Mobile Steering Wheel

Xiao Jin, Yuxuan Zhu, Qiaoan Shen

Department of Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—Our design is a wireless video game controller system capable of detecting gyroscopic input, dedicated to function with BeamNG.Drive. For driving simulation games such as BeamNG.Drive, having a steering wheel controller helps tremendously with the level of immersion users can get. However, such steering wheel controllers tend to cost more than 10 times the cost of a generic gaming controller. Our wireless steering wheel controller is designed to expand on the abilities of generic controllers, by adding 360 degree angle sensing to simulate turning a steering wheel, a 5 inch LCD display for vehicle information, and a 3D printed enclosure that resembles real steering wheels in race cars. The whole package weighs around 400 grams, around the average of what smart phones weigh nowadays.**

***Index Terms*—Design, controller, driving, gaming, racing simulator, steering wheel, wireless, gyroscope, accelerometer, Raspberry Pi, HDMI, LCD, BeamNG, PC**

I. Introduction

Simulated Racing, or simply Sim Racing, is a form of entertainment involving the simulation of driving or racing cars on computer games. Over the years, enthusiasts have evolved from using keyboard control, to dedicated game controllers, and now to full custom-made cockpits with 1:1 replication of real cockpits. Such sim racing cockpits often cost more than a few thousand USD, making it an extremely high bar of entry, and for this reason sim racing has been a very niche hobby. However, during the COVID-19 lockdown, an influx of newcomers surprised the sim racing community. As active members of this community, we were thrilled to see so many people interested in joining sim racing. It was unfortunate, however, for enthusiasts to learn that many people were turned away by the high cost of controller devices such as steering wheels and pedal sets. There simply isn't a controller on the market that combines the affordability of generic game controllers, and the immersive experience from an enthusiast-grade custom sim racing cockpit.

There is a void in the market for a cheap sim racing controller with immersive gaming experience, and our project is going to fill this void. As users of sim racing controllers ourselves, we brainstormed some use cases for such a product. The product will be used to control one of the most popular sim racing games, called BeamNG.Drive. It will resemble a steering wheel in a Formula One race car to introduce some level of immersion. It must be portable, in the sense that users can just stuff it in their backpack and take it on a trip. This means that such a product will not have force-feedback functionality, one of the reasons why some steering wheel controllers are very expensive. For people who are just trying out sim racing, however, force-feedback is not a necessity. This means that our product can be a free-floating, steering wheel shaped controller. Accurate steering input, on the other hand, is vital to sim racing. Traditionally when casually playing racing games, people simply use the arrow keys on keyboards to control acceleration, braking and turning. In order to achieve a high level of immersion in sim racing, controllers need to have analog acceleration, braking and steering inputs. Our product will have analog steering angle sensing capabilities, to simulate the effect of turning a steering wheel, even though there is no fixed axle the wheel should be attached to (steering column). Our controller will also support wireless connection to PCs, making it an ultimate package for maximizing portability.

Although portability is one of the biggest features of our controller, it is still important to retain some level of immersion. After all, it was the "simulated" aspect of sim racing that attracted people who used to play arcade style racing games. Compared with the XBOX Wireless Controller, which has a joystick for analog steering input, our product adds a more intuitive way of sensing the degree of tilt, making users' hand movement similar to turning the steering wheel in a real car. Our controller will also have a small form factor LCD display facing the user, so the user can choose to display some vital information about the car they are driving, just like how a real car's instrument cluster works. Compared with a more hardcore steering wheel controller such as the Thrustmaster T300RS, a very popular steering wheel controller setup amongst enthusiasts, our controller does not have a heavy base that needs to be clamped onto a desk, and that means unfortunately the force-feedback function is gone. Functionally, our product is very similar to how a smartphone racing game is played. On smartphones with Android or iOS operating systems, there are many racing games (Asphalt 9 for example) that use the phone's accelerometer to allow the user to control the car by turning the smartphone. On top of the smartphone experience, we are adding analog accelerator and brake control to raise the level of realness.

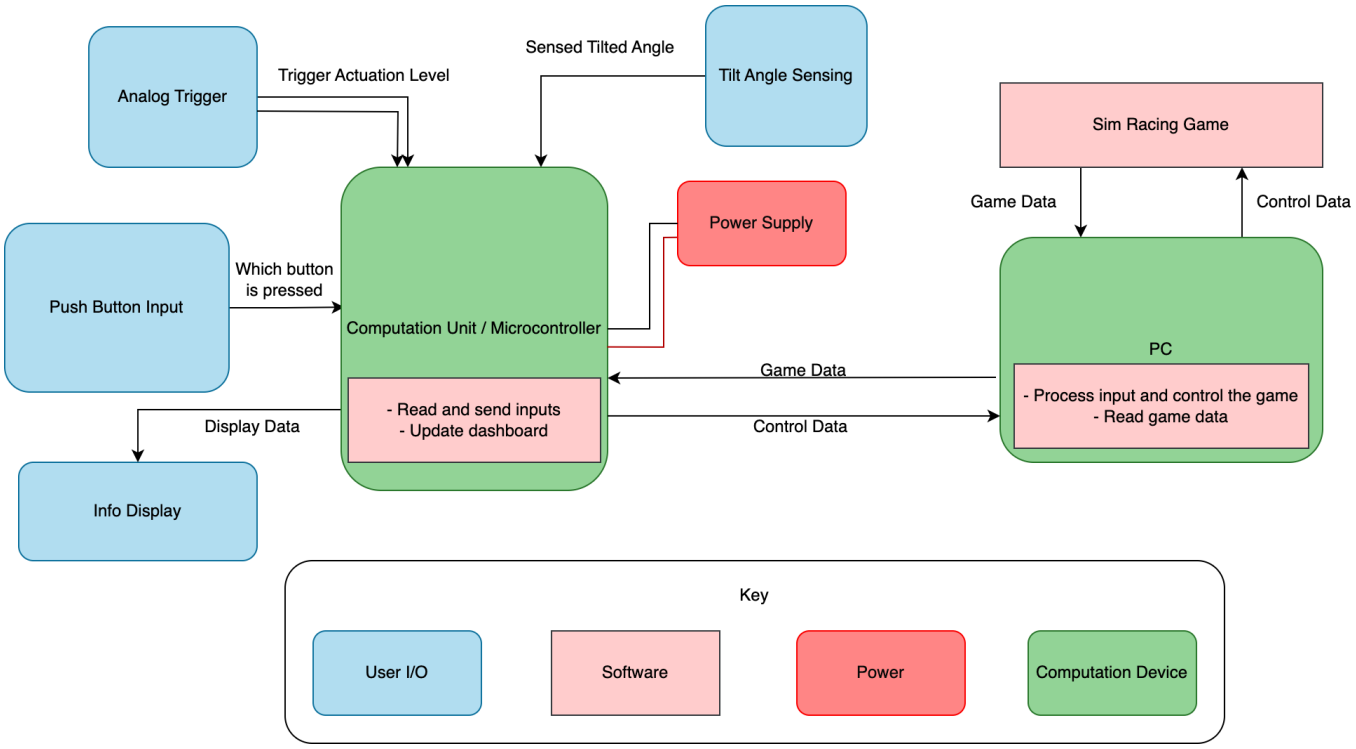


fig1. System Architecture Diagram

Overall, our controller eliminates some high-end features of existing steering wheel controllers, while retaining the core functionalities without breaking the bank, in a portable package that users can carry around in a backpack. The product borrows inspiration from racing games on smartphones. The goal is to significantly lower the level of commitment, both in terms of money and space, required to get started with sim racing, and to attract more people into the wonderful world of sim racing.

II. Use-Case Requirements

1) The controller should be able to detect the controller tilt angle at least within +180 degrees and -180 degrees from the horizontal position when users rotate the controller. This requirement stems from analysis of multiple steering systems, such as Formula 1 race cars and GT3 race cars. In a video showing the onboard camera of an F1 car [1], it can be observed that the steering angle never exceeded + or - 180 degrees from center. The same can be observed in a video showing a GT3 race car's driver's view [2]. For the competitive nature of the racing games our users will be playing, this requirement seems reasonable and well-justified.

2) The controller should be able to last for 8 hours in order to create a smooth gameplay without having users to charge frequently. From personal experience, the longest session we have played any racing game is below or just around 4 hours. We would like users to be able to play multiple sessions without having to charge the device. For example, if the user forgot to charge the controller after they played, then the next day they wouldn't miss a gaming session, say, with their best friend.

3) The controller should have remappable digital buttons, an accelerator input, and a brake input. The buttons on the controller can be programmed to simulate various functionalities on a real steering wheel. The brake and accelerator input should have a range of values, similar to real car gas and brake pedals.

4) There should not be perceivable input lag from the moment a user makes an input, to the moment the input is registered in the game. Controllers such as PS5 and XBOX controllers have no perceivable input lag. Their input lag metric is below 20ms [3], and it is reasonable to set our requirement as 20ms.

5) The weight of the controller should be less than 400g. Similar devices that users hold in their hands for extended periods of time, such as smartphones, usually weigh less than 400g, ranging from half of that to around 75% of that [4]. We tested with multiple smartphones, and discovered that a controller that weighs more than 400g will cause strain on the user's arms and hands as they hold it for a long time in the air. Many popular game controllers also weigh less than 400g.

6) The user should have a dashboard or a display that can show some vehicle related information. While playing in a competitive setting, ideally more of the PC's screen should be used to display the car and the environment the user is driving in. Having a display on the controller allows some information to be transported from the PC screen to the controller, freeing up valuable screen real-estate.

7) When the controller is parallel to the ground, the car

should move in a straight line. When the controller rotates to a certain angle, the steering wheel inside the game should rotate to another specific angle. The relationship between the tilt angle detected and the angle of the steering in the game should follow a predetermined equation. This is to simulate the effect of having a steering rack and a steering ratio in a real car. When the steering wheel turns X degrees in the car, the front wheels do not turn X degrees. Instead, the front wheels turn k*X degrees, where k is a constant determined by the mechanical properties, and k is usually a positive number smaller than 1.

8) When operating the accelerator or brake, the car is expected to accelerate or brake. The percentage of accelerator or brake being actuated is directly mapped to the percentage of the input the user is giving. This is the same principle as a gas or brake pedal in a real car, which ranges from being free to being fully depressed.

III. Architecture and/or Principle of Operation

Our controller will consist of several sensors for getting user input, an LCD screen for displaying key data from the game to the user, a computation device to translate sensor data to game controller output that is readable by a computer program, some wireless functionality for the communication between the PC and the controller, and a battery system for cordless power. (See Fig. 1)

For the hardware side, our controller will be getting user input from a range of sensors, including push buttons, analog triggers and most importantly for us, the tilt sensor. When the user pushes a button on the steering wheel, the computation unit will recognize this event as a button push. When the user depresses the analog triggers to a certain percent of travel, the computation unit will record that amount of actuation. When the user tilts the steering wheel to a certain degree to simulate steering in a real car, the tilt sensor will recognize the degree of tilt, and pass that data to the computation unit. The computation unit will also be responsible for deciding the content for display on the LCD screen. A battery system will be responsible for providing power to the sensors, the screen and the computation unit.

For the software side, a program will run on the controller to read data from buttons and sensors and run adjustment methods on the data to reduce inaccuracies. The program will establish a remote connection with the PC that runs the BeamNG.Drive game in order to send all the data from the controller to the PC. Also, the program on the controller should be able to read real time car information received from the PC so the controller can display data such as RPM, speed, and gear on the LCD screen.

The program on the PC side will also establish a remote connection to the program running on the controller so that it can receive the button and sensor information. After receiving the raw information, the program on the PC is responsible for translating the raw button information into an input format such that the game can understand. The translated inputs will move the car in the game. In addition, the program on the PC will aggregate game data from the game and send them remotely back to the controller to display the car information on the LCD display.

The software system architecture for the final implementation does not change much from the architecture in the design report. Though implementation details changed, the overall logic of data flow and communication between controller and the PC remains the same.

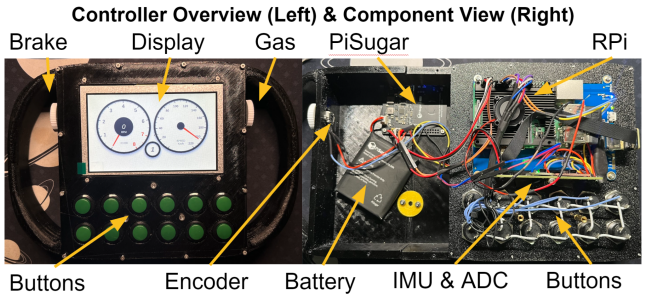


fig2. Overall system.

IV. Design Requirements

Based on the user-case requirements, we will determine the design requirements as stated below.

1) We should use a sensor capable of sensing changes in its posture (specifically angle). It should at least have a range of sensing of 360 degrees (+ and - 180). The angle sensing system also needs to consider the effect of the shaking of the user's hands. It should filter out small bursts of changes within a certain range, so that accidental shaking of the controller does not register as steering input. The angle sensing unit also should provide a stable output when the controller stays still, i.e. there should be no drift. The angle sensing unit should also be accurate, i.e. the actual angle of the controller and the sensed angle should be within a margin of error. After researching some IMUs, we think that + or - 1 degree error margin should be adequate.

2) The controller should have around 12 push buttons for digital inputs, and 2 analog inputs. This is the same number and types of inputs as an XBOX controller [5]. This should provide the user with enough inputs to properly control the game.

3) The controller should have an input latency no more than 20ms. In this case, the input latency represents the time it takes for the game to react once a button is pressed. Since 20ms is a time that's negligible for users to feel the latency, we need to find our method to transmit data within this time range. We need a fast and well-supported communication standard, and Bluetooth 5.0 is our choice based on the requirements above [6].

4) For the display/dashboard, we investigated the sizes of the displays on the steering wheels of Formula 1 race cars. They range from 5 to 9 inches in diagonal. We chose a 5 inch HDMI display. From personal experience of using

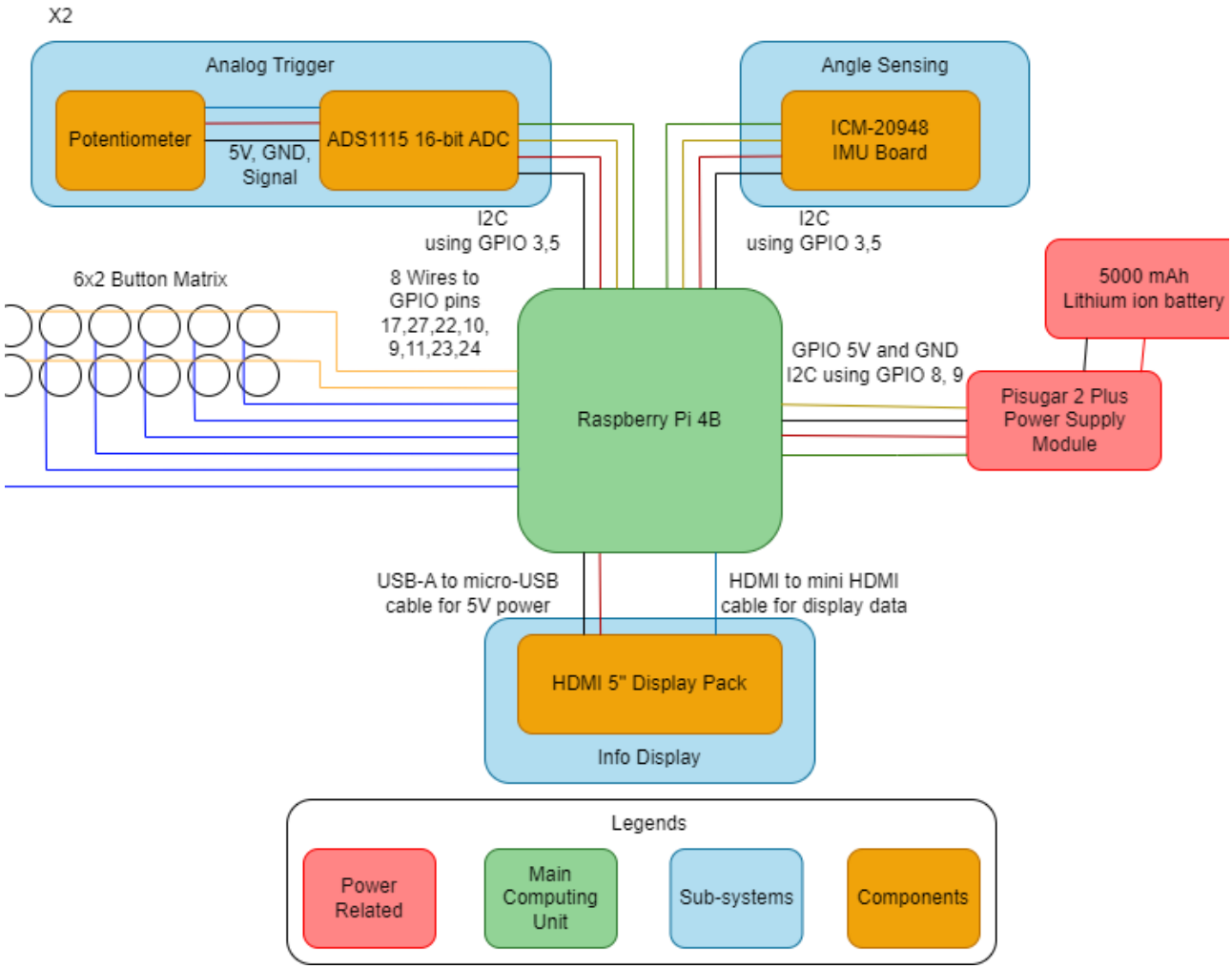


fig3. Hardware System Block Diagram

digital dashboards (screens) when playing sim racing games, the 5 inch size is more than adequate to display information such as engine speed, car speed, lap times, etc. Also, the HDMI interface makes communication with the display easy, since a lot of devices support some sort of HDMI.

5) Based on the requirements for user inputs, communication protocol, and the dashboard/display, we determine the requirements for our main processing unit. To wire 12 buttons, they require a minimum of 8 wires (buttons wired into a 4x4 matrix, while 2 are unused). The HDMI display needs an HDMI-capable computer to interface with it. There needs to be support for either I2C or SPI, since most embedded sensors use them for communication. The processing unit also needs to have support for analog input. Also, ease of coding should be taken into consideration.

6) To support an 8-hour battery life, we investigated the power consumption of a typical single board computer used in embedded scenarios, the Raspberry Pi 4B. It has a steady current drain of around 500mA [7]. A typical HDMI display, with its backlight on full, consumes about 500mA [8]. If the backlight is turned half way on, then it only consumes 370mA. The power consumption can be calculated by the formula $Capacity = I * Battery\ Life$ and here we calculate the data as $870mA * 8h = 6960mAH$. To satisfy the 8-hour battery life of our product, we determine that we need to use a battery no smaller than 6960mAH.

7) The controller enclosure design should follow the requirement that it should provide cutouts for 12 buttons, 2 analog inputs and a screen. To satisfy the 400g weight requirement, the 3D printed enclosure should be printed using composites, and designed in clever ways to minimize waste.

Subsystem	Use Case	Design Requirements
Enclosure	Digital buttons, 2 pedals, 400g limit	12 push buttons, 2 analog inputs, 3D print with composites
Sensing	360 degree rotation; Gas & Brake pedals; 180-degree change within 1	Angle sensing accuracy within ±1 degree; drift within ±1 degree in 60 seconds

	second.	
Software	20ms input latency	Bluetooth
Hardware	8 hour battery life	> 6960mAH battery
Hardware	-	Processing unit with I2C or SPI, Bluetooth, HDMI

V. Design Trade Studies

A. *Angle sensing: choice of IMUs*

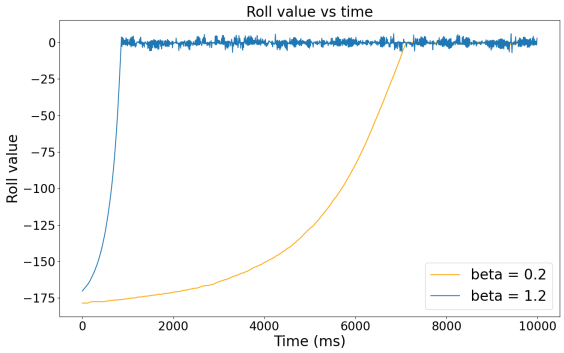
To satisfy the design requirement for sensing changes in angles, we need an IMU. IMU stands for Inertial Measurement Unit. They sense and report (changes in) an object's posture using a combination of accelerometers, gyroscopes, and sometimes magnetometers. We searched for several IMUs, and also got suggestions from Prof. Mukherjee about which IMUs to avoid. We compared the InvenSense ICM-20948 and Hillcrest Labs BNO085. They both provide 9-dof sensing, I2C interface, and are both in small packages, and they offer similar levels of performance. What eventually prompted us to choose ICM-20948 was that it claims to be "the world's lowest power 9-axis MotionTracking device" [9]. On the other hand, BNO085 offers fancy features that we do not need, such as optimizations for AR/VR [10]. Furthermore, SparkFun's ICM-20948 unit was cheaper than Adafruit's BNO085 at the time of purchase, and we want to ensure our design is much cheaper than those expensive racing simulators.

B. *Programming Language: Python over C*

C programming language is known for its speed which can contribute to our input latency design requirement to be less than 20ms. However, our implementation still chooses Python over C. The reason is that Python is simple to write and has many libraries that can be imported. Python also takes less time for debugging since the programming language takes care of memory allocation and garbage collection automatically. Since we only have around 7 weeks to implement after submitting the design report, we believe that Python can better help us meet our goals on time.

C. *Gyroscope learning rate: large over small*

The Madgwick filter uses gradient descent to update its estimation of the direction of the steering wheel based on accelerometer and magnetometer readings. The gradient descent algorithm approaches the target value with step size determined by the learning rate. A large learning rate allows the system to reach the vicinity of the target faster, but it will have overshoot problems. A smaller learning rate is more likely to be closer to the target value, but it will take longer time to reach the target. A comparison is shown in the picture below for what happened after we rotate the steering wheel for 180 degrees:



From the given picture, we can clearly see that when β=1.2, the algorithm responds to the change within 0.6 seconds, although it exhibits noticeable oscillation around the target. In contrast, when β=0.2, the algorithm completes the change after 3 seconds. Our use case prioritizes a quick response to rapid changes, so we opt for a larger learning rate. The oscillation can be filtered out during subsequent processing.

D. *User Interface: screen, buttons and encoders*

The choice of the HDMI screen was fairly simple. We searched on Adafruit and a number of screens of different sizes came up. We settled on 5 inches as the size, explained in the previous sections. We did have to choose between a touch screen version and a non-touch version. We eventually chose the non-touch version since we did not plan to implement touch screen UI, and the extra power consumption and complexity is not something we wanted to deal with. For push buttons, a similar strategy is applied. We chose 16mm momentary push buttons because, first of all, the size is large enough for human fingers to comfortably press; and secondly the nature of the pushing-releasing actions on a gaming controller dictates that we needed a momentary push button. For the analog inputs, we referred to a simple voltage divider circuit, with a potentiometer wired in to provide a varying voltage at the output end. A generic 3-pin potentiometer was chosen from Adafruit for its cheap pricing and ability to mount a knob.

E. *Hardware: choice of processing unit*

To handle the I/O needed for buttons, encoders and the screen, a main processing unit must be chosen. We debated between using a Raspberry Pi and an Arduino Uno. They are of similar sizes, so packaging is not a concern. The Arduino runs on C, and is faster than if we were to run Python on a RPi. However, since we are using Bluetooth and HDMI, and the RPi offers onboard support while the Arduino does not, we chose the RPi. This does bring a trade-off though. Since RPi does not have an onboard Analog to Digital Converter, our analog input circuit has to

be connected to an external ADC, then wired to the RPi using I2C or SPI. The higher energy consumption of the RPi compared to the Arduino Uno is also a drawback.

VI. System Implementation

A. Hardware Systems

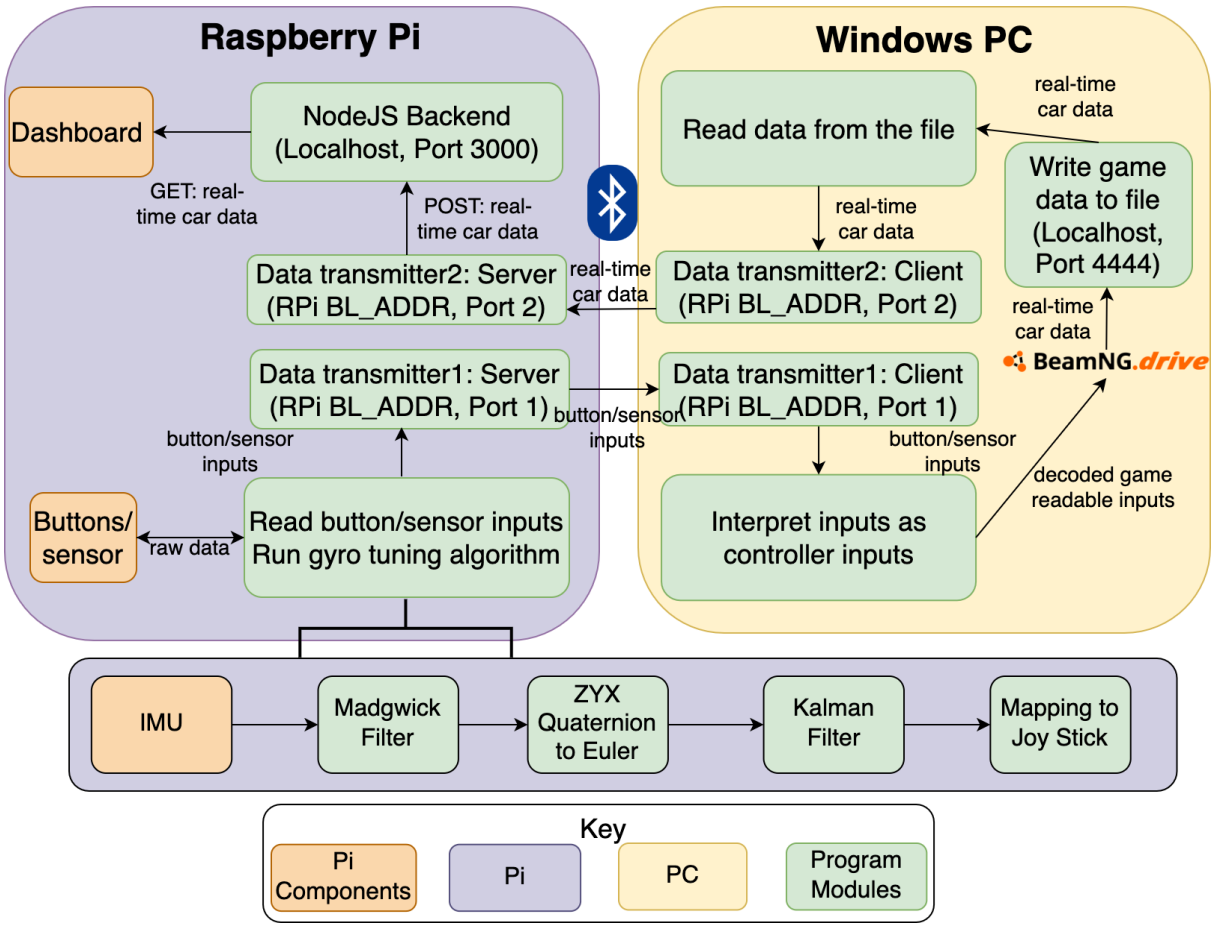
The Raspberry Pi 4B runs Raspberry Pi OS, and provides a platform for Python coding. The I2C function is enabled from firmware, and one I2C bus is readily available for communication. Originally, the IMU, the ADC and the PiSugar Power Management module all connected to the same I2C bus. From what we have learned, this shouldn't pose any problems. However, due to the way PiSugar is designed, it constantly occupies too much bandwidth of the single I2C bus, and was causing the IMU and ADC to intermittently drop connection. Fortunately, the RPi 4B has hidden I2C buses that are not activated. After using separate I2C buses (at the expense of 2 GPIO pins), the problem was solved. SparkFun and Adafruit both provide Python libraries for easy communication with their devices over I2C.

The HDMI display took very long to set up. Since it was a cheap unit, Adafruit states that it does not have a video scaler built-in. This means that when plugged into the RPi, the image was not full screen. Also, when first plugged in, the screen did not display anything, unlike many other HDMI displays, which are assumed to be plug-and-play. Eventually a lot of firmware tweaking had to be done for the RPi to correctly recognize the display.

The 12 push buttons were wired in a matrix configuration. The easiest way to wire buttons is to simply use one common GND, and one GPIO per button. However, since we do not have the luxury of a million GPIO pins, we chose a matrix layout. The 12 buttons are organized in a 2x6 matrix. Each row has one common wire, and each column does too. In software, each row is set to high, while a loop checks the input level on the column pins. If a button is depressed, the corresponding row and column pins will be connected. The software then figures out which buttons are depressed.

Setting up the potentiometers was very easy. The potentiometer has 2 pins, one for VCC, one for GND and another for output. VCC and GND were connected to 5V and GND on the RPi respectively, and output was connected to one of the input channels on the ADC. The ADC utilizes a Python library to interface through I2C, and the voltage value at the input channels can be read in Python.

Power delivery to the RPi is handled by the PiSugar Power Management HAT. It supports I2C and has a Python



library for checking battery charge level. Out of the box the PiSugar is bolted onto the back side of the RPi, making contact with all the power related GPIO pins. However, during final packaging, we found out that the height of the RPi combined with PiSugar was too large. We switched to mounting them separately, and using wires to connect them together. Originally we only connected one 5V and one GND wire. The RPi kept showing a low voltage warning, but the measured voltage at the 5V terminal was over 5V. It turned out that regardless of whether it's low on voltage or current, the RPi will always signal a low voltage warning. The single 5V and GND wires did not deliver enough current to the RPi. The problem got solved when we wired all of the RPi's power related GPIO pins to the PiSugar.

The last piece of the hardware puzzle is the enclosure design. We borrowed inspirations from GT3 and Formula 1 racing steering wheels, and went for a rectangular shape. The user would hold onto the 9 and 3 o'clock positions and never let go while using the steering wheel. Since the enclosure was going to be 3D printed, we weren't very confident with the strength PETG material can deliver. During design, margins were left fairly large to allow for any errors when assembling, and to avoid potential failure points. The screen and buttons were mounted on the top

face, while the rest of the hardware components were mounted inside the base enclosure. The top face screws onto the base enclosure for a full package.

B. Software Systems: Gyroscope Tuning

The goal of the gyroscope tuning algorithm is to accurately determine the steering wheel's orientation and filter out noise generated during the measurement and orientation evaluation process.

The algorithm receives data from the IMU-20948 board installed on our steering wheel, with the roll axis of the IMU representing the axis we want to measure for the steering wheel's rotation. The IMU provides angular rate data from the gyroscope, indicating the rotation speed of the device; acceleration data from the accelerometer, which measures linear acceleration along the x, y, and z axes due to motion and gravity; and magnetic data from the magnetometer, which captures the strength of the magnetic field along the x, y, and z axes of the sensor.

The gyroscope tuning algorithm employs a Madgwick filter [11] to determine the orientation of the steering wheel. The Madgwick filter is a popular sensor fusion algorithm used for determining the orientation of objects in 3D rotations by combining data from accelerometers, gyroscopes, and magnetometers. The algorithm transforms and combines the data from these sensors into quaternions to calculate the 3D rotation. It uses accelerometer and magnetometer data to apply gradient descent, minimizing the error between the predicted orientation based on the current quaternion and the readings from these two sensors. Then the algorithm fuses the step from the gradient descent with gyroscope data to compensate for gyroscope drift, ultimately integrating the combined quaternion and returning the normalized quaternion output. The Madgwick filter is a complex algorithm, so we decided to use functions from the madgwick_py library by the Cognitive Systems Lab (CSL) of the Karlsruhe Institute of Technology [12]. To better tune the Madgwick filter here, we need to carefully decide the learning rate for the gradient descent here. We choose beta = 0.8 as its learning rate, as this allows the Madgwick filter to give quick response to rapid rotations, while limiting the oscillation of the gradient descent results in a reasonable range. We will describe the reason behind in detail in the design trade-off section.

The gyroscope tuning algorithm converts normalized quaternion values to Euler angles in the ZYX (yaw, pitch, and roll) sequence. The design here aims to address the gimbal lock problem of Euler angles, which occurs when two rotational axes become aligned, leading to a loss of one degree of freedom in the system. In our design's scenario, the gimbal lock problem arises when the pitch angle approaches 90 degrees, which means we place the steering wheel's screen parallel to the ground, if we use the commonly used XYZ (roll, pitch, yaw) sequence quaternion-to-Euler conversion, which cannot generate meaningful roll axis rotation readings in this case. By adopting the ZYX sequence, the algorithm mitigates the influence of gimbal lock and provides reliable orientation when the steering wheel's screen is near parallel to the ground. This is achieved by changing the sequence of rotations performed along the axes, which, in turn, alters the scenario in which gimbal lock occurs.

The gyroscope tuning algorithm employs a Kalman filter to mitigate measurement and processing noises resulting from the Madgwick filter. The Kalman filter aims to stabilize the output from the Madgwick filter, which can vary due to measurement errors in the IMU and oscillations from the gradient descent step of the Madgwick filter. In each iteration, the output from the Kalman filter is derived by adding the filter's current state, which serves as the orientation prediction, to the product of the filter's uncertainty and the difference between the new measurement and the prediction, and the uncertainty is changed by the processing noise in each iteration. By optimally combining predictions based on the current orientation estimation and new measurements from the IMU sensors after processing by the Madgwick filter, the Kalman filter reduces its uncertainty and iteratively refines the state estimates.

The gyroscope tuning algorithm linearly maps the filtered angle readings in degrees to analog readings, simulating the analog output of joysticks within a range of +/- 32,000. This range represents the joystick's movement from the leftmost to the rightmost side. Additionally, the output of the gas and brake knobs is mapped from a range of 0 to 255 that satisfies the 0.5% accuracy design requirement to a range of 0 to 1, which is required by the joystick's triggers as their inputs. A deadzone is applied for gas and brake knobs' readings that are less than 10 or greater than 245, setting the

readings to 0 or 255, respectively. This design choice accommodates users who may not be able to fully press the throttle or brake, ensuring that they can effectively control the input.

C. Software Systems: Communication & Data Flow

To establish the connection between RPi and PC, a client and server model with Python Socket is used. In our implementation, RPi acts as the server and PC acts as the client. The protocol used to achieve Bluetooth communication is RFCOMM protocol which supports simple and reliable communication between two devices. The protocol is supported by the Socket library. In order to connect the two devices, RPi creates a socket object, binds itself with RPi's Bluetooth Address and listens on port 1. Then, the PC creates another socket object, connects to RPi's Bluetooth Address on port 1, and starts receiving and sending packets.

In the actual implementation, as shown in the software block diagram, two servers are created on RPi and two clients are created on the PC. The server and client connected via port 1 is used for communicating input information and controlling the game. The second server and client connected via port 2 is used for sending car information back to RPi from the PC. The reason for this separation is to ensure that our controller inputs can be sent to the PC and control the game with as least interference as possible. The flow of sending car data and updating the dashboard therefore can be separated and run independently on another client and server process.

The packet format sent between RPi and PC is JSON. The button and sensor information are first stored in a Python dictionary which is then changed into JSON format on RPi. The RPi then sends the packet to the PC which decodes the data back into a Python dictionary to read the raw inputs. The car information sent from PC to RPi is in the same format. The RPi converts the JSON information from the PC into a Python dictionary to access gear, speed, and RPM information.

To establish connection between the main Python program running on PC and the game running on PC itself, outgauge mode is enabled on the game. The outgauge mode allows the game to act as a server to listen for connection on localhost, port 4444. Our helper program on the PC connects to the game on port 4444 at localhost to continuously read and write car information into a file without blocking. This allows our main program to only need to read the most current car information from the file and send it to RPi without contacting the game that takes extra time. Through this implementation, we are able to separate the data sending to RPi logic and the game data reading logic on the PC.

To establish connection between main Python program and the frontend dashboard implemented in HTML, CSS, and Javascript, with its source code from Codepen [13], on RPi, a NodeJS simple backend framework Express is used to act as a middleman that can handle POST request of most recent car information from the main Python program and GET request from the frontend Javascript that wants to update the HTML on the car dashboard. Once the main Python program on RPi receives the JSON car data from the PC, the main program makes a POST request to the Express backend running on localhost port 3000 and continues to listen for the next car information without taking extra time to communicate with the frontend. Once the car data is updated in the Express backend, the frontend Javascript running separately on the browser makes a GET request to the backend at localhost port 3000 to update the data. Since Express supports handling concurrent requests and updates, we are able to separate the data receiving and dashboard updating logic on the RPi.

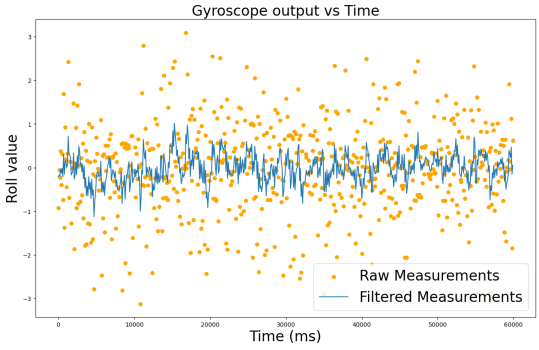
D. Software Systems: Key Mappings & Game Control

To control the game on the PC, the program uses vgamepad (Virtual Gamepad) library that can emulate an XBOX360 controller [14]. The library provides function calls that allow users to specify which button is pressed or released and in what direction have the joysticks moved by entering 0 or 1 for button pressing and x and y coordinates for joystick movements. In order to translate raw inputs that correspond to XBOX360 configurations provided by the library, the program maps the raw button values to the button and joystick values corresponding to the XBOX360 specified by the library. Once the raw inputs are mapped, then the program calls the button and joystick update methods provided by the library to control the game. Each update is done once a JSON packet of raw inputs are received from RPi on the PC. Once the updates are done, the main program then goes to the next loop and listens for the next JSON packet for updates. Since the loops are run in an expected input latency, there are frequent update method calls within a second just as if a gaming controller is actually controlling the game.

VII. Test, Verification and Validation

A. Results for Gyroscope Drift

Gyroscope drift is a common problem with orientation estimation based on angular velocity from gyroscopes. We can measure the gyroscope drift by placing the steering wheel on a flat surface, and then check the output of the gyroscope tuning algorithm for over 60 seconds. To pass the test we need the algorithm output to change within the range of +/-1 degree after 60 seconds.



From the graph we can see that after 60 seconds, the reading changed by 0.3 degrees, which is smaller than our 1 degree goal. Throughout the 60 seconds period, the distribution of the filtered output is within the +/-1 degree range of goal. The noise from raw measurements limits us from exceeding our target, since the large learning rate makes the Madgwick filter overshoots and cannot give us a stable measurement.

B. Results for Gyroscope Accuracy

We measured the gyroscope accuracy by placing the controller in various predefined orientations. We chose roll angles of 0 degrees, 90 degrees, and -90 degrees. These positions were checked by referencing the Measure app in iOS. In each of the positions, we hold the controller for 60 seconds and check the smallest and largest output. The values can be seen in the table below with degree as the unit:

Real angle	Smallest output	Largest output
0	-1.2	1.1
90	91.0	89.2
-90	-90.0	-88.9

The controller was able to output values in the neighborhood of the desired values. Though it does not meet the requirements for difference within ±1 degree, the project is still able to limit the range within ±1.2 degrees. This is limited by the oscillation of Madgwick filter output due to its high learning rate to achieve a fast response for rotations.

C. Results for Input Latency

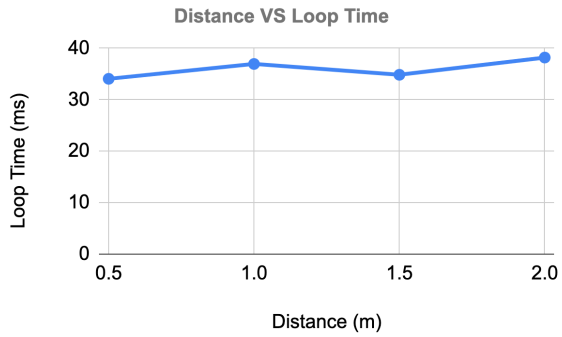
Input latency is measured by measuring the time it takes to run one loop on the main Python program that handles input reading and communication on the RPi. The total loop time is equal to the sum of button and gyro input processing time and message round trip time (RTT) which represents the time it takes to receive an acknowledgement from the PC after sending a packet to the PC from RPi as shown in the equation below

$$Loop\ Time\ =\ Input\ Processing\ +\ RTT$$

Since the RTT includes the time for the RPi to handle the acknowledgement sent from the PC, we approximate the input latency by dividing the loop time by 2 such that

$$Input\ Latency \cong Loop\ Time\ /\ 2$$

We measured the average of 1000 loop time for each distance at 0.5m, 1m, 1.5m, and 2m between the 2 devices to simulate different gaming environment settings. Based on the results, the average loop time at each distance is lower than 40ms, indicating that our approximate input latency is less than 20ms, as shown in the figure below.



D. Results for Final Product Layout

Based on the final product layout, we achieved the goal by incorporating 12 buttons through a 2 x 6 button matrix in the middle of the controller and two analog knobs for acceleration and brake. The case is 3D printed as specified.

E. Results for Weight

The weight for the controller is measured on a scale. The total weight is 660g which exceeds the specification of 400g. One main contributor to the weight is the 3D printed enclosing. We have measured that if the enclosing is printed with thinner walls and tighter packing of hardware, there will be 150g less enclosing weight contributing to the total weight. If we have enough time, we can also reduce the weight by using PCB instead of breadboard and wiring connections.

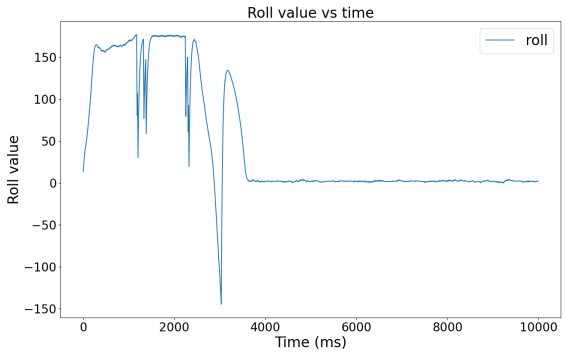
F. Results for Battery Life

The battery life we calculated is around 4.6 hours. Using a multimeter at the battery terminals, we measured the steady power draw of our system to be around 4 Watts. Our battery is 5000 mAh, or 18.5 Wh at the rated voltage. This gives us roughly 4.6 hours of battery life. This does not meet our design goal. We originally had a design requirement of a battery with size > 6960 mAh. However, during parts ordering and implementation, we found out that, firstly, the PiSugar does not come with a battery larger than 5000 mAh. Then we considered using an alternate battery and doing some custom wiring. This was not implemented because the extra weight a heavier battery would bring is too much for our system. We want to prioritize the comfort of our users. The shorter-than-designed battery life is one of our unfortunate trade offs. In the future, however, instead of using a bigger battery, we can look into how to lower the power consumption of our system. Maybe a screen with a slower hardware refresh rate, a processing unit with more dedicated features (less bulky than RPi), or some software tricks to put unused hardware to sleep.

G. Results for Response Time of Rotation

Our use case requires the project to be able to reflect a 180-degree rotation within 1 second. To test this, the steering wheel is placed at -180 degrees on the table, and the tester rotates it as quickly as possible. We then examine

the log of the gyroscope output to determine if it reflects the change within 1 second. The plotted result is shown below:



From the plot we can clearly see that the reading changes 180 degrees in around 0.8 second, which is within the 1 second goal. The oscillation due to the high learning rate of the Madgwick filter stops us from performing better, since if we increase the learning rate larger it will give us a higher oscillation and makes our noise filter work harder.

VIII. Project Management

A. Schedule

Please see the attached Project Schedule.

B. Team Member Responsibilities

Xiao Jin was responsible for building the hardware system. He researched and purchased the hardware components. He soldered the wiring for the components with the RPi. He also designed and 3D printed an enclosing shell for the controller and packed all the components inside.

Qiaoan Shen was responsible for handling sensors such as gyroscopes and analog pedals. He conducted research on orientation estimation and rotation representation. He implemented algorithms to process sensor readings and fine-tuned the results for our use cases.

Yuxuan Zhu was responsible for implementing the communication program between RPi and PC, PC Python program and game, and RPi Python program and frontend dashboard code. He modified the code for the dashboard and implemented a NodeJS backend to handle car data from the Python program.

C. Bill of Materials and Budget

Please see the table Bill of Materials at the end.

D. Risk Management

1) The biggest risk factor in our project is the tuning of the gyroscope output. We previously have had no experience with a gyroscope, and having to use an algorithm to consolidate potentially all 9 channels of data could be very challenging. In the event that a complicated algorithm does not work as expected, or if we run out of time, we will consider making compromises on the performance of the controller, for example, downgrade from sensing 3-axis movement to only sensing 1-axis movement.

2) Bluetooth communication is another risk factor in our project. Though there is a well-known package called Pybluez for Bluetooth communication in Python, the package is now currently maintained and does not support the newest Python versions. Initially there was a lot of trouble installing Pybluez and importing the package in code as system errors appeared frequently. Before finding the less documented Bluetooth socket currently used in the project, the mitigation strategy was to use USB cable instead of Bluetooth. If a basic communication couldn't be established between the two devices, we wouldn't do further testing and our project would fail.

3) Loss of code on the hardware is another risk factor. If the code developed is not backed up in time and there is a hardware issue that removes all the codes, then all the progress will be lost. To ensure that codes are available, we pushed all our code to Github regularly. We also created separate branches for programs related to communication and programs related to hardware and gyro tuning so that we can all make progress separately without constantly causing merge conflicts.

4) Hardware itself is another risk factor. The RPi might break, the sensor might be broken, and the screen might be smashed. In order to have a second layer of security, we borrowed another RPi4 from the ECE Department and bought another LCD screen. Not only additional hardware components provide prompt replacements when things fail, they also allow a second member to develop using the hardware simultaneously.

IX. Ethical Issues

Since our project is a gaming controller, there are naturally concerns towards misuse or addiction to games.

First of all, our intended use is for users who love driving with a physical steering wheel while playing car driving games without worrying about buying a bulky and expensive steering wheel set connected to a PC through wires. However, the worst case scenario relates to users who spend so much time that they cannot differentiate a real steering wheel from our mobile steering wheel.

A specific scenario could be that the user remotely connects the steering wheel to a PC and starts playing. The game the user is playing is a racing game that has a global ranking online and the user wants to rank No.1. In order to achieve this goal, the user plays day and night without going out of his home. When the user starts to drive a real car to buy some groceries after days of lockdown, the user still feels like driving a mobile steering wheel. However, when turning, the user only rotates the wheel to an angle that can turn a car in the game but not in real life, so the user hits the other cars and is fatally injured.

Second of all, people who may easily get addicted to the game will be vulnerable to our product. If someone plays the game too much with our controller and doesn't drive in

the real world frequently, they may get too tired physically, but cannot realize that mentally because they are highly excited. Long term, this can cause health concerns that can potentially lead to horrible outcomes such as death.

X. Related Work

Our whole idea stems from the experience of using smartphones to play racing games. Racing games on smartphones often utilize the gyroscopic data to translate tilting of the phone to steering the cars.

XI. Summary

A. Future work

We have two main plans for the future. As we realize a smartphone has all the hardware components we have, we can possibly incorporate it into our design. We also want to redesign the shell of the steering wheel to be more ergonomic.

B. Lessons Learned

The final product turned out to be more complex than we initially envisioned. Most of the technologies were new to us, but through learning, we managed to grasp them. Integration proved to be the most challenging aspect, and we should always allocate ample time for integration and testing in future projects. For future students who wish to continue working on this project, we would like to recommend focusing on the gyroscope tuning section, which is the core of the project. The steering wheel relies on the potentially unstable results of this section to control the car in the game, and hence it's the most difficult part of the project.

Glossary of Acronyms

IMU - Inertial Measurement Unit
ADC - Analog-to-Digital Converter
PC - Personal Computer
RPi - Raspberry Pi

References

[1] FORMULA, "The fastest lap in F1 history - Lewis Hamilton's pole lap | 2020 Italian grand prix | pirelli," 05-Sep-2020. [Online]. Available: https://www.youtube.com/watch?v=2f1PtJV0vIs. [Accessed: 06-May-2023].
[2] J. Baldwin, "Drivers Eye At Spa Francorchamps In A McLaren 720s GT3," 16-Sep-2022. [Online]. Available: https://www.youtube.com/watch?v=B4JAEGYEqQo. [Accessed: 06-May-2023].
[3] Frame Counting, "PS5 DualSense vs Xbox series X/S controller input lag test - just the controllers," 26-Dec-2021. [Online]. Available: https://www.youtube.com/watch?v=UXS_0tub_Jk. [Accessed: 06-May-2023].
[4] "Smartphones weight ranking - comparison list," Techrankup. [Online]. Available: https://www.techrankup.com/en/smartphones-weight-ranking/. [Accessed: 06-May-2023].
[5] C. Hoffman, "Bluetooth 5.0: What's Different, and Why it Matters," How-To Geek, 27-Feb-2018. [Online]. Available: https://www.howtogeek.com/343718/whats-different-in-bluetooth-5.0/. [Accessed: 06-May-2023].
[6] "How much power does the Pi4B use? Power Measurements," RasPi.TV, 25-Jun-2019. [Online]. Available: https://raspi.tv/2019/how-much-power-does-the-pi4b-use-power-measurements. [Accessed: 06-May-2023].
[7] Adafruit Industries, "HDMI 5" Display Backpack - Without Touch," Adafruit.com. [Online]. Available: https://www.adafruit.com/product/2232. [Accessed: 06-May-2023].
[8] "Xbox Wireless Controller," Xbox.com. [Online]. Available: https://www.xbox.com/en-US/accessories/controllers/xbox-wireless-controller. [Accessed: 06-May-2023].
[9] GENERAL DESCRIPTION, "World's lowest power 9-axis MEMS MotionTrackingTM device," Sparkfun.com. [Online]. Available: https://cdn.sparkfun.com/assets/7/f/e/c/d/DS-000189-ICM-20948-v1.3.pdf. [Accessed: 06-May-2023].
[10] Datasheets.com. [Online]. Available: https://www.datasheets.com/en/part-details/bno085-ceva--inc-405190273. [Accessed: 06-May-2023].
[11] S. O. H. Madgwick, A. J. L. Harrison and R. Vaidyanathan, "Estimation of IMU and MARG orientation using a gradient descent algorithm," 2011 IEEE International Conference on Rehabilitation Robotics, Zurich, Switzerland, 2011, pp. 1-7, doi: 10.1109/ICORR.2011.5975346.
[12] J. Böer, "madgwick_py: A Python implementation of Madgwick's IMU and AHRS algorithm." Accessed on May 5, 2023, [Online]. Available: https://github.com/morgil/madgwick_py
[13] Chmood, "Car speedometers with engine sound," Codepen.io. Accessed on May 5, 2023. [Online]. Available: https://codepen.io/Chmood/pen/MaBZdM.
[14] Y. Bouteiller, vgamepad: Virtual XBox360 and DualShock4 gamepads in python.

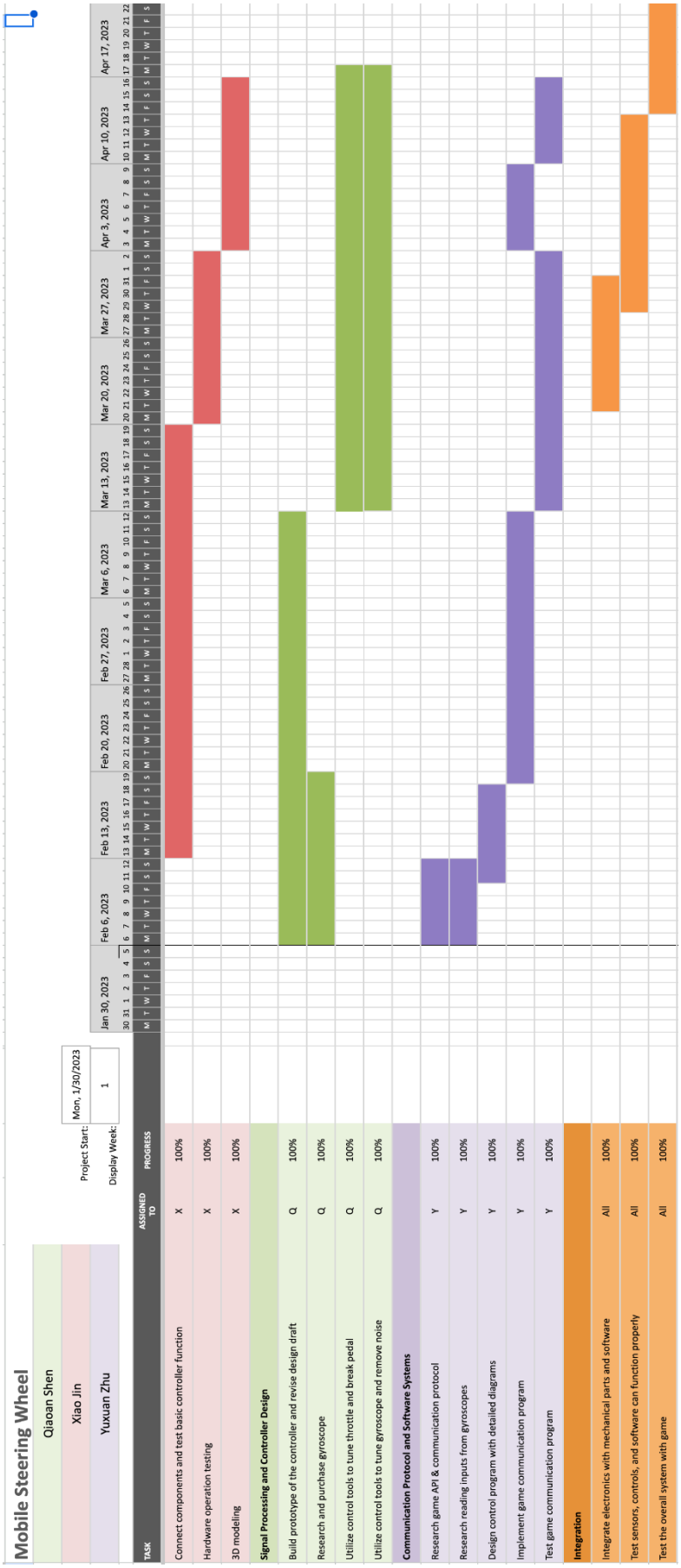


fig. Team Schedule

Name	Price	Quantity	Cost	Notes
16mm Panel Mount Momentary Pushbutton - Green	$0.95	20	$20.14	
HDMI 5" Display Backpack - Without Touch	$59.95	1	$63.55	
SparkFun 9DoF IMU Breakout - ICM-20948 (Qwiic)	$18.50	2	$39.22	
Potentiometer with Built In Knob - 10K ohm	$1.25	4	$5.30	
ADS1115 16-Bit ADC - 4 Channel with Programmable Gain Amplifier	$14.95	2	$31.69	
Pisugar 2 Plus: Battery for Raspberry Pi 3B/3B+/4B	$49.99	1	$52.99	
eSUN PLA+ Filament 1.75mm	$22.99	2	$48.74	
Geekworm Raspberry Pi 4 7mm Embedded Heatsink with Fan	$9.99	1	$10.59	
USB to Micro USB Cable, 90 Degree Right Angle	$6.99	1	$7.41	
Micro HDMI to Standard HDMI Cable	$12.72	2	$26.97	Only used 1
Various lengths copper wires and heatshrinks	-	-	-	
M2x6 Hex Screws	-	18	-	
Total			$306.59	

Bill of Materials