Sensing & IOT Report

CSAIM Project

Rob Garland 14/01/2020

CID: 01340769

Using accelerometer sensor and Gamestate API to provide useful feedback on the Aim of the the player in the FPS Title Counter Strike:Global Offensive.

Coursework 1: Sensing

1 – The Problem

Figure 1: SK Gaming professional Counter Strike team winning the 2017 ESL One Cologne Tournament

Counter Strike:Global Offensive (CS) is a strategic eSports First Person Shooter title. The game relies on precise mouse movements in order to 'hit' other players resulting in kills, these precise movements are essential to any success in the game. Professional players have built up muscle memory over years and many thousands of hours to be confident hitting, what is essentially a small area of pixels on the screen which correspond to the enemy player.

The eSports scene is taking on a life of its own, with hundreds of thousands of viewers in CS tuning in on the Twitch streaming service, and crowds of over 10,000, to watch professional teams in action in major tournaments, often in competitions with prize pools worth millions of dollars.

It is therefore recognised that the ability to analyse these movements and provide insights for game improvement will be important at both the elite and casual level. As the game is competitive, you want to improve, so that you can win!

Hence, this this project was created to use sensors in an IoT setting around the mouse and use API's to get data from the game with the intent of providing analysis and tips to improve, like devices that measure elements in traditional sports, such as concussion monitors, and distance trackers in Rugby and Football, respectively.

2 – Hardware Set-Up

The set-up of the project consists of four core elements, the first is the Adafruit MMA8451 accelerometer sensor which is located on the mouse using a 3D printed clip, see Figure 2. An Arduino Uno, which is essentially used as an ADC to grab data from the accelerometer over I2C. The Arduino Uno is then connected and powered by a Raspberry Pi 3B+ using a USB-A to USB-B lead. This is used as the serial bus between the Arduino and Pi. The final component is the PC, Counter:Strike is a PC game and so a laptop or PC is required to play the game, and get the live outputs.

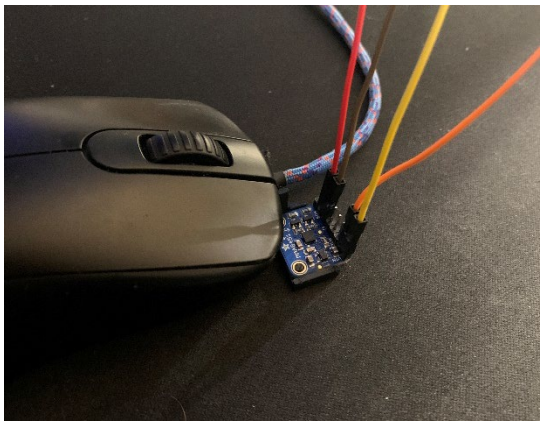

Figure 2: Sensor location on my Zowie S1 mouse.

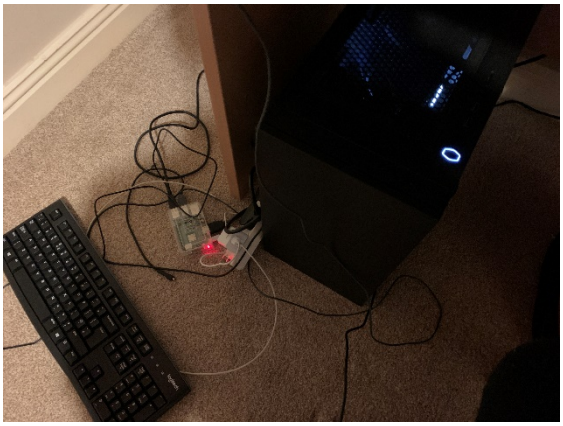

Figure 3: Additional Hardware components PC and Raspberry Pi 3B+

3 – Overall Sensing System

Figure 4: Desktop Hardware set-up, Arduino can be seen.

The overall sensing system uses a blend of API's and the hardware mentioned previously to create the data required for useful analysis. Libraries were installed on Arduino in order to allow it to convert the ADC values obtained from the accelerometer into SI units which are then sent to the Pi via the serial bus. There is an interaction between the PC and Pi via the sockets module in Python. Local hosts are used to create a HTTP server on the PC to collect the game data through an API called Game State Integration (GSI), pre created python libraries were used to set this up and an in-depth description was found on reddit here, to help learn how to use it. Multithreading is used on the Pi in order to allow a buffer to store data from the accelerometer while it is not required, until prompted by the PC that there has been a game state change, when this occurs, the current buffer of accelerometer data is added

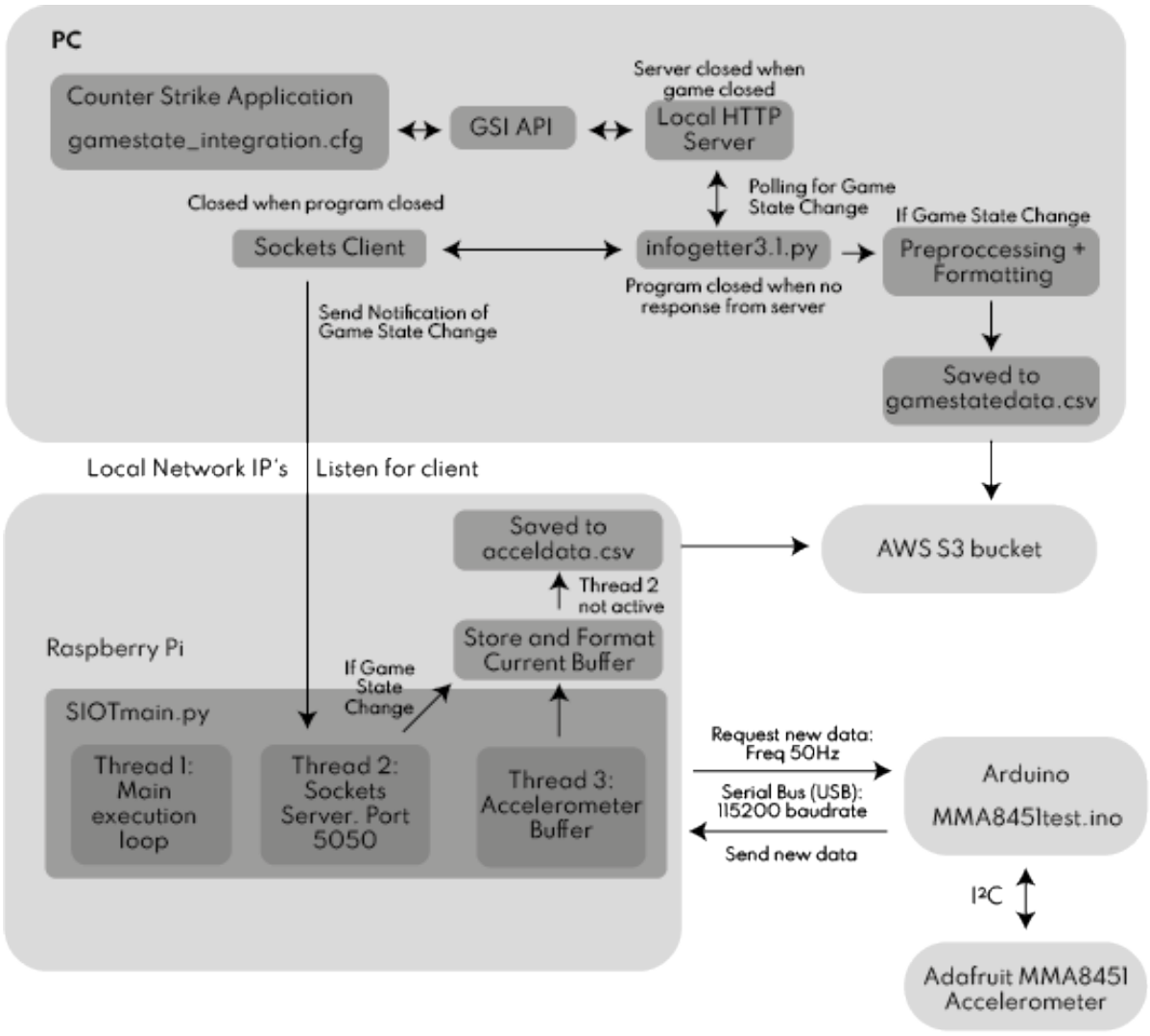

Figure 5: Sensing Set-Up overview

to the file. At the end of the process, upon the game closing, the AWS boto3 python SDK was used to backup/upload the data, stored in CSV format to an S3 bucket. Figure 5 shows the system in more detail. All programs were written in Python 3.

4 – Sensor Set-Up and Data Acquisition

4.1 Getting Gamestate Data

The Game State Integration API used pre created libraries available here.

4.1.1 – create cfg file

The first part of the API involved creating a .cfg file placed in the cfg part of the local Counter Strike game file. (C:\Program Files (x86)\Steam\steamapps\common\Counter-Strike Global Offensive\csgo\cfg) . This file lets the game know to what data categories to output to the local HTTP server. In my case I was mainly interested in the data with a "1" next to it in Figure 6.

```
"data"
    {
        "provider"                  "1"
        "phase_countdowns"          "0"
        "map_round_wins"            "0"
        "round"                     "0"
        "bomb"                      "0"
        "map"                       "1"
        "player_match_stats"        "1"
        "player_position"           "0"
        "player_weapons"            "1"
        "player_state"              "1"
        "player_id"                 "1"

        "allgrenades"               "0"
        "allplayers_id"             "0"
        "allplayers_match_stats"    "0"
        "allplayers_position"       "0"
        "allplayers_state"          "0"
        "allplayers_weapons"        "0"
    }
```

Figure 6: .cfg file (1 = data will be collected)

4.1.2 – infogetter.py (here) – Sensing and Preprocessing Game State Data

Launched with the game using a batch file, the way in which the data was collected live from the game is found in the getGSIdata() function in my infogetter.py file. Effectively, once the server has been started and a connection established, the server set up is also part of the infogetter file, the program polls the server at a rate of 33.33Hz (selected as fastest weapon rate of fire is the Negev at 1000 RPM, thus, to sample at the Nyquist rate sample period = 60/1000 = 0.06/2 = 0.03 (1/0.03 = 33.33Hz)).

If the state of the player is 'playing' (this is when the user is not in a menu!) then the program enters a while loop. The current gamestate is stored in my desired format using the info() function. Once attained, this is appended to a list of data collected in that session and stored to the 'previous' or 'previous1' variable.

After an initial collection period of a few days, it was noticed that I had a habit of quick switching between weapons, all that was changing between game state was the weapon, there was no change to any other category of note, this data was not useful, and would be problematic to have when getting accelerometer data. To combat this I added additional comparison/validation checks to the program to ensure that quick switching (effectively noise) was filtered from the data, only when there was a change to a category of interest is data appended.

This collection upon game state change was primarily done as a way of optimising the storage size of the output csv's, with the data polling at the rate it does, there would be hundreds of thousands of datapoints if one were added every period over the course of a session, this would result in slow download/upload times and affect the efficiency of the dataanalysis module. This way, the points from one session keep the csv in the KB range rather than the 10's of MB, also, it ensures that the data added is all relevant, if it were simply added at every time period much of the data would be discarded anyway.

4.2 Sockets server/client

Within the infogetter.py file is also the set up of a sockets client to communicate over my local network between the PC and the Pi. Also found in the SIOTmain.py (here) file on the pi is a system that is split into a start() function which does the initial set-up of the server. There is a blocking line (server.accept()) in the main program loop that will only be passed when a new

client requests connection. Once this is passed, a new thread opens which runs the handle_client() function, which sets up communication between SIOTmain.py and infogetter.py. If a game state change is detected messages are sent using the send() function in infogetter, the message will consist of a single byte ("1","0","!") if the message is a 1 then the gamestate has changed, SIOTmain will handle this message and conduct the corresponding process. If "0" nothing will occur, and if "!" the handle_client() function will close the thread (this is the disconnect message), the "!" is only sent during the closing lines of infogetter.py, in order to disconnect the client, so that issues do not occur when the game is reopened, as SIOTmain runs continuously.

4.3 Getting Accelerometer Data

4.3.1 – SIOTmain.py – Sensing from Gamestate

As mentioned in section 4.2 the handle_client() function is running on a thread when the server.accept() line has been passed and connected = True. The function has some protocols once a particular message is received. If "1", the protocol is to empty the current buffer into the list acting as data storage (.pop()). If the message is "!", the thread is closed.

4.3.2 - SIOTmain.py – Multithreading and sending data get request to Arduino

I – Buffer

Within this program, a thread is opened in the start() sequence that causes a live buffer of accelerometer data to be collected, the functions of note for the buffer are livedata() and adddata(tic,toc). The livedata function runs an infinite while loop that calls the adddata function once the sampling time period, in this case 0.02s, has elapsed, calculated using the 'tic' and 'toc' variables, which are updated at points in the execution, to do this the python time module is used. If the buffer is currently full, the oldest datapoint is removed and the new one appended. Allowing for a continuous buffer to be created.

II – Serial Set-Up and calling data

Using the python serial module, a serial connection is set-up in the start() function to the Arduino. To minimise the time delay in transfer, as data is collected in such minuscule time periods, a high baudrate (115200) was used. Data is then sent and received from the Arduino using commands from the serial module. A data request is sent as the byte "1". Using ser.flush() all current data is to be cleared from the serial queue before sending new data, to ensure clean communication. Data from the Arduino is sent as a string and so using eval() it is turned into a useable list, this is the formatted and appended to the buffer.

III – MMA8451test.ino – I2C on Arduino and Serial printing upon receipt

This is the code running on the Arduino, simply, it is an infinite loop that waits for a "1" to be received through serial before getting the current accelerometer event. The adafruit unified sensor library, and MMA8451 library were installed on the Pi so that when the file was uploaded to the Arduino, the sensors_event_t event and mma.getEvent(&event) functions would work. Once a "1" is received the current event is recorded, and formatted in as few characters as possible. Using event.acceleration.x (x interchangeable with y or z) that events' acceleration in SI units is created, this is then sent back to the Pi using Serial.print.

IV – Data Quirks

The accelerometer data collected has a baseline acceleration of around on average of -0.15 and -0.25 m/s^2, this means that it was not optimal to get positional data directly as integrating over such a long time period would cause a big offset, to combat this I could add a moving

average filter to the buffer on the Pi, this would allow me to filter the annoying offset and clean up the overall data.

4.4 – Pushing to AWS S3 via boto3

When the game closes, the session is deemed to have ended. The disconnect message is sent from infogetter.py to the Pi. During the initial lines of code in infogetter and SIOTmain, the s3 bucket is created as a boto3.client() object. In the final lines of code in infogetter the s3.upload_file() is used to send the latest gamestate.csv to the bucket. Likewise, in SIOTmain, once the disconnect message is received, the csv is closed and uploaded to AWS using the same method as the infogetter program.

5 – Raw data analysis in app

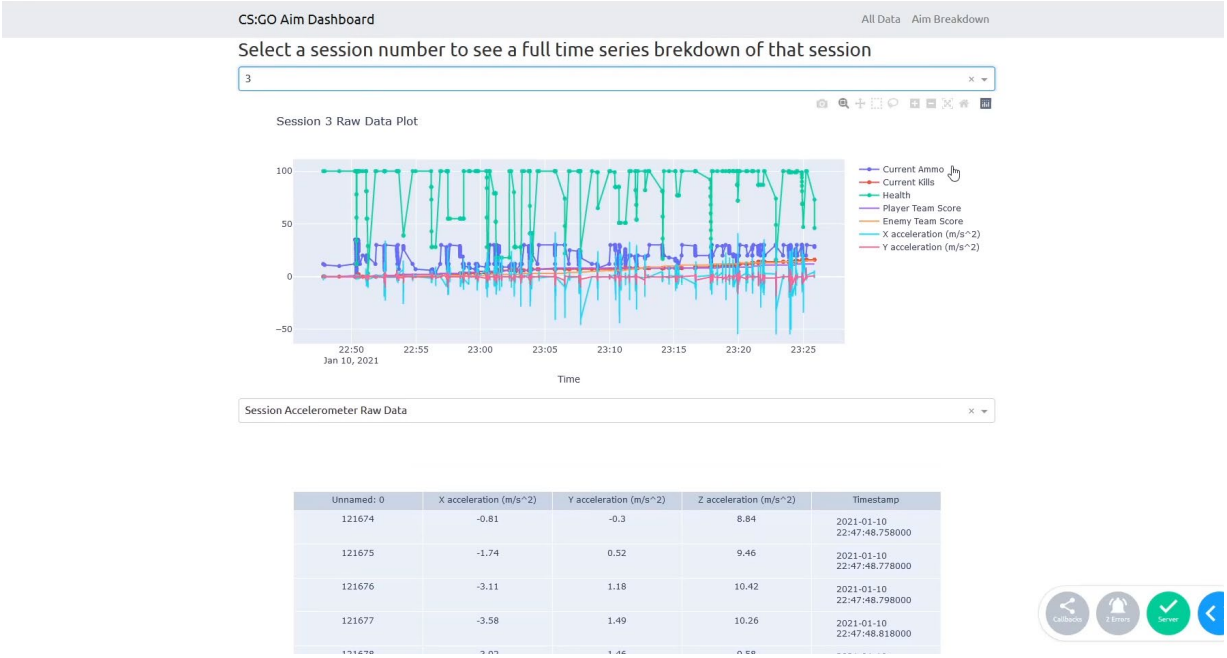

Figure 6: Raw Data Visualisation

The data can be visualised by session (each data set in gamestatedata.csv has a session identifier), in the app. This data is presented as a time-series. As can be seen in the graph we can see a correlation between changes in x and y acceleration as bullets fired changes.

dataanalyser.py is a module that was created in order to conduct data analysis and breakown the data to create insights. This function will be demonstrated in coursework 2 as it fuses the two data sources.

The timestamps were stored at the point of collection for both the accelerometer and game state data – the format is made using the datetime python module, this makes the data more readable to the eye than the initial UNIX timestamp would.

Coursework 2: ‘Internet of Things’

1 – Context

1.1 What do people want to know?

CS offers many possibilities, it is strategic, it is about using utility to manoeuvre your opponents around a map, all actions have consequences. The are many applications already created (e.g. scope.gg / leetify.com) that offer details into your positioning on maps, your kill differentials and ways to garner your impact on a game, and what you need to think about to improve. It is a big subsection of the current eSports scene, and many services offer paid models to get the most detailed stats!

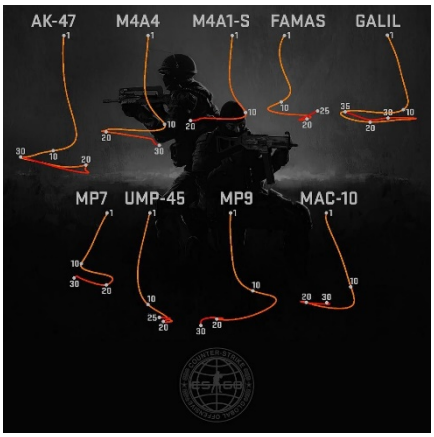

Figure 1: Various CS spray patterns. Numbers represent where your mouse must be relative to the start point at a particular bullet fired.

There is also a level of skill required in the game, you can be the best strategist in the world, however if you are unable get the necessary kills to win a round, there is little hope that you will win. I saw a gap in the existing solutions where they do not have a hardware component that analyses exactly how good you are at aiming.

Like traditional sports, you may conduct a warmup, a series of minigames that allow you to refresh your muscle memory before entering the game, it would be interesting to know how you performed so that you can warmup more optimally too.

1.2 – CS’s convoluted shooting mechanics

Counter Strike has a ‘high skill ceiling’ this is part of what makes it so popular, this in part is due to the way in which you shoot and kill opponents. Each weapon has a ‘spray pattern’ this is a pattern that you must approximate with your mouse to ensure that each bullet travels to the place in which you were initially aiming. There are also many methods of shooting; tapping which is exactly that, shooting one bullet at a time; spraying/bursting, which is holding down mouse1, but the categorisation is dependent on the number of bullets; there is also categorisations within these types of shooting, see Figure 2. For example, certain players may be extremely good a spraying but not great at getting kills from taps. Therefore, in order to become a more complete player, it goes to say that it would be interesting to know what you were strong and weak at with your aim!

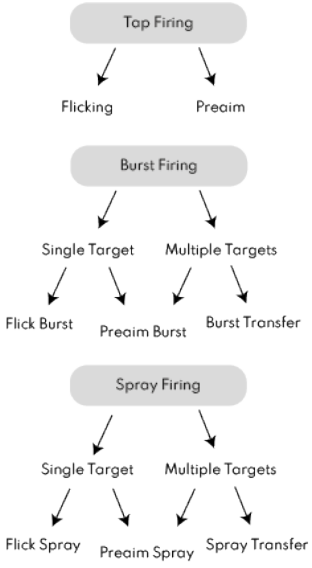

Figure 2: Shooting Categorisation in CS

2 – Overall App Outline

Figure 3, opposite, shows the basic app structure, when the index.py file is started, the two latest versions of the data csv files are downloaded into the working directory, these files are then passed through the dataanalyser module which interprets the data, outputting an additional csv of ‘useable’ data. This file includes categorisations of sprays, total bullets fired, and shooting type to kill ratio. The app was created using libraries/modules from plotly (plotly_graph_objects, dash_core_components, dash_html components and dash_bootstrap compenents). Data frames that were used to create of metrics and graphs use the pandas module. The app style is imported from dash bootstrap components.

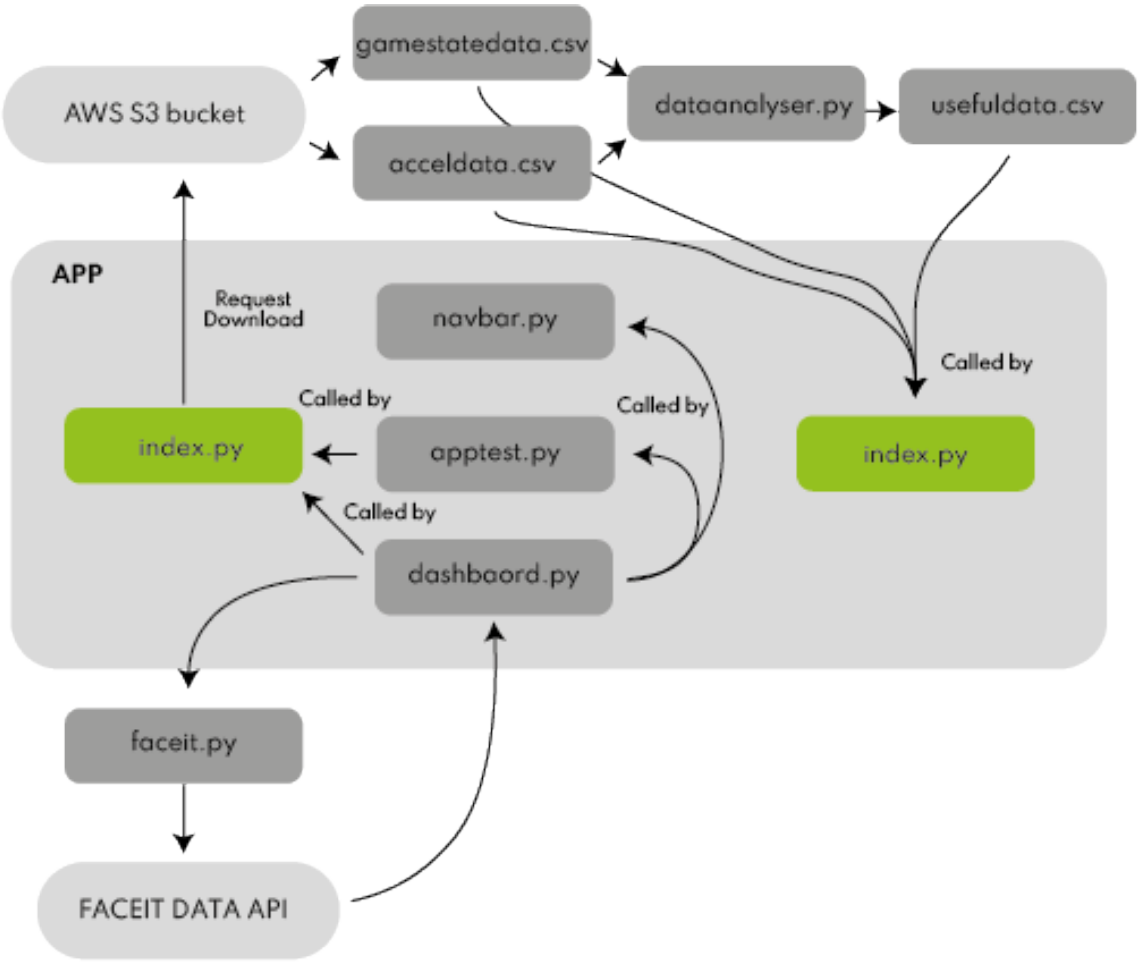

Figure 3: App Outline

2.1 – Index

Index.py (here) acts as the initial and root file of the app, within the file, callbacks are used to update components based on inputs from dropdown menus. It also accesses the S3 bucket on start-up downloading the latest data.

2.1 – Dashboard

In the app, the dashboard acts as the primary hub for the user to view their data, using the FACEIT.com data API, data is pulled from the matchmaking service. The last 8 matches, along with some key metrics and comparison to the 8 games prior to this 8 is also shown. If the metrics show a slump, then it is inferred that the user should practise more before playing competitively. The dashboard is called to the index file using the Homepage() function when the href = '/dashboard'. Also in the dashboard is the results of the data analysis, it shows the minutes I played in the last session, the total bullets fired in the session, and rounds played in the session, figure 4 shows a test session and so the data represents this short session.

If this session were removed from the data, the last session column would show data from some actual games. The graph also shows the number of taps, bursts and sprays found in the previous session. You can then see the breakdown of this data by navigating to the aim breakdown section. The layout of the page can be seen in dashboard.py, it uses a series of dash html elements to position the data, in the prototype the data is shown using html tables, the next step would be to improve the formatting of the tables.

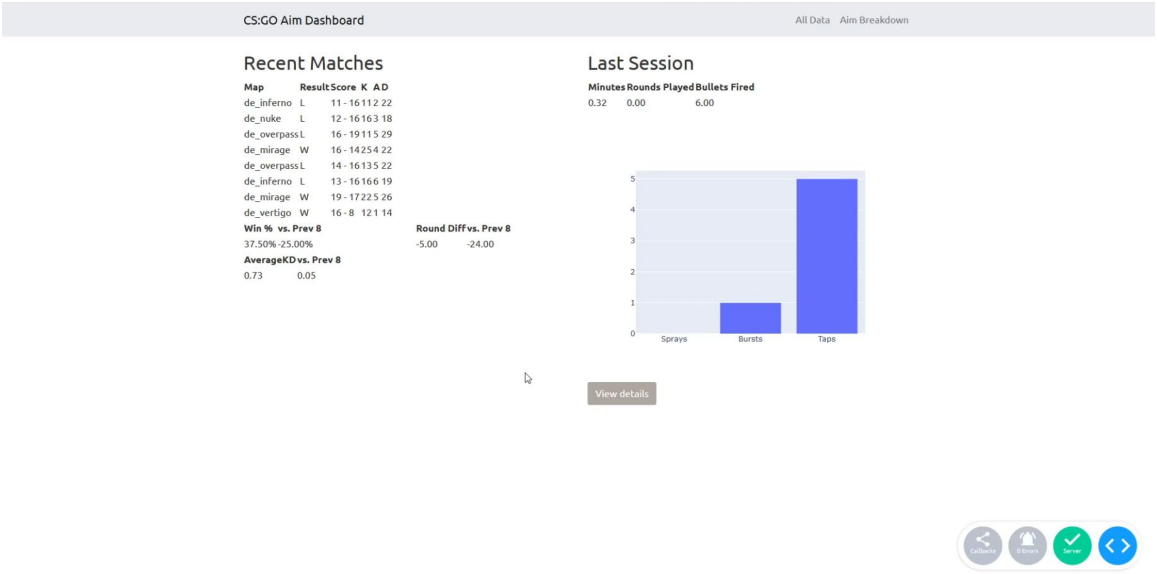

Figure 4: Dashboard Prototype

2.2 - All Data

The all data page allows the user to see the raw data found in the session, Figure 6 from Coursework 1 shows this page. Using the plotly scatter graph object the graph is updated based on the dropdown. Along with this, the table is a plotly table object, also updated based on the dropdown. Plotly gives default graph interactivity meaning the lines can be removed and you can zoom in on a particular section of the graph.

By hovering the user can see all the data points at that specific timestamp. The functionality of this page is found in the apptest.py file, within the App() function, which is a layout object, when called to the index, this becomes the new page. Within apptest.py there is also a function called build_graph_and_tables, this is called in the index callback update_graph, it interactively creates the graph objects based on the input parameters of the function.

2.3- Aim Breakdown

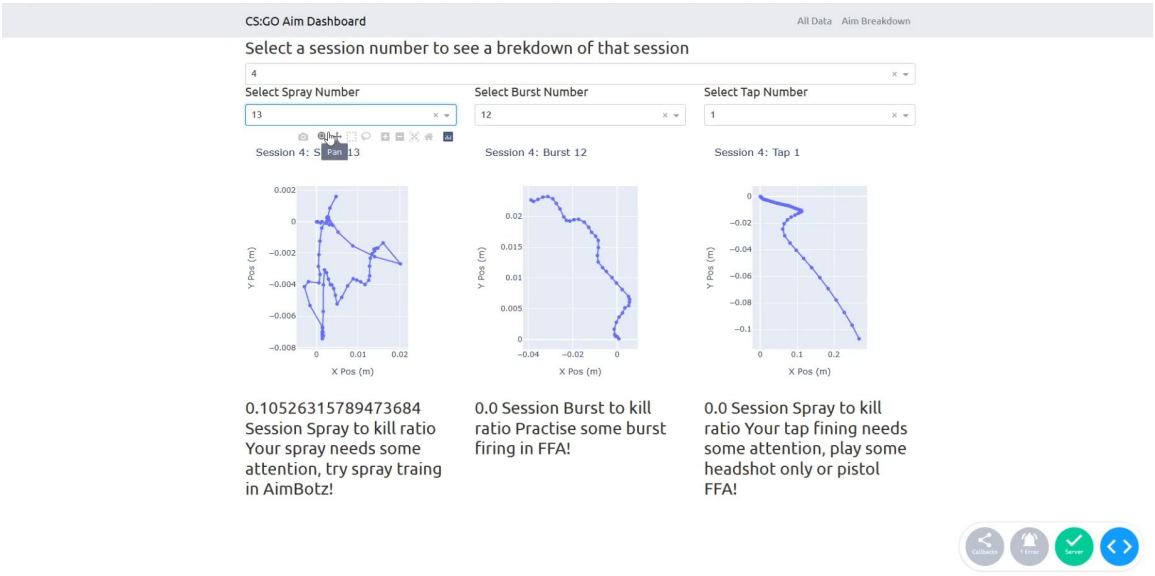

Figure 5: Aim Breakdown

The aim breakdown page shows the outputs of the data analyser, based on session, and shooting type. In this page, the positional data during the time period of the selected object is shown. Below this text showing the session shooting type to kill ratio is shown, this gives the user visual data to evaluate the success of their session. Along with this an insight is produced based on how the data stacks up compared to previous sessions, if the data shows a worse shooting type to kill ratio than the average for other sessions the app will will recommend that you try a specific type of practise that is tailored to improving your abilities at a particular shooting method.

The function that is called by the index file for this page is Breakdown(). In the Index callbacks, the function called is build_spray_tables, this is what is used to produce the updates to the page, it outputs the graph objects and html header objects. The graphs can be hovered over, showing the additional data of InstanceKills and Weapon Name.

3 – Cross - Correlation (dataanalysis.py, getpositions.py)

In order to create the aim breakdown two 'modules' were created which are called by the app. dataanalysis.py contains a function that checks the gamestatedata csv for a change in current ammo of -1 between two points for each weapon in the dataset. This is done in two for loops, this has compute time of $O(n^2)$ which is far from optimal.

However, it does work for this purpose. When the change in ammo is identified as -1, this means a bullet has been fired, thus the instance bullets = 1. The instance start time is the recorded at this point, the loop the continues, if the next point has a time differential to the previous point less than the threshold, the loop continues, until this is not the case. When this is not the case, the end time is found.

There are various scenarios as can be seen in the code where an Instance is deemed to end, at this point the instance can then be interpreted as a 'Tap, 'Spray' or 'Burst', the accelerometer data is then sliced between the Instance Start Time and Instance End time, and using getpositions.py, the accelerometer data is integrated to produce position data. This is then appended to the usefuldata list, along with additional metrics such as total bullets, Instance Kills and so on to be interpreted by the app into useful metrics.

This cross correlation of the game state data and accelerometer data is useful as it allows the user to see exactly the motions they undertook, allowing them to fine tune, for example if the spray pattern identified for an ak47 as seen when the graph is hovered over in aim breakdown, does not resemble the corresponding pattern shown in Figure 1, then the user knows that their spray needs some work.

4 – Opportunities

4.1 – Machine Learning

This project has opportunity for ML to be integrated, an open-source platform such as TensorFlow could be used to create a model. For this model to be created a detailed dataset containing the data that represents each of the shooting types would have to be created. Currently this project only focusses on the top level shown in grey in Figure 2. Potentially with the help of ML, the project can start to predict and classify the bottom level objects in Figure 2, such as flicks (adjustment to a target), preaims (opponent walks into crosshair, limited adjustment required) and transfers (adjustment between targets).

In order to effectively implement ML, it should be put on a central device, as it will be using the two datasets.

4.2 – Battery powered endpoint (potentially push)

The current hardware set-up is convoluted, it would be more optimal to have a single wireless battery powered endpoint, collecting the data, and the analysis done in the cloud. Arduino Nano 33 IOT shown in Figure 6, has an onboard IMU, this means that with wireless connectivity, the data could be sent to a database without the need for the Raspberry Pi at all. This would also remove the annoyance of the additional dupont wires.

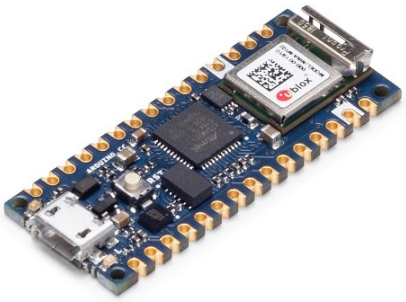

Figure 6: Arduino Nano 33 IOT

4.3 – DynamoDB for quicker app

An additional improvement would be to use a live database such as AWS DynamoDB for data collection, rather than simply uploading to S3. This would offer the opportunity to show the user live readings, and live analysis, which would be arguably more useful as it would be provided whilst playing the game, allowing you to adjust your aim based on the readings of the system. Th user would be able to see that my last spray was sub-par, and exactly what I did, rather than going off 'feel'.

4.4 – More App detail/Mobile Phone App

Lastly, the app prototype is quite rudimentary, it would be impressive to improve the formatting of the app so that the data can be observed, especially on the dashboard, more cleanly. Also, on the aim breakdown page, indications on the positional data of when each bullet was fired would allow the user to give more context to the graphs, helping to improve understanding.

If the user has a single monitor, and we can see live data, it is not optimal to have a web app, I would be more beneficial to have a mobile app, or a small screen that can be placed on the desk so that the user can view the insights without having to tab out of the game.

4.5 – Additonal Endpoints

An eye tracking device could be implemented to get data on the direction of someone's gaze during a match, this paired with the accelerometer data could provide you with all you need to become an 'aim god'.

5- Conclusion

The goal of this project was to be able to provide users with an insight on the positional data of their mouse during the different shooting types found in the Counter Strike game. The app does provide positional data, however improvements such as bullet indicators and a greater range of insights could benefit the user. Raw data from multiple sources has been pulled together. Continuing to collect data and use that app after this project, could bring fantastic improvements to my game, and potentially reach into the elite markets where every bullet counts!

Appendix

GitRepo Code and Data (read Readme.md) - https://github.com/robgarland/SIOT-CSAIM

Presentation Link - https://drive.google.com/file/d/1GjyrViWexktbyUbrs2oXsgTD1ml1Sely/view?usp=sharing