The Design and Testing of Electronic Drum Sticks

Joseph Fitzgerald, Syracuse University Computer Engineering, Aidan C. Franits, Syracuse University Electrical Engineering, Mark Martello, Syracuse University Electrical Engineering, Anthony Magari III, Syracuse University Computer Engineering

Abstract**—**The goal of this project was to create a set of remotes that can function as electronic drum sticks, that can be used by a drummer to mime the motions of real drum sticks, but without actually striking anything.

I. Introduction

For our 2019 Senior Design project, we chose to design and construct a set of electronic drum sticks. The project was completed within two college semesters at Syracuse University's Center for Science and Technology, senior lab, with help from JLCPCB in Hong Kong and the student machine shop at Syracuse Universities Link Hall. In this report, we discuss the specific hardware and software required to complete this project, and the design iterations we went through, so that anyone wishing to replicate the project will have a solid idea of how to do so.

Traditional drum sets are expensive, bulky, and loud. For example, a basic/beginner-level drum set typical starts at $200. It will also take up at least 24 square ft, which is a significant amount of space consumed to most every household or music classroom. This makes it difficult to transport drums. Upgrading a drum set in number of drums would amplify these problems. Also, the volume of drums is dependent on a drummer's strength/speed of a swing. If a drummer needs to limit their volume, their only option is to physically play softer.

II. Problem Description

We sought to design and produce a pair of Electronic Drum Sticks with an associated computer program to mimic playing a real drum set and eliminate some of the issues associated with traditional drum sets.. The end goal was for users to swing our sticks as if "air drumming" and be able to produce realistic drum sounds, all without making physical contact with anything. To do this, we looked to create Electronic Drum Sticks that had the capabilities of electronic remote sensors with the relative shape, size, and weight of traditional drum sticks. The electronic sticks needed to track user movements, record their movements, process the data, and send the data to our server. Specifically, the program would track the area in which a stick was located relative to the drum set created by our program. If the sticks were swung in a zone, that was designated as a drum, with a certain acceleration and was at a certain roll angle, a drum sound would play. There would be multiple zones to represent multiple drums, like a traditional drum set. It was also important to design the Electronic Drum Sticks such that their size, shape, and weight were as close as possible to that of traditional drum sticks in effort to create a realistic feeling for users. In the end, we wanted users to be able to "air drum" while holding the electronic drum sticks and output accurate drum sounds, without need to make physical contact with anything.

III. Solution

The Electronic Drum Sticks use a Bosch BNO055 IMU (inertial motion unit) to determine how the drummer is moving them. This information is fed to a Raspberry Pi Zero W computer, which determines whether a virtual drum has been "struck" or not. Using a TCP Python Client Server model, information is transmitted via WiFi from the Raspberry Pi Zero W's to a Server running on a PC/Laptop. The server interpreted this information and made one of four drum sounds, depending on the yaw angle of the stick.

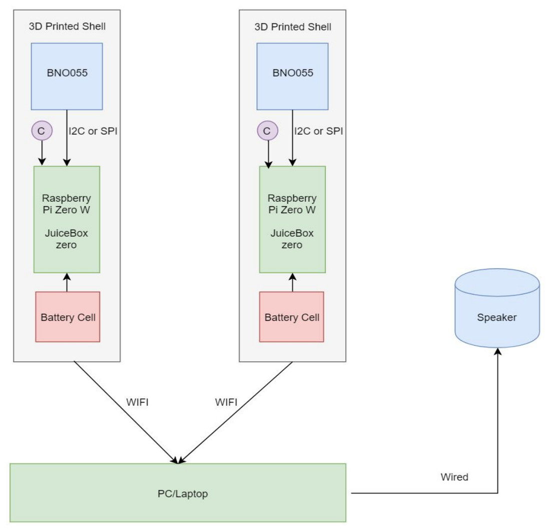

Fig. 1. A block diagram of the project and component layout inside each electronic drumstick

A. *Bosch BNO055*

We chose the Bosch BNO055 IMU for this project for its high accuracy and precision, and reasonable size. This IMU uses an accelerometer, gyroscope, and magnetometer to determine displacement and rotation [1]. It can be easily wired to work with a Raspberry Pi [2].

B. *Raspberry Pi Zero W*

The heart and mind of our electronic drumsticks is the Raspberry Pi. Each Raspberry Pi is connected to the Bno055 sensors and receive necessary data used in the algorithm and client code used in the operation of our overall system.

C. *Python Client/Server and Audio Output*

The two Raspberry Pi single board computers are engaged in a client/server relationship with the host PC that the electronic drumsticks are connected to. The Raspberry Pi's integrated within each drumstick serve as clients to the Host PC's server.

The Host Server is a Python program running on a designated Host PC utilizing multithreaded handling of each client that will connect to said server. It follows the basic Python template of creating/initializing a socket and port number to receive incoming socket communication. It then runs the socket listen() function to wait for incoming socket communication attempts. We added the integer parameter "4" (referred to as a "backlog") within this method to ensure that the server would not allow for more than 4 unaccepted connections at a time to exist. If this set backlog amount is exceeded then the socket will refuse further communication until pre-existing communication attempts are handled.

Each socket connection attempt is processed by threading a new instance of our algorithm method to receive, and derive an appropriate audio output from data sent by our clients via socket communication.

In regards to the clients, each is a simple implementation of the standard Python client template. With each instance of the program containing two major components. The code needed to initiate communication with the designated server program and the code needed to assemble a necessary message in byte format (as needed by socket communication).

To run the Python client program from the Raspberry Pi, we established SSH communication with the raspberry pi and executed the files via the SSH client terminal. Upon bootup of the Raspberry Pi, we wanted to run the client program. This would result in our client code running automatically whenever the Raspberry Pi turned on. However, we were unable to add in this component of our project before the demonstration. As a solution, we used SSH communication to each Raspberry Pi's Ipv4 address assigned by the internet connection each were linked to. While this was fine in short term tests, we quickly found that whenever the Raspberry Pi disconnected from its internet connection and then later reconnected, it was assigned a new Ipv4 address, meaning that sustained SSH connection was inconvenient compared to more automatically executed solutions.

In regards to setting up the socket communication, the client needed to match the port number set by the pre-established server as well as the Ipv4 internet address assigned to the host machine running the server.. Once the Ipv4 address and the port number are confirmed, the client creates a socket object (according to Python's socket template) and connects to the socket established by the server and (at the time of running the client program) and actively listening for socket communication.

Once client/server communication via the sockets is established, the clients send single digit integers provided by the data processing algorithm (running within the Raspberry

Pi client). Each digit represents a expected audio output in response to perceived user action by the data processing algorithm.

The primary reason for having our client machines sending integer messages was to output the correct audiofile with minimum lag (hence the smallest possible file being sent between client and server). With this in mind, we had all of the audio files our system were expected to replicate located on a host computer (tom.wav, snare.wav, high-hat.wav, and cymbal.wav). The reason we needed for these audio files to be located on the Host PC rather than on the Raspberry Pi clients is because, due to past experience by Anthony, sending audio files of any size via wireless socket communication can result in unacceptable system response times to user action. By keeping the audio files on the host machine, where they will be executed to produce the required audio output to simulate conventional drumstick interactions, we intended to help optimize our design's response time and reduce the number of instances of unacceptable lag between user action and system audio reaction.

The clients, instead of sending over the entire expected audio file, will instead send a single integer, either "1,2,3,4", to the server and signal the server to execute one of the four audio files.

How this would work in practice is, after client/server socket communication is established, the client would recognize which audio file to execute via the algorithm. It would then send a single digit integer message to the server, and, once the sent message is received, the server would execute whatever audiofile is set under that particular integer instance.

To actually execute/play an audiofile we chose the "playsound" function located within the Playsound Python library. This function takes the location of the audiofile we want to execute and runs the file to its completion. While we wanted to make it so that the audio files were executed in a manner similar to the performance of actual drums, we were unable to finish every aspect of our system in time for our demonstration.

To correctly simulate a cymbal crash, for example, we would begin the execution of an audio file simulating the sound. While the file was executing, we would concurrently check if the client program indicated that the cymbal "hit action" was executed again before the audiofile was finished running. If that were the case, then we would terminate the current execution of the audiofile and begin a new instance of the audiofile being executed. Since most of our audio files relating to cymbals were 10-15 seconds in length we didn't want to deal with forcing the player to listen to a cymbal crash for 10 seconds on end before they could play another note.

Unfortunately, we were unable to implement either a threading or multiprocessing Python method required by this kind of IO audio file execution. Instead, we simply edited the audio files so that they would execute for the minimum amount of time required to simulate the note… while cutting out "fade" aspects of a note (primarily inherent in the cymbal audiofile). The snare drum, tom drum, and high-hat audio files required little editing as they were already the aforementioned minimum size, however we managed to cut the cymbal audio file from its original 10 second duration to a more bearable 3 second duration. This wasn't ideal compared to our original intent but a reasonable improvement over previous editions of our project.

In addition to editing the length of our audio files to reduce lag between user action and system audio reaction,, we discovered that editing the amplitude of the audio files via the Audacity multi-track audio editor allowed us to increase the perceived volume of our audio output and improve the overall simulation between our system and a conventional drum set.

D. *Algorithm*

To achieve our goal of replicating a drum set we gathered data from the BNO055 IMU, interpreted the data and made decisions based on predefined criteria. For our drums we required that the algorithm use CPU resources effectively and efficiently to reduce latency in our system.

In our system we created helper functions based around the library functions provided to us by Adafruit. From these functions we added debugging statements to help us pick suitable criteria for determining where the drum set was in relation to the user and when to play sound. These debugging statements also allowed us to locate and eliminate issues in our software as well as identify potential hardware issues.

We created two functions called play() and play_drums(). These functions act as the heart of our algorithm. The play() function utilized our helper functions and read in the euler angles as heading, roll, and pitch and the linear acceleration in its X, Y, Z components. Through testing we learned that the euler angles returned by the BNO055 IMU would help us create our drum set. The heading was the side to side motion of the stick. Using this we subdivided the 180 degrees in front of a user into four sections each 45 degrees in size. Each 45 degree section would be a drum (Figure 2).

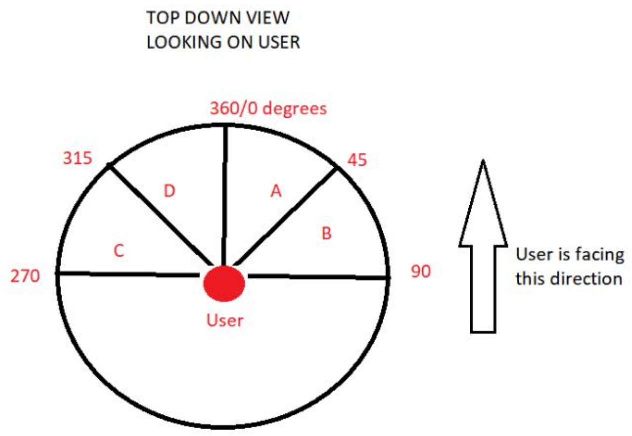

Fig. 2. Top down view of drum sections based on heading.

We then had to determine, when the user was in a specific 45 degree section, what point in the swinging motion was the stick at (Figure 3).

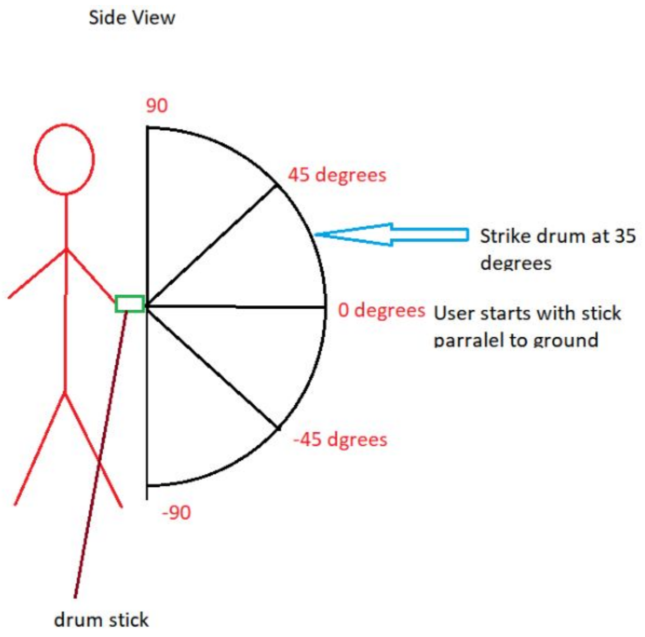
Side View
90
45 degrees
Strike drum at 35 degrees
0 degrees
User starts with stick parralel to ground
-45 dgrees
-90
drum stick

Figure 3. Side view of the activation zone determined by the roll of the stick.

Through testing we found that we could use the roll of the stick to pick a range in which the play_drum() function would be called. One issue with using this method is that once a range was picked, this range would be passed through twice. Once on the swing up and once on the swing down of the stick. To counter this issue we found that using the linear acceleration we could determine when the user was swinging the stick down which is the only time we want the play_drums() function to be called. Using the Z component of the linear acceleration we could make it so that only on the swing down could the play_drums() function be called. Using nested if-else conditionals we could determine if a user was in a specific 45 degree section, what the roll of the stick was and if they were swinging the stick up or down. If the stick was being swung downward greater than a specified linear acceleration, in the range of activation and in one of the four drum sections then the play_drums() function was called. The play_drums() function would take in a value that was passed into it in the play() function and then handle the transmission of data to our server. The value passed in would correspond to the drum audio file that we wanted the server to play through the speaker. Using if-else conditionals we determined which integer value to send to the server. Once the integer value was sent, the server would use an if-else conditional to play a audio file corresponding to that integer.

E. *Printed Circuit Board*

The printed circuit board was designed using the EasyEDA tool, and manufactured by JLCPCB, both chosen out of convenience. We made a physical slot for the Raspberry Pi to sit in, which we worked into the design of the 3D-printed enclosure (see section E). The board had solder pads and holes for a male header (to connect to the Raspberry Pi and JuiceBox), a button (used to start and stop the program), and the BNO055. Cutouts in the enclosure were used to hold it in place, as the board did not have bolt holes.

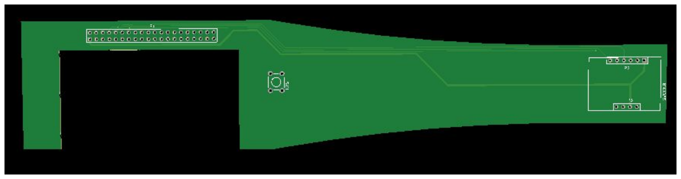

Fig. 4. A 3D rendering of the PCB, drawn by EasyEDA. Note the cutout for the Raspberry Pi. The battery sits behind the PCB in the final design.

F. *3D-Printed Enclosure*

The 3D-printed enclosure was used to hold the circuit board, Raspberry Pi, and battery. It was designed on Autodesk Inventor and fabricated with 3D printers in the machine shop in Link Hall at Syracuse University, with assistance from Mr.

Timothy Breen. The enclosure went through two iterations. The first was bulky and took 17 hours to print. The second was more compact, more comfortable in the user's hand, and the board fit in it better, so we ended up using it. Both versions included bolt holes to hold the two halves of the stick together and bolt holes to hold the Raspberry Pi in place.

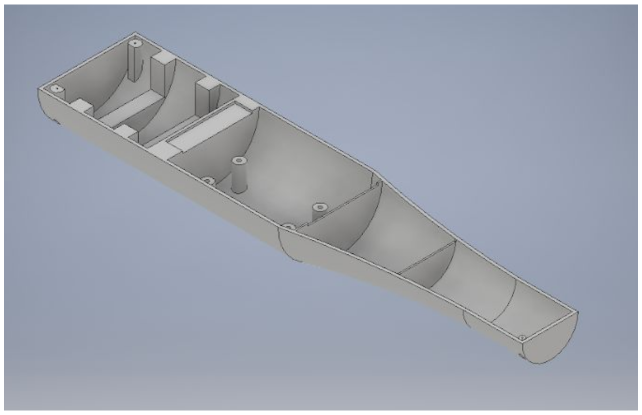
Fig. 5a. The bottom shell of the original design.

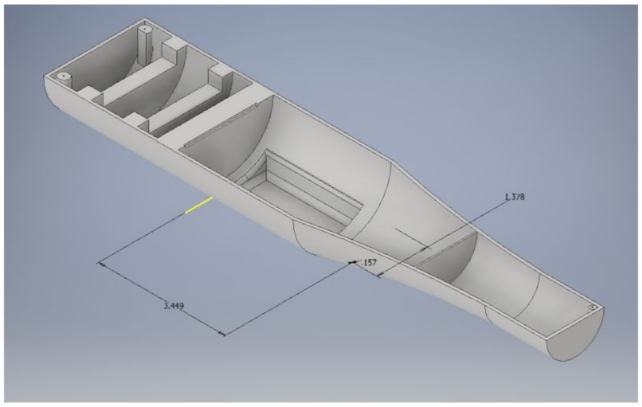

Fig. 5b. The top shell of the original design. The original design had a cutout for the Raspberry Pi.

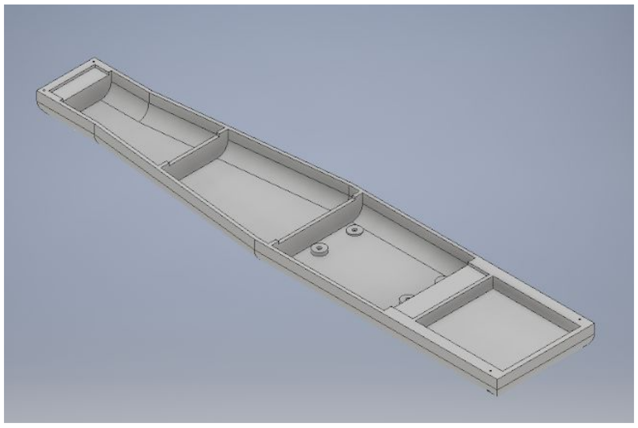
Fig. 6a. The bottom shell of the used design. This design incorporated a much more shallow well for the battery and the Pi posts were much lower.

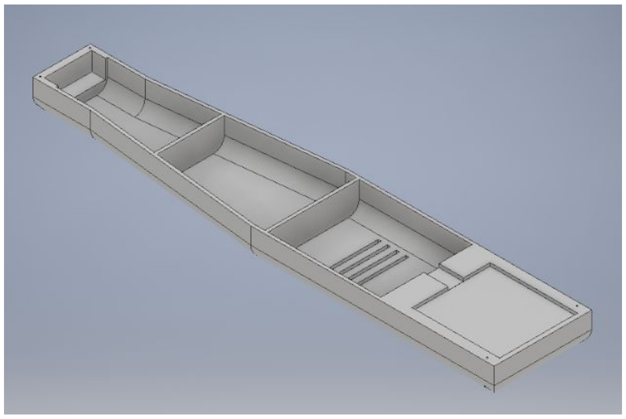
Fig. 6b. The top shell of the used design. No separate cutout for the Pi was needed. We added vent slots for it, however.

IV. Demonstration Results

At the demonstration, we were able to produce drum sounds based on and in direct reaction to the movements of the drumsticks. We demonstrated them ourselves and also let others play with them to help demonstrate the potential flexibility of the project and advocate for the convenience of the design. After a brief period of acclimating to the required actions needed to trigger the algorithm to translate user action to audio output. Most users were able to semi-reliably use the electronic drumsticks, although the project did demonstrate a number of issues. Among the list of problems, a drum would sound twice when the user only intended to play it once, while other times it would not play at all. A glaring issue we experienced was when the Ipv4 addresses of both Raspberry Pi's changed and we were unable to run the Client Program on both due to a interruption in SSH communication.

We eventually fixed this by unscrewing the drumstick shells, removing the Raspberry Pi's, and connecting them to a monitor and usb hub in order to run an "ifconfig" command in each's terminal and learn the correct Ipv4 address of each in order to continue correct SSH communication. If the host machine's Ipv4 address changed then it was less of an issue since the SSH communication was maintained and we could just edit the intended Ipv4 address each client initiated socket communication to via the SSH programs (to find the Host Machine's Ipv4 address, we just needed to open the CMD.exe terminal and run the "ipconfig" command and look for the Ipv4 address under the current internet connection).

While we eventually resumed correct operation in one of the drumsticks, we highly recommend that anyone who replicates our project implements the recommended actions specified in section C (Python Client/Server and Audio Output). Namely, having the Client Program run on startup/boot of the Raspberry Pi. While there could be an issue of the client not knowing the correct Ipv4 address of the server, it could instead connect to some third party resource to find the correct Ipv4 address of its host. We were unable to correctly implement this, but automatic Ipv4 address acquisition and updating are required to make this project and any others based on it remotely convenient for any amount of extended time.

For our weight and size specifications we met our goal and through a series of revisions improved upon our original goals. Our original specification for the size of the device was 33cm long, 7 cm wide at the base and 4cm wide at the top and 7 cm high. Using tighter tolerances we got the size of the stick down to 31.3 cm in length, 6.35 cm wide at the bottom and 3.75 cm wide at the top and a height of 3.4 cm. This revision in size contributed to significant weight loss. The weight of the device was lower then we had expected and allowed for prolonged use of the device without fatigue. The layout of the components, specifically the battery, contributed to comfort and ease of use. Since the battery, the heaviest component, rested in the users hand the weight distribution was comfortable and not over cumbersome.

One aspect of our design that we originally wanted to implement but were unsuccessful in doing so was allowing the user to play the drums facing any direction. To play the drums the user needed to start by facing 0 degrees, the start point seen in Figure 2, which was determined by the calibration of the magnetometer. This start location was based off the calibration of the magnetometer in the sticks which aligned the sensor with the magnetic field. The user had to find where this start position was by using trial and error to find where the drum set was. Once the starting position was located users could play the drums freely but this design required the user to take an extra step before playing the drums. Before the Open House demonstration we attempted to alter the algorithm to use the current position the user was facing and create an offset that could be used to make the drums around the users current heading. We unfortunately ran into issues that could not be resolved before Open House and we were unable to add this feature.

For our design latency was an important issue that we actively, from the beginning of the design process, aimed to minimize. We aimed to have a latency of less than .1 seconds from the time of the strike of the drum to the output of the sound by the server. To do this we limited the amount of function calls in our code and kept the complexity of the code to a minimum. The largest source of latency that we identified would come from the transmission of data from the clients to our server. To reduce the time of transmission we reduced the number of bits sent to the server and used multi-threading on the server. From the time of the strike to the output of the sound we observed an average latency between .1 and .01 seconds. There were some times when the latency would exceed 1 second but no more than 5 seconds. This was due in part to spikes in CPU usage by the programs that would cause the Raspberry Pi's to lag. This issue was not frequent but was in some instances noticeable.

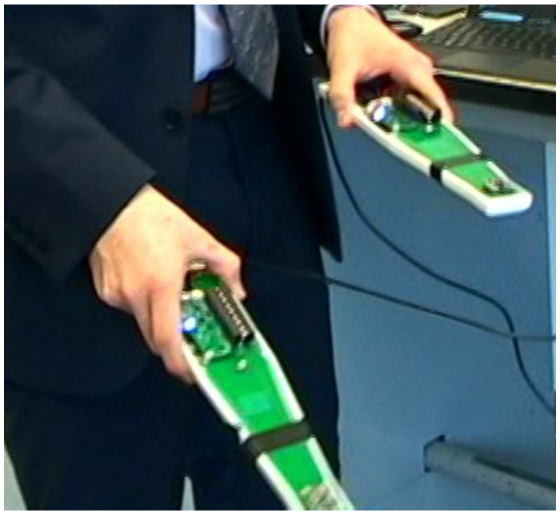
Fig. 7. The drum sticks being operated by Joseph Fitzgerald. The top half of each electronic drumstick were removed for testing and some of the interior components are visible. Near Joe's hand is the Juicebox Zero Power cape connected above the Raspberry Pi. The Raspberry Pi is connected to a set of ribbon wires to a PCB. The PCB contains both a push button (the white dot off of the Juicebox) and the Bno055 (blurry blue component farthest from Joe's hand near the tip of the drumstick).

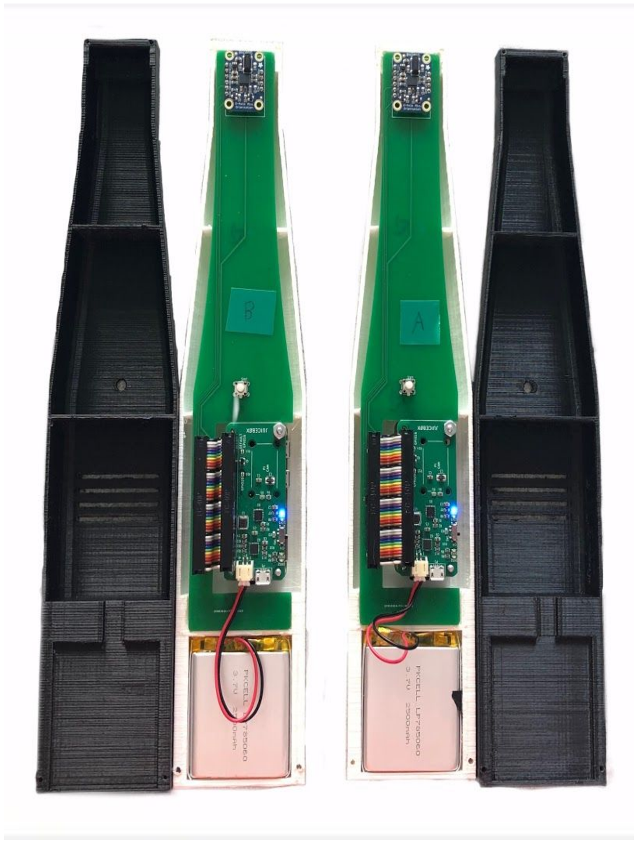
Fig. 8. Close up of the internals of the final design.

V. Conclusion and Discussions

In the end, we were able to create the Electronic Drum Stick which did work to produce realistic drum sounds based off the motion of users. The Electronic Drum Sticks worked wirelessly, tracked users' motions, interpreted the data, and produced audio outputs. While it did work relatively accurately, it was not quite to the level of a traditional drum set or electronic drum set. The main issue we encountered was either a double-sound or no sounds when their should have been one sound. Also, at the open house our we lost the ability to SSH into the raspberry pi's onboard the sticks due to the IP address of the each pi changing. However, our team worked in a timely manner to solve this issue and get our sticks working again. Our Electronic Drum Sticks worked well and with some more fine tuning, could only improve to provide users an accurate "air drumming" experience. For instance, we could rearrange the design for a more comfortable fit in the user's hand, by putting the Raspberry Pi on top of the battery (with different mounting). Another change that we may have yielded better results would be to have a true multi-threaded server instead of using the hotfix of running a server per stick. Improvements to the server code could also have decreased latency to make the sticks feel more like traditional drum sticks. To increase the realistic nature of the sticks we could have also included vibration motors to give the user haptic feedback when striking the drums created by our software. The project highlights the culmination of each team members experience and knowledge in their respective majors. Through effective team communication and hard work our team was successful in designing and producing a functioning set of Electronic Drum Sticks.

References

[1] "BNO055 Intelligent 9-axis absolute orientation sensor," Bosch, Nov. 2014. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/BST_BNO055_DS000_12.pdf, Accessed on: Apr. 19, 2019

[2] T. DiCola, BNO055 Absolute Orientation Sensor with Raspberry Pi & BeagleBone Black, Adafruit. [Online]. Available: https://learn.adafruit.com/bno055-absolute-orientation-sensor-with-raspberry-pi-and-beaglebone-black/software