ECE3140/CS3420 Final Project: “Snake”
Haley Lee (hal64), Jonathan Gao (jg992)

Introduction

Our final project is an adaptation of the classic Snake arcade game, implemented using a 32x32 RGB LED matrix and an accelerometer. In this game, the snake moves around the grid with the objective of growing as long as it can without dying. The snake grows by obtaining “food,” which are randomly generated dots on the grid. The snake dies if it runs into any of its own body segments or into the boundaries of the board.

We were successfully able to recreate this game on the 32x32 LED matrix, using the accelerometer on the FRDM-K64F board to control the snake’s direction.

Here is our video: https://youtu.be/8qrlVaZiNZA

System Diagram

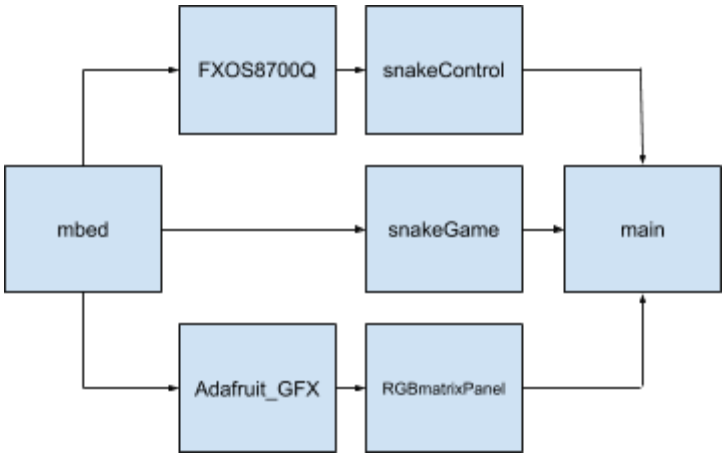

Figure 1: Software system block diagram

In order to implement this game, we made adaptations to three external libraries:

- Mbed: all of the files include mbed, which is the library we used to simplify pin connections between the RGB LED matrix and the FRDM-K64F microcontroller.
- FXOS8700Q: the library used to read data from the 3-axis linear accelerometer built-in to the FRDM-K64F.
- RGBmatrixPanel: the library used in order to connect, set up, initialize, and draw to the RGB LED matrix.
- Adafruit_GFX: though this library allows us to draw shapes, its primary use was the font for the words we displayed on the matrix.

We also wrote three additional files:

- SnakeControl: this file polls the accelerometer for data and determines the direction which the user intends to move the snake.

- SnakeGame: this file is where all of the game design is implemented, from generating a snake, moving it, generating food, checking for collisions, and updating the snake length.
- Main: this file is where the pin connections are set up, panel objects are created, and the data obtained from snakeControl is linked to snakeGame in order to successfully implement the game. This file includes a call to attach_us which triggers the interrupt handler every 200 us, during which the data from the accelerometer is obtained and the snake is allowed to update.

The major architectural choice we made was to program in C++. This allowed us to make object oriented design choices, such as creating separate classes for the game, controlling the game, etc., and make our code more abstract.

Hardware Description

Bill of Materials:

Item	Source	Product ID	Quantity	Price
32x32 RGB LED Matrix Panel - 6mm pitch	Adafruit	1484	1	$39.95
Proto Shield for Arduino			1	*Already had
8 pin header pins			2	*Already had
FRDM-K64F Accelerometer			1	*Already had

The shield was obtained in order to avoid soldering wires directly to the board. Though it is an Arduino shield, the pins match up to the FRDM-K64F. We then soldered the two header pins to the shield so that we could connect the input jumper cable from the board. On the shield, we soldered wires from the header pins to the correct pins, following the pin layout on the following diagram:

ECE3140/CS3420 Final Project: “Snake”
Haley Lee (hal64), Jonathan Gao (jg992)

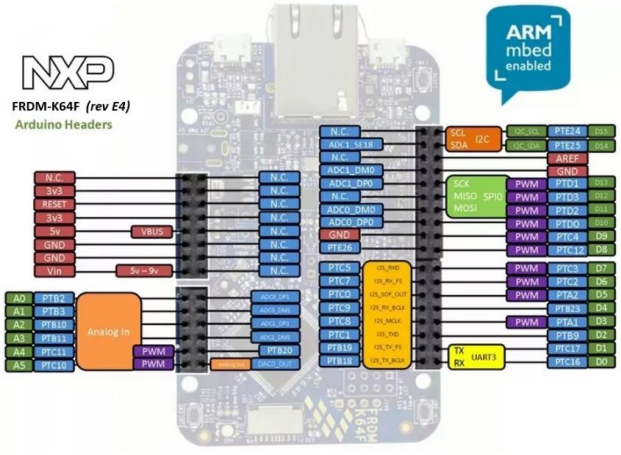

Figure 2: NXP header pinout for FRDM-K64F

Ultimately, our pins were connected from the RGB LED matrix panel to the FRDM-K64F as such:

- LAT → D10
- OE → D9
- CLK → D8
- B2 → D7
- G2 → D6
- R2 → D5
- B1 → D4
- G1 → D3
- R1 → D2
- A → A0
- B → A1
- C → A2
- D → A3

Detailed Software Description

The software was primarily written in C++, an object oriented language, and thus we tried to stick to object oriented programming conventions. When designing the program, we tried abstract and encapsulate as much of the program as we could. We started by working on the RGB matrix LED panel.

RGBmatrixPanel:

The RGB matrix panel was sourced from Adafruit, who along with the product also provided some documentation and a tutorial for how to interface with the board. The board was driven by LED driver chips that drove 16 LEDs at a time per chip. Each chip drove two 16 LED rows, of which are interlaced together to mitigate tearing during refreshes. Our RGBmatrixPanel

class was adapted from an RGBmatrixPanel library created in 2014 from the Mbed online libraries. Because the code was old and the mbed libraries were not compatible with this library, we had to rewrite the library to work with the most recent mbed-os library.

The RGBmatrixPanel class did not work not due to logical errors in the code, but rather due to an update to the mbed-os library that changed the functionality of DigitalOut and the Ticker classes. The RGBmatrixPanel class used BusOut, which uses DigitalOut, to drive the GPIO pins. We discovered this error by looking through the mbed-os repository. The updated mbed-os libraries made DigitalOut’s write() method atomic, where it required a mutex lock to call the function. However, because RGBmatrixPanel used the Ticker class, a class written to link a function call to a periodic interrupt handler, we ran into an error because Ticker does not allow for calls to locks or waits in its attached function. To keep a high refresh rate display, we decided to refactor the RGBmatrixPanel library to work without relying on DigitalOut and BusOut.

By looking through the mbed-os repository, specifically in the drivers folder in fsl_gpio.h, we were able to write another function setPin() that emulated the DigitalOut write() function but without a mutex call. The setPin() function for the K64F writes a 1 to the LSB of the PCOR or PSOR register for the respective GPIO pin, addressed by the pin number added to the GPIO base address. The updateDisplay() function of RGBmatrixPanel also had to be rewritten to accommodate for setPin(). For simplicity, we decided to stick with creating DigitalOut objects to exploit its constructor to set up the GPIO pins.

The RGBmatrixPanel class extends the Adafruit_GFX class, which has some prewritten functions for font rendering, shape drawing, and color manipulation from Adafruit. This is their generic class provided for all of their displays. RGBmatrixPanel implements the function drawPixel, which Adafruit_GFX uses in its methods.

The RGBmatrixPanel class interfaces with the board by first designating the row by a 4-bit bus, setting the latch, then clocking in the color information 6 bits at a time for each pixel.

snakeGame:

This class handled all the functions and logic for the game itself. This included a linked-list implementation for the snake, the food, and functions to check collisions, update the snake on each tick, and to interface with other classes to render the snake.

The linked-list implementation of the snake used a struct SEG which held the fields x, y, next and prev. The x and y fields were for its location on the board, then the next and prev were pointers to the next and previous SEG.

The game functioned on a tick based system, where each step of the game was initiated by a call to updateSnake(). This function handled all the movement and make calls to checkCollision

to detect changes in game state. The updateSnake() function was passed a direction as a parameter, defining the next direction the snake moves in. This movement was done by prepending a new SEG to the from of the snake, only removing the tail SEG if a food condition was not met.

The food was generated using a random number generator seeded with the current time.

To speed up our program, we used a head and tail pointer that was kept updated.

snakeControl:

The snakeControl class handled the input from the accelerometer. The accelerometer read values passed as a struct, thus we wrote a pollAccel() function to update a direction field that gets returned by getDirection(). The raw accelerometer values were filtered and parsed to accurately determine the direction the board was tilted in.

Main:

The main class merged all the above classes together. In this class we created a RGBmatrixPanel object with all the pins passed into the constructor, a snakeGame object to handle the game mechanics, a snakeControl object to handle the accelerometer measurements, and the Ticker objects for refreshing the display and refreshing the snake.

To draw the snake on the board, we passed the pointer to the linked-list for the snake and drew the snake pixel by pixel on each draw() function call. The same was done for the food. The main class then also handles all the intro rendering and the game over screen. We used a while loop in main to handle the snake mechanics due to polling of the accelerometer also requiring locks.

Code:

Most of the code was written by us. Excluding the libraries, we wrote all of the above classes besides the code that wasn’t modified in RGBmatrixPanel.

<u>Testing</u>

The classes were tested individually using the serial monitor or the matrix panel before being integrated into the rest of the project. After completing the game, we tested the game mechanics iteratively, fixing collision mechanics, tuning control sensitivity and refresh rate, and implementing a score display for the game over screen.

RGBmatrixPanel:

The matrix panel was first tested using the serial monitor, then by drawing every pixel one by one.

snakeGame:

ECE3140/CS3420 Final Project: “Snake”
Haley Lee (hal64), Jonathan Gao (jg992)

The snakeGame class was tested using the matrix panel and by playing the game. We tested each game state, each manner of collision, and multiple digit scores. The state where the board fills up entirely was not tested because we are not good enough to reach it.

snakeControl:

The controller was tested by tilting the board and reading output to the serial monitor.

Results and Challenges

We achieved what we proposed and more. The snake game works well and the accelerometer as an input device works much better than expected. The overall design was fairly similar to the one we proposed. The most difficult portion of this project was getting the matrix to work with the board. After struggling to figure out the way the matrix takes inputs, we decided to switch to using mbed for its libraries. However, when the RGBmatrixPanel library didn’t work, we ended up troubleshooting for a really long time. It wasn’t until we read through all of the mbed repository, figured out how the library works and the file hierarchy, and then working our way around the library by injecting our own code that we were able to get the board working. Although mbed did help a lot, it helped mostly by providing a skeleton in which we were able to build our functions from. Next time, we want to try to add more difficulty to the game, possible with variable speed and some color manipulation as well.

Work Distribution

The work was divided among the team. Jon spent a lot of time writing the game logic and the RGBmatrixPanel class, while Haley wrote the accelerometer class and the intro and game over logic. We worked together in person, sharing an mbed account to write code.

References, Software Reuse

References:

- Adafruit Wiring Tutorial

Libraries:

- Mbed
- RGBmatrixPanel
- FXOS8700Q
- Adafruit_GFX