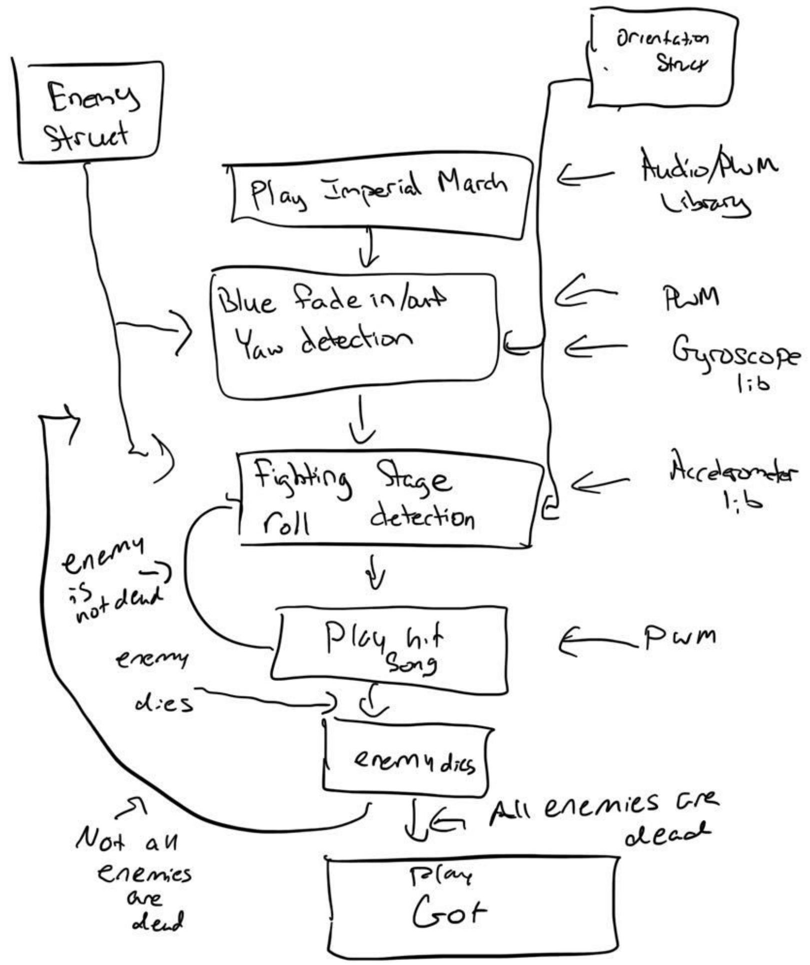
Sword Fight Game

By Laura Sizemore (ls797) and John Ly (jtl222)

1. Introduction

We made an interactive game using motion tracking. Using a sword prop that's connected to the FRDM-K64F board as one or our devices, we created a game where the player fights against hidden enemies by using a toy sword and LED's to find and defeat them. First, the players will have to find the hidden enemies. We will randomly generate the enemy locations each time the game is reset. To track these motions we will use the FRDM board together with a *FXOS8700CQ* 3D Accelerometer + 3D Magnetometer combo sensor combined with a *FXAS21002C* 3-Axis Digital Gyroscope for a total of 9 degrees of freedom. This will enable us to detect sword orientation through the yaw component and use that to find the enemy. In stage one, the sword will blink blue, indicating an enemy is around, but they haven't found it yet. To find the enemy, the player must move in a circle until the sword glows white, indicating he/she has found an enemy. When this happens, the player must swing the sword to "kill the enemy", who will be right in front of the player. We will use the acceleration plus the roll component of orientation to determine if the player successfully "hit." The sword flashes red upon each hit and plays a "hit sound." When the player kills the enemy, the sword will flash green, and the player can then continue looking for enemies. We also added audio components using a speaker and PWM to provide a better immersive experience for the user. When the player starts the game, Star Wars' "Imperial March" will start playing. When the player hits an enemy, a "hit sound" of our own creation will play. Finally, When the player finally kills all enemies, we play part of the Game of Thrones theme song and flash the LED's colorfully.

2. System diagram

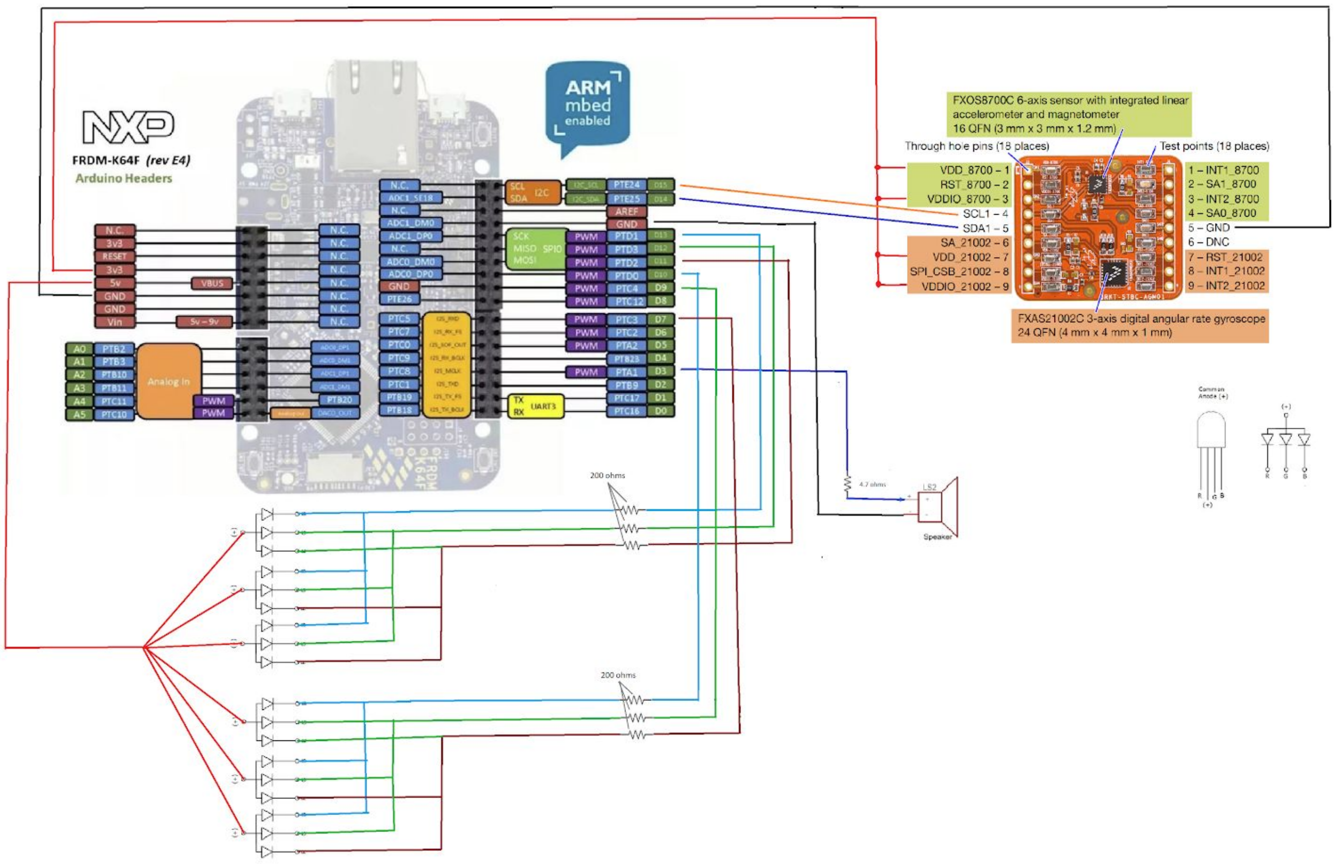


Our system consists of a toy sword, a FRDM K64F board, a BRKT-STBC-AGM01 9DOF sensor, a small speaker, 6 LED's, and a battery bank. The toy sword contains a hilt, which will hold the board, and a clear plastic piece containing 6 RGB LED's. We will connect the FRDM board to the 6 LED's, a small speaker, and the 9DOF sensor. We will be using the I^{2}C protocol to send and receive data from the board to the motion sensor. To power the board, we will connect it to a battery bank with a very long USB cable. The battery bank power the board through the USB cable, which will then power the sensor, speaker, and LED’s. The sensor will send gyroscope and accelerometer data to the board. The board will then use that to calculate the orientation. If the player is in stage one, or the finding enemy stage, the board will determine if the player has found an enemy or not. If the player is in stage two, the board will use that information to determine if the player has killed the enemy or not.

We chose to use polling over interrupts for our data collection. Since we wanted to have our data to come in after each calculation for the prior data value was complete, interrupting every time the data came in would be overkill. In addition, we considered using system calls or concurrency to go from stage one to stage two to add complexity. Once again, this proved to be unnecessary for what we wanted, as we wanted the flow from stage one to stage two to be linear, and we never wanted stage two to happen before stage one. Due to this, we chose to not use interrupts/scheduling/real-time

3. Hardware description

Schematic



Bill of Materials:

Item	Part Number	Price (not including tax)
FRDM K64F	Part #: 935326293598	Provided
BRKT-STBC-AGM01 FXOS8700CQR1/FXAS21002CQR1 Specialized Sensor/Gyroscope Sensor Breakout Board	Part #: 935326729598	Free (originally $12.97 but was refunded. Purchased on Arrow.com)
Minecraft Light-Up Adventure Sword	ASIN: B07847SRCV	$14.99 (on clearance at Target, originally $30.95)
Wires	n/a	Free
Resistors (6) 200ohms (1) 4.7ohms	n/a	Free
Electrical Tape	n/a	Free (friend gave to us)
Total Cost		$14.99

4. Detailed software description

Data Structures
Orientation: This struct held our various orientation data calculations. It consisted of three floats: pitch, yaw, and roll. We chose to use a struct here as we needed to save various forms of this data, so it was easier to initialize multiple orientation structs as opposed to making multiple arrays.

Enemy: This struct holds two integers: the enemy's health, and the enemy's location (a number between 0-7), In addition, similarly to the process_t struct, we hold a pointer to the next enemy. The enemies are held in a linked list called mobs. In order to update mobs when an enemy is added/removed, we added an enqueue and dequeue function. Like lab 3, enqueue takes one parameter, an enemy pointer current, and adds it to the end of the queue. If the queue is empty, it sets mobs to be that pointer. Otherwise, it sets the endQueue pointer mobsTail's next to be current, and then updates endQueue. For dequeue, we either return null if the queue is empty, or we set mobs to be the current mob's next pointer, and return the original mobs. The enqueue and dequeue was heavily based off the code Laura's group wrote for lab 3.

Finding Stage Code
For both stages, we have a while loop that continues while there is a current enemy, or that dequeing doesn't return null. The implementation of the finding stage starts with getting the location of the current enemy's struct. From there, we collect one axis of gyroscope data, which we do with the FXAS21000 library. To convert the angular velocity from the gyroscope into the yaw component of orientation, we take the integral of that angular velocity. We did this by multiplying the new yaw value by the update rate, deltaT, and adding that to the old value. We found the deltaT through trial and error as it varied with our code's length and the board's Output Data Rate (ODR). While we used the CalPoly paper linked below to understand the math, we wrote the velocity-to-orientation conversion code ourselves. Since this outputted a yaw value with a range of k*[-180, 180], where k is the number of times you've spun in a full circle, we changed our range to be [0, 360] by taking the modulo of the current value by 360 and, if that value was negative, adding 360. Finally, as our enemy's location was a number between 0-7 to represent the eight circle slices around the user where an enemy could be, we divided our yaw value by 45, as 360/8 = 45. We checked if this final value was equal to the current enemy's location, then we moved into the fighting stage. Otherwise, we stayed in the finding stage.

Fighting Stage
In this stage, we used a while loop that kept this player in this stage until the enemy's health was depleted. Here, we used the accelerometer to keep track of the player's attacks. We used the FXOS8700Q library to collect and read the sensor's accelerometer values using the I2C protocol. To ensure there was as little noise as possible in our data, we made a low-pass filter that took the average of 15 collections of accelerometer data. After that, we calculated the orientation's roll component of the sword by taking the arctan of the x component of acceleration divided by the square root of the y component squared plus the z component squared. Once again, we got the math behind the formula from the CalPoly paper linked below, but we converted the math into C++ ourselves. To determine if the player has hit, we looked at the roll value plus the acceleration. If the roll was greater than -70 (so the sword has gone almost 90 degrees) and the acceleration was greater than negative 1G, or if the acceleration down was greater than -.7G and the combined y and z components of acceleration were greater than one. We found those values through playtesting, as these values generally connected with what a hit would be if an enemy existed in real life. If the enemy was hit, we decremented its health. If it died, then we flashed the green LED's three times, and dequeued the next enemy. If that dequeued enemy was null, we knew the game was done and went to the victory code. Otherwise, we went back to the finding stage. All of this logic we wrote ourselves.

LED's and Music
To make sure our game indicated what was going on, we added LED and music cues. For the LED's, we used the pwm library in MBED so we could make the LED's to be lit at any value between 0-1 instead of just 0 or 1, allowing us to create cool wave effects with the lights. This was primarily used in the finding stage, where we made the sword blue light fade in and out. We then used the LED's in the sword to indicate what stage the player was in. If the sword was

pulsing blue, the player was in stage one. If the sword was pure white, the player was in stage two. In stage two, we used red flashes to indicate whenever an enemy was hit, and green flashes to indicate when a player killed an enemy. To generate the blue pulses, we used an if statement to determine whether or not we were fading in (adding .01 to the LED color) or fading out (subtracting .01 from the LED color). We also use the pwm library to generate the music. To generate recognizable music, we found piano sheet music online, and used a Wikipedia "piano to frequency" list to convert the piano notes into frequencies that PWM could play. To start the game and set an ominous mood, we start the game by playing Star Wars' Imperial March. Every time a player hits an enemy, we further reinforce that they succeeded by playing the "hit" song. Finally, to indicate to the player that he/she won, we play part of the Game of Thrones theme song. We did not actually write the sheet music for Imperial March or Game of Thrones, but we did write it for the hit song. The majority of the "playNote" function and code for Imperial March we found from an online tutorial. We then took that original function, changed it for our needs, and then used it to generate the hitNote and Game of Thrones music.

To communicate with our 9DOF sensor, we used the I2C protocol. Setting up and using this protocol was done for us with the FXOS8700Q (for the accelerometer) and FXAS21000 (for the gyroscope) libraries, which provided functions and defined constants for us to easily set up I2C.

5. **Testing**

Our testing fell into three components. First, we used PuTTY to make sure our raw sensor data and our orientation calculations were reasonable. This involved many printf statements and our own logic. For example, when we were getting Magnetometer values over 2000, we knew our sensors were wrong. This form of testing proved especially useful for testing Yaw calculations, as when we first calculated and printed Yaw, we found that the values were all over the place, leading us to instead use the gyroscope for Yaw, which led us to much more consistent numbers.

Once these sensor values worked, we changed values in main.cpp, to test our game logic and orientation calculation. To do so, we removed our random generators for the number of enemies and enemy locations and made them constant. This allowed us to know how many/where our enemies were, so when playing the edited main's game, if we were in location 0, the location of the first enemy, and we were still in stage 1, then we knew we had an issue with orientation. In addition, if we only saw 2 enemies and had three, then we knew we had an issue with either our logic to switch from enemy 1 to enemy two, or our enqueue/dequeue was wrong. When we wanted to try out different enemy combinations, like what would happen if all enemies were in one location or if there was only 1 enemy, we would change that number. We only added the random generator back in when we were certain our game logic was working for a variety of situations. Finally, we tested the music + entire game through playtesting. With regards to the music, we would run the game, and decide if the songs sounded like what we wanted. We also repeatedly playtested our game. This made sure that our random generators were working as expected. Finally, we also did some user testing on our game. This tested that our game was usable to someone not familiar with the codebase.

While hardware testing for with the accelerometer and gyroscope was all accomplished on the K64F through PuTTY outputs, testing for the LEDs and speaker were initially started on an Arduino Uno. The Arduino helped us understand that we were using common anode RGB LEDs and that wiring LEDs in parallel with 200ohm resistors was a viable option while still giving out enough brightness. As for the speaker, premade Arduino tone libraries helped us understand that we could use PWM signals for our audio generation on the K64F.

6. Results and challenges

Generally, we succeeded in what we proposed. We successfully built a game that allowed players to find and kill enemies, and plays different sounds depending on the actions of the user. While in our proposal we intended to use quaternions, we instead used Euler angles, as they worked better with the sensors we had. That being said, we used orientation data to find the enemies, and acceleration values to determine if we hit. The biggest difference between our proposal and our implementation was that instead of using an SD card to play WAV music files, we used PWM to play singular tones. We also indicated to the user what was happening in the game through the use of LED's. The two most complicated parts of the design was wiring everything together and getting orientation values to be reasonable. With regards to the wiring, it was difficult to understand the necessary wiring of the BRKT-STBC-AGM01 without researching about the all of the necessary voltages for each sensor on the board in addition to understanding how to wire the SCL and SDA data lines between the breakout board and our K64F whilst making sure that we were using the proper I2C addresses. The other difficulty was powering the speaker through the right pin. At first we tried using the onboard DAC0_OUT pin but found that we couldn't get a loud enough signal without using an amplifier so we ended up switching over the PWM method with our music being stored in flash as opposed to an SD card.

With regards to orientation, getting the sensors to not bounce around too much and to get our orientation data, especially yaw, was very difficult. Because our sensor was not oriented like usual (our Y and Z were essentially swapped), we had to partially rederive the formulas from the linear algebra to make sure our formulas accounted for the differently oriented board. In addition, as Yaw usually requires the magnetometer, we found it very difficult to get the magnetometer to stay calibrated and not completely mess up Yaw. In the end, we found that using a complementary filter with just the gyroscope data for Yaw was more consistent then using the magnetometer and accelerometer + the gyroscope to calculate Yaw.

In the future, we would make sure that we orient the board correctly, as this would have mitigated a lot of problems. In addition, one of the big reasons we switched to MBED was that we wanted to have an SD card play music. However, after a couple days of trying to get this working, we realized that wavplayer and SDCard libraries needed an amplifier to play music from the FRDM-K64F. By the time we realized this, we did not have enough time to find the piece and rewire the board. Finding out this issue took us a lot of time that could have been used to add further complexity to the game.

7. Work distribution

Hardware wise, our project consisted of the breakout board, the FRDM-K64F board, and the 6 LED's connected together. Software wise, our code was split into three parts: the finding stage, the fighting stage, and the music. The main part of the finding stage consisted of calculating yaw to determine where the player is relative to the starting point and if the player was in the "quadrant" the enemy was in. The main part of the fighting stage involved using the accelerometer to calculate the roll component of orientation, and then using that to determine if the player has hit the enemy. The music consisted of various functions that played different songs, and the playNote function. To write our code, we used MBED's online compiler. We chose to use MBED over uVision because initially we wanted to use an SD Card to play music, and our current uVision license could not work with SD Cards. We shared an account in order to share code between each other, and we used Facebook Messenger to communicate with each other. With regards to work distribution, John was in charge of wiring everything, getting the I2C protocol set up, and the finding stage. Laura was in charge of creating all necessary data structures, the music, and the fighting stage. In order to make sure everyone had an active role, we used pair programming to complete the code. Whoever was in "charge" of a particular piece of code was the driver, while the other person was the observer. The one exception to this was the wiring. For this section John mainly completed it himself, with Laura only helping when more than two hands were needed to solder the wires.

8. References, software reuse

Data Sheets

- https://www.nxp.com/docs/en/data-sheet/FXAS21002.pdf
- https://www.nxp.com/docs/en/data-sheet/FXOS8700CQ.pdf

Libraries

- https://os.mbed.com/users/JimCarver/code/FXAS21000/
- https://os.mbed.com/components/FXOS8700Q/

Manuals/Other Material:

- https://musescore.com/user/158751/scores/2163051 (sheet music for GOT theme song)
- https://www.musicnotes.com/sheetmusic/mtd.asp?ppn=MN0017607 (sheet music for Imperial March)
- https://learn.sparkfun.com/tutorials/mbed-starter-kit-experiment-guide/experiment-9-pwm-sounds (tutorial on PWM sounds)
- http://en.wikipedia.org/wiki/Piano_key_frequencies
- https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?referer=https://www.google.com/&httpsredir=1&article=1422&context=eesp

9. YouTube Link: https://youtu.be/v2KSji-28eU