Hit It!

Stephen Pupa, Shreya Ramesh, George Whitfield

Department of Electrical and Computer Engineering,
Carnegie Mellon University

***Abstract* — Popular rhythm games such as Rock Band or Guitar Hero do not incorporate both portable hardware and/or have pre-defined music libraries that cannot account for the large variety of music genres. Hit It! aims to rectify both these issues by introducing an involved game interface, an active hardware apparatus for users to interact with, and the ability to have beat maps be automatically generated at the user's request. These features enable Hit It! to give players an engaging experience in which they hit drums to the beat of any song they choose while being scored higher or lower based on their performance.**

***Index Terms* — Beat Maps, Digital Signal Processing, Embedded Systems, Entertainment, Graphical User Interface, FSR**

I. Introduction

"Hit It!" is a drum based rhythm game that offers a middle ground between other rhythm games on the market. Currently, the market is filled with rhythm games that either do not involve hardware or rely on a cumbersome and expensive setup that is inaccessible to some audiences. Furthermore, these games typically do not provide the user the ability to import their own songs, further limiting the demographic it appeals to based on the songs they offer.

Our project, Hit it!, offers a small and portable drum apparatus and the ability for users to input their own song. Hit It! analyses the frequency content of the player's song to generate a beatmap, allowing the user to play to the rhythm of their favorite songs. By incorporating the creation of beat maps based on user-inputted songs, the demographic can extend to all ages.

II. Use-Case Requirements

We identified 6 use case requirements that have helped us better define success in meeting our objectives.

A. *Compactness*

The hardware should fit into a standard 15 liter backpack. The ability to fit within standard carrying devices (backpacks) is crucial for enabling users to transport the system with ease. The convenience of transportation also factors into the user's impressions of the game outside of their experience with the game itself.

B. *Effective Beat mapping*

The game should play user provided songs and generate beatmaps to go along with them. The beatmaps are accurate to the input song; an accuracy of at least 80% will feel true to the provided song. The duration of beat mapping relates back to initial impressions and encumbrances to the user. If the game requires a lengthy and tedious setup process when trying to use certain in-game features, users will avoid that feature altogether. The beat mapping also needs to be accurate, otherwise it would be dishonest to call it a beatmap, as an inaccurate beatmap defies the whole point of having the beatmap to start with.

C. *Frame Rate*

The game should run faster than 30 frames per second. Frame Rate is an important factor relating to user enjoyment, as a slow frame rate will lead to "choppy" visuals and a worsened user experience. This factor will likely conflict directly with graphical quality, as higher quality graphics will require more time to modify each frame.

D. *Latency*

The latency between the user hitting the drum and the game recognizing the input should be less than 70 ms. Latency is a combined criteria that assesses the total delay within both hardware and software from the time a user provides input to the time that input is displayed back to the user. This is an extremely important criteria as rhythm games are fundamentally reliant on timing, so delays in those timings will be very noticeable

E. *Ease of Setup*

The user should be able to set up the game in less than one minute. The ease of setup characterizes the user's initial impressions of our game. A cumbersome and tedious setup process will establish a negative opinion of new users from the start.

F. *Drum Recognition Accuracy*

The hardware should correctly identify user drum hit inputs 99% of the time. Drum recognition accuracy is the number of times the drum module registers a hit and passes

that information to the computer divided by the total number of times the drum is hit. This criteria most accurately defines the "consistency" of our system by assessing how often the system performs as it has been defined to.

III. Architecture and/or Principle of Operation

There are two main components of the software as illustrated by the block diagram (Appendix Figure 12): beat tracking in Python and gameplay code in C++. For the C++ portion, Open Graphics Library (OpenGL) is used to create the gameplay code. OpenGL is a relatively low level tool used for creating video games that adds technical complexity to the development process.

The Python code analyzes all of the songs that have been placed into the *songs* folder, and exports their beatmap data as JSON files. Then, when the player selects a song to play, the corresponding JSON is read by the C++ code and loaded into the game.

The primary goal of the beat tracking Python script is to export a series of timestamps where the supposed beats are detected. In order to do this, the audio file must be provided as an uncompressed wav file. Which beat is mapped to a particular drum is dependent on the frequency of the beat detected. The time stamps, the BPM, and the drum mappings are then entered into a JSON. This data is read by the gameplay code at runtime.

The hardware portion of this design is focused upon a series of "Piezoresistive Sensor Drum" modules, each of which consists of a 3D printed drum housing and a custom PCB upon which all the drum's circuitry is soldered.

In addition to these 4 drum modules, there is also a separate "Central Module" that houses the ESP32 microcontroller development board and its supporting circuitry soldered to a perf board. This module serves as the interface between the drums and the computer by modifying the raw data received by the drums into a more digestible form for the game program. Furthermore, the central module also has a 3D printed outer shell, however its form factor is much simpler than the drums, as it is effectively just a small box with holes for connectors, whereas the drum housings need a large circular opening for the drum pad and smaller holes in the corners for their several feedback LEDs. The interaction between these 5 total modules is depicted below in Figure 1.

The design of both these modules has shifted since the design report. For the central module, we originally planned for it to be similar in width/length to the drum modules, which would've aided in easy storage, however after gaining more experience with the costs of 3D printing, we decided this convenience wouldn't be worth doubling the module's cost. Regarding the drum modules, the only changes that have occurred to the housing were the addition of corner holes to account for the feedback LEDs and the removal of the crossbar that supported the FSR in the prototype drum design. The custom PCB already serves this purpose, so having the crossbar in addition would be redundant.

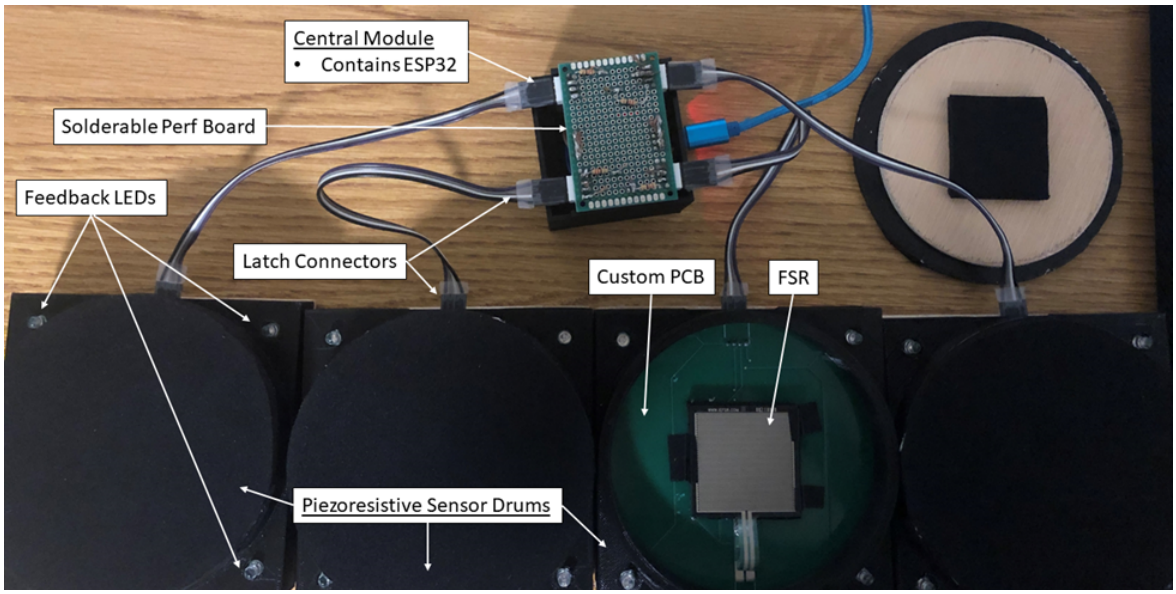

Figure 1: Overall System Architecture (Hardware System).

The circuitry within both these modules is depicted in Appendix Figure 13. The "Piezoresistive Sensor Drum" module contains the FSR and a 1 MOhm resistor that acts in tandem with the FSR to form a voltage divider, which is core to the drum's operation in creating analog voltage changes. Additionally, each drum module has been outfitted with 4 green feedback LEDs since the design report. These LEDs receive signals from the ESP32 when the drum's output voltage is high, which allows them to turn on and provide visual feedback to the user when the drums are hit. Furthermore, these LEDs serve an excellent debugging purpose, as both developers and users can observe the LEDs upon hitting the drum to verify the drum is working properly and that their hit was registered.

The voltage outputs from these drum modules feed into the "Central Module" also shown in Appendix Figure 13. These voltages connect to the GPIO pins of the ESP32, which has access to an internal Analog to Digital Converter, allowing it to convert the analog voltages received from the drums into digital signals. The results of this conversion are then transmitted out of the ESP32 chip using the UART serial communication protocol into a UART-USB interface chip, which then sends that information to the connected computer using the USB serial communication protocol. As mentioned above, the incorporation of feedback LEDs since the design report necessitated additional functionality from the ESP32. Most of these changes occurred within the ESP32's Arduino code, however a new 220 Ohm resistor was also needed for each LED power line to limit the current sourced from the ESP32 to the drum LEDs.

IV. Design Requirements

Each of the use-case requirements specified earlier has an associated quantitative metric that defines what it means for this project to "succeed" at satisfying the user requirements. However, the design side of the project's implementation provides additional restrictions and specifications on those requirements:

A. Latency

The goal duration between drum hit (input) and GUI response (output) is no more than 70 milliseconds. For the software stream, the frames per second goal of 30 has the consequence of increasing the software latency to 33 milliseconds by default, as any inputs received after a frame update can not be displayed through the GUI until the next frame update, which at worst could occur 33 milliseconds later (33 milliseconds = 1/30 seconds). 33 milliseconds is much higher than the expected runtime of the gameplay loop, so this stream's latency is constrained to 33 millisecond as long as the FPS remains 30.

Therefore, the hardware stream has the remaining 37 milliseconds to work with, which is divided up between the FSR, the ESP32's ADC, and the UART-USB interface. Based on the hardware specifications of the FSR,[5] latency should be negligible as its rise time is on the order of microseconds. Therefore, the ADC and UART-USB interface need to be under those 37 milliseconds when combined.

B. Drum Recognition Accuracy

The drums should recognize over 99% of valid drum hits. This requires a definition of "valid hits," which when coming from a user perspective should only be anything the user wanted to portray as a hit. During the design report, we used a study of forces on drumsticks[19] to reach a target of 10 N for this goal. The reason we thought this value would be good is because within said study, they provide a graph of force vs. time while striking a drum, and following the initial force peak, there are several smaller force peaks caused by vibrations. None of the secondary force peaks reached 10 N, so we believed considering all forces above this threshold to be valid would maximize our recognition rate without gaining any false positives.

The problem with this study is that the swing velocity used by the drummer in the force test (7 m/s)[19] was much higher than the swing speeds some users were actually applying to the drums during user testing. Therefore, we had to rethink this metric. After watching an instructional drumming video,[17] we found that drumsticks should be resting a few inches above a drum while at rest for optimal performance. As such, we chose a new metric of 3 inches for our swing distance, rather than the 10 N metric from before. Having a distance metric is more useful as well, as measuring force applied from contact with a moving object can be quite difficult, and with a height metric, we can simply use gravitational force from the optimal height as a stand-in for what the lightest user hits should apply in terms of force. Furthermore, regarding the issue of unintended vibrations creating false positives, we implemented solutions within the code to ignore such hits that occur too close in proximity to each other to be valid, eliminating that concern altogether.

The metric of over 99% recognition is most heavily correlated to the reliability of our FSR and the design of our drum pad modules. Assuming our FSR works to the specifications provided in its datasheet,[5] there should be no issues with it recognizing forces so long as they are properly applied.

C. Ease of Setup and Compactness

The system should take no longer than 1 minute to set up. This duration starts from when the user touches the modules and finishes when the game is operating and interactable for the user. The system should also be small for ease of portability.

There are two main aspects to this setup time, physical and electrical. The physical setup time is primarily defined by factors such as untangling wires and connecting modules together. The solution we incorporated was to bundle the cables together, reducing the likelihood of cords becoming tangled. Furthermore, setup time also depends on portability of the hardware apparatus, which further increases the importance of satisfying that use-case requirement.

Regarding the electrical setup time, this includes the laptop and microcontroller boot times. For the laptop, we expect it to be turned on either prior to setup or during the physical setup process above. Regarding the ESP32 module, all it needs is to be connected to a powered computer to fully boot and connect to the computer. Furthermore, the ESP32 has a boot time on the millisecond to second range,[3] which is far quicker than the minute goal we have from the use case requirements, so this shouldn't pose an issue.

D. Effective Beat mapping

The game should be capable of transforming provided songs into playable beatmaps. These beatmaps should take no longer than the song's duration to produce and should match sample beatmaps with over 80% accuracy.

The first challenge is the time it takes to create a beatmap from the user's song. Having it take several hours would severely decrease the user's enjoyment of the game, so the goal is to create a beat map from a user-inputted song in the length of the audio clip or one iteration over the song. To achieve this, an amplitude threshold-based onset beat detection algorithm is used. The suggested algorithm is to determine the consistent noise spikes and consider the beat amplitude. However, user inputs may have to be restricted with the usage of this algorithm. Coupled with dynamic programming to lessen the recalculation time for similar calculations, there is only a need to loop through the audio file once. These factors should assist in ensuring the time it takes to create the beat map is at or below the length of the song.

The second goal is having the beat map accuracy over 80% when compared to the beat maps created by the industry standard package (Librosa). There are several challenges with this. There is a high potential to detect beats where there are none, and the algorithm may also miss some beats. To meet this criteria, the aforementioned algorithm will also be used. By also using the BPM to ensure that counted beats roughly have the expected timings, the beat accuracy should be above 80%.

E. Frame Rate

The game should run at 30 frames per second (FPS). 30 FPS was the target goal set by what potential users have come to expect from the gaming market. However, from a design perspective, this means that the gameplay loop of the main codebase, including the communication with the user's hardware, must take no more than 33 milliseconds. This shouldn't be a difficult goal to attain under the right circumstances, however certain other game design criteria could potentially limit the system's ability to reach this goal. The most concerning tradeoff is likely going to be that between frame rate and the graphical quality/intensity.

Due to how the GUI displays images to the user, creating more visually engaging graphics often requires greater code complexity and as a result more calculations to be performed by the code each cycle of the gameplay loop. As such, going into the project we expected there may come a point where further increasing the visual quality of the game results in the framerate dropping below the 30 FPS target.

V. Design Trade Studies

A. Beat Mapping

The main package that the beat mapping algorithm depends on is Librosa, an audio-processing package tool. The primary use of this package is to determine the beats per minute (BPM) of the audio file.[14] This package was used instead of a manual beats per minute determination due to latency concerns. Because a majority of BPM tracking algorithms depend on a combination of fourier transformations and loops into the transformations, the expected latency would have doubled the time for the total beat mapping computation.

Determining the beat mapping algorithm requires the calculation of the average energy throughout the audio file [2]. This is calculated through the summation of the squared intensities over the audio file. This is then normalized and compared to an instantaneous energy. This does require looping throughout the wav files.

Because the algorithm for determining the time stamps of the beats is a loop based approach, one of the tradeoffs in this algorithm is accuracy for speed. Ordinarily, these strategies should be avoided due to the desire for a lesser time of computation. However, because a majority of these values within the loop are recalculated, dynamic programming can be used to lessen the computation time. Accuracy was prioritized over speed because the inclusion of the dynamic programming should allow for a faster computation time than otherwise. While this does sacrifice efficiency in space, the use case requirements for the project do not set a limit for drive space. This approach maximized time savings as well as accuracy.

B. Graphics Programming Tradeoffs

The gameplay for *Hit It!* is implemented in C++ with OpenGL. Video games are typically created nowadays using game engines such as Unity or Unreal. OpenGL is a more low-level tool for displaying video game graphics compared to game engines, which adds technical complexity to our project. Although OpenGL stands for "Open Graphics Library", it is not actually a library but rather a specification for the interface between the software and GPU.

There are several alternatives that we could have used instead of OpenGL for graphics displaying. One popular alternative is PyGame, which is a Python library for making games. PyGame also comes with built-in sound support, which OpenGL does not. We also could have used DirectX or Metal, which are graphics interfaces for Microsoft and Apple platforms. OpenGL was chosen over these alternatives because 1) the team has experience with OpenGL, 2) there are numerous cross-platform OpenGL libraries available (this is not the case for DirectX and Metal because they are OS-specific), 3) programs in C++ with OpenGL run faster PyGame (due to Python being a slower language), and 4) C++ is common for development in the video games industry.

C. Hardware Trade Studies

For the hardware portions of this project, there were several comparisons that needed to be made. However, the most important decisions were centered in five areas, the drum pad material, the FSR, the microcontroller, the drum housing material, and the circuitry base.

Neoprene vs. Natural Rubber

Regarding the choice of drum pad, our team looked at several options with good elasticity and eventually settled between two materials, those being Neoprene and Natural Rubber. Neoprene is a synthetic version of rubber known for its resilience and elasticity, which makes it an excellent material for products like bouncy balls. Natural rubber has many of the same positive qualities regarding its elasticity, however it doesn't have the same resilience as Neoprene, although this isn't an issue since it will not be subjected to extreme temperatures or conditions for our purposes. Our team's original goal was to purchase samples of both, allowing their differing functionalities to be tested in a prototype drum firsthand, however, the cost ended up being the deciding factor here, as purchasing natural rubber sheets through Amazon was an order of magnitude higher than obtaining Neoprene rubber sheets. This circumstance made the decision between the two quite easy, as our team couldn't justify spending half of our budget on a material we may or may not end up using in the final product.

Flexiforce Pressure Sensor vs. Interlink Electronics FSR

The main qualities needed in the FSR for this project were response time and reliability, which derive from the design requirements of latency and drum recognition accuracy respectively. Our team's two main FSR candidates, those being the Flexiforce Pressure Sensor and the Interlink Electronics FSR 402, both satisfied these criteria, however there were also important secondary considerations.[4-5] Specifically, the Flexiforce Pressure Sensor had the benefits of high precision and a large force sensing range of approximately 0 to 445 Newtons, whereas the Interlink Electronics FSR 402 had the benefits of being much cheaper as well as having a larger sensing area. However, the FSR 402 has a much smaller sensing range than the Flexiforce sensor, as it can only reliably measure from 0.2 to 20 N, and in case the project needed to use different levels of force during a post-MVP stage of the project, our team opted to prioritize the Flexiforce sensor. However, after performing some preliminary tests on both sensors, it became clear that the user's force was dampened to such an extent through a layer of Neoprene that the vast majority of the Flexiforce sensor's measuring range was going unused. This eliminated the primary reason for selecting the Flexiforce sensor, and as such our team decided to focus our further efforts on the FSR 402 sensor instead since it performed similarly for a much cheaper price.

ESP32 vs. Arduino

Regarding the selection of microcontroller, our team narrowed down the choices to two options. The first was the ESP32 designed by Espressif and the second was the Arduino Zero made by Arduino. These two boards share many

similarities, as both are 32-bit microcontrollers that operate at 3.3 Volts, and which are programmable using the extremely accessible and user-friendly Arduino IDE. However, there were a few defining factors which ultimately solidified the ESP32 as the preferable microcontroller.[1,3] Firstly, the Arduino Zero uses an ATSAMD21G18 which gives it a clock rate of 48 MHz, whereas the ESP32 uses a Tensilica Xtensa LX6 microprocessor chip, giving it a clock rate between 160 MHz and 240 MHz depending on the operating mode. Due to how having low latency is a core design requirement of this project, the ESP32 is the preferable option here. Secondly, the ESP32 is overall much cheaper to purchase at around $11 for some development boards whereas the Arduino Zero is closer to $40. There are certainly cheaper Arduino options available, but even the cheapest Arduino boards tend to be around $20, which is still higher than the ESP32 development board. Lastly, our team also has great deal of experience working with the ESP32 microcontroller, which would result in time savings of the course of the project. These three factors are what ultimately solidified the ESP32 microcontroller as the preferred option for this project.

3D Printing vs. PVC Piping

One tradeoff that our team experienced was between 3D printing and using available materials such as PVC pipes for developing our drum module housings. The original idea was to use 3D printing for all of the drum modules and the central module, however the first round of prototype printing came back more expensive than we originally anticipated, as 3D printing one prototype module cost approximately $55. When we extrapolated this to 5 more modules, we expected to incur an additional $275 in expenditures. This prompted an exploration of alternatives, specifically the use of PVC piping and wood for our drum module housings. This alternative would've been an order of magnitude less expensive, however it had the substantial downside that our team would need to invest much more time and energy to achieve an inferior product. The primary tradeoff here wass price vs. time and effort, and in the end, we decided to make due with 3D printing as we valued the additional time and energy much higher. Luckily, smart revisions to our 3D designs allowed us to trim a significant amount of the expected printing costs, shrinking the original $275 estimate down to only $164, which further validated our choice to take this route.

Custom PCB vs. Solderable Perf Board

Custom PCBs were used within the drum modules due to the large space between the several components and need for an exposed solder pad to attach the FSR to. However, the price of these PCBs was slightly higher than expected (approximately $54 for 1 set) and the manufacturing/shipping delay took around 3-4 weeks. As such, when the need arose for a circuit board within the ESP32 module, another custom PCB wasn't an appealing option. Instead, we chose to use a solderable perf board, which had the two main benefits of being far cheaper ($5 or less) and quicker to assemble/solder by hand (2 days). This saved much time during the final days of this project, during which a custom PCB likely wouldn't have arrived in time. A depiction of this solderable perf board can be seen below in Figure 2.

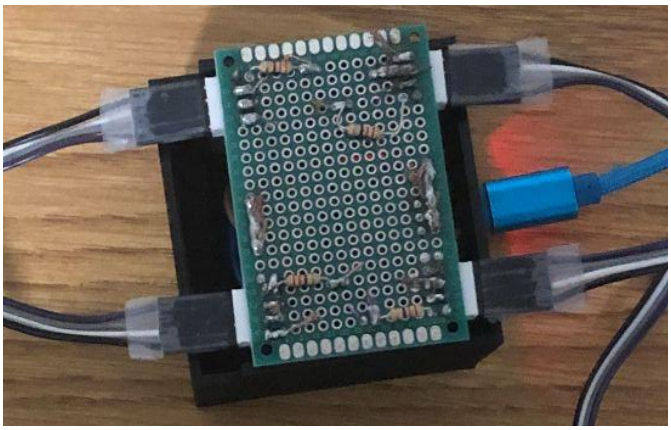

Figure 2: FSR resting on Neoprene square at center of PCB.

VI. System Implementation

Hardware System Implementation

As mentioned within Section III, the "Piezoresistive Sensor Drum" circuitry is soldered to a custom PCB and placed within the 3D printed drum housings depicted in Figure 1. The way the drum modules operate is by having the FSR leads be soldered to the PCB while its sensor rests on a small square of neoprene attached to the PCB center. This is depicted below in Figure 3.

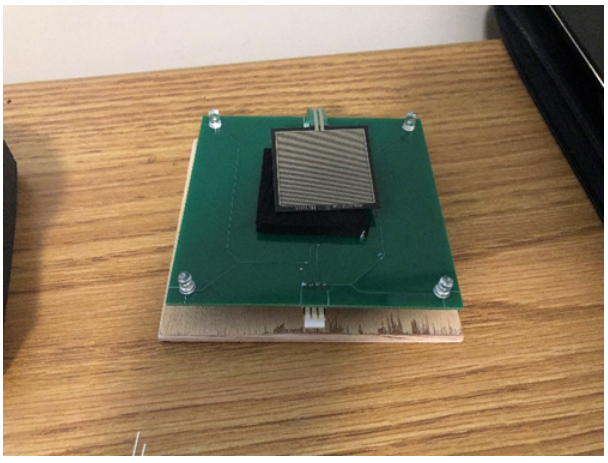

Figure 3: FSR resting on Neoprene square at center of PCB.

The sensor being raised on this square creates a lever system with the drum pad and the FSR, as when force is applied to one end of the drum, the adhesive attachment of the drum pad's far end keeps the pad from rising. This redirects the majority of the force applied to the drum pad through the FSR no matter where on the drum pad the user hits. This system is not perfect however, as the neoprene's elasticity allows for some flexibility on the drum's far end, meaning not all of the force will go into the FSR. Furthermore, forces applied very close to the drum's edge will also have much of their force lost through the load-bearing walls of the drum module. A brief diagram depicting this can be seen below in Figure 4.

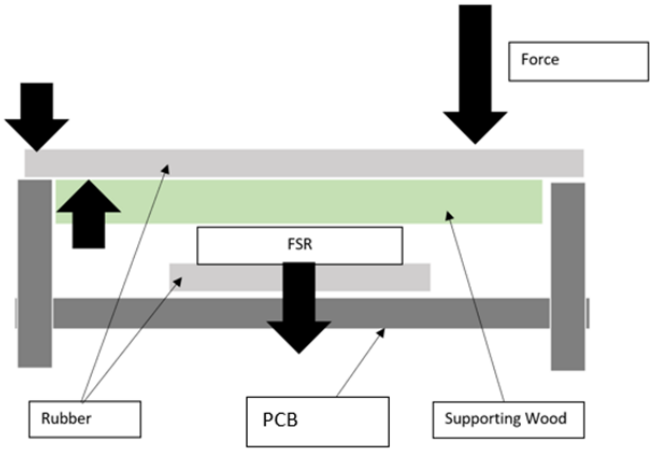

Figure 4: Force redirection into the FSR.

Each drum pad consists of two materials, Neoprene rubber and wood. The Neoprene slightly dampens the applied force, giving the drum a more satisfactory feeling when hit while also protecting the FSR from direct contact with the user, whereas the wood provides the necessary rigidity needed to perform the lever effect described previously, as otherwise the Neoprene would simply bend without redirecting the forces into the FSR as desired.

Regarding the circuitry within the drums, the FSR forms a voltage divider with a 1 MOhm resistor, so that when the FSR receives a force and its resistance drops (from approx. 10 MOhm to near 0 Ohm), the voltage at the node between the FSR and the resistor will change drastically. This changing analog voltage, as well as the GND and VCC (3.3 Volt) nodes which participate in the voltage divider are passed through the opening in the modules base. However, since the design report an additional node has been added to this group, specifically the LED voltage line which powers the drum's feedback LEDs. Originally, it was planned that the voltage divider output would also be responsible for driving these LEDs, so that another line wouldn't need to be fed between each drum and the central module, however due to issues sourcing the required amount of current through the FSR and issues related to LED illumination duration (discussed more below) we decided that having the ESP32 handle these power lines would be the preferable option. Therefore, each drum has 4 nodes (GND, VCC, LED Power, Voltage Output) that are passed out of their base through a locking pin header, which reduces the possibility of suddenly disconnecting with a module during use. A depiction of this pin header can be seen below in Figure 5.

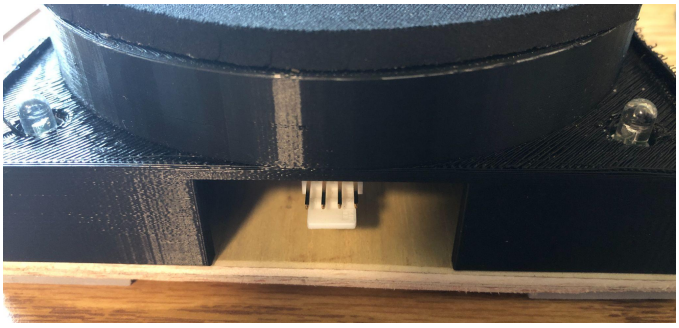

Figure 5: Locking Pin Header within drum module

The voltage divider outputs from the drum modules each feed into the ESP32 development board module through its GPIO pins, specifically those which can be used with the ESP32's internal Analog to Digital Converter. This allows the ESP32 to read said pins and get their respective voltages represented as digital values. From here, the ESP32 will then package that information into a single byte and transmit it using the UART serial communication protocol to the UART-USB interface chip present on the development board, which allows for the message to be converted to the USB communication protocol before being forwarded through the board's USB-C port to the connected computer. The only component in this stream that requires programming is the ESP32 microcontroller chip, which is programmed using the Arduino IDE due to the ease of use and convenient helper functions that IDE provides. As mentioned previously, the ESP32 has also gained responsibility for powering the drum LEDs since the design report, which it also performs using its GPIO pins. Each drum's respective LED power pin is turned on once the ESP32 reads a high voltage from the drum's output line, however, the changes within the drum voltage dividers happen near instantaneously and disappear just as fast. As such, if these LEDs power pins were to turn off as soon as the voltage divider voltage disappeared then the LEDs would hardly be turned on for a perceivable amount of time for the user. To remedy this, the Arduino code simply sets a timer for each drum's LED power pin whenever a high voltage is read from that drum, and once that timer reaches zero the LED power will be set low again. This allows the LEDs to

remain on for a long enough period of time for users to react to their presence.

Software System Implementation

There are several criteria that must be met for a song to be imported into the game. Otherwise, the song will either fail to load in the game properly or the generated beatmap will feel awkward and it will not follow the music properly.

- The song must be encoded as a .wav file with 44.1 kHz sample rate
- The song must have a constant tempo
- The song can not have strong impulse aperiodic noises

Beat Map Generation

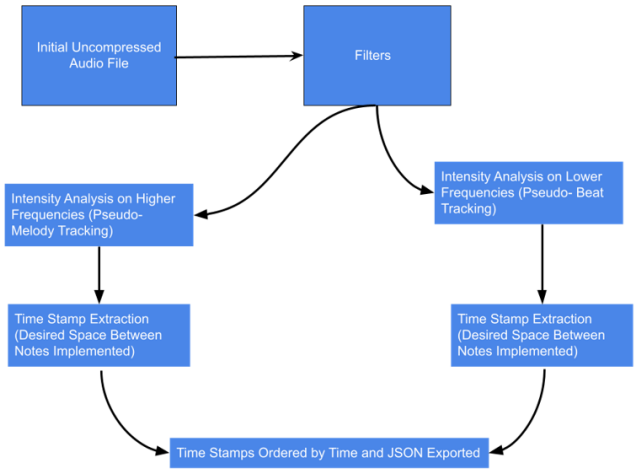

Figure 6: Overall Pipeline for Beatmap Generation

Multiple packages were used to create the beat map generation script. Firstly, Librosa is an audio-processing software package written in Python.[14] This package was used to obtain the beats per minute (BPM) of the audio file. A majority of the remaining digital signal processing was done with relatively simple python packages, including Numpy, Scipy, and MatPlotLib [10,13]. The BPM was then used to determine the approximate amount of time between beats in the audio file. In this case, twice the amount of time of seconds per beats according to the BPM was required to detect a note

The game has four drums. Originally, four filters were used to divide the audible frequencies into four categories: low, medium-low, medium-high, and high. Respectively, the trial frequencies were in the ranges: 60 Hertz to 250 Hertz, 250 Hertz to 500 Hertz, 500 Hertz to 2000 Hertz, and 2000 Hertz to 4000 Hertz. These values were largely guided by referenced online resources as well as testing that had been conducted[16]. However, after various playtesting sessions, it was found that the division of the audio file into four bands was too difficult to play. To remedy this, the audio file was divided into two frequency bands using Scipy filters: low pass filter that filtered for frequencies below 500 Hertz and a high pass filter that filtered for frequencies above 500 Hertz. This division allowed a pseudo-melody tracker to be created. By regarding the division of the audio files as two separate files, individual intensity analyses could be performed on both files to create beat tracking and melody tracking. The intensity analysis is described below.

The expected amplitude is determined using an amplitude threshold-based onset beat detection algorithm. Currently, the assumption is that the audio files have a static BPM. By determining the average energy over the entire file and comparing it to the instantaneous energy, beat energy can be calculated as a larger change between intensities when compared to other points in the audio file. This can then be mapped to the time [2]. One issue with this method is the implicit assumption that the BPM is static, as well as the constant scalar value by which average energy is multiplied by that the instantaneous energy must be greater than. However, a predictive algorithm can be implemented in which the genre of the audio file can be determined, allowing for a more accurate scalar value. In order to send the timestamps of the beat tracking algorithm to the main gameplay loop, the time stamps themselves were exported into a JSON format that is able to be read by the game code.

Gameplay Code

The gameplay is implemented using C++. CMake is used to compile and link source files, as well as all of the external libraries used. The graphics were created using OpenGL, which is a popular interface for rendering graphics. Additional libraries are used for reading files and playing audio. C++ was chosen as the language for the gameplay code because it is common in the video game industry, and the members of the team are personally interested in using C++ with OpenGL.

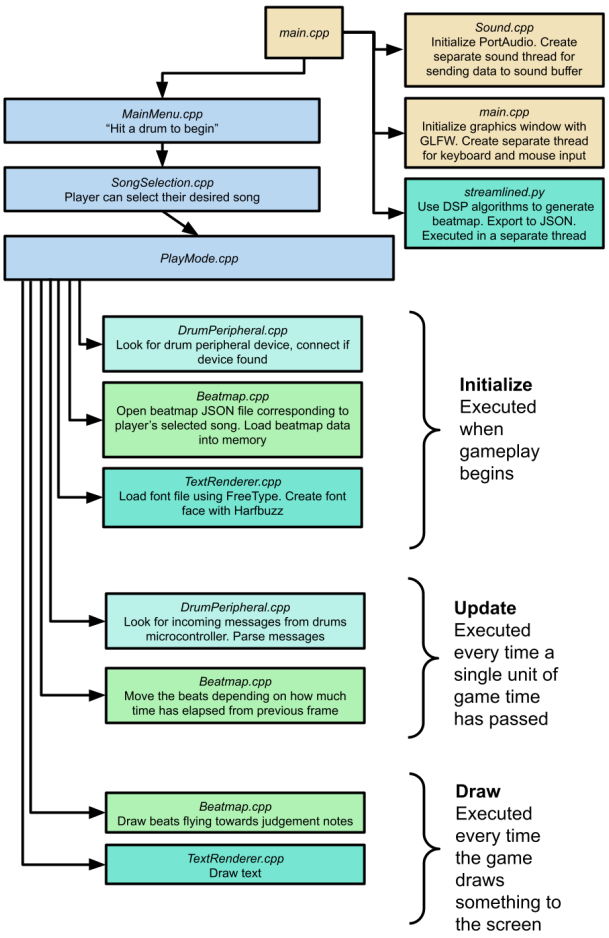

Figure 7: Diagram of gameplay code flow. This is not a comprehensive list of all of the files in our project but rather an overview of how the most important files interact with each other. For a complete list of our gameplay files, please refer to our GitHub page.

Hit It! uses several external libraries for writing graphics code:

- GLAD - Implementation OpenGL functions
- glm - Math utility library for graphics programming
- GLFW - Cross platform window creation

For file I/O and playing audio, the following libraries are used:

- libaudiodecoder - reading WAV files
- jsoncpp - reading beatmap files encoded as JSON
- PortAudio - real time sound rendering

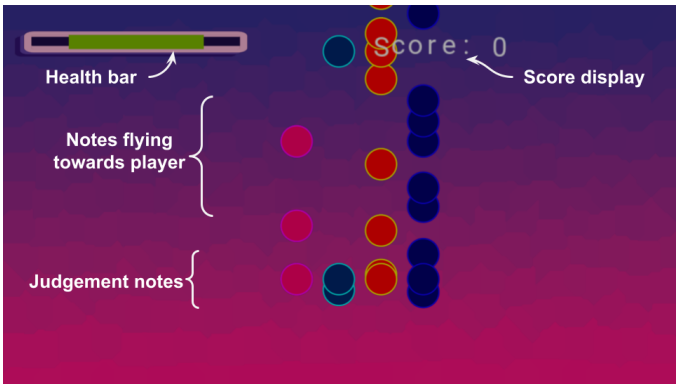

Figure 8: Screenshot of gameplay GUI.

The GUI for the song selection and gameplay was heavily inspired by *Dance Dance Revolution* (DDR). The current design for the song selection screen is very similar to the song selection GUI from DDR. The health bar in our gameplay screenshot is placed at the same location as the health bar in DDR.

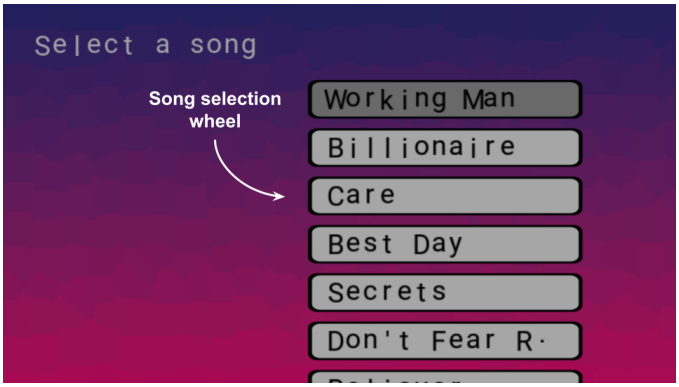

Figure 9: Screenshot of song selection GUI

VII. Test, Verification and Validation

Most of the use case requirements and design requirements outlined previously were defined through some form of quantitative metric. Keeping these in mind, we performed a series of tests to verify the extent to which our final design met our target goals. The methods we used to test, the results we achieved, and how they compared to our expectations are described for each metric below.

Tests for Latency

There are two primary intervals which constitute the latency within our design, that from the instant the drum is hit to the moment that information is sent out from the hardware, and that from the moment the software can read that information to the point it can act accordingly. As discussed previously in the design requirements section, our target goal for latency overall was 70 ms, with 33 ms being dedicated to

the software assuming operation at 30 FPS and the remaining 37 ms being available to the hardware.

Regarding the hardware testing, we used an iPhone camera under the "slow-mo" setting to record at a rate of approximately 100 frames per second (10 ms intervals). Using this high-frame count, we were able to count the frames from the instant the drumstick made contact with the drum pad to the point an oscilloscope could read the message being sent. This is slightly less precise than we expected it to be, as in the design report we expressed that we hoped the frame rate would be closer to 240 fps, however this rate was still serviceable, and the results can be seen in Table 1 below.

	Trial 1	Trial 2	Trial 3	Trial 4
Response Time (ms)	40	40	30	40

Table 1: Hardware response times

Using these results, we had an average response time of 37.5 ms for the hardware, give or take 5 ms in either direction due to the somewhat low precision of the measurements. This is higher than we were expecting, as we believed the high clock speed of the ESP32 would easily be able to perform the simple Arduino loop quickly. However, within these trials we also observed that the feedback LEDs turned on near instantaneously, no more than 1 frame after the drumstick made contact. Since the LEDs turning on was the final action performed within the code, this informed us that reading the voltages GPIO pins did not have a significant delay and that it was the act of sending the data from the ESP32 causing the delay. This unfortunately meant that code optimization wouldn't be capable of reducing this latency by a significant amount.

However, we were able to make up for some of this delay on the software side of the project, as our gameplay code could run at 60 FPS rather than the 30 FPS we originally planned. This meant that only 17 ms were being consumed by the software side of the design, and therefore when we add the 42.5 ms worst case hardware latency to that value our full system latency comes out to around 60 ms, which is below our desired target of 70 ms.

Playtesting

Since Hit It! is a video game, playtesting is critical to test the system in front of users to ensure that the game is easy to play. The team conducted playtesting sessions during the week of April 18. During these sessions, the team asked CMU students who have never played the game to play several rounds of Hit It! and also connect wired to the microcontroller. After the test session, students completed a survey asking them about their experience.

According to the survey, 75% of playtesters felt that the drums were behaving strangely. This was a concerning piece of feedback and as a result, we focused on enhancing the drum hit recognition more before performing our official recognition tests. As a result, the drums were much more responsive during testing and during the public demo on May 6, 2022. Since the data collected from the survey is qualitative, we cannot use this data to directly measure whether we have met a use case requirement.

Tests for Drum Recognition Accuracy

To test the drum recognition accuracy, our original goal in the design report was to test specific force quantities in a variety of different locations along the drum to ensure that all forces above a target threshold of 10 N counted whereas those below that threshold would not. However, as discussed in the design requirements section, we realized this metric was problematic once we began testing. As such, we defined our tests to instead validate the accuracy of swings beginning at a height of 3 inches above the drum, as that is an optimal resting position while drumming normally.[17] To evaluate the drum's using this new metric, we hit the drums in two different ways. The first was manually from the height of 3 inches above the drums, as this would be the most accurate representation of the forces the drum would experience from an average user. The other way we hit the drum was using exclusively gravity's effect on the drumstick from the 3-inch height, as this simulated the weakest forces we might expect a user to apply. Additionally, we also hit the drum in two different locations, the center and the edge, which represented our expected most and least optimal use scenarios respectively, as hits near the center would likely have higher recognition rates since they occur directly above the sensor. The results of this testing can be seen in Table 2 below.

	Drum 1	Drum 2	Drum 3	Drum
Manual - Center	250/250	250/250	250/250	250/250
Gravity - Center	250/250	250/250	250/250	250/250
Manual - Edge	242/250	235/250	236/250	247/250

Table 2: Drum recognition accuracy results

Each category of testing received 1000 trials across all 4 drums. The averages didn't vary based on the way the drum was hit, however they did vary based on location. The averages for the different locations are 100% for centered hits and 96% for edge hits. This aligned well with our expectations as we had little fear that hits near the center would be recognized properly and it was hits near the edge that concerned us. However, although 96% is less than our target of 99%, we still considered this to overall be a successful showing by the drum modules, as the most concerning failure would be in which one has no way of guaranteeing that their hits will count. As long as the user has a reasonable and near guaranteed way of ensuring that their inputs will register (hitting near the center) then the use case requirement regarding the consistency of the recognition accuracy will be satisfied.

Tests for Ease of Setup

To test this criterion, we had random participants with no prior experience with our project attempt to setup our system during one of our user-feedback and playtesting sessions. These results were recorded and averaged, which can be seen in Table 3 below.

Trial 1 (s)	Trial 2 (s)	Trial 3 (s)	Trial 4 (s)	Trial 5 (s)
119.00	26.50	36.08	32.00	44.92
Trial 6 (s)	Trial 7 (s)	Trial 8 (s)	Trial 9 (s)	Trial 10 (s)
25.20	38.30	38.40	37.40	27.94

Table 3: User testing set-up times

Averaging these results yields 42.57 seconds, which is below our target of 1 minutes and aligns well with our estimates going in. However, one important thing to note is that not all trials met the 1-minute target, as the first result was nearly twice as long. During this specific trial, the user had a lot of trouble getting the connectors into the latches. The most probable explanation for this is that they were being overly cautious with the system to not break it, as when we expressed to other participants that the system was rigorous enough to withstand moderate applications of force, they seemed to have a much easier time. From the development perspective, this tells us that information as to how much force one can feel comfortable applying without fear of breaking the system should be included within the instruction set that describes how to set up the system.

Tests for Compactness

Testing the compactness of the system was rather simple. All that we needed to do was measure the dimensions of each module and sum their volumes together. Table 4 below shows the results of these measurements.

Module	Width (cm)	Length (cm)	Height (cm)	Volume (cm^3)
Drum	12	12	5	720
Central	6	6	6	216

Table 4: Drum and Central module dimensions/volume

Total Volume (cm^3) = 4 * (720) + (216) = 3096 cm^3 = ~3.1 Liters

The total volume of the entire system comes out to about 3.1 liters, which is far below our target of 15 liters we specified during the design requirements. This lined up with our expectations, however, we also believed a more qualitative evaluation of this test derived from the use-case requirement of the system fitting in a backpack would be valuable to assess. Luckily, the system was carried within a backpack nearly every time it needed to be transported, so once again the design passed the test without much issue, which aligned with our pre-testing expectations.

Tests for Beat Tracking Timing

To test for the length of the beat map generation algorithm, a series of fifty beat maps were generated by the final

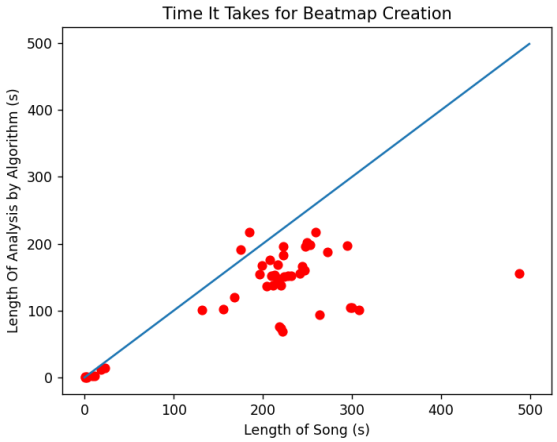

Figure 10: Beatmap generation time vs. song length

algorithm. The "time" module from Python was used to measure the time that the user-inputted song was identified by the algorithm to the moment that the JSON was created. The algorithm operated faster than its goal 96% of the time. In the following graph, the blue line represents the user requirement for the project. It maps the length of the audio file. The red dots represent the duration it took for the beat map generation for a variety of time points. One issue with this test is the similarity of lengths of given audio files. Because there were ranges within this testing (between 300 and 500 seconds) that no audio files of a given length were used, the data surrounding this range will have to be assumed to follow the same range as the data within this range. Overall, the length of the generation of beat maps was met for this project.

Tests for Beat Map Accuracy

Testing for beat tracking accuracy relied largely on two different avenues. Firstly, the Librosa module has a built-in beat tracker. The values that the algorithm for this project creates for the beat tracker can be compared to the Librosa model. On average, the Librosa module generates approximately 400 beats for a three minute audio file. In comparison, the beat tracking algorithm generates approximately 600 beats for a three minute audio file. Notably, the Librosa module appears to create time stamps for their beats with a required time between seconds per beats (from the BPM). This suggests that they are not closely following the rhythm of a specific song. However, following the Librosa module as a baseline, this suggests a 67% accuracy rate with the other notes being counted as false positives.

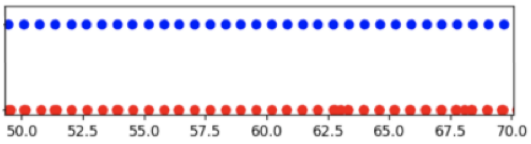

Figure 11: Librosa timestamps vs. beatmap algorithm

The above figure represents a sample taken from an instrumental song. The blue dots represent Librosa's timestamps outputs and the red dots indicate the time stamps from the beat map generation algorithm designed for this project. The values on the y-axis are redundant and were created to designate some space between the two colors for clarity for the user. As can be clearly seen, the red dots largely line up with the blue dots. However, false positives do exist, shown by the clustering of red dots at the 62.5 second time stamp. This is likely caused by the project's need to closely track rhythms and melody changes rather than a simple beat.

Frame Rate

To test that the system is achieving the goal of 30 FPS, a timer was added to the gameplay render loop in the C++ code. This timer was able to record and output the time between frames, which therefore allows for the average frame rate over a long period of play to be easily calculated. The average frame rate was 61 frames per second.

VIII. Project Management

A. Schedule

The schedule for this semester can be seen in Appendix Figure 14. Largely, the team was able to keep up with the schedule. As the project progressed, the Gantt chart was updated to reflect specific ideas or goals that were being implemented at the time. Furthermore, the minimum viable project deadline was moved to a week later than originally anticipated due to issues with integration.

Some post-MVP goals were not included in the final project due to time constraints. For example, initially, the game would have been cross platform. There were various cross platform packages that did not work as anticipated that made this goal impossible. Similarly, the original idea for melody tracking would have used a completely different algorithm than beat tracking. However, through experimentation and time constraints, the melody tracking implemented in this project was sufficient for the user's needs and was used throughout the project.

The goals of the project were accomplished by the set deadlines and the responsibilities of each member largely stayed the same.

B. Team Member Responsibilities

Shreya

- Implementation of musical analysis algorithms for beatmap generation
- Exporting beatmap to JSON

George

- Implementation of gameplay: placement of notes on screen, handling player input, calculating player's score, song selection
- Implementation of real time playback using PortAudio
- Game design of gameplay

Stephen

- Design and Manufacture of Drum pad/ESP32 modules
- Design of FSR circuitry
- Programming of ESP32

C. Bill of Materials and Budget

The budget for our project can be seen in Appendix Figure 15. The most noticeable feature is that it is mostly dominated by the 3D printing costs, which in total were around $220 and a little more than half of our total expenditures. However, with these expenditures the total amount spent was still only around 65% of the total $600 we had been allotted. Within the BOM, items that were not strictly needed within the final design are highlighted with blue and items that were not part of our plans during the design report are highlighted in olive. The only unplanned expenditure was the use of custom PCBs, however that item still wasn't entirely unplanned as it was an avenue we were considering during the design report. All other unplanned items were scavengable from our own personal electronic supplies, however their costs wouldn't have been substantial (probably around the $20 range).

D. AWS Credits

AWS credits were not used for this project. The game is an individual offline experience. There would be no advantages to hosting the game on AWS.

E. Risk Management

The primary risks we anticipated from this project's outset were that the OpenGL library would fail, the drums would not be able to recognize a majority of the hits, and the beat map generation would be unable to generate any valid beats. Luckily, these failures were for the most part unencountered, so many of the risk mitigation strategies held in the design report were not necessary. However, throughout the implementation of the project, there were several other risks that occurred that needed to be accounted for.

One such example is the implementation of the drum modules, as after soldering one of the PCBs, it was discovered that we made a mistake that were not easily correctable. Without risk management, this would have meant that another PCB would've needed to be ordered. However, we ordered extras PCBs initially to mitigate this risk and were able to avoid long shipping delays. In the case of PCBs, ordering extra is often not too much of a financial cost, as they are produced in batches anyways, so we viewed this extra cost as worthwhile insurance.

Another concern was with computer failure, or human error and losing the code written. To manage this risk, we used GitHub consistently throughout the project. By implementing version control, any changes made that may have caused damage to the game or any changes that would lose important progress were easily reversible. Overall, by using GitHub, there was no chance that the entire project would be lost.

Overall, there were a variety of risks that the project posed. Although none of the major risks we predicted in the design report did not happen, they helped us to predict issues that may have occurred and mitigate these risks. Because of the risk management conducted, the project was successful.

IX. Ethical Issues

As a product intended for solely recreation purposes, Hit It! is not predisposed to serious adverse ethical impacts. However, there are certain qualities of this game that could create minor ethical concerns to certain people, although other games in the market also have similar concerns..

One example is the inclusion of custom hardware for playing our game. Although custom hardware makes a rhythm game more engaging, custom hardware could also exclude those who can't use that hardware. For example, some people with physical disabilities may struggle using drums, especially when compared to using a keyboard or controller. However, this potential concern is not isolated to Hit It!, as it's ubiquitous within all games that use custom hardware. For example, someone in a wheelchair clearly can't play DDR nor could an amputee play Guitar Hero in the intended fashion. These aren't concerns that can be engineered away using clever design, as these custom input methods are core to the identity and experience of those games. Mitigations could certainly be implemented alongside these custom controllers. The way this has been accounted for in Hit It! is to have a secondary input option of keyboard presses which also map to the beats in-game. This broadly widens the forms of acceptable inputs, as most USB controllers allow for button remapping, which allows for the user to play the game in the way most comfortable to them if they so choose.

Another possible ethical concern is addiction, a common problem with video games. Once a product is in the customer's hands there is little the producer can do to ensure it is used properly, making it difficult to mitigate. However, some games do take steps to limit the amount players can play in one sitting. The most notable examples of this are games such as Clash of Clans or FarmVille. They use stamina meters that recharge slowly over time, which prevents users from performing too many actions at once. However, these systems are almost always accompanied by predatory microtransactions and/or addictive gambling, which are generally considered to be worse. One could of course implement such a feature without predatory financial practices, however there are many other factors that must be considered when adding such a system, such as the inconvenience to responsible players and the ease at which addicted players could surmount these artificial hurdles. As

such, we haven't opted to include any mitigation measures for this issue within our game for two primary reasons. Firstly, there is little precedent for these features in other rhythm games, which much of our design inspiration comes from, and secondly, we don't feel addiction is particularly problematic within Hit It! due to the lack of online features that stimulate competition within the game itself.

X. Related Work

One game which shares a lot of similarities with our product is the franchise rhythm game Rock Band.[9] Specifically, both our design and Rock Band have controllers modeled after drums, in which there are 4 drum pads that the user is meant to hit in rhythm with the notes appearing on the screen.

However, while our designs share the same number of drum pads, the Rock Band controller also has an additional input in the form of a bass pedal. This bass pedal adds extra complexity and difficulty to the game as users need to split their focus between their foot on the bass pedal and the drums they hit with the drumsticks in their hands. During the design report, we considered taking inspiration from this secondary mode of user engagement, however we ultimately chose to prioritize other features post-MVP.

One notable downside of Rock Band is the lack of song flexibility their product has, as although they provide a large and varied list of popular songs, such as "Caught Up In You" by .38 Special and "I Bet My Life" by Imagine Dragons, their website also has a tab where users can request songs to be put in the game. The presence of this tab highlights the major limitation of their product which our design is aiming to solve, as when a user wants to experience a specific song in game, the best they can do is message the developers of the product themselves and hope that they'll be able to obtain the rights. The fact that Rock Band felt the need to accept this input also proves the importance of our system's beat mapping functionality, as by avoiding this process altogether our product saves users the significant and unreliable hassle of messaging developers directly.

Additionally, there exists another project this semester that shares many similarities with ours, specifically Team A3's project Flex Dance. The two core similarities between our projects are that both use custom hardware setups based on Force Sensitive Resistors and how both are rhythm games. The main differences between our projects though is that our project has the additional features of beat mapping software whereas their project has a much greater emphasis on their hardware setup.

XI. Summary

In summary, *Hit It!* is a rhythm game where the user hits drums to the beat of a song. The game design is inspired by popular rhythm games such as *Rock Band* and *Dance Dance Revolution*. The technical complexity of our project comes from our implementation of the drum hardware, signal processing algorithms, and use of low level graphics libraries. The user interest in this game is primarily because of the active and engaging hardware and software experience. Furthermore, because users are able to input their own songs into the game, the game creates a more personalized experience. Because some of these design traits are currently lacking in the market, the introduction of this fun, personalized experience allows users of all ages to play the game.

The system was able to meet a majority of our design requirements, although there were some issues with the beat mapping accuracy. Noticeable limits on the beat mapping accuracy were in terms of the type of audio file submitted. A file had to have a wav extension or the tracker would be unable to analyze it. Furthermore, although no wav input could have caused the program to crash, pop songs and instrumental audio files were better suited to the beat tracking algorithm.

There are several ways in which the game aesthetics and functionality could be improved in the future. As of now, the score tracking is dependent on how closely the user hits are to the beat. However, in a different iteration, score calculations could also be tied to intensity of the hit, double hits, or a variety of other factors. In general, the gameplay could be enhanced by adding visual and auditory feedback based on the user's drum hit force. Finally, the beat map generation algorithm could also be improved. The current version only has one level of difficulty. However, in a different iteration, users could be able to choose multiple different difficulty levels for each individual song. Unfortunately, because the members of our team are moving to different parts of the country post-graduation, it would be extremely difficult to continue working on the project in the future. As such, the project is complete for our team.

The lessons learned in this project were numerous. We found communication was extremely important in the success of the project. Mediums such as Discord and GitHub allowed the team to keep track of progress made. Open source software and tutorials helped us become familiar with languages and technical aspects of the project that we would have been unable to do alone or with previous classwork done.
Overall, the creation of Hit It! was an informative and entertaining experience for us to create. Each of us learned far

more about our various disciplines than we had known before and we hope to carry these lessons into our future endeavors.

Glossary of Terms

Beatmap - the sequence of notes that fly towards the user during the game

Rhythm game - games (typically video games) which require the player to move their body in time with music

Beats - The groove of a song. These are the musical rhythmic hooks that the listener feels when listening to a piece of music.

BPM - Beats per minute. This is also referred to as the tempo. The term "beat" as used in BPM does not refer to the "beats" as defined above, but rather it refers to the steady musical pulse that the listener feels when hearing music.

Judgment Notes - In rhythm games, the "judgment notes" are the notes on the GUI that the beatmap notes fly towards; the player must strike the game hardware when the beatmap notes align with the judgment notes.

FSR - Acronym for Force Sensitive Resistor. These resistors are usually made of piezoresistive materials, which give them the property of having different resistances under different levels of pressure/force. Generally, the resistance of FSRs decreases as more force is applied.

ADC - Acronym for Analog to Digital Converter, which converts an analog signal into a digital one.

References

[1] "Arduino Zero." Arduino Online Shop, Arduino, <https://store-usa.arduino.cc/products/arduino-zero.>

[2] "Beat Detection Algorithms." *GameDev.net*, http://archive.gamedev.net/archive/reference/programming/features/beatdetection/index.html.

[3] "ESP32 Series Datasheet." Rev. 3.8. Espressif Systems. Pg. 8-14. <https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf>

[4] "FlexiForce® Standard Model A201." ZFLEX A201-100. Rev. A. Tekscan. Pg. 1-2. <https://cdn.sparkfun.com/datasheets/Sensors/ForceFlex/FLX-A201-A.pdf>

[5] "FSR® 400 Series Data Sheet." PDS-10004-C. Rev. 2. Interlink Electronics. Pg. 2-3. <https://cdn2.hubspot.net/hubfs/3899023/Interlinkelectronics%20November2017/Docs/Datasheet_FSR.pdf>

[6] Glad used for implementation of OpenGL functions. Feb 2022 https://github.com/Dav1dde/glad

[7] GLFW used for creating OpenGL context and window. Feb 2022 https://www.glfw.org/

[8] glm used for mathematical helper functions with OpenGL. Feb 2022 https://github.com/g-truc/glm/releases/tag/0.9.9.8

[9] "Harmonix Music Systems, Inc.." Rock Band Rivals, Harmonix Music Systems, Inc., <https://www.rockband4.com/.>

[10] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. *Array programming with NumPy*. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2. (Publisher link).

[11] jsoncpp used for reading JSON beatmap files. Feb 2022 https://github.com/open-source-parsers/jsoncpp

[12] libaudiodecoder used for decoding music files. Feb 2022 https://github.com/asantoni/libaudiodecoder

[13] McFee, Brian, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. "librosa: Audio and music signal analysis in python." In Proceedings of the 14th python in science conference, pp. 18-25. 2015.

[14] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, et al. (2020) SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17(3), 261-272.

[15] PortAudio used for audio playback. Feb 2022 http://www.portaudio.com/docs/v19-doxydocs/index.html

[16] Smoot, Jeff. "Understanding Audio Frequency Range in Audio Design." *CUI Devices*, 4 June 2020, https://www.cuidevices.com/blog/understanding-audio-frequency-range-in-audio-design#:~:text=The%20generally%20established%20audio%20frequency,octave%2C%20you%20double%20the%20frequency.

[17] Sutherland, Jeff, director. Drums - Hand Position and Approach. YouTube, YouTube, 5 Aug. 2011, https://www.youtube.com/watch?v=A1rbnMsy7F8. Accessed 22 Apr. 2022.

[18] Vries, Joey de. *Learn Opengl - Graphics Programming Learn Modern Opengl Graphics Programming in a*

Step-by-Step Fashion. Kendall & Welling, 2020. Feb 2022.

[19] Wagner, Andreas. “Analysis of Drumbeats: Interaction between Drummer, Drumstick and Instrument.” *Master's thesis at the Department of speech, music and hearing*, Kunglia Tekniska Högskolan, 2006, pp. 19–23.

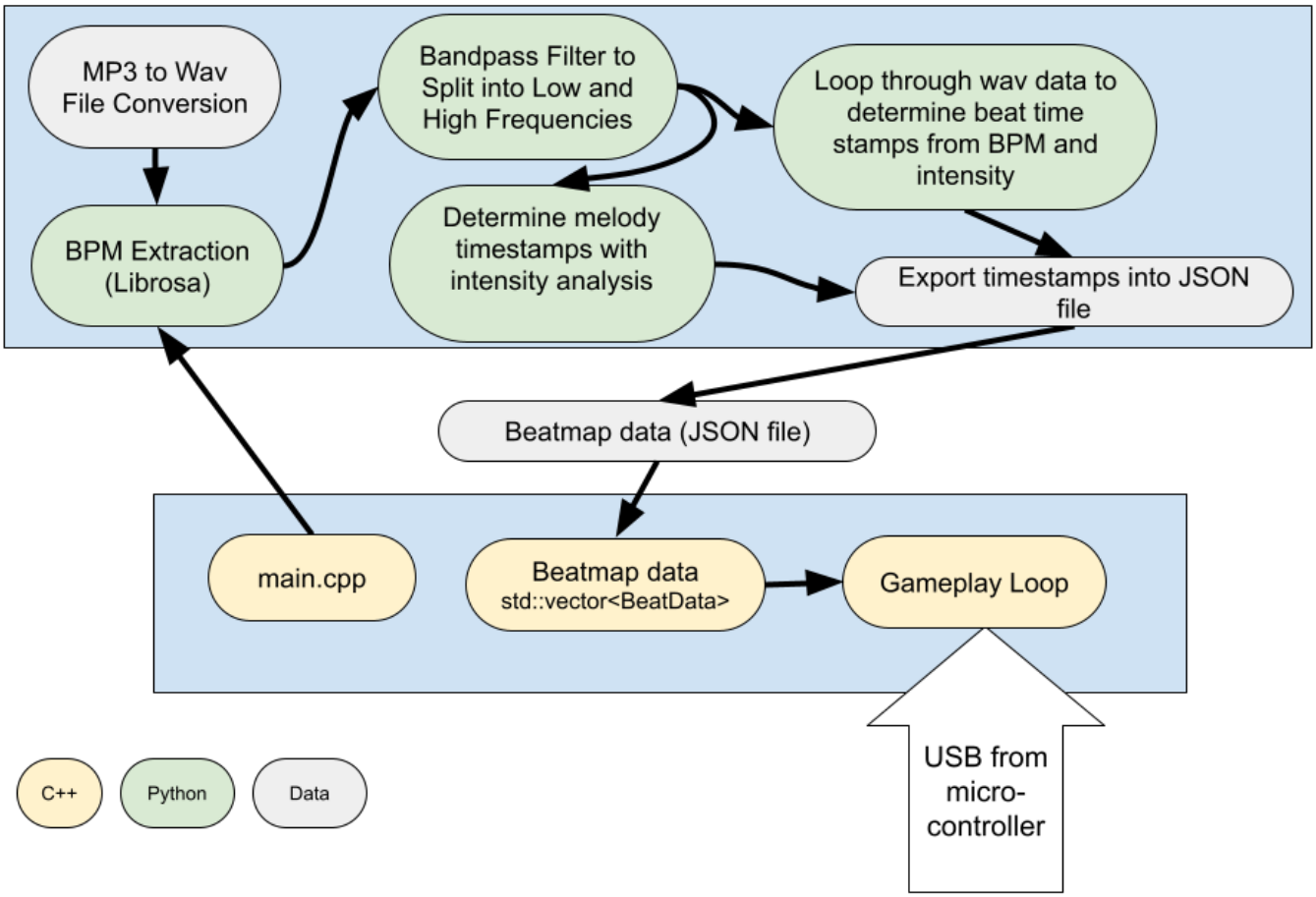

Figure 12: Block Diagram of the Software information stream

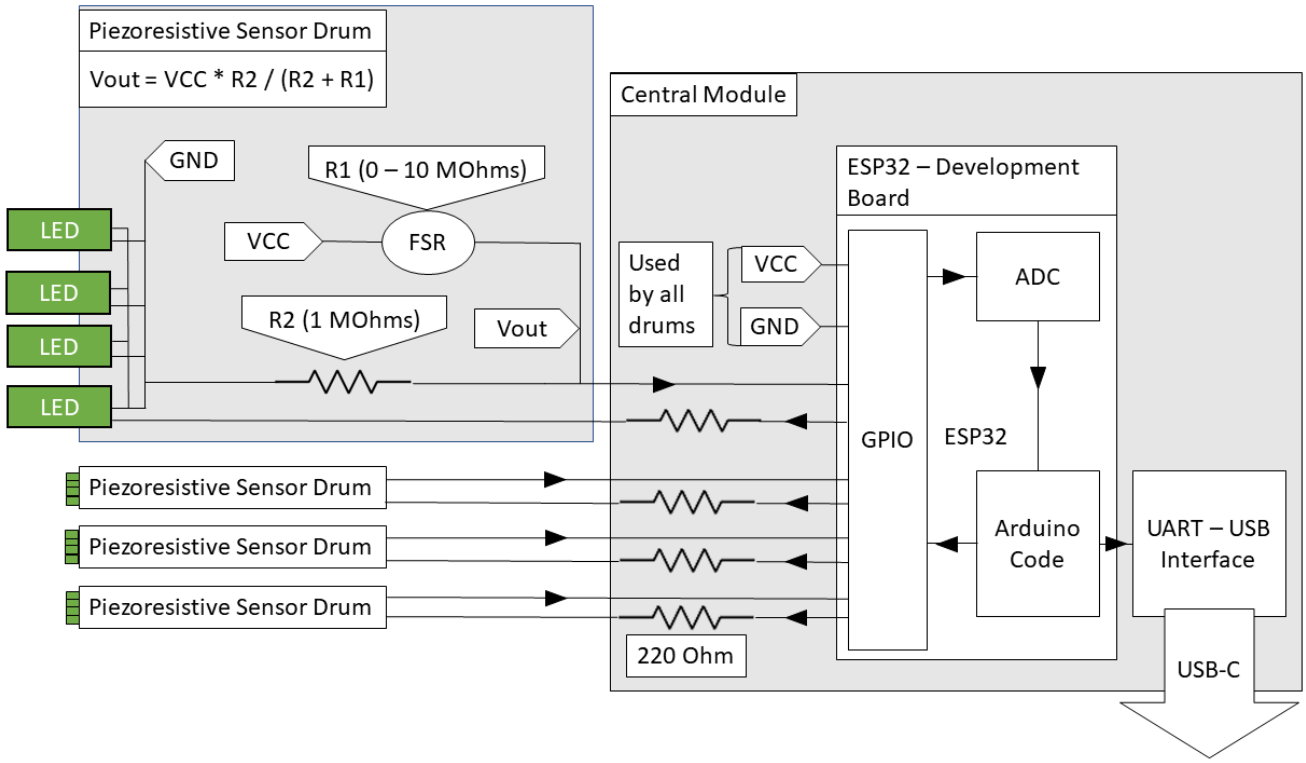

Figure 13: Block Diagram of the Hardware information stream

Figure 14: Gantt chart depicting the progress over project lifespan

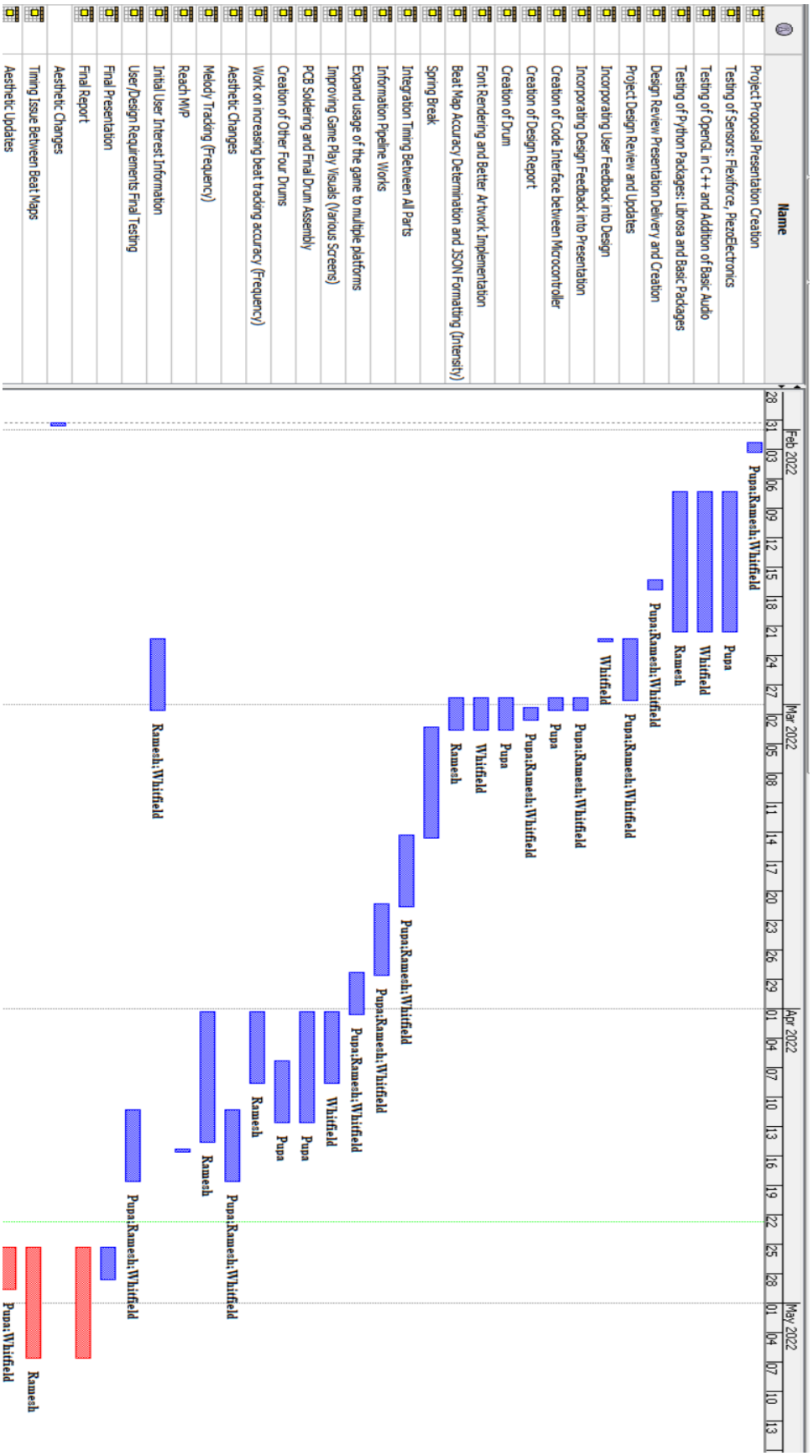

Figure 15: Bill of materials (BOM) showing total expenditures and areas of expenditure over project lifespan. Item name refers to the name depicted on the purchase website.

Prototyping and Initial Testing Phase Supplies

Qty	Item Name	Manufacturer	Description	Distributer	URL	Unit Price	Shipping + Tax	Total Price
1	FlexiForce Pressure Sensor - 100lbs.	FlexiForce	Piezoresistive force sensor	Sparkfun	https://www.	$19.95	$13.68	$33.63
5	MCP6004-E/P	Microchip Technologies	Op Amp Quad Low Power Amplifier	Arrow Electronics	https://www.	$0.63	$6.63	$9.80
1	DZS Elec 15PCS 35mm Piezo Disc Transducer	N/A	Cheap backup teasting piezo transducers	Amazon	https://www.	$8.99	$0.00	$8.99
1	10-Pack 6mm Rubber Neoprene Square Padding Sheets	Genie Crafts	Neoprene rubber sheets for drum pad testing	Amazon	https://www.	$9.99	$0.00	$9.99
1	HiLetgo ESP-WROOM-32 ESP32 ESP-32S Development Board	HiLetgo	ESP32 development board	Amazon	https://www.	$10.99	$0.00	$10.99

Final Product Development Materials

Qty	Item Name	Manufacturer	Purpose	Distributer	URL	Unit Price	Shipping + Tax	Total Price
10	22-12-4042	Molex	Isolating Circuitry from user and/or harm	Mouser	https://www.	$0.68	$7.99	$34.17
5	OY105KE	Ohmite	Having custom PCBs will reduce chance of circuitry damage	Mouser	https://www.	$2.96		
20	C503B-GAN-CB0F0791	Cree LED	Mapping the internal circuitry of each module	Mouser	https://www.	$0.23		
5	FSR Model 406	Interlink Electronics	Making the IE FSR connections more reliable	Interlink Electronics	https://buyi	$3.99	$9.99	$35.92
2	FSR Model 402	Interlink Electronics	Locking Pin Headers which will be placed at module interfaces	Interlink Electronics	https://buyi	$2.99		

3D Printing Expenses & PCB Manufacturing Expenses

Qty	Item Name	Manufacturer	Unit Price	Total Price
1	Prototype Drum Module	Tech Spark at CMU	$56.50	$56.50
4	Final Drum Modules	Tech Spark at CMU	$36.00	$144.00
1	Central/Control Module	Tech Spark at CMU	$20.00	$20.00
5	Custom PCBs	PCBWay	$20.00	$53.69

	Not included/needed for final design
	Not planned during design report

Previously Owned

Qty	Item Name	Purpose	Example URL
1	DaFuRui Small breadboard kit	Debugging exclusively	https://www.amazon.com/DaFuRui-tie-F
1	ELEGOO 120pcs Multicolored Dup	Debugging and use within central module	https://www.amazon.com/Elegoo-EL-CF
1	Velcro Strips (Melsan 1x4 inch Hoo	Easily attachment and detachment of drums	https://www.amazon.com/Melsan-inch-
Undef.	Miscellaneous Scrap Wood	Bases for drum modules	N/A
1	Universal Proto-board PCBs 4cm x	Soldering of central module circuitry	https://www.adafruit.com/product/4785

Category	Total Cost
Prototyping and Initial Testing Phase Supplies	$73.40
Final Product Development Materials	$70.09
3D Printing Expenses	$220.50
PCB Manufacturing Expenses	$53.69
All Categories	$417.68