move in the respective direction and should not move into spaces occupied by or outside of the maze. If the avatar gets to the bottom of the screen, the player wins and is notified as such. If the avatar can no longer descend (e.g. getting stuck in a bucket), the player loses and is notified as such.

Design

In my mind, there were *six* major problems that needed to be solved. I needed to:

1) Control the Avatar's movement,
2) Detect collisions with the maze and its edge,
3) Generate a random maze
4) Draw the generated maze
5) Detect when the avatar was soft-locked (impossible to descend),
6) Detect when the player has won

To implement problem one, I utilize two different functions: roll() and scaleControllerMovement(). The roll function takes in one value, *xMag*, which is the controller's X-axis gyroscope, clamps it between negative one and one, and then returns the arcsine of that value. This return value is then fed into my scaleControllerMovement function, which takes in a single value *rad*. In my game, this function returns a value between -3 and 3 based on how far the controller is rotated.

With both functions combined, we get a value between -3 and 3 based on the rotation of the controller. This value directly maps to the avatar's speed allowing for more fine or rough control depending on the player's wishes.

Problem two is slightly more complicated. I will speak about this further in the comments section and why my move_avatar() function is more complex than it needs to be in its current state. With regards to horizontal movement, all I do is check a space ahead in the direction that the avatar is currently moving. If that space is a wall, then we do not update the avatar's position. If it is, we increment the avatar's X position by *deltaX*. At the bottom of the move_avatar() function I check if the avatar's X position is out of bounds of the maze; if it is, I set the X position to the nearest X in bounds. With regards to vertical movement, I check if the space directly below the avatar is empty, and, if it is, I update the avatar's Y position by 1.

Generating a random maze for problem three wasn't too hard. We don't know any algorithms or anything to generate an actually interesting maze, so we just randomly place walls. It is important to note that in my main function I utilize time.h along with srand to ensure that the maze is random each run (srand(time(NULL))). If you do not do this, you will get the same maze for each compiled version of the program! Then, in my generate_maze() function, which takes in an integer for difficulty, I iterate through the columns and rows of the maze. At each step I generate a random number between zero and ninety-nine. This is then compared to the provided difficulty value. If the random number is below the difficulty value, then we set that space to wall. Otherwise, nothing happens. This allows for higher difficulty values to directly correlate to harder experiences.

For problem four, we iterate through the columns and rows of the maze yet again. At each step, we check if the given space is a wall or an empty space. Depending on which it is, we print the corresponding character.

Problem five was probably the most difficult conceptually, but when you sit down to think about the problem, it becomes easier. I do the following checks in a function called *stuck()*: I check to the avatar's right, and if that space is empty, I then check the space right below that. If that space is empty, then we break from the loop, as the avatar is not stuck. If the second space checked is a wall, then I check two spaces to the avatar's right. If that space is empty, then we check the one below. So on and so forth. We do this until we either find a wall that is on the avatar's Y level, or we find a gap for the avatar to go through.

Last but not least, detecting when the player has won. All we need to do for this problem is check if the avatar's Y position is at the last Y value in the maze. The default number of rows is seventy-two, so when the player is at Y 71, they should win. My code sets a win flag equal to

one when this condition is true. This stops the do-while loop from running and lets the code continue to a print statement. This print statement describes if the player has either won or lost, depending on the state of the win flag.

Testing

To verify that my program was behaving correctly, I fashioned a few tests.

1) Run, close, and re-run the program. Each time you do this, make sure the maze is being generated and is done so randomly.
2) Ensure that when you run the game the avatar is placed at the top-center of the maze.
3) Watch the avatar as it falls with a lower difficulty. Ensure that it falls at a consistent rate and does not skip a step or double up on movements.
4) Tilt the DualShock 4 left, and the avatar should move to the left.
5) Tilt the DualShock 4 right, and the avatar should move to the right.
6) Try to move into the walls of the maze and into the edges; the avatar should be unable to go through these spaces.
7) Complete the maze and make sure that a win condition is possible.
8) On a higher difficulty, find a bucket where it would be impossible to descend further and place the avatar inside. If the game ends after placing the avatar inside, the soft lock detection works.

My program successfully passes all these tests.

Comments

I really liked this lab. It felt like an actual problem that we were solving every step of the way. Video games have always been an interest of mine, so this lab was interesting doubly-so. I also enjoyed how this lab felt as a culmination of all the prior labs. We are starting to put everything together into one neat program. The requirements in this lab document were clearly laid out and there is nothing I would change.

I will say, however, that I did end up making things difficult on myself. I wanted my character to move at a variable speed depending on the overall tilt of the controller. This seems like an easy problem at first but has a few issues. My first naïve solution was to move the character more spaces at a time. This created an issue where the player could skip through walls. So, I had to create a collision detection system that would check ahead of the player for walls, and, if there were any, the game would stop the player and slowly move it towards the wall instead. It ended up working nicely.

I then realized that a much easier solution would be to decrease the time interval that decides when the player moves laterally. I implemented this and it works perfectly, meaning that my entire collision detection system is way too bloated. The code is still there as a safety, but it should be unnecessary.

Another problem I ran into was implementing a moving average. Following the skills I learned in the previous lab, I created a moving average that just *would not* work for some reason. The code is still there for the moving average, but I ended up just moving on to get the lab done. The final version does not use the average to calculate the roll.

Questions

1) I did not implement my collision detection in the same way as described in the lab documentation, however, it still works in roughly the same way. When I detect if the avatar can go left or right (in my move_avatar() function), I check one space to the left or right of the player respectively, and if there is a wall I don't allow the avatar to move any further. This is done by simply not updating the avatar's X position.

   To detect whether the player could fall or not, I do much the same thing except I check the space directly below the avatar. If space is empty, the avatar's Y position is updated, otherwise it is not.

   At the end of move_avatar(), I do a quick check to see if the avatar's X position is within the bounds of the maze. If it is not, I set it to the nearest possible value. (i.e. if the avatar's X position is less than 0, I set it to zero. If the avatar's X position is greater than 99, I set it to 99).

2) I have code written for this in my stuck() function. Basically, I check to the avatar's right, and if that space is empty, I then check the space right below that. If that space is empty, then we break from the loop, as the player is not stuck (yet). If the second space checked is a wall, then I check two spaces to the avatar's right. If that space is empty, then we check the one below. So on and so forth. We do this until we either find a wall that is on the avatar's Y level, or we find a gap for the avatar to go through. We iterate this on the left side of the avatar as well. This will make it so that the player loses when the avatar is stuck inside of a bucket.

   To add this to the flowchart, we could add it as a parallel route to the "Did I Win?" check. "Did I Lose?" could run through the aforementioned algorithm, and, if it is

found that the player is stuck, we go to “Announce & Exit”. If the player is not stuck the game simply continues onwards.

Implementation

home > wesley06 > CprE1850Lab > lab8 > C lab8.c

```c
// WII-MAZE Skeleton code written by Jason Erbskorn 2007
// Edited for ncurses 2008 Tom Daniels
// Updated for Esplora 2013 TeamRursch185
// Updated for DualShock 4 2016 Rursch

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

// 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 ' '

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

// POST: Generates a random maze structure into MAZE[][]
// You will want to use the rand() function and maybe use the output %100.
// You will have to use the argument to the command line to determine how
// difficult the maze is (how many maze characters are on the screen).
void generate_maze(int difficulty);

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

// 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);

/* This function takes in the current x and y positions of
 * the avatar, and then the change in x and y of the avatar.
 * Then, this function will move the avatar using the deltaX
```

```
 * Then, this function will move the avatar using the deltaX
 * and deltaY, all while taking into consideration collisions.
 * It will also delete the avatar character from where it last
 * was on screen.
 */
void move_avatar(int* curX, int* curY, int deltaX, int deltaY);

// Returns 1 if the current position is stuck.
int stuck(int curX, int curY);

// Returns -1 for num < 0, returns 1 for num > 0, returns 0 for 0
int sign(double num);

// Moving Average Functions
double avg(double buffer[], int numItems);
void updateBuffer(double buffer[], int length, int newItem);

// Controller Movement Functions
// PRE: -1.0 < x_mag < 1.0
// POST: Returns tilt magnitude scaled to -1.0 -> 1.0
// You may want to reuse the roll function written in previous labs.
double roll(double xMag);

// Returns a value between -3 and 3 depending on the roll of the controller.
int scaleControllerMovement(double rad);

// 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;
    }

    // Generate a seed based off the current time so the rand() function is more random
    srand(time(NULL));

    // Get difficulty from first command line argument
    int difficulty = atoi(argv[1]);

    // Player Data
```

```
    // Player Data
    int playerX = 50;
    int playerY = 0;

    // Controller Data
    int time, deltaX, moveTimer;
    int lengthofavg = 10;
    double gx, gy, gz;
    double agx; // Average gx
    double xBuffer[100]; // 100 is a placeholder

    // Flags
    int started = 0;    // Has the player started moving?
    int movedX = 0;     // Have we already moved on the x-axis in this time frame?
    int movedY = 0;     // Have we already moved on the y-axis in this time frame?
    int won = 0;        // Have we won the game? (reached the bottom?)
    int isStuck = 0;    // Are we stuck?

    // Read gyroscope data to get ready for using moving averages.
    for (int i = 0; i < lengthofavg; i++) {
        scanf("%d, %lf, %lf, %lf", &time, &gx, &gy, &gz);
        updateBuffer(xBuffer, lengthofavg, gx);
    }

    // Setup Screen
    initscr();
    refresh();

    // Generate and draw the maze, with initial avatar
    generate_maze(difficulty);
    draw_maze();
    move_avatar(&playerX, &playerY, 0, 0);

    // Event loop
    do
    {
        scanf("%d, %lf, %lf, %lf", &time, &gx, &gy, &gz);

        agx = avg(xBuffer, lengthofavg);

        deltaX = scaleControllerMovement(roll(gx));

        // Read data, update average
```

```
126         // Read data, update average
127         updateBuffer(xBuffer, lengthofavg, gx);
128 
129         // Avatar Placed & Waiting
130         if (abs(deltaX) > 0) {
131             started = 1;
132         }
133 
134         moveTimer = 200 - (45 * sign(deltaX) * deltaX);
135 
136         // Is it time to move?  If so, then move avatar
137         if (time % moveTimer < 20 && movedX == 0 && started == 1) {
138             move_avatar(&playerX, &playerY, 1 * sign(deltaX), 0);
139             movedX = 1;
140         } else if (time % moveTimer > 50 && movedX == 1) {
141             movedX = 0;
142         }
143 
144         // Begin Fall
145         if (time % 250 < 25 && movedY == 0 && started == 1) {
146             move_avatar(&playerX, &playerY, 0, 1);
147             movedY = 1;
148         } else if (time % 250 > 125 && movedY == 1) {
149             movedY = 0;
150         }
151 
152         if (playerY > 70) { // Have we won?
153             won = 1;
154         } else {
155             isStuck = stuck(playerX, playerY);
156         }
157     } while(won == 0 && isStuck == 0);
158 
159     // Give the player a second to see that they either won or lost
160     while (time % 1000 < 950) {
161         scanf("%d, %lf, %lf, %lf", &time, &gx, &gy, &gz);
162     }
163 
164 
165     endwin();
```

```
    endwin();

    // Print the win message
    if (won == 1) {
        printf("YOU WIN!\n");
    } else {
        printf("YOU LOSE!\n");
    }

    return 0;
}


void generate_maze(int difficulty)
{
    int tRand; // temporary random integer

    for (int i = 0; i < NUMCOLS; i++) {
        for (int j = 0; j < NUMROWS; j++) {
            tRand = rand() % 100;
            if (tRand < difficulty) {
                MAZE[j][i] = WALL;
            } else {
                MAZE[j][i] = EMPTY_SPACE;
            }
        }
    }
}

void draw_maze()
{
    for (int i = 0; i < NUMCOLS; i++) {
        for (int j = 0; j < NUMROWS; j++) {
            switch (MAZE[j][i]) {
                case WALL:
                    draw_character(i, j, WALL);
                    break;
                case EMPTY_SPACE:
                    draw_character(i, j, EMPTY_SPACE);
                    break;
            }
```

```
205             }
206         }
207     }
208 }
209
210
211 // PRE: 0 < x < COLS, 0 < y < ROWS, 0 < use < 255
212 // POST: Draws character use to the screen and position x,y
213 // THIS CODE FUNCTIONS FOR PLACING THE AVATAR AS PROVIDED.
214 //
215 //    >>>>DO NOT CHANGE THIS FUNCTION.<<<<
216 void draw_character(int x, int y, char use)
217 {
218     mvaddch(y,x,use);
219     refresh();
220 }
221
222 void move_avatar(int* curX, int* curY, int deltaX, int deltaY)
223 {
224     // The below line is likely unnecessary because collison will stop
225     // the case in which we need to redraw parts of the maze. Just in case I suppose.
226     // Erase previous avatar
227     draw_character(*curX, *curY, MAZE[*curY][*curX]);
228
229     // Is there a wall within range 'deltaX' ?
230     int wallThere = 0;
231
232     /* I wanted my player to move at varying speeds, so
233      * this is how I implemented collision. Basically it
234      * works by checking ahead a number of spaces equal to
235      * deltaX. If it finds a wall, it sets wallThere to 0
236      * and begins slowly moving the player towards the wall
237      * until it is flush with it. This was the solution I
238      * was most happy with and is actually pretty common in
239      * pixel perfect 2D games.
240      */
241     if (deltaX > 1) {
242         for (int i = *curX + 1; i <= *curX + deltaX; i++) {
```

```
242         for (int i = *curX + 1; i <= *curX + deltaX; i++) {
243             if (MAZE[*curY][i] == WALL) {
244                 wallThere = 1;
245             }
246         }
247 
248         if (wallThere == 1 && MAZE[*curY][*curX + 1] == EMPTY_SPACE) {
249             *curX += 1;
250         }
251     } else if (deltaX < -1) {
252         for (int i = *curX - 1; i >= *curX + deltaX; i--) {
253             if (MAZE[*curY][i] == WALL) {
254                 wallThere = 1;
255             }
256         }
257 
258         if (wallThere == 1 && MAZE[*curY][*curX - 1] == EMPTY_SPACE) {
259             *curX -= 1;
260         }
261     }
262 
263     // Update X
264     if (MAZE[*curY + deltaY][*curX + deltaX] == EMPTY_SPACE && wallThere == 0) {
265         *curX += deltaX;
266     }
267 
268     // Can I fall?
269     // As long as there is an empty space directly below the avatar,
270     // the avatar will be moved downward by deltaY
271     // Update Y
272     if (MAZE[*curY + deltaY][*curX + deltaX] == EMPTY_SPACE) {
273         *curY += deltaY;
274     }
275 
276     // Boundry checking
277     if (*curX < 0) {
278         *curX = 0;
279     } else if (*curX > 99) {
280         *curX = 99;
```

```
        *curX = 99;
    }

    // Place new avatar
    draw_character(*curX, *curY, AVATAR);
}

int stuck(int curX, int curY)
{
    int rWallFound = 0; // Right wall found?
    int lWallFound = 0; // Left wall found?

    if (MAZE[curY + 1][curX] == WALL && MAZE[curY][curX + 1] == WALL && MAZE[curY][curX - 1] == WALL) {
        return 1;
    }

    if (MAZE[curY + 1][curX] == EMPTY_SPACE) {
        return 0;
    }


    // We take the current position and go all the way to the right, checking for any gaps.
    // We then do the same thing going to the left. If there are none, the player is stuck,
    // as there is no possible way for the player to go any further down. Game over.
    for (int i = curX + 1; i < 100; i++) {
        if (MAZE[curY][i] == EMPTY_SPACE) {
            if (MAZE[curY + 1][i] == EMPTY_SPACE) {
                break;
            }
        } else {
            rWallFound = 1;
        }
    }

    for (int i = curX - 1; i >= 0; i--) {
        if (MAZE[curY][i] == EMPTY_SPACE) {
            if (MAZE[curY + 1][i] == EMPTY_SPACE) {
                break;
            }
        } else {
            lWallFound = 1;
```

```
            lWallFound = 1;
        }
    }

    if (lWallFound == 1 && rWallFound == 1) {
        return 1;
    }

    return 0;
}

int sign(double num)
{
    return (num > 0) - (num < 0);
}

double avg(double buffer[], int numItems)
{
    double temp = 0.0;
    for (int i = 0; i < numItems; i++) {
        temp += buffer[i];
    }

    temp /= numItems;
    return temp;
}

void updateBuffer(double buffer[], int length, int newItem)
{
    for (int i = 0; i < length - 1; i++) {
        buffer[i] = buffer[i + 1];
    }

    buffer[length - 1] = newItem;
}

double roll(double xMag) {
    if (xMag > 1.0)
        xMag = 1.0;

    if (xMag < -1.0)
        xMag = -1.0;

    return asin(-xMag);
}

int scaleControllerMovement(double rad)
{
    int temp = 0;
    if (rad < (M_PI/15) && rad > -(M_PI/15)) {
        temp = 0;
    } else if (rad < (M_PI / 4) && rad > -(M_PI/4)) {
        temp = sign(rad) * 1;
    } else if (rad < (3 * (M_PI / 7)) && rad > -(3 * (M_PI/7))) {
        temp = sign(rad) * 2;
    } else {
        temp = sign(rad) * 3;
    }

    return temp;
}
```