Caprice: A Motion Controlled Synthesizer

Authors: Jeffrey Li, Jason Hsu, Michael Wang: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—Caprice is a three-part motion controlled synthesizer comprised of a smartphone, laptop, and VR controller. The smartphone is to be controlled by the left hand to handle pitch selection and chromatic shift while the VR controller is to be held by the right to handle note separation, octave shift, note toggle, and filter toggle. The laptop serves to handle motion-processing and audio output for the whole system and at the same time is connected to the VR controller via the Web Bluetooth API and to the smartphone via web sockets. Touch sensors in the smartphone will be polled for pitch selection, while the gyroscope in the VR controller will be utilized in classifying motion for control of effects. The touchpad sensors from the VR controller will be utilized in detecting swipes and holds for the other functions—namely, menu navigation, chromatic shifting, octave shifting, and effects toggling.**

***Index Terms*—Samsung Gear VR Controller, MIDI, Motion Controlled Synthesizer, Web Bluetooth API, BLE, Sockets, Motion Detection, React Native, IMU Motion Sensor**

1 INTRODUCTION

Caprice is a new instrument that utilizes the versatility of a VR controller to allow more expressive playing for novice musicians. Traditional instruments often require hundreds of hours of practice for players to get to a point where they are comfortable and familiar with the instrument. By utilizing natural expressive motions of the hands, Caprice allows players to skip hours of familiarizing themselves with the instrument, and serves as a good gateway into slightly more advanced instruments such as the violin or guitar.

Due to the nature of being an instrument, a major goal for development is to minimize the latency between receiving a user's input to the output of sound. Humans can only tell two sounds apart when they are more than 30ms apart. However, digital recording can have latency of up to 40ms [2]. Therefore, we aim to achieve at most 40ms latency for Caprice. In addition, we also aim to achieve 92% accuracy in classifying motion from the VR controller.

With this new instrument, many of the features we are implementing are inspired from several instruments and are combined to give the end-user a seamless experience. For the act of triggering notes, we looked to stringed instruments such as the violin in how a player would bow the instrument. With Caprice, the user would also make a bowing motion with the VR controller, but instead of placing the bow on the strings to play a note, he or she would hold the trigger button. For note selection, we looked to the guitar in how the user holds the neck of the guitar. Similar to the guitar, a user would hold the phone with their palm facing up and select notes in a similar manner a guitar player would. For the actual note selection, we drew inspiration from fretted instruments in the way that the notes are in ascending order of a scale. With eight buttons, each button would represent a note on a particular scale.

2 DESIGN REQUIREMENTS

The 40ms requirement is derived from the latency of low-end digital instruments, where latency tends to be around 20-40ms [2]. Since the player is not physically producing the sound (ie. plucking a string, blowing a note), this latency will be less noticeable. To test this, we will time different components of our code and try to minimize each separately.

First, we will want to measure the latency between an action on the Gear VR controller and response on the laptop. This latency is something we can't control but will want to see how much this will contribute to the overall latency. To test this, we measured the latency between every connection in our system. First, we measured the latency for the Gear VR controller Web Bluetooth connection to our Javascript frontend. In regards to web sockets, we measured the time between each web socket sending input and our backend socket receiving this input. The sockets that we tested for included: GearVR notification to backend, smartphone presses to backend, and backend to Tone.js.

Then we will use timer code in Python to measure the time of buffer processing for gyroscope motion detection, as well as touchpad swipe detection. Within the server, we utilized several methods to minimize processing time, such as parallel processing and several socket optimizations. Lastly, we also wanted to measure the latency between the frontend receiving a MIDI message to sound generation, but due to the asynchronicity of Javascript, we had no reliable way of capturing this data.

A 92% classification accuracy was determined through a research paper [5], where 50 distinct motions are classified with a forearm sensor, outfitted with the same sensors we are using (gyroscope, accelerometer). In this research, a maximum classification accuracy of 92% was attained. Therefore, we also aim to reach this classification accuracy, although our classification set is significantly simpler (up, down, left, right). To test this, we will measure the accuracy of swiping over 400 reps.

3 ARCHITECTURE OVERVIEW

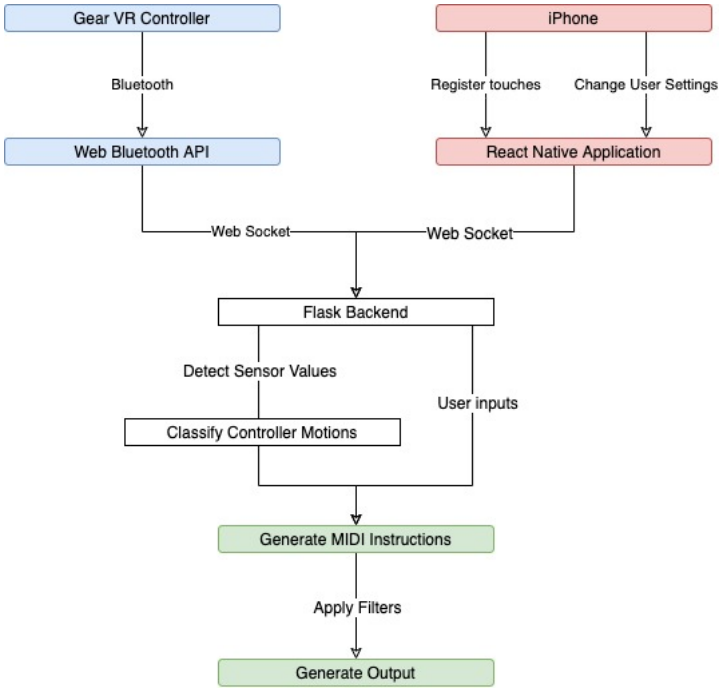


Figure 1: Overall System Diagram

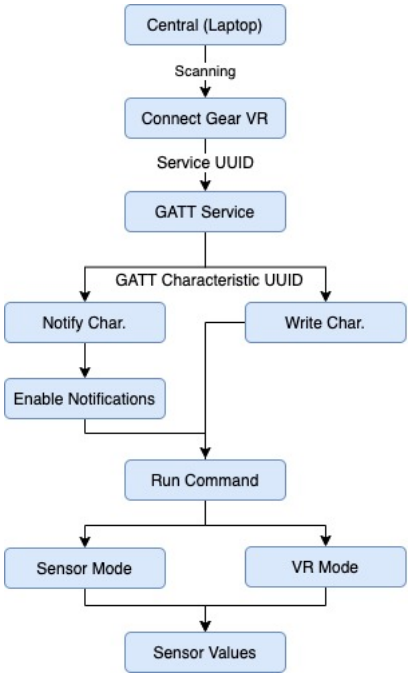


Figure 3: Bluetooth System Protocol

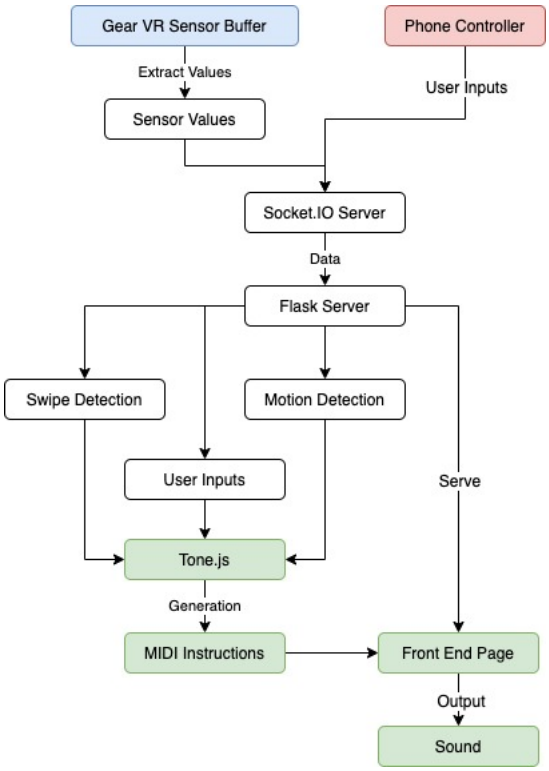


Figure 2: Laptop System Diagram

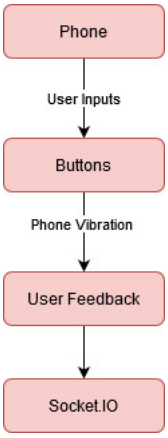


Figure 4: Smartphone System Protocol

Figure 1 describes the overall design of the system, from both controllers to sound output. Figure 2 goes into further detail of the implementation of the Flask backend, which signals feed into which modules, and the processing required in each module. Figures 3 and 4 describe which values each controller monitors, the processing performed on them, and how the values are passed into the Laptop system.

4 DESIGN TRADE STUDIES

4.1 Laptop Subsystem/Web Bluetooth

Initially, Caprice's design was to be solely contained within a phone app, which makes it very portable. After basic tests on the feasibility, we discovered that Gear VR controllers have an encrypted Bluetooth protocol that disables it from connecting with non-Gear VR bluetooth devices. A workaround, however, was found in Web Bluetooth, which is a relatively new technology that didn't exist when Gear VR was first released. We suspect that since Web Bluetooth didn't exist when Gear VR was designed, the Gear VR Bluetooth protocol does not block Web Bluetooth connections.

Since Web Bluetooth is only accessible on computer browsers, we had to pivot from a phone-centric design to a laptop-based system. There was also a consideration of how audio can be generated on a mobile device, and we concluded that we are not confident to achieve the same flexibility of sound on a phone than we could on a computer.

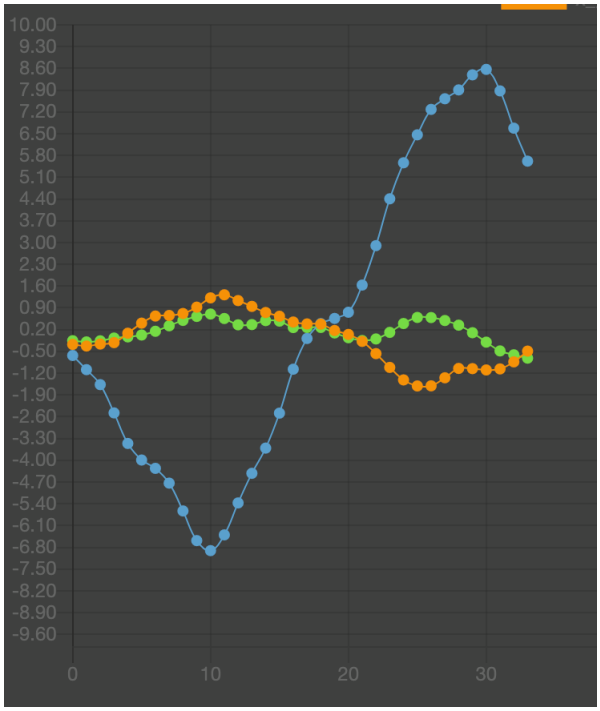


Figure 6: Gear VR Gyroscope Data

4.2 Gear VR Controller

For the right hand controller, we had to make a decision between a smartphone or a more standard handheld controller. Specifications-wise, smartphones and VR controllers both have sensors that we need (gyroscope, accelerometer, touch registering), and both devices have similar sensitivities in their sensors.

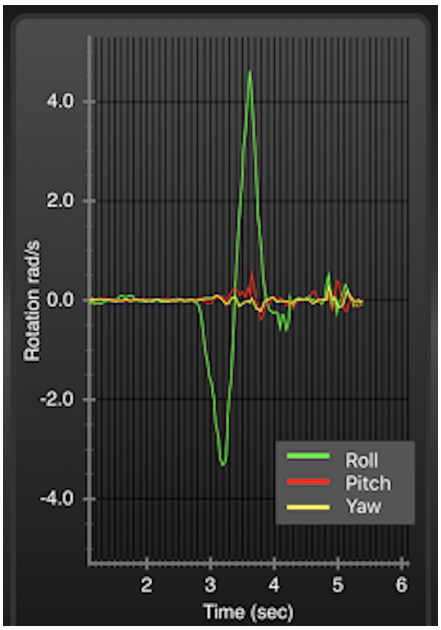


Figure 5: iPhone Gyroscope Data

After considering the performance, usability, and cost, we decided to use the Samsung Gear VR Controller. With similar performance as an iPhone, the Gear VR provides better ergonomics and price, at only $8.99 per controller. The design of the controller is much more compact and easy to hold than a smartphone, and the physical buttons provide much more tactile feedback than a touchscreen can, so a user can use it easily without having to look at the controller every time before they press a button.

4.3 Tone.js

Tone.js is the library we decided to use to convert MIDI to sound. Before *Tone.js*, we also looked at *PyAudio* and *MIDI.js* to output sound. *PyAudio* is a very low-level audio package, and was hard to interface with MIDI generation. *PyAudio* requires samples to be provided in a recording (ie. mp3, wav, etc.), which added a lot of unneeded complexity in understanding how to work with these filetypes. Audio synthesis in *PyAudio* using samples was noisy and had high latency, so we decided to use an external audio package to handle sound generation. *MIDI.js* is a package very similar in functionality, but lacks important features such as effects chains and pitchbending. Moreover, *MIDI.js* is a deprecated package that was last updated 5 years ago, and does not have widespread usage as *Tone.js*.

4.4 Peak Detection Algorithm

To determine change in motion for note separation, we need to look at the peaks and troughs of the accelerometer values. In these values, we will be able to tell when there's

a change of motion when the Gear VR controller experiences a change in direction. Since we are not particularly concerned with the acceleration values per axis, we plan on combining the three different values with an absolute value sum. This way, we only need to identify peaks (and not troughs) as the resulting graph will denote any significant motion in any direction as a peak.

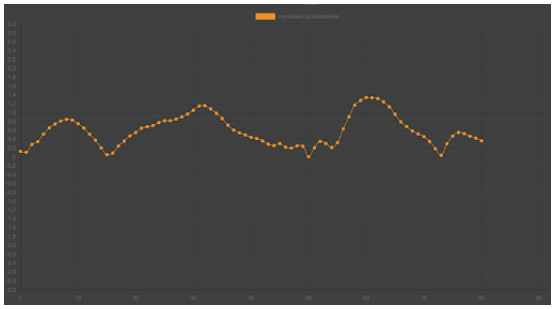
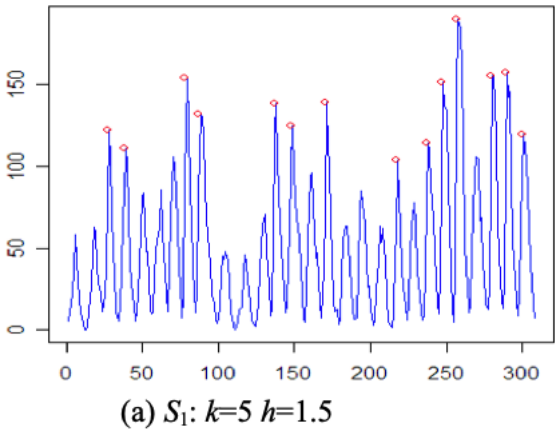


Figure 7: Absolute Value Sum of Accelerometer Values

We considered three different approaches to solving this: a ML classifier [5], a dispersion[4] based sliding window algorithm, and a sliding window algorithm with a max peak function [3].



(a) S_1: k=5 h=1.5

Figure 8: Example Peak Detection (Palshikar)

The ML approach was the first algorithm we looked into, but we quickly decided against using it due to concerns about latency and processing time. To do the ML approach, we would need to receive several sample ticks of accelerometer data before classifying the motion, which would add on a fundamental source of latency along with the latency associated with the convolutional neural network. Also, as mentioned earlier, we are not so much concerned with classifying specific gestures, but more so change in motion.

At a conceptual level, the dispersion based algorithm keeps track of a moving mean and standard deviation of a sliding window as the data points come in. If a new datapoint exceeds a certain threshold of standard deviations away from the moving mean, it can be labeled as a peak. We will test and modify different parameters in order to give us the best accuracy for detection. Some of these parameters include the size of the window we are looking at and the threshold for how far the standard deviation should be from the mean.

The final algorithm also looked promising, and had a decent success rate in determining peaks, as pictured in Figure 8. In terms of the algorithm, it performs a peak function denoted by S, which computes the average of the maximum among the distances between the input data and its k left neighbours and the same with the k right neighbors. This essentially indicates how significant this particular data point is in respect to its neighboring points. These values are then used in a similar fashion to the previous algorithm, where the mean and standard deviations peak detection is performed on the peak function's (S) results. Finally, the found peaks are filtered within the window size to remove potential consecutive peaks in a short time frame—we may choose not to perform this part of the algorithm.

4.5 Giving Up Motion Classification

As mentioned in the Design Review, we were implementing a motion classification algorithm with accelerometer values from the Gear VR controller. However, after 4 weeks of pursuing a good peak detection algorithm and still not having a reliable classifier, we decided that we should cease work on motion detection, as there was no clear solution in sight. Instead of spending more time on a subsystem that we can't guarantee to be working, we chose to move on to other parts of the project so we can still finish on time. The most significant drawback from this decision was us giving up an entire sensor (accelerometer) in the VR remote, and now we aren't using it to its full capabilities.

4.6 Making a Controller-Accessible Frontend

After stopping pursuit of motion classification, we decided that we needed some additional functionality to replace it. Aiming for improvements in user experience, we user-tested our project and discovered a significant component of usability and smoothness was affected by the fact that users had to put down their controllers to use the frontend. To fix this, we decided to reprogram our frontend and backend to make the entire system VR controller-accessible. This put a lot more work on our plate, as now we also had to go back and redo several parts of our backend, as well as needing to put a lot more thought and iteration into our frontend design.

5 SYSTEM DESCRIPTION

5.1 Phone Controller

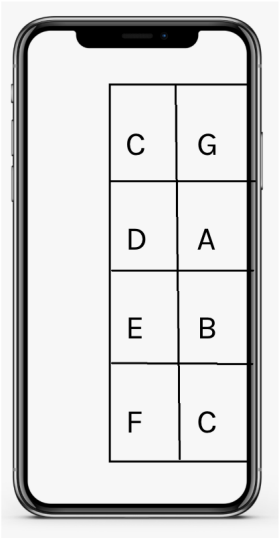


Figure 9: Note Selection (C Major)

The phone controller was built upon React Native. Initially, the phone was meant to be used as the place for the central processing of all aspects of the system. After much consideration, this plan was not used because of the lack of library support for other parts of our system. For example, with React Native, there wasn't much support for MIDI instruction generation. The biggest reason however was the inability to establish a stable connection between the controller and the Samsung Gear VR controller. As a result, we chose to use the phone only for note selection, similar in manner to how a violin or guitar player would select notes.

Using React Native, we combine the native development processes of iOS and Android with React. Our final design includes 8 buttons built in React Native that will represent the keys within a given octave on a particular scale. By default, the scale will be in C Major, but there is support to change the scale that 8 notes are in through the laptop subsystem. Multiple buttons can also be pressed at once to give the system support for polyphony.

The status of all this data will be sent in a stream to the laptop or music box. To send this data, we will utilize an external React Native web sockets library called Socket.IO. The phone will act as a client and the laptop will act as the server. After establishing this socket connection, data can be streamed from the phone to the laptop. A limitation is that the phone must be connected to the same WiFi network as the laptop. For the phone to find the laptop and connect to it, the phone must know the IP address of the laptop. On the first time connecting, the user will be prompted to enter the IP address of the laptop and be saved for future connections. Note that latency between the phone and laptop will also depend on the user's internet connectivity.

5.2 Samsung Gear VR Controller

Figure 10: Filter and Octave Selection

The Samsung VR controller incorporates many sensors into a simple package. For the sensors that we will utilize, it features a 9 DOF IMU sensor which includes a 3-axis accelerometer, gyroscope, and magnetometer. For the buttons, it features a trigger button that the index finger will press to trigger a note. It also has volume up and down, home, and back button which we will customize for our own liking. Additionally, it has a touch pad that has a resolution of 100x100 and can be used for custom features we will implement such as swipe recognition and filter selection.

To connect to this controller, we did quite a bit of research and found previous work done on this exact problem we had. The main problem was that Samsung didn't have any official API to connect to this remote so we had to find a way to reverse engineer the connection process. The most success we found was from the work of jsyang [1], where he was able to successfully connect to the controller from a web application and use sensor fusion to display the controller in 3D space.

The Samsung controller is known as a BLE (Bluetooth Low Energy) device and to connect and read from such devices, we need to utilize the GATT profile of the device. In general, all BLE client devices connect to a host device with the built in GATT profile on the host. Within a profile, GATT characteristics are grouped together into a list of GATT services based on its utility. An example GATT characteristic could be the current temperature reading, and it could fall under a sensor data service. Each service consists of the ability to read, write, and subscribe to a characteristic in order to receive notifications when values change for that service. To see the services and characteristics available on any given device, we used the Light Blue mobile application (or any BLE scanning application). Using this, we were able to find the hex values of the service/characteristics pertaining to the sensor data. We were able to confirm these values with the service values that jsyang used for his reverse engineering experiment.

Now, to connect to the remote, we now have the service we need to subscribe to and listen for notifications on. To begin receiving data, we need to write two specific commands: VR Mode enable and Sensor Mode. These commands are represented as a two byte array and for these two modes specifically, the byte arrays are 0x80 and 0x10

respectively. Once we begin receiving data, the sensor values are encoded into a 32 byte array buffer and must be decoded on each notification. For example, for the accelerometer's X, Y, Z values, they can be found at the 3rd, 4th and 5th byte in the buffer. We were able to find out the map of sensor values to buffer indices from the work jsyang did with decompiling the Gear VR Input APK. By doing this, he was able to view Samsung's APK source code and determine how they set up the buffer before sending it.

After several tests, we saw that on a moving average over 10 minutes, notifications are received from the remote at 68.6 notifications per second, which is approximately one notification every 0.015 seconds. With this rate of data input, we decided to use websockets to stream this data to the Flask backend.

5.3 Laptop (Music Box) Subsystem

With the laptop, we want it to serve as the central location for data flow and processing. A Python Flask server will serve as the central location for hosting and receiving data. We chose to run our processing through a server since it offered the ability to host the web application that would receive data from the Gear VR remote. On the Flask server, sockets are enabled by using the Flask-SocketIO package. This package essentially wraps a normal flask web server with a sockets server that can accept bi-directional communications between clients and server.

Once the server is running, the pages being served by this web server can act as clients and send messages to the server. In our case, when notifications are received from the remote, the data buffers are immediately sent as messages to the socket server. With the user's phone controller, it also acts as a client. For the laptop to connect to the phone and begin receiving messages, first they must be connected to the same network. Then, when the system on the laptop first is started, a prompt will display the laptop's IP address for the user to input on the phone.

Once we receive the extracted Gear VR sensor values, we will perform tap and swipe detection, as well as process gyroscopic data, and determine appropriate actions based on this. To do this, we used a linear classifier to classify directions of swipes, and simple thresholds to classify taps in different areas of the touchpad. The gyroscope on the VR controller outputs angular acceleration, which is hard to incorporate with effects and other functionalities. We transform angular acceleration to angular velocity by performing a simple integration with our data values by maintaining a running sum on a fixed window of the time-series gyroscope data. The angular velocities per axis are then transmitted to the frontend, where it will perform different functions based on which effect it is applied to.

With the processed data, the Flask backend will send simple commands such as "play note 40", "enable effect x with parameters a, b, c", through a socket to Tone.js running on the Javascript frontend. Tone.js will then generate an initial waveform, transform it through a configurable effects chain, and output the audio through a speaker.

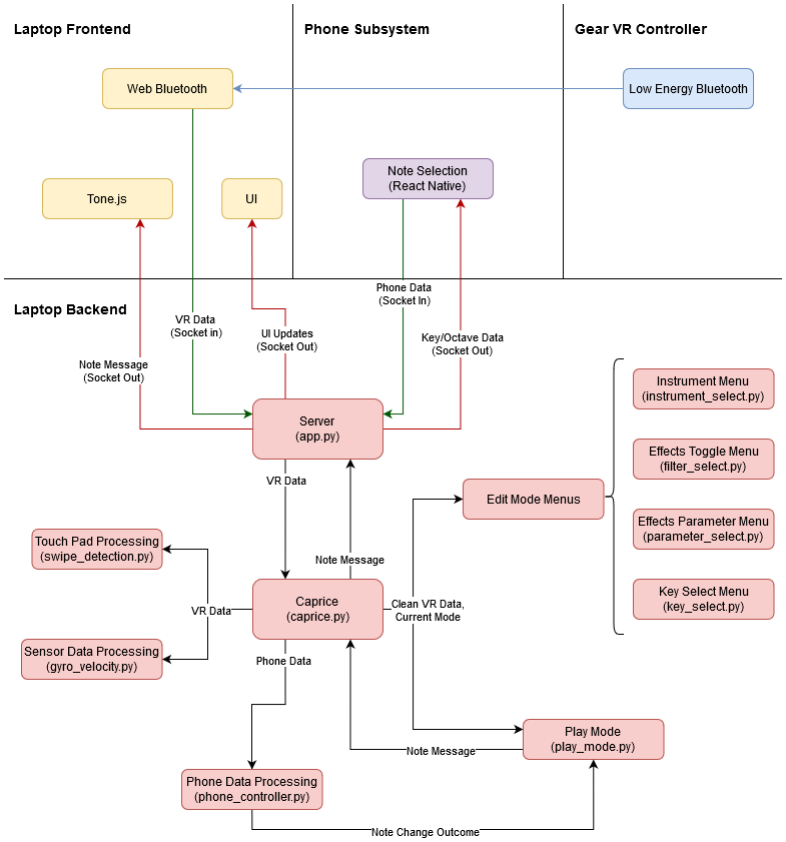


Figure 11: Laptop Subsystem, Detailed

After our decision to implement a controller-accessible frontend, we had to rethink and redesign our existing backend. A good way we found to partition and abstract this was to break down our frontend and backend into states. For example, there are two main modes of operation for *Caprice*: *Play* and *Edit* mode. These states are maintained and synced from frontend to backend. Play mode is the mode users should be in while playing, since it maps all of our VR controller inputs to playing functionalities (filter selection, octave shifting, chromatic shifting, note separation). Edit mode is where users can customize their settings for different effects, change their mappings in filter selection, change their current instrument, as well as change the key and mode of the phone controller.

Edit mode is then broken down further into smaller substates, one for each menu the user could be in while in edit mode. Supported menus include: instrument selection, filter selection, filter parameterization, key/mode selection. In each of these substates, VR controller inputs would do something slightly different. For example, in instrument select, swiping in any direction would move the cursor to another instrument, but in parameter select, swiping up/down will change the current selected parameter for the current effect, swiping right/left will iterate through possible values for the current parameter, and tapping in any direction will move to cursor to another effect.

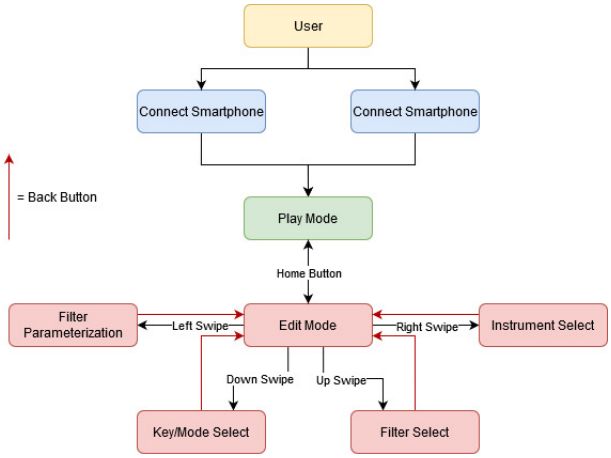


Figure 12: User Flow Diagram, Play/Edit

5.4 Custom Grip for Smartphone

The custom grip for the smartphone is meant to improve the ergonomic feel and hold to the left hand controller while diminishing the risk of dropping the smartphone during play. We envision the primary part of the grip to be a reduced bookend-type surface, such as one piece rests in the left palm while the other supports the smartphone as pictured below. The part in contact with the palm will be secured via a Velcro strap that runs across the user's knuckles, while their fingers remain free to play on the smartphone screen. As for connecting the smartphone to the flat surface of the bookend, we also plan on using Velcro straps on the back of the smartphone case and on the surface of the bookend.

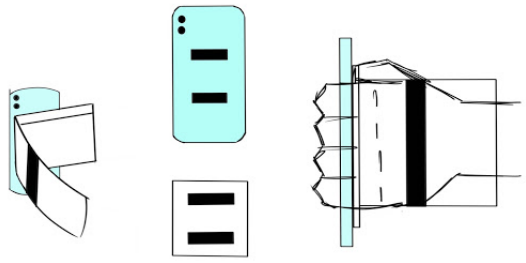


Figure 13: Side View, Interface Between Phone and Bookend, and Bottom View of Custom Grip

Note: The fabrication of the custom grip was not pursued due to the loss of access to facilities post-lockdown.

6 PROJECT MANAGEMENT

6.1 Schedule

A major roadblock we underestimated was how long it took to successfully reverse-engineer the GearVR controller to start reading sensor values from it. As a result, most development on the controller side was pushed back by 3 weeks. However, development of the audio generation remained on schedule. We also had an unanticipated shift to centering the user experience on the laptop side. As a result, we dedicated more of our time later in the project to developing the front end systems this experience.

6.2 Team Member Responsibilities

The division of labor mostly adheres to the original plan. Michael is doing his work mostly on the controller and backend, including motion classification and the reverse engineering of the controller. Jason worked on the generation of MIDI messages and output of sound/sound generation. Jeffrey worked on frontend for the music box (laptop), reverse engineering Bluetooth protocols, and the phone application.

6.3 Budget

Since this project is mostly software-based, most of the budget is spent on creating the grip for the phone. For a more detailed bill of materials, refer to Fig. 10 in Appendix A.

6.4 Risk Management

One risk we had to handle was how difficult it was to generate audio from scratch. We realized it would be near impossible for us to implement an audio system that can achieve polyphony and still minimize latency in the time we had. Therefore, we decided to fall back on using a pre-existing audio library (Tone.js) to generate audio for us; allowing us to focus on more features, better motion classification, and designing a better user experience.

Another risk we have to consider was the latency of the system in general including the Bluetooth and socket aspects of the project. This was a risk for our system because having too much latency may be noticeable for the end user and make it a less desirable experience. We decided that making the system wireless was a huge plus because it made the experience more enjoyable and easily accessible as well. Ultimately, we were able to measure the latency of all subsystems and found that Bluetooth actually wasn't the bottle neck in terms of latency. In fact, the largest culprit was WiFi and speed of the socket between phone and laptop subsystem.

7 RELATED WORK

The basis for this project was inspired by *AUUG Motion Synth*, a Shark Tank Australia product. The *AUUG* can act as a MIDI controller, DJ controller, as well as an instrument by itself. However, the instrument subsystem is relatively underdeveloped and hard to use in a performance setting than its other modes. The *AUUG* is also only a one-handed device, where its functionality is best

utilized with another device, such as a digital instrument or DJ software.

AUUG's primary focus is on interpreting the user's motions in a preexisting musical context (ie. the notes are determined somewhere else), and Caprice's focus is on creating a new musical instrument that the user can play whatever the notes they want.

8 SUMMARY

Overall, the goal with Caprice is to create a motion controlled synthesizer using three components: A smartphone, laptop, and VR controller. We hope to general sounds based on user inputs and motions to create a new experience for musicians of any skill level. With a smooth frontend that has been implemented with respect to the VR controller, users can navigate a slew of settings effortlessly and customize their own sound and playing.

References

[1] *Gear VR Controller Reverse Engineering*. URL: https://jsyang.ca/hacks/gear-vr-rev-eng/.

[2] *Monitoring Latency (How Low Can You Go?)* URL: https://ask.audio/articles/monitoring-latency-how-low-can-you-go.

[3] Girith Palshikar. "Simple Algorithms for Peak Detection in Time-Series". In: (2014).

[4] *Statistical Dispersion*. URL: https://en.wikipedia.org/wiki/Statistical_dispersion.

[5] Terry Um. "Exercise motion classification from large-scale wearable sensor data using convolutional neural networks". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017), pp. 2385 –2390.

Appendix A

Item	Quantity	Price Per Unit	Cost	Purpose	Supplier
Energizer AAA Batteries (24 pack)	1	13.58	13.58	Batteries for controllers	Amazon
Strenco Velcro Tape	1	13.52	13.52	Strap for phone controller	Amazon

Figure 14: Bill of Materials