**Wiimaze**

**LAB 9**

**SECTION A**

**SUBMITTED BY:**

**Brian Reber**

**SUBMISSION DATE:**

**12/4/2009**

**Lab Problem**

Our goal is to use the Wiimote to help us create a maze game. The development of this game will help us learn about two-dimensional arrays, loops and top-down program design.

**Analysis**

The goal of this lab is to create a maze of characters using a two-dimensional array, and to use the roll/pitch of the Wiimote to control where the avatar moves in the maze. We will be learning about how to better use loops, and two-dimensional arrays in the process of making this program.

**Inputs**:

There will be data coming in via the Wiimote and Wiiwrap.

**Outputs**:

The output should be a maze on the screen, and an avatar moving down and across the screen according to the orientation of the Wiimote.

**Design**

Necessary Formulas:

* Average – (sum of all items) / (total number of items)

Algorithm:

1. First off, we will need to get from the command line the difficulty that the user wants to use for the current game. This will be an integer between 0 and 100.
2. We will then make a maze (a two-dimensional array of characters) using a random number generator. To get the numbers in the range we want, we will mod the random number we are given by 100 to get it between 0 and 100. We will then check and see whether that number is less than the difficulty value. If it is less than that value, we will put a wall there, otherwise it is a space.
3. We then print out the maze to the screen by iterating through the array and calling the given function for each space in the array.
4. We then fill our buffer for the x values given by the Wiimote. We chose to use the x direction because it made sense to use the roll of the Wiimote to control the avatar.
5. Once the buffer is filled, we go into the while-loop that is our game-play. Each time through the loop we will get the data from the Wiimote, update the buffer, and compute the average of the values in the buffer. We then check to see if it is time to move (we use a counter that increments each time through the loop). If it is time to move, we will call the move avatar function.
6. In the move avatar function, we first erase the previous avatar. Then we check to see if the average of the accelerations in the x-direction (the average of the buffer) is greater than our tolerance value or less than the negative of our tolerance. This way we know that the Wiimote is rolled and we need to move right or left depending on whether the value is negative or positive. We then update the x position. If the Wiimote is not rolled, we assume we are moving down and we update the y position. We then check to see if there is a wall at the updated position – if there is we revert back to the original position (the one we passed in to the function). We then draw the avatar at the new position.
7. The move avatar function returns either 0 or 1. It will return 0 if we are successfully able to move. It will return 1 if we are stuck. In our version, it checks to see if there is a wall on the right, left and bottom of the current position. I will talk in the comment section at the end how we would have implemented the full stuck feature.
8. We then return to the do-while loop. This is the end of what happens in the loop. So we check to see that we aren’t stuck, we haven’t reached the bottom of the screen, and that the user didn’t press the A button. If all 3 of those conditions have been meet, we go through the loop again.
9. If one of those conditions hasn’t been meet, we will exit out of the loop, and print out the status of the game. Either the user won, they are stuck, or they didn’t finish the game (they manually exited out by pressing the A button).

**Testing**

We ran into a few small problems when writing this program.

* One problem that everyone had was dealing with the Ncurses library. This limited what we could do on the first lab day. Once this was properly installed, things went much better.
* Another problem we had was with being off by one. We had originally started with a blank line at the top so that the avatar could move left or right at the beginning (so they weren’t stuck from the start). We later realized that this was causing our avatar to act weird – it would stop moving when it was one space away from a wall. We took this line out from the top, and it started working correctly.
* We also had some trouble getting the program to end when we reached the bottom of the screen. We would get like 4 lines past the end of the maze and it would sit there for a little while, then sometimes it would say we won, and sometimes it would say we lost. I think we changed the while loop conditions and it worked.
* We also spent a lot of time trying to figure out the best combination of numbers for the size of the buffer and the amount of times to go through the while loop to get the best game play. I think we finally came upon a good combination.

**Comments**

Lab Questions

* To check whether it is safe to go right or left, we use the average of the accelerations in the x-direction. If the average is greater than our tolerance, which we set to .15, we will move right. If it is less than the negative of our tolerance, we will move left. If neither of these conditions is met, we will move down. Once we set the x and y values, we check to see if there is a wall there – if so, we will revert back to the values before we updated them.
* In order to check to see if the player is stuck, we would need to check in all the directions we are able to move to see if there is a way out of that position. This can get quite complicated because we would need to recursively check and see if the next position we would move to has a place to move. Since this was really complicated and hard to picture, we just said that you were stuck if there was a wall directly to your right, left and below you.

**Other Comments**

In general, I thought that this lab somewhat difficult to figure out. Once we got the ncurses library working, it became much easier. We were pretty much able to figure out what needed to be done before we were given the flow chart. We had to make some minor revisions to our code when we got the flowchart, but it pretty much worked before that. One of our main problems was figuring out how to best implement the time delay. We didn’t want it to be too slow, but we also wanted to have the Wiimote be pretty responsive. I think that we have done a pretty good job at figuring out a good balance.

**Implementation**

// Nathan Brinkman and Brian Reber

// WII-MAZE Skeleton code written by Jason Erbskorn 2007

// Edited for ncurses 2008 Tom Daniels

// Headers

#include <stdio.h>

#include <math.h>

#include <ncurses/ncurses.h>

#include <unistd.h>

#include <stdlib.h>

#include <time.h>

// Mathematical constants

#define PI 3.14159

// Screen geometry

// Use LINES and COLS for the screen heigth and width (set by system)

// MAXIMUMS

#define SCREEN\_WIDTH 100

#define SCREEN\_HEIGHT 50

// Character definitions taken from the ASCII table

#define AVATAR 'A'

#define WALL '\*'

#define EMPTY\_SPACE ' '

// Number of samples taken to form an average for the accelerometer data

// Feel free to tweak this

#define NUM\_SAMPLES 25

// Not what you use for your delay.

#define LOOP\_IDLE 15

// The value used to slow down the program

#define TIME\_DELAY 45

// Tolerance for "Dead Zone"

#define TOLERANCE .15

// 2D character array which the maze is mapped into

char MAZE[SCREEN\_WIDTH][SCREEN\_HEIGHT];

// POST: Generates a random maze structure into MAZE[][]

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 < SCREEN\_WIDTH, 0 < y < SCREEN\_HEIGHT, 0 < use < 255

// POST: Draws character use to the screen and position x,y

void draw\_character(int x, int y, char use);

// PRE: -1.0 < y\_mag < 1.0

// POST: Returns tilt magnitude scaled to -1.0 -> 1.0

float calc\_pitch(float y\_mag);

// The function that reads in the data from the wiimote

int read\_acc(float\* a\_x, float\* a\_y, float\* a\_z, int\* t,

int\* Button\_one, int \* Button\_two, int\* Button\_B, int\* Button\_A);

// This function updates the average buffer

double updatebuffer(double buffer[], int length, double new\_item);

// This function averages the values in the buffer array.

double avg(double buffer[], int num\_items);

// This moves the avatar

int move\_avatar(double avg\_x, int\* current\_x, int\* current\_y);

// Main - Run with 'wiiwrap2.exe /T /A /B' piped into STDIN

int main(int argc, char\* argv[])

{

// acts as a counter for array input

int count = 0;

// an array for the averaging buffer

double x[NUM\_SAMPLES];

// a counter to slow the program

int time\_counter;

// magnitude values of x, y, and z accelerations

float a\_x, a\_y, a\_z;

// variables to hold the button statuses; b\_junk is just a temp variable

int b\_a, b\_b, b\_home, t, b\_1, b\_2, b\_junk;

// the values of the initial coordinates of the AVATAR

int current\_x = 50, current\_y = 0;

// average values for x and y

double avg\_x;

// The difficulty of the maze

int difficulty;

// a variable that allows the program to quit if the AVATAR is stuck

int stuck = 0;

// Get the difficulty from the command line

if (argc>1)

{

sscanf(argv[1], "%d", &difficulty);

printf("You entered a difficulty of %d\n", difficulty);

}

else

{

printf("Enter a difficulty on the command line\n");

return(-1);

}

if (difficulty < 0 || difficulty > 100)

{

printf("Invalid Difficulty\n");

return(-1);

}

// setup screen

initscr();

refresh();

// Generate and draw the maze, with initial avatar

generate\_maze(difficulty);

draw\_maze();

// AVATAR Placed and Waiting

draw\_character(current\_x,current\_y, AVATAR);

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

// Loads the first buffer for x, y, and z arrays with data from the wiimote

for(count = 0; count<NUM\_SAMPLES; count++){

read\_acc(&a\_x, &a\_y, &a\_z, &t, &b\_1, &b\_2, &b\_b, &b\_a);

x[count] = a\_x;

}

//Event loop

do

{

// Read data, update average

read\_acc(&a\_x, &a\_y, &a\_z, &t, &b\_1, &b\_2, &b\_b, &b\_a);

// updates the buffers & computes average

updatebuffer(x,NUM\_SAMPLES,a\_x);

avg\_x = avg(x, NUM\_SAMPLES);

// Begin Move?

if((time\_counter % TIME\_DELAY) == 0)

stuck = move\_avatar(avg\_x, &current\_x, &current\_y);

usleep(LOOP\_IDLE); // leave this here to smooth out game play

time\_counter++;

} while((current\_y < (SCREEN\_HEIGHT)) && !b\_a && !stuck); // Did I win? Am I stuck?

// Print the win message

endwin();

// If we are stuck we will let the user know.

if(stuck)

printf("YOU GOT STUCK!\n");

// If we won, we will congratulate the user.

if(current\_y >= SCREEN\_HEIGHT)

printf("!CONGRATULATIONS!\n!!!! YOU WIN !!!!\n");

// If the user exits out by pressing the A button this is displayed.

else

printf("YOU DIDN'T FINISH!!\nYOU LOST THE GAME!!!");

return 0;

}

// PRE: 0 < x < SCREEN\_WIDTH, 0 < y < SCREEN\_HEIGHT, 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();

}

void draw\_maze(void){

int i;

int j;

for(i=0; i<SCREEN\_HEIGHT; i++){

for(j=0; j<SCREEN\_WIDTH; j++){

draw\_character(j,i,MAZE[j][i]);

}

}

}

void generate\_maze(int difficulty){

int i;

int j;

int random;

// seeds the random function with time from the computer

srand(time(0));

// Initializes the Maze array with spaces

for(i=0; i<SCREEN\_HEIGHT; i++)

{

for(j=0; j<SCREEN\_WIDTH; j++)

{

MAZE[j][i] = EMPTY\_SPACE;

}

}

for(i=0; i<SCREEN\_HEIGHT-1; i++)

{

for(j=0; j<SCREEN\_WIDTH; j++)

{

random = rand() % 100;

// Uses difficulty entered by user

if(random<difficulty)

MAZE[j][i] = WALL;

else

MAZE[j][i] = EMPTY\_SPACE;

}

}

// makes sure the AVATAR's beginning position is clear

MAZE[50][0] = EMPTY\_SPACE;

}

int read\_acc(float\* a\_x, float\* a\_y, float\* a\_z, int\* t, int\* Button\_one,

int \* Button\_two, int\* Button\_B, int\* Button\_A)

{

int junk; // a junk variable for unneeded buttons

int HOME; // variable for home button

scanf("%d,%f,%f,%f,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d",

t,a\_x,a\_y,a\_z,Button\_A,Button\_B,&HOME,&junk,&junk,Button\_one,

Button\_two,&junk,&junk,&junk,&junk);

return 0;

}

double updatebuffer(double buffer[], int length, double new\_item)

{

int i;

for(i=0; i<length; i++)

buffer[i] = buffer[i+1];

buffer[length] = new\_item;

}

double avg(double buffer[], int num\_items)

{

int i;

double sum = 0;

double avg\_value;

for(i=0; i<num\_items; i++)

{

sum = sum + buffer[i];

}

avg\_value = sum/num\_items;

return avg\_value;

}

int move\_avatar(double avg\_x, int\* current\_x, int\* current\_y)

{

int original\_x = \*current\_x;

int original\_y = \*current\_y;

// Erase Previous AVATAR

draw\_character(\*current\_x, \*current\_y, EMPTY\_SPACE);

// Left Move?

if(avg\_x < -TOLERANCE)

// Update X

\*current\_x = \*current\_x - 1;

// Right Move?

else if(avg\_x > TOLERANCE)

// Update X

\*current\_x = \*current\_x +1;

// Update Y

else \*current\_y = \*current\_y + 1;

// Can I Fall? (Check to see if the space we are moving to contains a WALL)

if( MAZE[\*current\_x][\*current\_y] == WALL ||

\*current\_x < 0 || \*current\_x > SCREEN\_WIDTH-1)

{

\*current\_x = original\_x;

\*current\_y = original\_y;

}

// Place new Avatar

draw\_character(\*current\_x, \*current\_y, AVATAR);

// If the AVATAR is stuck this allows the program to end

if (MAZE[\*current\_x-1][\*current\_y] == WALL &&

MAZE[\*current\_x+1][\*current\_y] == WALL &&

MAZE[\*current\_x][\*current\_y+1] == WALL)

{

return 1;

}

// Returns 0 if we are able to successfully move

return 0;

}