Project write-up

System Explanation

Project Flappy Bird is a recreation of the infamous flappy bird mobile game on a 16x32 LED matrix and powered by a Beaglebone. Unlike the mobile game, our game uses voice to control the bird by the use of an electret microphone.

Everything we print out to the screen is basically by modifying a 2D array of 16x32. The LED display driver has 2 buffers, 1 is used by the display thread to show on the LED display while the other is used to be accessed by other threads to copy the array (new screen) to the buffer. This is to avoid bottleneck since copying 512 elements may take a long time.

The screen can print any strings with letters a-z (each letter on a 5x5 grid). We provide an interface to print out the string in any specified position. If we don't have enough screen space for the next character, it will be automatically omitted.

Threads

The program runs with 1 main thread that shows game screens and checks for the right type of inputs every frame, and 3 background threads including a UDP listener thread, a sound sampling and noise filtering thread, and a thread to display number on the 14-seg display.

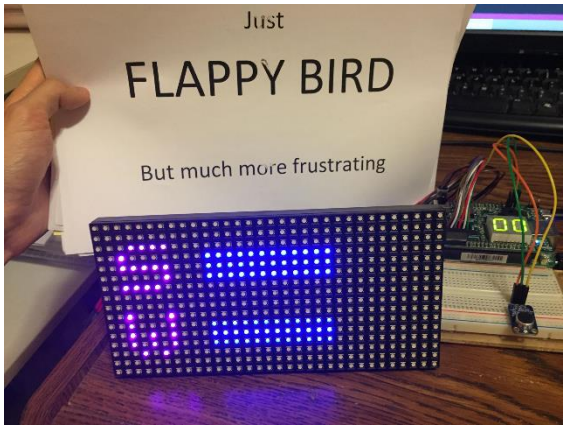
Background threads are initialized by the main thread (with threading stuff is encapsulated in their own module). During the initialization for the UDP thread, main thread passes a flag so that UDP thread can use it to trigger game shutdown. It is basically a callback "variable" so that the main thread knows and calls other threads to clean up and then closes the game.

Game screens

Welcome screen

Simply just the original flappy bird logo. Player can proceed to the next step by moving the joystick to any direction.

Set up screen



There are 2 factors that will affect the gameplay: speed and weight.

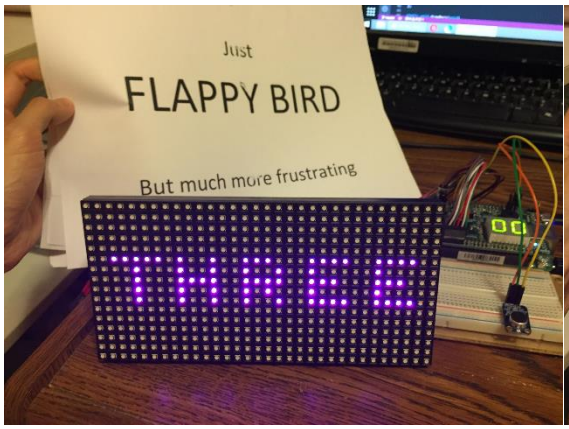
- Speed decides how fast the bird will fly (range 20-100, higher means faster)
- Weight decides how fast the bird will fall due to gravity (range 20-100, higher means heavier)

Here the player can use the joystick to go through all preset bird stats that we provide by **holding** up or down. The bars show the stats visually (longer means bigger number). At the current moment we have 5 presets (40-40, 50-50, 60-60, 70-70 and 80-80 for Speed and Weight)

Apart from that, the player can use the potentiometer to change the color of the bird, which reflects on the color of the bar at this stage. We have 7 colors: red, green, blue, light blue, white, purple and yellow.

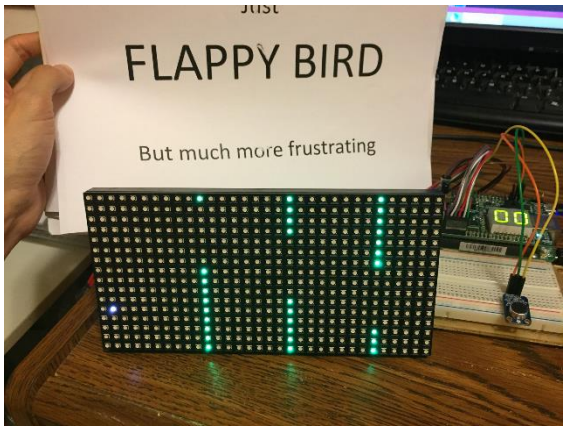
To confirm the settings and proceed, the player has to **hold** left or right for 0.2s. Every action with joystick requires a hold to avoid input mistakes from the player.

Pre-Gameplay screen



It’s basically a count down from 3 to 1 on the screen (every screen is 3 seconds). Texts can be in any color but we choose purple since it looks the best for the game (not too bright or too dim).

Main gameplay screen



The gameplay is exactly the same as the original version (in terms of mechanics – i.e. making it frustrating and rewarding) except for the input. The player has to use voice to control the bird jump. The louder the noise the player makes, the higher the jump. There is a cool down between 2 jumps (to avoid spamming). When the bird reaches the top, the remaining power of the jump that is not processed yet is cancelled to avoid the bird getting stuck on the top for too long. The game is implemented with a loop that simulates frames. We make it run every 5ms (200 frames a second). Every jump, gravity pull, or move (right) is performed using a frame counter. When the counter reaches a certain number, we will perform the action. The jump is performed every frame but gravity and move vary based on the current set up.

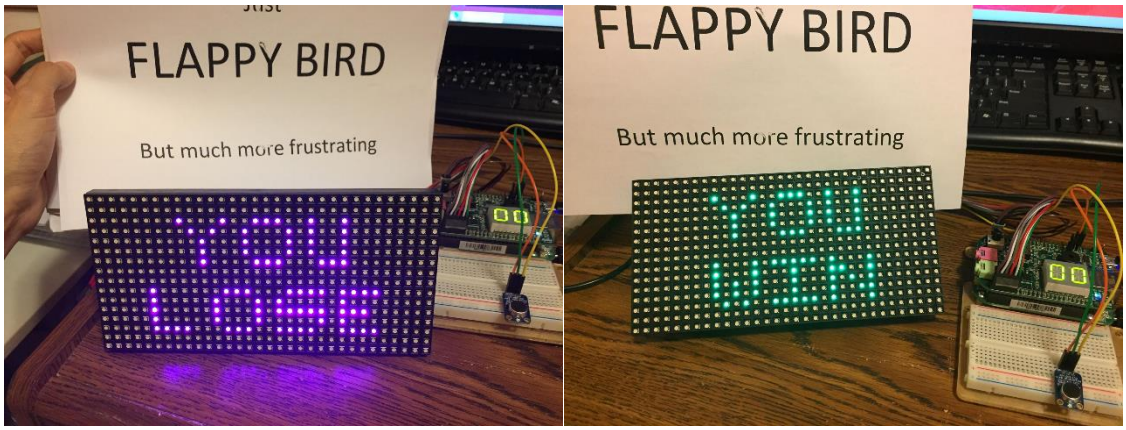
For the input, the background environment noise is filtered out using an algorithm similar to moving average filtering. We take the average of the 400 most recent samples, then calculate if the newest sample passes the threshold or not. If it does then we treat it as the player's sound, otherwise we will replace the least recent sample with that new number, recalculate the average. To make it efficient, we use a circular array to hold 400 samples. We also cache the sum and average (by simply deducting the least recent sample from the sum, adding the newest sample to the it, then dividing by 400 to get the new average) so we don't have to loop through the whole array every time.

Score is displayed on the 14-seg display on Zen Cape. Every 10 scores, the difficulty is increased (speed += 10), and all the pipes will change color. Of course, we make sure that pipe color will never be the same as the bird color, even from the start of the gameplay screen. The score is kept until the next play starts to let user know their score.

By default, the max score to win the game is 100. But we also provide a cheat system using UDP (port 12345, command maxScore x with x is the new max score) to make it easier to win the game. This will be reset every time we start the main game screen. Also, we only allow to set it higher than the current score to avoid bugs.

We can also use UDP to shut down the game by sending "stop". The game will be shut down after the end screen in order not to interfere the current gameplay. We let user finish their game before we shut it down. This feature is ensured to clean up everything to avoid daemon threads hanging around.

End game screen



The game prints either “You Lose” or “You Win” depending on the situation for 3 seconds. After that, the game goes back to welcome screen.

Things that did not work well

Our project is heavily reliant on sound which makes it less reliable in noisy environments like the Mezzanine. Because of that, the microphone often picks up noises from people talking next to the player, leading to faulty behavior in the game. But if we make it less sensitive enough to filter out those noises, the user will have to scream quite loud (or in our demo, the simulated sound has to be louder). However, since our phone can only reach to a certain level of loudness, we decide to accept some of the environment sounds to mess up with the game. As long as people who are very close to the player (radius of less than 0.2m) stay silent, the game will run well.

LED matrix display flickers due to the fact that we have quite a number of threads and it is the limitation of the hardware itself. This is explained in the guide that we follow to make this work. An easy fix would be moving everything into 1-2 threads but that would make the code messy and hard to debug so we decided to have it flickers but playable and keep the code clean.

Feature Table

Description	Host/Target	Completeness	Code	Notes	Contributor
LED matrix display	T	4	C	Modified from the driver given in the guide (80% sample code, 20% us) Works but flickers due to having too many threads.	Pham
Joystick	T	5	C	Modified from assignment 1	Pham
Buzzer	T	2	C	Created based on PWM guide (100% us but using system commands provided by the guide)	Nirag

				Works when tested independently, not with the project	
Button	T	1	C	Works on terminal but signal seems to be unpredictable when using C to access the file	Nirag
Potentiometer	T	5	C	Modified from A2D guide	Pham
Microphone with noise filtering	T	4	C	Code is mostly modified from the A2D guide Filtering algorithm is 100% made by us Works will in quiet environments. Less reliable in noisy areas	Nirag, Pham
14 segment display	T	5	C	Modified from the code of assignment 2, different pattern to avoid digits being upside down	Pham
UDP listener	T	5	C	Modified from the assignment 2-3 UDP code	Pham
Printing strings to screen	T	5	C	Letters are encoded based on various sources online and tweaked to improve kerning	Nirag, Pham
Game screens	T	5	C	Creates the 5 game screens (100% our code)	Pham
Game mechanics	T	5	C	Recreates the original game with a change in input readings (100% our code)	Pham