Voice Control Snake Kernel Module

Yujie Zheng (yjz122@bu.edu)

Yimo Zhao (zhaoyimo@bu.edu)

https://github.com/Yujie2023/EC535_Project.git

Abstract

This project aims to implement voice-controlled gameplay for the classic game Snake using a voice control module and a beaglebone board. With the whole system, players can use voice commands to direct the movement of the Snake character without relying on traditional keyboard or controller inputs. The project begins by capturing the player's voice commands through a voice control module, which are then transmitted to the BeagleBone development board for processing. A program running on the board is responsible for interpreting the voice commands and translating them into movement actions for the Snake character within the game. This enables players to control the Snake's movements using simple verbal commands, enhancing the game's entertainment value and interactivity. Players can also opt to use traditional buttons alongside or instead of voice commands for game control.

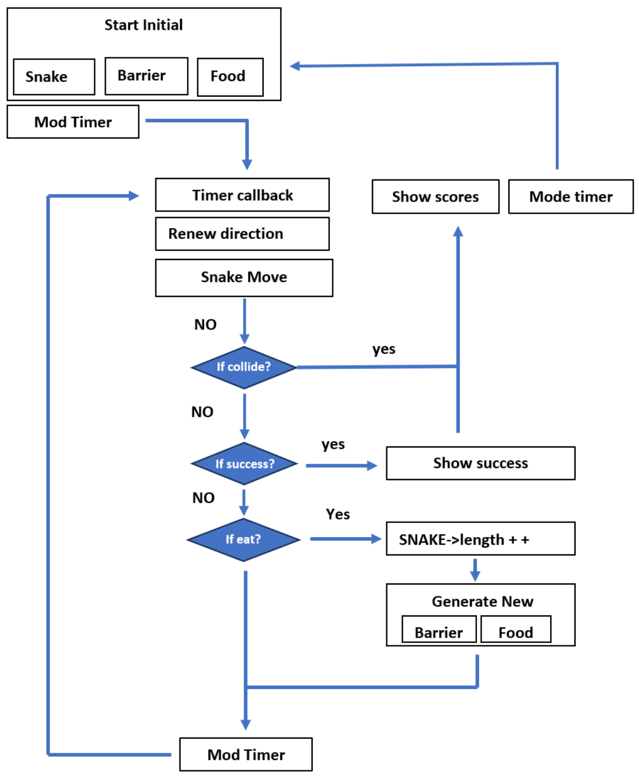
1. Introduction

The gameplay of this game is very similar to the classic Snake, but it allows players to use either the voice control module or regular buttons. Upon booting up, the game first displays a blue screen and then initiates gameplay after a 15-second waiting period. In the game, there are three types of colored squares: blue representing the snake, red representing food, and white representing obstacles. Initially, the snake consists of 2 segments, and players can control the snake's direction by using voice commands ("UP," "RIGHT," "DOWN," "LEFT") or pressing buttons. Through voice commands, the snake can turn in the corresponding directions after the voice control module outputs signals to the BeagleBone. When the snake eats food, its length increases, and its movement speed accelerates.

The game ends if the snake's head collides with an obstacle or any part of its own body. Upon failure, the screen displays the current score, which is proportional to the square of the snake's length. After displaying the score for 7 seconds, the game reinitializes the snake, food, and obstacles, and restarts. Victory is achieved when the snake's length reaches the predefined value in the code. Upon winning, the screen first displays "SUCCESS" for three seconds, then shows the score for four seconds, and finally restarts the game. The maximum length of the snake is 100 segments, and the highest score achievable is 10,000 points.

The voice recognition module captures audio signals from the external environment through its built-in microphone or external audio input interface. These audio signals are then converted into digital signals for further processing. The module contains pre-trained voice recognition models for offline processing. During the model matching stage, information extracted from the audio features is compared and matched with these models to determine the most likely voice recognition results. Finally, the voice control module outputs corresponding GPIO signals to beaglebone pins to achieve voice control functionality.

2. Design flow



Frame Buffer Graphics:

The game directly manipulates the framebuffer to render game graphics such as the snake, food, and barriers. This is achieved using structures like fb_info and fb_fillrect to specify drawing operations.

GPIO and Interrupt Handling:

The game uses General Purpose Input/Output (GPIO) pins to read input from the output of the voice control system or physical buttons connected to the system. Each GPIO input is associated with a direction change for the snake. Interrupt handlers are set up for each button to process button presses asynchronously.

Timers:

The module uses kernel timers to manage game timing. The snake's movement is updated based on the timer, providing a consistent game speed and response to user input.

Memory Management:

Kernel memory allocation functions (kmalloc, kfree) are used extensively to manage memory for game objects dynamically.

Input Event Handling:

An input handler is registered to capture and process keyboard or other input device events. This part is crucial for extending or customizing input handling beyond the predefined GPIO input.

Character Device Interface:

A character device is registered, allowing user-space applications to interact with the module through device files. This could be used for starting/stopping the screensaver, adjusting settings, or querying game state.

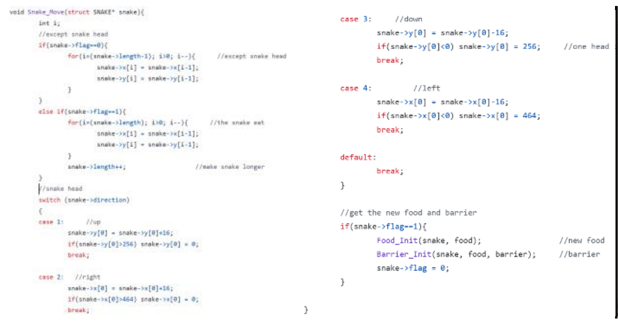
3. Discussion of failure

The original plan was to make a voice-controlled ambient light, but since the image in the lab5 file did not include the PWM driver file, the LCD screen in the experimental materials was used to change the project back to a voice-controlled snake game. Other than that, no unplanned issues were encountered.

4. technical report of success

4.1 Snake movement

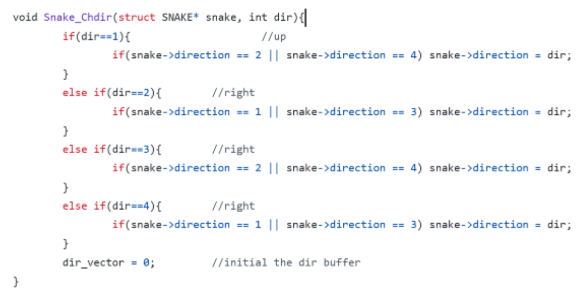
The Snake_Move function shifts each snake segment forward to the next segment's position, advancing the snake. If the snake has eaten food (snake->flag == 1), it grows by adding a segment. The snake's head updates based on its direction, with boundary wrapping handled seamlessly. After eating, the function initiates new food and barriers placement, resetting the flag for the next move. This ensures continuous movement and incremental difficulty in the game.ovement.



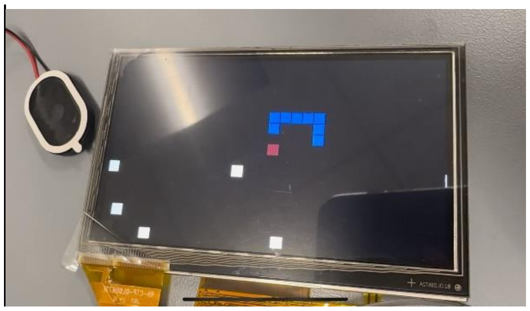
```
void Snake_Move(struct SNAKE* snake){
        int i;
        //except snake head
        if(snake->flag==0){
                for(i=(snake->length-1); i>0; i--){      //except snake head
                        snake->x[i] = snake->x[i-1];
                        snake->y[i] = snake->y[i-1];
                }
        }
        else if(snake->flag==1){
                for(i=(snake->length); i>0; i--){      //the snake eat
                        snake->x[i] = snake->x[i-1];
                        snake->y[i] = snake->y[i-1];
                }
                snake->length++;                       //make snake longer
        }
        //snake head
        switch (snake->direction)
        {
        case 1:      //up
                snake->y[0] = snake->y[0]+16;
                if(snake->y[0]>256) snake->y[0] = 0;
                break;

        case 2:   //right
                snake->x[0] = snake->x[0]+16;
                if(snake->x[0]>464) snake->x[0] = 0;
                break;

        case 3:      //down
                snake->y[0] = snake->y[0]-16;
                if(snake->y[0]<0) snake->y[0] = 256;      //one head

        case 4:         //left
                snake->x[0] = snake->x[0]-16;
                if(snake->x[0]<0) snake->x[0] = 464;
                break;

        default:
                break;
        }

        //get the new food and barrier
        if(snake->flag==1){
                Food_Init(snake, food);                //new food
                Barrier_Init(snake, food, barrier);    //barrier
                snake->flag = 0;
        }
}
```

4.2 Change direction function

The Snake_Chdir function updates the direction of the snake based on user input while ensuring that the snake cannot reverse into itself. The function takes two parameters: a pointer to the snake structure and the intended direction (dir). It only allows changes perpendicular to its current movement. This prevents the snake from moving directly back into its own body, which would typically result in a game over condition. After setting the new direction, the function resets the direction buffer (dir_vector) to zero, preparing it for the next input.

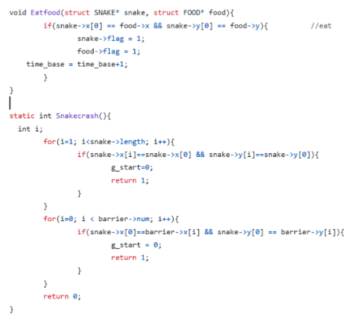


```
void Snake_Chdir(struct SNAKE* snake, int dir){
        if(dir==1){                     //up
                if(snake->direction == 2 || snake->direction == 4) snake->direction = dir;
        }
        else if(dir==2){        //right
                if(snake->direction == 1 || snake->direction == 3) snake->direction = dir;
        }
        else if(dir==3){        //right
                if(snake->direction == 2 || snake->direction == 4) snake->direction = dir;
        }
        else if(dir==4){        //right
                if(snake->direction == 1 || snake->direction == 3) snake->direction = dir;
        }
        dir_vector = 0;         //initial the dir buffer
}
```



4.3 Hit barrier or Eat food

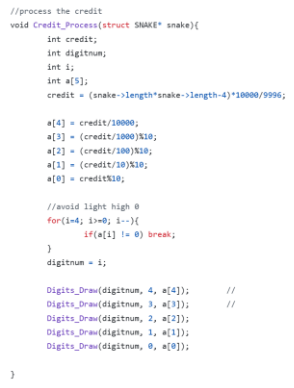
The Eatfood function checks if the snake's head has collided with the food item. If the coordinates match, indicating consumption, it sets the snake's flag to 1 to trigger growth in the next move and marks the food's flag as 1 to signal it's been eaten. The Snakecrash function checks for collisions by iterating through its segments from the second one onwards. If the head's coordinates match any other segment's, it sets g_start to 0 and returns 1 to indicate a crash. It then checks if the snake has collided with any barriers on the field.



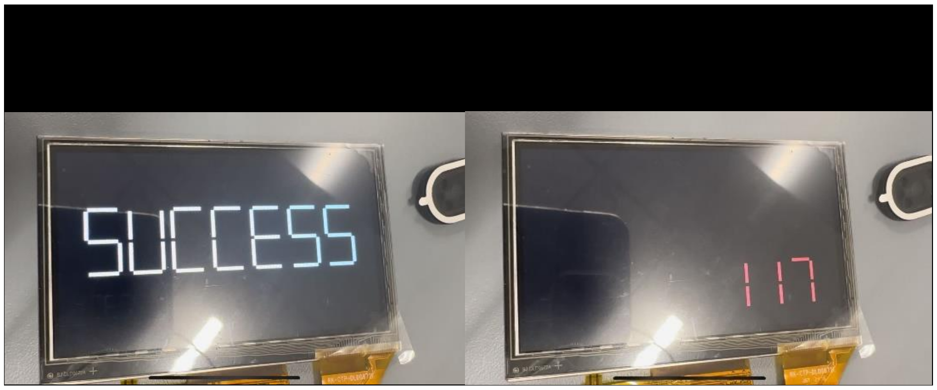
```
void Eatfood(struct SNAKE* snake, struct FOOD* food){
        if(snake->x[0] == food->x && snake->y[0] == food->y){          //eat
                snake->flag = 1;
                food->flag = 1;
    time_base = time_base+1;
        }
}

static int Snakecrash(){
  int i;
        for(i=1; i<snake->length; i++){
                if(snake->x[i]==snake->x[0] && snake->y[i]==snake->y[0]){
                        g_start=0;
                        return 1;
                }
        }
        for(i=0; i < barrier->num; i++){
                if(snake->x[0]==barrier->x[i] && snake->y[0] == barrier->y[i]){
                        g_start = 0;
                        return 1;
                }
        }
        return 0;
}
```

4.4 Show credit

The Credit_Process function calculates the game score based on the snake's length, breaks it into individual digits, and omits leading zeros. It then calls Digits_Draw for each digit to display them on the screen. The Digits_Draw function allocates graphical elements for each segment of a digit and sets their color and position based on the digit's value.



```c
//process the credit
void Credit_Process(struct SNAKE* snake){
        int credit;
        int digitnum;
        int i;
        int a[5];
        credit = (snake->length*snake->length-4)*10000/9996;

        a[4] = credit/10000;
        a[3] = (credit/1000)%10;
        a[2] = (credit/100)%10;
        a[1] = (credit/10)%10;
        a[0] = credit%10;

        //avoid light high 0
        for(i=4; i>=0; i--){
                if(a[i] != 0) break;
        }
        digitnum = i;

        Digits_Draw(digitnum, 4, a[4]);        //
        Digits_Draw(digitnum, 3, a[3]);        //
        Digits_Draw(digitnum, 2, a[2]);
        Digits_Draw(digitnum, 1, a[1]);
        Digits_Draw(digitnum, 0, a[0]);

}
```



4.5 Screensaver_Init

Device Registration: Attempts to register a character device for the module. If it fails to obtain a major number, it logs an error and exits the initialization.

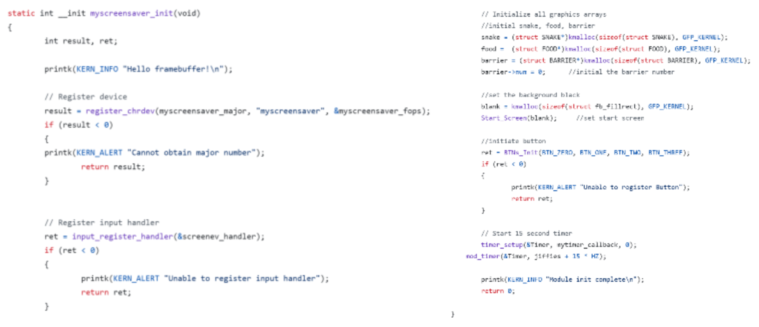
Input Handler Registration: Registers an input handler for capturing keyboard or other input events. If this registration fails, it logs an error and exits.

Memory Allocation: Allocates memory for the game's primary structures: snake, food, and barriers, initializing them in the process.

Graphics Initialization: Allocates memory for a framebuffer rectangle to set the background screen, and then initializes the start screen.

GPIO Initialization: Attempts to initialize GPIO buttons used for controlling the game. If this fails, it logs an error and returns.

Timer Setup: Sets up a kernel timer to trigger game updates, initializing it to fire 15 seconds after module start.



```c
static int __init myscreensaver_init(void)
{
        int result, ret;

        printk(KERN_INFO "Hello framebuffer!\n");

        // Register device
        result = register_chrdev(myscreensaver_major, "myscreensaver", &myscreensaver_fops);
        if (result < 0)
        {
        printk(KERN_ALERT "Cannot obtain major number");
                return result;
        }


        // Register input handler
        ret = input_register_handler(&screenev_handler);
        if (ret < 0)
        {
                printk(KERN_ALERT "Unable to register input handler");
                return ret;
        }
```

```c
        // Initialize all graphics arrays
        //initial snake, food, barrier
        snake = (struct SNAKE*)kmalloc(sizeof(struct SNAKE), GFP_KERNEL);
        food =  (struct FOOD*)kmalloc(sizeof(struct FOOD), GFP_KERNEL);
        barrier = (struct BARRIER*)kmalloc(sizeof(struct BARRIER), GFP_KERNEL);
        barrier->num = 0;      //initial the barrier number

        //set the background black
        blank = kmalloc(sizeof(struct fb_fillrect), GFP_KERNEL);
        Start_Screen(blank);     //set start screen

        //initiate button
        ret = BTNs_Init(BTN_ZERO, BTN_ONE, BTN_TWO, BTN_THREE);
        if (ret < 0)
        {
                printk(KERN_ALERT "Unable to register Button");
                return ret;
        }

        // Start 15 second timer
        timer_setup(&Timer, mytimer_callback, 0);
    mod_timer(&Timer, jiffies + 15 * HZ);

        printk(KERN_INFO "Module init complete\n");
        return 0;

}
```

5. Refenrence

https://github.com/JimmySong95/EC535_project.git

https://github.com/chagge/vad.git