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);
```

```c
// 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,%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;
}
```