CG4002 Computer Engineering Capstone Project
2022/2023 Semester 2

"Laser Tag++"
Final Design Report

Group **BO7**	Name	Student #	Primary Component	Secondary Component
Member #1	Lee An Sheng	A0223222Y	HW Sensors	Comms Internal
Member #2	Wang Zihan	A0214345M	HW AI	Comms External
Member #3	Izdiyad Farhan Bin Zuri	A0217371J	Comms Internal	HW Sensors
Member #4	Zhuang Jianning	A0214561M	Comms External & Game Engine	HW AI

Section 1 System Functionalities

1.1 User Stories

As a…	I want…	So that…
Player	To join a game with other players	We can play against each other
Player	To use gestures to execute in-game actions	I can control and interact with the game
Player	To get visual/aural feedback of my in-game actions, such as when I have fired a shot off	I will know what is happening in the game and the feedback to my actions
Player	A gun which shoots laser bullets	I can deal damage to the opponent
Player	A grenade to be thrown	I can deal greater damage to the opponent behind cover
Player	A shield to be activated	I can protect myself from incurring damage
Player	A motion-activated reloading action	I can reload the gun when magazine is empty
Player	A wearable vest with target sensor	The opponent can deal damage to me
Avid gamer	Intuitive motion controls with sensors and gun only	I can get a more realistic shooting experience
Avid gamer	My actions to be detected and registered quickly and reliably	I can play the game seamlessly
Avid gamer	To not face random disconnection from the game & to be able to reconnect and resume gameplay easily	I do not need to worry about interruptions to my gameplay

1.2 Feature List

With only four members, the visualiser component of the game is not included in the assessment for our group. Nonetheless, we would like to implement visual and aural hardware indicators to acknowledge and display a player's in-game action being successfully executed.

1. A sound will be made by a buzzer on the gun once a shot has been fired by the player to notify him/her of the action being successful.
2. LEDs on the vest target sensor would light up to notify the player of his player health points or his shield activation.

1.3 Use Case Diagram

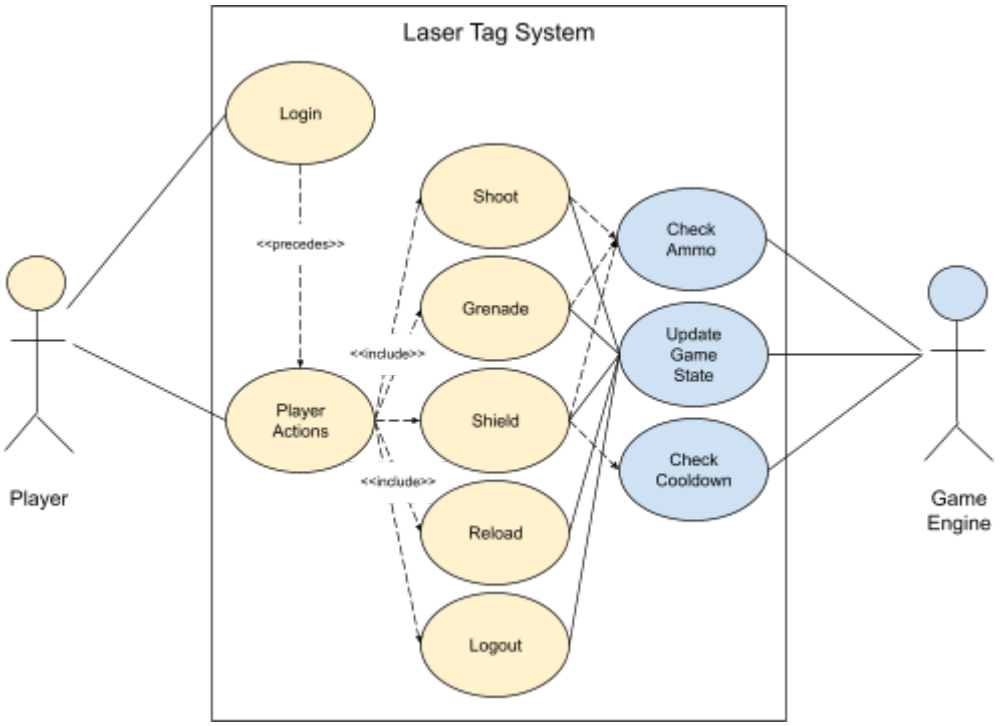


Figure 1.1: System Use Case Diagram

Section 2 Overall System Architecture

2.1 UML Diagram

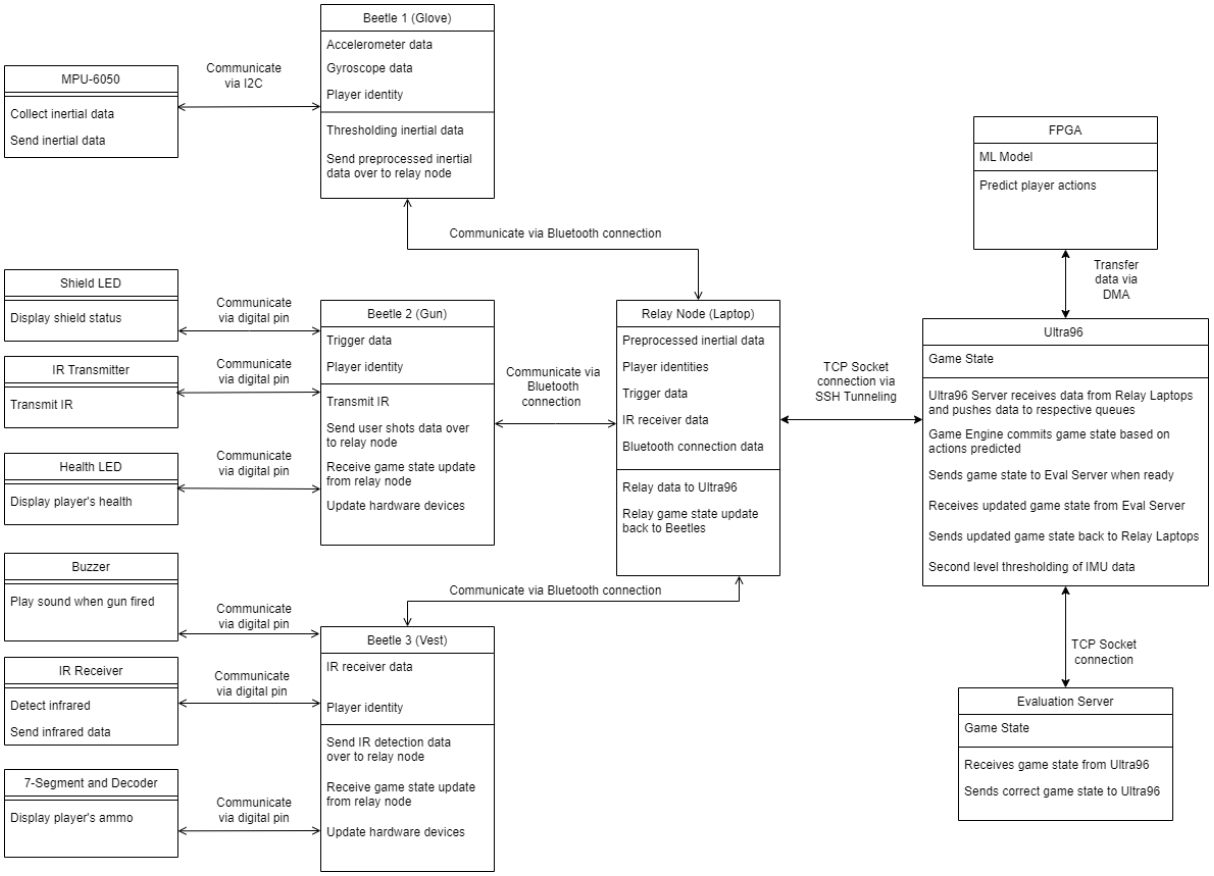


Figure 2.1: The UML Diagram

2.2 System's Final Form

Our system's final form consists of three physical pieces of equipment that each player would be equipped with: a wearable vest, a glove and a gun. The description of each component is detailed below.

Vest:

- Coloured high-visibility vest with pockets
- Beetle with infrared receiver on the left chest
- Pocket to store auxiliary components
- Battery pack attached outside of pocket for easy access
- White LED strip to indicate that the player's shield has been activated
- A LED strip to indicate player's health points

Glove:

- Plastic container at back of the hand to store Beetle, IMU sensor and auxiliary components
- Battery pack velcroed outside container, on glove for easy access

Gun:

- A lightweight pistol that allows for fast and agile movement
- A button trigger mechanism retrofitted in the gun
- Beetle with infrared transmitter in gun barrel
- Battery pack velcroed outside of gun for easy access
- A buzzer will sound when shot is fired
- A 7-segment LED to indicate the amount of ammo left.

Visualiser:

- Due to the lack of a fifth member, the software visualiser will not be implemented

Figure 2.2, 2.3, 2.4 below illustrate the system's hardware final form for vest, glove and gun respectively.

Figure 2.2: System's Hardware Final Form (Vest)

Figure 2.3: System's Hardware Final Form (Glove)

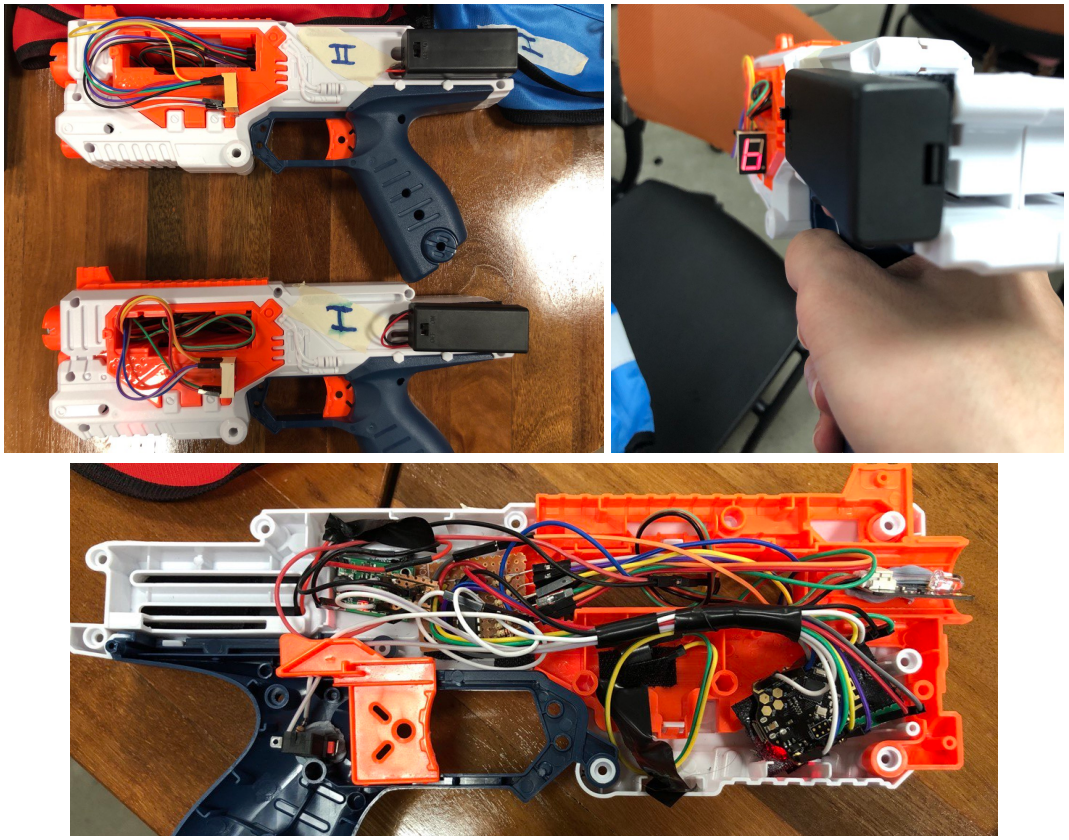


Figure 2.4: System's Hardware Final Form (Gun)

2.3 High-Level Running Process

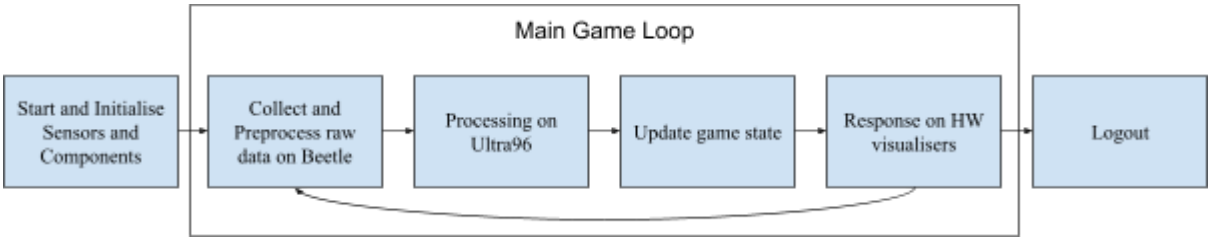


Figure 2.5: System's High-level Running Process

The laser tag game would run in a loop of collecting player data, processing the data, and updating the game state in the following steps:

1. Start and initialise sensors and components
2. Collect and preprocess sensor data
    - Collect raw data from sensors (MPU/push button/IR transmitter and receiver)
    - Preprocess the raw data on the Beetle via sampling/hashing to reduce the amount of data sent to the laptop.
    - Relay processed data from Beetle to laptop via Bluetooth
3. Processing on Ultra96
    - Relay processed data from laptop to Ultra96 via SSH tunnelling
    - Feed data through a pre-trained neural network and output the action taken.
    - Time-series data is gathered by recording the accelerometer and gyroscope readings of the group members doing the specific actions for a specified number of sample frames.
    - Data is split into training and test sets to train and evaluate the performance of the model via a confusion matrix using accuracy and precision.
4. Update game state
    - Game state is updated in the game engine based on the action detected
    - Relay game state to Evaluation Server
    - Relay response from Ultra96 to laptop and laptop to Beetle
5. Output response on Beetle on LEDs and buzzers. (in place of SW visualiser)
6. Repeat steps 1-6
7. Logout

Section 3 Hardware Sensors

This section will explain the different hardware components required for the game and how they will be connected. The handling of different power requirements and appropriate battery pack suggestions will also be included in this section.

3.1 The Components

This section will be explaining the various essential components to be used.

3.1.1 DFROBOT Beetle BLE (DFR0339)

Figure 3.1: The DFROBOT Beetle BLE (DFR0339) [1]

Datasheet: Beetle BLE (DFR0339) Datasheet [1]

Figure 3.1 shows the Beetle BLE that we used to retrieve multiple sensor data and to forward those data to our relay node via bluetooth. The datasheet will be teaching us how to configure and get started. Each player will have 3 Beetle BLE on them. One on the vest, one on the glove and one on the gun.

3.1.2 DFROBOT Digital Infrared Receiver (DFR0094)

Figure 3.2: The DFROBOT Digital Infrared Receiver (DFR0094) [2]

Datasheet: DFROBOT IR Receiver (DFR0094) Datasheet [2]

Figure 3.2 shows the infrared receiver we used. It will be powered by the Beetle BLE (Vest). Its goal is to receive infrared shots from opponents and relay that data to the Beetle BLE (Vest).

3.1.3 Programmable LED Strip (RAINBOW LED 1X5 MODULE WS2812)

Figure 3.3: RAINBOW LED 1X5 MODULE WS2812

Datasheet: RAINBOW LED 1X5 MODULE Datasheet [3]

Figure 3.3 shows the programmable LED we used. It will be powered by the Beetle BLE (Vest). Its goal is to update and show the player's current health points. Blue light will represent health points from 10 to 50 while red light will represent health points from 60 to 100.

3.1.4 LED Strip

Figure 3.4: LED Strip

Datasheet: LED Strip Datasheet (Similar Product) [4]

Figure 3.4 shows the LED we used. It is non-programmable. It will be powered by the Beetle BLE (Vest). Its goal is to light up all at once whenever the player's shield is up.

3.1.5 MPU-6050 (Accelerometer + Gyroscope)

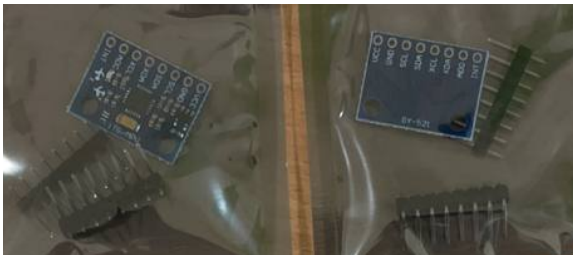


Figure 3.5: The MPU-6050

Datasheet: MPU-6050 Datasheet [5]

Figure 3.5 shows the MPU-6050 which is a 3-axis accelerometer and 3-axis gyroscope module. This module will allow us to collect data on the speed and the angular velocity of the players' hand movements relative to the 3-axis. These data will then help us to determine what action the players are doing. This component will be connected and powered by the Beetle BLE (Glove). Communication between the MPU-6050 and the Beetle BLE will be via I2C.

3.1.6 DFROBOT Digital Infrared Transmitter (DFR0095)

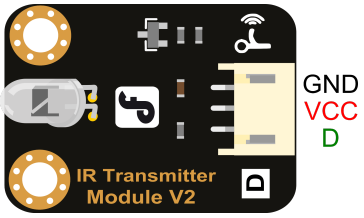


Figure 3.6: The DFROBOT Digital Infrared Transmitter (DFR0095) [6]

Datasheet: DFROBOT IR Transmitter (DFR0095) Datasheet [6]

Figure 3.6 shows the infrared transmitter that the project used. It requires line of sight to the receiver in order for the receiver to receive the signal. This component will be powered by the Beetle BLE (Gun).

3.1.7 Buzzer

Figure 3.7: The buzzer

Datasheet: The Buzzer Datasheet (Similar Product) [7]

Figure 3.7 shows the buzzer that the project used. It will sound off whenever the player shoots, adding an element of realism. This component will be powered by the Beetle BLE (Gun).

3.1.8 Switch

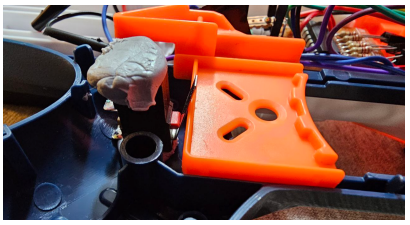


Figure 3.8: The Switch

Figure 3.8 shows the switch that the project used. It is a Normally-Closed-Normally-Open switch. Only the common pin and the Normally Open pin will be used. The switch is retrofitted into the gun, acting as the trigger mechanism between the player and the internal circuits. When the switch is pressed, the gun will then transmit the infrared shot.

3.1.9 BCD-to-Seven-Segment Decoder (Texas Instruments SN74LS47N) and 7-Segment LED

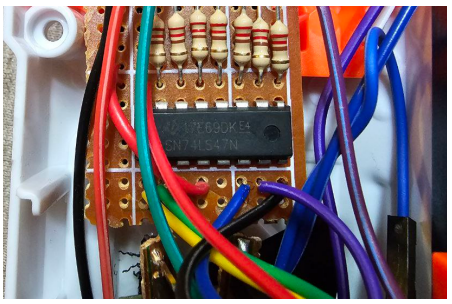


Figure 3.9: The SN74LS47N by Texas Instruments

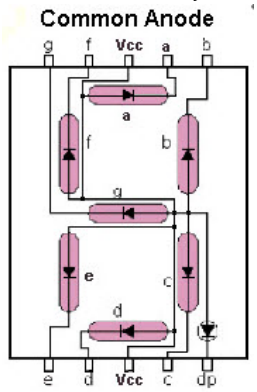


Figure 3.10: The Common Anode 7-Segment LED Display [8]

Datasheet: Texas Instruments SN74LS47N Datasheet [9]

Figure 3.9 and Figure 3.10 shows the Texas Instruments SN74LS47N decoder and the common anode 7-segment LED respectively. The purpose of the 7-segment LED is to show the user how much ammo they have left. The decoder was used as it allows me to use only 4 I/O pins of the beetle BLE instead of 7 to control the LED output. 4 pins are required as numbers 0-9 require 4 bits. Both the LED and the decoder will be powered by the Beetle BLE (Gun).

3.1.10 Battery Holder

Figure 3.11: The Batter Holder

Figure 3.11 shows the battery holder used for the project. It is meant for 2 AAA batteries. It also comes with a switch for easy access and setup. All equipment will be using this battery holder.

3.1.11 DC-DC 3V to 5V Converter

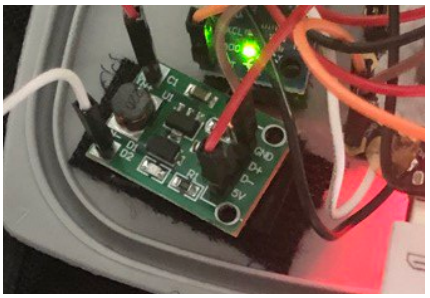


Figure 3.12: The Step Up Converter

Datasheet: DC-DC 3V TO 5V Converter (Same Product but Different Seller) [10]

Figure 3.12 shows the DC-DC 3V to 5V Converter used for the project. As many components require 5V, this is used to step up the voltage of our power supply. 3V is used instead of 5V so as to reduce the number of batteries required, making the system lighter in weight. All equipment will have a converter.

3.2 Pin Tables

The tables below will show how the different Beetle BLEs and components will be connected. All Beetle BLEs will be powered by the step up converter. Pins on the same row are connected together.

3.2.1 Beetle BLE (Vest)

Beetle BLE (Vest) Pins	Infrared Receiver (DFR0094) Pins
D2	DATA

Step Up Converter	Infrared Receiver (DFR0094) Pins
5V	VCC
GND	GND

Beetle BLE (Vest) Pins	LED Strip Pins
D3	VCC

GND	GND

Beetle BLE (Vest) Pins	Programmable LED Strip Pins
D4	DATA

Step Up Converter	Programmable LED Strip Pins
5V	VCC
GND	GND

3.2.2 Beetle BLE (Glove)

Beetle BLE (Glove) Pins	MPU-6050 Pins
5V	VCC
GND	GND
SDA	SDA
SCL	SCL

3.2.3 Beetle BLE (Gun)

Beetle BLE (Gun) Pins	Infrared Transmitter (DFR0095) Pins
D3	DATA

Step Up Converter	Infrared Transmitter (DFR0095) Pins
5V	VCC
GND	GND

Beetle BLE (Gun) Pins	Buzzer Pins
D5	VCC
GND	GND

Beetle BLE (Gun) Pins	Switch (With 200 Ohm Resistor to GND)
5V	Switch Side 1
D4	Switch Side 2 with 200 Ohm Resistor

Beetle BLE (Gun) Pins	SN74LS47N
5V	VCC
GND	GND
A0	A
A1	D
A2	B
A3	C

Beetle BLE (Gun) Pins	7-Segment LED
5V	VCC

SN74LS47N	7-Segment LED
Pins A to G	Pins A to G

For specific location of pins A to G, refer to the datasheet and information given in section 3.1.9.

3.3 Schematics

This section will focus on the schematics, how the devices will be connected to one another.

3.3.1 Beetle BLE (Vest)

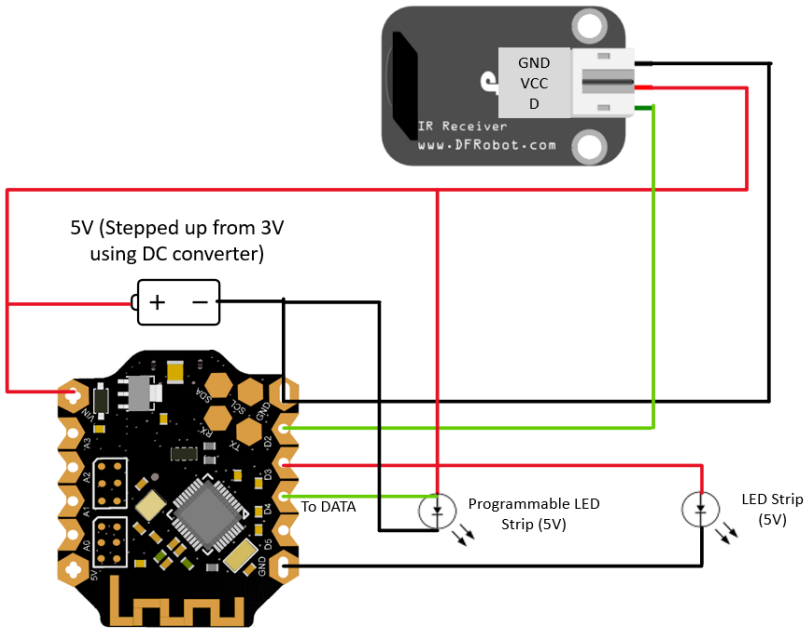


Figure 3.13: The Schematic for Beetle BLE (Vest)

3.3.2 Beetle BLE (Glove)

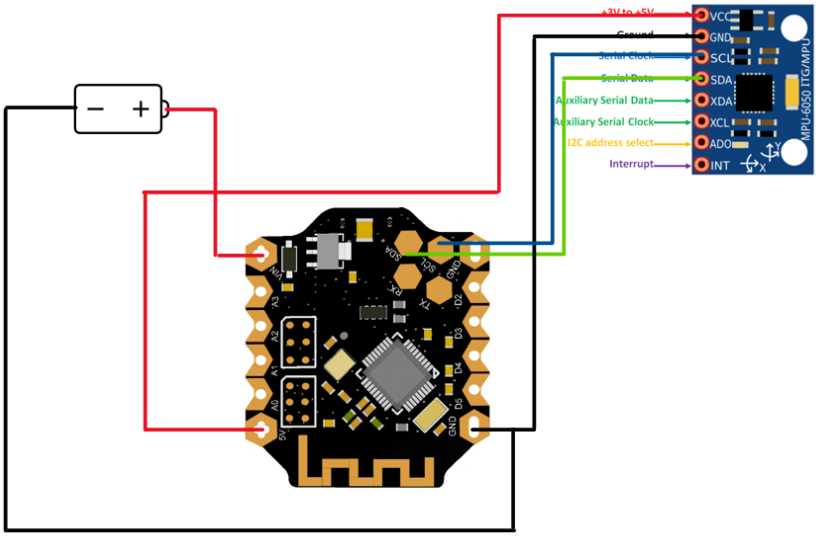


Figure 3.14: The Schematic for Beetle BLE (Glove)

3.3.3 Beetle BLE (Gun)

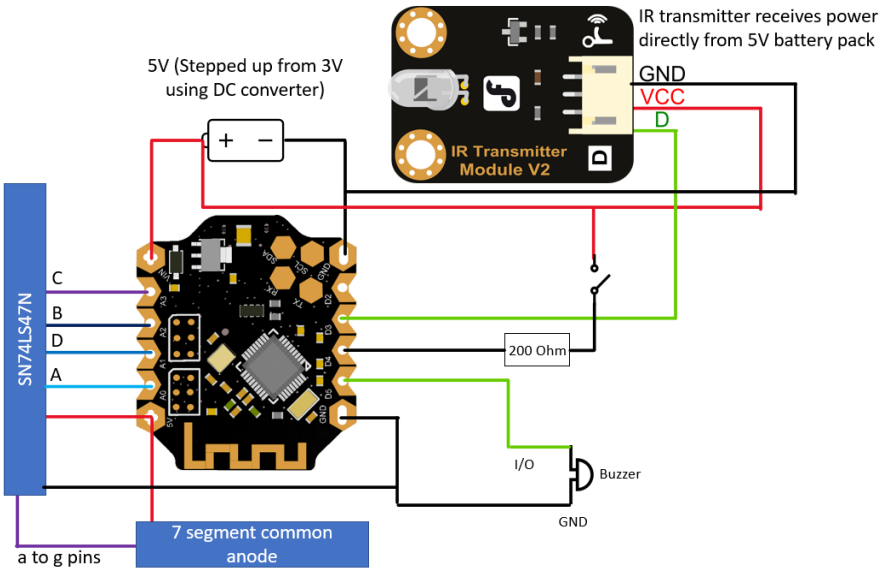


Figure 3.15: The Schematic for Beetle BLE (Gun)

3.4 Operating Voltages and Currents

This section focuses on the power requirements of various components. Battery packs capable of supporting intended loads will also be discussed.

3.4.1 Components' Power Consumption

Component	Operating Voltage	Operating Current	Operating Power
Beetle BLE	5V (From datasheet) [1]	10mA (On average) [11]	5V * 10mA = 0.05W

IR Receiver (DFR0094)	5V (From datasheet) [2]	1.5mA (From datasheet of similar product) [12]	5V * 1.5mA = 0.0075W
Programmable LED (RAINBOW LED 1X5 MODULE WS2812)	5V (From datasheet) [3]	5 * 18mA = 90mA (Because 5 LEDs used) (From datasheet) [3]	5V * 90mA = 0.45W
LED Strip	5V (From datasheet of similar product) [4]	125mA (From datasheet of similar product) [4]	5V * 125mA = 0.625W
MPU-6050	5V (From datasheet) [5]	3.8mA (DMP not used) (From datasheet) [5]	5V * 3.8mA = 0.019W
IR Transmitter	5V (From datasheet) [6]	60mA (From datasheet of similar product) [13]	5V * 60mA = 0.3W
Buzzer	5V (From datasheet of similar product) [7]	30mA (From datasheet of similar product) [7]	5V * 30mA = 0.15W
BCD-to-Seven-Segment Decoder (Texas Instruments SN74LS47N)	5V (From datasheet) [9]	7mA (From datasheet) [9]	0.035W (From datasheet) [9]

3.4.2 Battery Pack Consideration

Battery Pack Location	Total Current Required	Suggested Battery Pack
Vest	Beetle BLE + IR Receiver (DFR0094) + Programmable LED (RAINBOW LED 1X5 MODULE WS2812) + LED Strip = 10mA + 1.5mA + 90mA + 125mA = 226.5mA	2x AAA 1.5V batteries in series With a DC-DC converter, providing constant 5V to system
Glove	Beetle BLE + MPU-6050 = 10mA + 3.8mA = 13.8mA	2x AAA 1.5V batteries in series With a DC-DC converter, providing constant 5V to system

Gun	Beetle BLE + IR Transmitter + Buzzer + BCD-to-Seven-Segment Decoder (Texas Instruments SN74LS47N) = 10mA + 60mA + 30mA + 7mA = 107mA	2x AAA 1.5V batteries in series With a DC-DC converter, providing constant 5V to system

3.4.3 Rational for Suggestion

- Most demanding equipment is the vest. Note that calculations are assuming every component is constantly on but that is not the actual case.
- A DC-DC converter should be used as it is not very practical to have more than 2 batteries for each battery pack. The extra weight might not be very ergonomic.
- A DC-DC converter can provide reliable constant voltage.
- AAA batteries are smaller and lighter.
- AAA batteries typically have a capacity of 1100mAh, allowing it to easily support 1 hour of laser tag gameplay. Theoretically, 2x AAA 1.5V batteries can last about 2.91 hours given the load calculated ($\frac{1100}{226.5 * 5 / 3} = 2.91$). However, that's a very optimistic view and provided that the battery is very reliable.
- 2x AAA batteries will ensure greater reliability and the 3V they provide will ensure compatibility with more DC-DC converters in the market.

3.5 Libraries Used

This section focuses on some of the libraries that can be used when programming each device.

3.5.1 Libraries used for Beetle BLE (Vest)

- IRLib2 by cyborg5. This is the library used to assist us in receiving IR signals. Link: Github Source Code [14]
- FastLED by FastLED. This library allows us to program programmable LED Strips to show health points. Link: Github Source Code [15]

3.5.2 Libraries used for Beetle BLE (Glove)

- MPU6050.h by Jeff Rowberg. This library gave us an easier time in retrieving inertial data. Link: Github Source Code [16]
- I2Cdev.h by Jeff Rowberg. This library allowed us to utilise I2C connection easily. Link: Github Source Code [17]

3.5.3 Libraries used for Beetle BLE (Gun)

- IRLib2 by cyborg5. This is the library used to assist us in sending IR signals. Link: Github Source Code [14]

3.6 Change Made Over the Course of the Semester

3.6.1 Vest

Initially, there were very few visual feedback components added. Only a single LED was used to show that the player had been hit. For single-player game, the single LED was

removed and a Programmable LED strip was used instead to show the player's current health points. A LED strip was also added that lights up whenever the player's shield is up. For subsequent evaluations, the same setup was used. There was also a clash between the FastLED library and the IRremote library as they happen to use the same timers, resulting in me using IRLib2 instead of IRremote. Vest final form can be seen in Figure 2.2.

3.6.2 Glove

A container is used to house all the components together. Initially, I decided to stick the container to the glove directly using double sided tape. However, upon testing, I realised that the flexing of the hand can result in IMU readings spiking. This was due to the flexible container lid and the close proximity of the IMU chip to the hand. As a result, from single-player game onwards, a styrofoam tape is used to create a gap between the chip and the hand. Refer to Figure 2.3 for the addition of styrofoam tape.

For single-player game, yaw, pitch, and row data was used to measure the players' hand movement. However, we realised that yaw, pitch, and row data are heavily impacted by the players' orientation, making it hard to obtain data for training and testing. 6-axis values were then used instead for 2-player game onwards.

From 1-player game onwards, the first layer of thresholding is also implemented in the Beetle BLE to detect a start of move. Glove final form can be seen in Figure 2.3.

3.6.3 Gun

Initially, there is only a buzzer to let the player know a shot has been fired. The integration of components for the gun was also not ideal. All the components were placed outside the gun. After the first progress check, I decided to place all the components inside the gun so as to protect the components and to make the gun more presentable. A better switch was also used for the trigger. After the first progress check, I also decided to have the guns emit distinct IR signals. This is so that there will not be any friendly fire and interference from external sources. Before single-player game, a 7-segment LED is also added to show the user their ammo count. Same gun setup was used for subsequent games. Gun final form can be seen in Figure 2.4.

3.7 Lessons Learnt

- It is important to ensure libraries do not have conflicts. I did not look out for library conflicts during planning and implementation, resulting in time wasted in bug finding and solution finding. The specific libraries that clashed were IRremote and FastLED where they share the same timers. An alternate IR library, IRLib2 was used.
- There were times where equipment would fail all of a sudden. For example, a wire will detach from the solder point. Such experiences have led me to understand the importance of maintenance and that it is something that has to be done.
- Being able to think out of the box and come up with effective solutions is very important as it led me to solve the IMU issue mentioned above, where flexing of the glove led to IMU data spiking.
- I could have done a more thorough research on components to buy. For example, there exist battery holders with built in DC step up converters. Such components could have come in handy for me as it can reduce my implementation time.
- Integration of hardware and internal comms was interesting as I got to know partially how internal comms work. However, it adds an extra layer of debugging to do.

- While setting up the switch trigger for the gun, it was important to do switch debouncing. Also the digital pin used to read switch input should be set to INPUT_PULLUP, making use of the chip's internal resistor and 5V supply. This is to prevent floating states and the gun randomly triggering.

Section 4: Hardware AI

This section on Hardware AI details the development process of a neural network that classifies the hand gesture actions of a player using data collected from an IMU sensor on their hand, its implementation in a Python environment, and deployment of the model on the FPGA on the Ultra96 board.

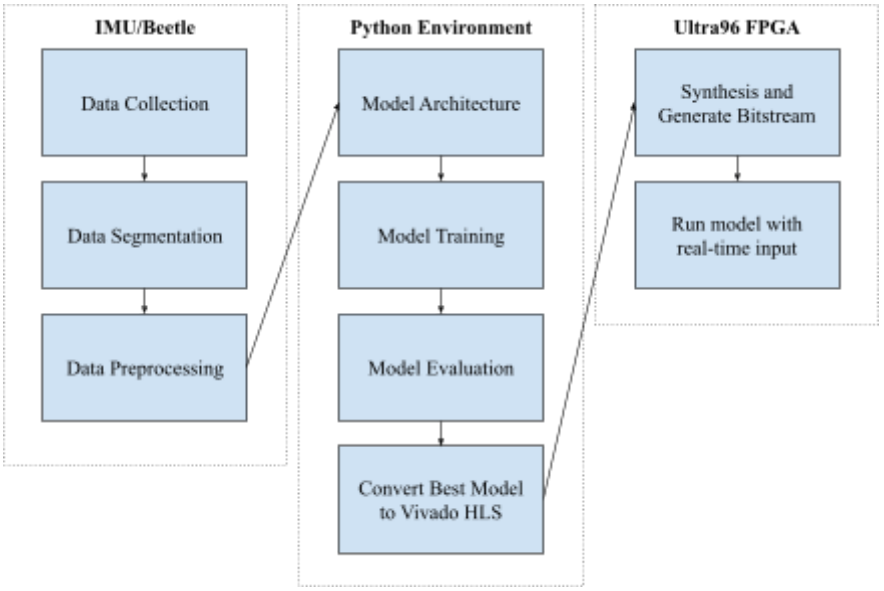


Figure 4.1 - Hardware AI Development Process

4.1 Data Pipeline

4.1.1 Data Collection

Raw data is collected from an MPU sensor mounted on the glove of the players. The MPU contains an accelerometer and gyroscope that can measure the time-series acceleration and angular velocity of the player's movement in three axes each [18]. This data captures the characteristics of a movement corresponding to a (or lack of) player action.

There are 4 possible actions: Shield, Reload, Grenade, and Logout. There is also a state where no actions are being performed, Idle. For good model performance, a large amount of varied data was collected for each action. This was achieved by measuring each movement forty times per person, with different speeds, angles, and with different people, which gave a robust dataset to train the model.

There should be an equal number of samples recorded for each class (4 action states and 1 inaction state) such that the data for training the model is balanced, as imbalanced data may negatively affect the performance of classification models which may be inherently biassed to a majority class [19].

4.1.2 Data Segmentation

For the transmission of the data packets, as well as the structuring of input parameters into the model, the real-time raw data was divided into segments of constant size, based on time

intervals. This segment should not be too short as it may not capture an action being fully performed. It should also be not too long as it would increase the size of the data packet, or be able to capture multiple actions.

Segments may be in the order of hundreds of milliseconds based on the average time taken to complete an action. However, considering the varied speeds that an action could be performed in different conditions or by different players, the length of this segment was finalised to be 25 frames at 25Hz, corresponding to 1 second.

4.1.3 Data Preprocessing

Raw data collected from the MPU sensors may be noisy due to vibrations. Preprocessing would be useful in removing the noise through filtering, masking, or spectral analysis. Furthermore, preprocessing may be needed for feature engineering that may simplify data transformations and improve model accuracy [20].

An initial movement threshold was on the Beetle, such that when the average of any of the 6 movement indicators of the 10 frames exceeded the average of the next 10 frames, the subsequent 25 frames would be sent to the Ultra96 for prediction.

There was a secondary thresholding on the Ultra96, that calculates the sum of absolute differences in the z-axis acceleration between each adjacent frame in the 25-frame packet. If the sum exceeds the threshold, the packet will be sent for classification. This helps to filter out 'tail' actions, where the primary threshold was exceeded on the Beetle due to it recording the tail end of a preceding action.

The data was then normalised to adjust for values on different scales and bring it closer to a normal distribution, which improved model performance.

4.2 Model

4.2.1 Model Architecture

The classification model was a fully connected feedforward multilayer neural network. This is one of the simplest neural network architectures, which would simplify the conversion and deployment of the model on the FPGA.

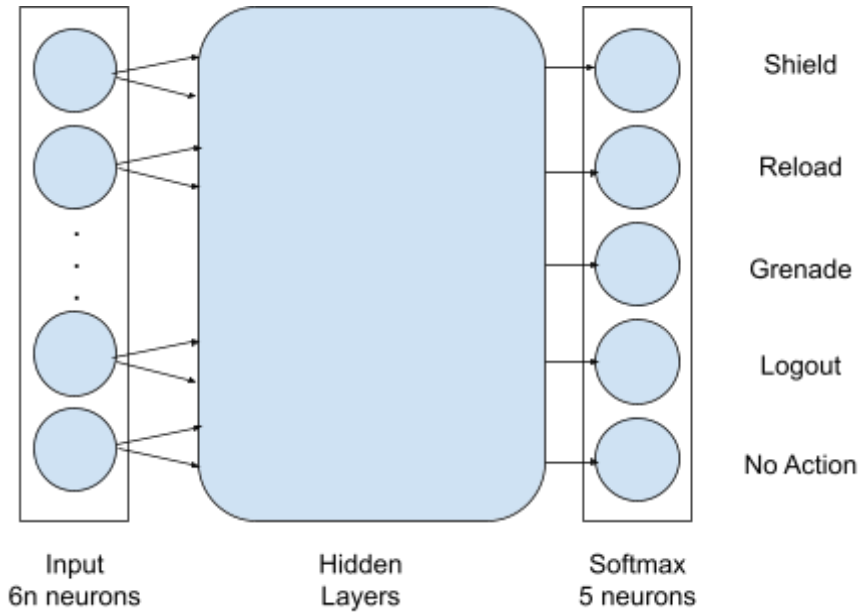


Figure 4.2 - Model Architecture

The input layer consists of 6*25 neurons, where 25 is the number of samples in one time segment, that would consist of six input parameters in each segment from the 3-axis data from both the accelerometer and gyroscope. The input data was flattened before feeding it into the neural network.

The model consists of two hidden layers, with 64 and 32 neurons respectively. ReLU activation function was used for both layers to overcome possibilities of vanishing gradients

The output layer consists of a dense layer with 5 neurons with softmax activation corresponding to the classes: Shield, Reload, Grenade throw, Logout, and No action. It converts the vector of weights going into the layer into a vector of probabilities that are proportional to the relative scale of each class. Then, argmax was used to determine the class with the highest probability, which would be the predicted output action from the model.

The model was implemented using the Keras Python package, which provides functions and abstractions to easily build, compile, and train the model.

4.2.2 Model Training

After the time-series accelerometer and gyroscope data was collected, the dataset was split into a training and testing set in the ratio 80:20.

Next, the model was defined using Keras layers using the input and output layers specified in the architecture along with two Dense layers. The final model was trained for 10 epochs.

4.2.3 Model Evaluation

The model was evaluated using the test data by comparing the accuracy and precision of its predicted classifications. A confusion matrix was generated to understand the classification patterns of the model.

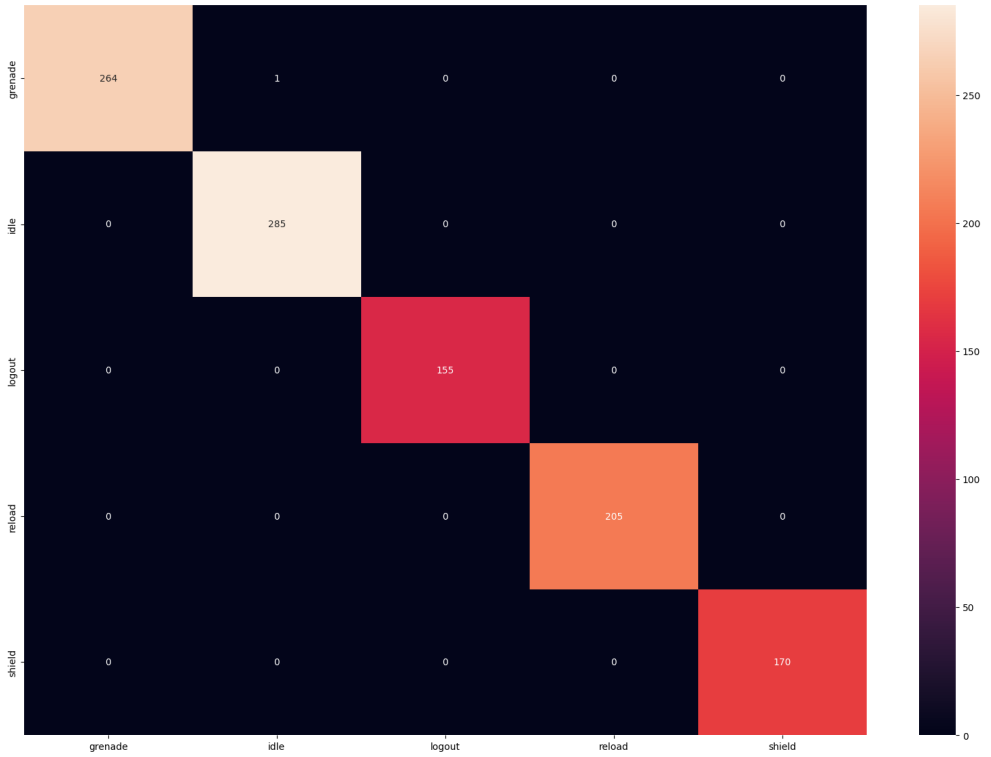


Figure 4.3 - Confusion Matrix of final model

4.3 Ultra96 FPGA Deployment

4.3.1 Realising the Neural Network

The final trained neural network was transferred to the FPGA by manually copying the weights and biases of each layer and connecting them in a C++ script. This script was used as the source file in the Vivado HLS. Directives such as loop unrolling were set to optimise the latency of the program.

During the conversion, Vivado HLS will run the following steps:

1. C simulation - Verify C program implements the Keras model correctly
2. C/RTL synthesis - Synthesise RTL of C program
3. C/RTL co-simulation - Verify RTL corresponds to C program
4. Export IP - Export HLS IP block.

These steps ensure that the model is correctly synthesised on the FPGA, retaining almost the exact same classification behaviour. It would export a HLS IP block which can be directly imported onto Vivado, where a bitstream would be generated to allow the model to run.

In Vivado, the IP block was connected via a Direct Memory Access (DMA) module to the main processor, allowing data to transfer between them.

4.3.2 Run model with real-time input

An Application Programming Interface (API) was developed on the Ultra96 with two functions, init() and predict(), which would initialise the bitstream and DMA block as well as run the neural network prediction respectively.

The driver program on the Ultra96 that receives the data packets from the Beatles passes them as input to the model after normalising the data with the fitted scaler during training.

The predicted action is relayed to the game engine, where its internal data and logic can be updated.

4.3.3 Evaluation of design timing, power, and area

To evaluate the performance of the model, we need to consider the time taken for a forward pass prediction, the power used, and the area of the FPGA used in terms of its hardware resources (Flip Flops, Look Up Tables, RAM).

Timings can be determined by tracking on the Ultra96, the time taken from what input data is passed to the model, and when the output classification is received. More detailed timing characteristics may also be found in Vivado's "Report Timing Summary" [21].

Vivado can estimate power at various stages of design from synthesis onwards using the "Report Power" function [22]. This was useful in determining the power distribution and loads across the components on the board, ensuring it does not exceed power thresholds.

4.4 Developments, Issues Faced and Solutions

Initially, it was unclear what would be the most efficient way to deploy the neural network model on the FPGA. Exploration using the hls4ml python package to convert the keras model to HLS was abandoned after feedback from peers and tutors about possible bugs integrating with the Ultra96 system, which meant reverting back to the manual specification of the model in a C++ file. This later proved helpful as defining the neural network in C++ allowed us to interface with the ap_axi_sdata.h and hls_stream.h header files, giving us the option to define FPGA I/O manually, which was important to get DMA communications working.

Another issue we faced was in our data processing, specifically how we chose to use the yaw pitch and roll derived from the measurements from the IMU sensor, instead of the 6-dimension raw accelerometer and gyroscope data. It was initially assumed that these new features can encompass the movement information fully in fewer dimensions, reducing the communication workload between the Beetles and the relay nodes. However, prediction accuracy with it was poor, in large part due to data fluctuating depending on the user's directional orientation in space (sensors give very different readings for the same action if the user is facing a different direction). Ultimately, we reverted back to using the 6 raw dimensions of data, albeit with a slower data transfer frequency. Nonetheless, accuracy was much higher, even with the reduced frequency.

Section 5 Internal Communications

This section on Internal Communications outlines how communication was established and maintained between the Arduino Beetles and the laptop. Evolution from the initial subsystem, task/thread management on the Beetles/laptop, BLE communication protocol as well as handling reliability issues will be covered.

5.1 Task & Thread Management

5.1.1 Task Management on Arduino Beetles

The table below details the tasks that the Beetles would be responsible for, relevant for Beetle-laptop communication.

Task	Beetle	Hardware Sensor	Description
sendACK()	All	-	Sends ACK in response to packet sent from laptop for handshaking
resetBeetle()	All	-	Resets Beetle programmatically upon receiving 'R' packet from laptop; connection timeout mechanism
sendGun()	Gun	IR Transmitter	Sends when GUN shoots and transmits IR
sendVest()	Vest	IR Receiver	Sends when VEST receives IR shot
sendMotionData()	Glove	MPU-6050	Sends IMU data
displayAmmo(...)	Gun	-	Shows remaining bullets from updated game state on 7-segment display upon receiving 'U' packet from laptop
Player Health LEDs	Vest	-	Shows player health on LED strip upon receiving 'U' packet from laptop
Shield LED	Vest	-	Shows shield being active on white LED upon receiving 'U' packet from laptop

5.1.2 Thread management on Laptop

For concurrency to be achieved, the laptop will have to connect and maintain communication with 3 Beetles for each player simultaneously. We implemented multithreading with the use of Python's **threading** module - for thread creation and management. There can be one thread dedicated to each Beetle for sending and receiving data to/from the laptop. The Beetles associated with the IR receiver and gun trigger sensor would not be required to constantly send data due to the periodic nature of getting shot by another player and shooting at another player. On the other hand, the Beetle with the MPU would need to be sending data more frequently and in greater size.

5.2 Beetles - Laptop communication protocol

5.2.1 Handshaking

A 3-way handshake protocol is executed whenever a connection is being established between a Beetle and the relay laptop. Firstly, the laptop sends a 'H' packet to Beetle. Secondly, the Beetle receives said 'H' packet and sends an ACK packet in response. These two steps signal that both devices are ready to transmit/receive data. Thirdly, the laptop sends its own ACK packet to confirm that the handshaking is complete. Thereafter, (sensor)/(game state) data transfer can take place between them.

5.2.2 Packet Types

Packet Type	**Packet Identifier**	**Description**
Handshake	'H'	Sent from laptop to Beetle to initiate 3-way handshake
ACK	'A'	Can be sent by either laptop or Beetle to acknowledge packets being received
Updated Game State	'U'	Sent from laptop to GUN or VEST Beetle containing information such as remaining bullets, player health, shield health from the updated game state
Gun Data	'G'	Sent from GUN Beetle to laptop to indicate IR transmitter has shot
Vest Data	'V'	Sent from VEST Beetle to laptop to indicate IR receiver has been hit
Motion Data	'M'	Sent from GLOVE Beetle to laptop containing IMU Data
Reset	'R'	Sent from laptop to Beetle to trigger reset function in Beetle after 10 H packets receive no ACK response

5.2.3 Packet Format

In BLE communication, the portion of the data packets we are concerned with that defines the application data is the ATT Payload. In Bluetooth 4.0 which the Beetle uses, the maximum ATT Payload is 20 bytes [23]. For simplicity and consistency, we fixed the size of all data packets we send to be 20 bytes. We can include the packet identifier, sensor data relevant to equipment the Beetle is attached to and packet CRC checksum. If not all 20 bytes are utilised, the remaining bits are filled with padding. In the Arduino code below, an ACK packet consists of 1 byte packet identifier, 18 bytes of padding and 1 byte of CRC checksum.

```
66 void sendACK()
67 {
68   crc.restart();
69   handshake_data.packetType = 'A';
70   Serial.write(handshake_data.packetType);
71   crc.add(handshake_data.packetType);
72   makePadding(18);
73   handshake_data.checksum = crc.getCRC();
74   Serial.write(handshake_data.checksum);
75   Serial.flush();
76 }
```

5.3 Handling reliability issues

5.3.1 Packet Fragmentation

During integration testing, data packets from the GLOVE Beetle occasionally faced data fragmentation. This was due to the transmitting frequencies experimented with being very high (> 50Hz). Once a data packet is fragmented, it fragments up successive packets and leads to packet corruption.On the relay laptop, incoming packets are added to the receiving buffer. If the filled buffer is shorter than 20 bytes, it is recognised as a fragmented packet. It then waits for another data packet to arrive and get added to the buffer till the latter is 20 bytes full. The 20 bytes long packet is then sent for further processing and the buffer is cleared.

```
        self.buffer += raw_packet

        if len(self.buffer) < 20:
            COUNT_FRAG_PKT[self.macAddress] += 1
            # print('PKT FRAGMENTED:', COUNT_FRAG_PKT)

        else:
            # send non-fragmented (full) packet for processing
            self.manage_packet_data(self.buffer)
            # clear buffer to prepare for next packet
            self.buffer = self.buffer[20:]
```

Secondly, the MPU normally prints out some initialising statements on the serial monitor. Given that transmit/receive of packets were based on serial communication, the statements ended up being sent in packets to the laptop. We could only reduce it to one particular statement “>...>...” on the serial monitor. On the relay laptop, this issue was handled by recognising the statement and clearing it from the receiving buffer as seen below.

```
        # clear MPU initialising printing
        if b'>.' or b'..' in self.buffer:
            self.buffer = self.buffer.replace(b'>.', b'')
            self.buffer = self.buffer.replace(b'..', b'')
```

5.3.2 Packet Corruption

We used Cyclic Redundancy Check (CRC) functionality to check data packets being received by the laptop from the Beetles for corruption. On the Beetles, CRC checksum is computed for every data packet being sent to the laptop. On the relay laptop, packets being received are sifted through CRC check as seen below. The check independently computes CRC checksum for the packet and compares it to that in the packet. If the check returns true, the packet is allowed to proceed for further processing. Otherwise, the packet is corrupted and dropped.

```
def crcCheck(self, full_packet):
    checksum = Crc8.calc(full_packet[0:19])
    if checksum == full_packet[19]:
        return True
    return False
```

5.3.3 Potential Disconnections

The Bluetooth connections between the Beetles and the laptop may fail due to various factors such as being out of range, interference from other Bluetooth devices, low battery, etc [24]. For a start, given that the Beetles/laptop are sufficiently powered and Bluetooth connection is broken, the Beetle would send out advertising packets. Connection will be restored if the laptop is able to receive them. The Beetles would be ready to transmit sensor data once the 3-way handshake has been completed.

In addition, we implemented a connection timeout on the relay laptop. If up to 10 'H' packets have been sent and no ACK packet was received to continue and complete handshaking, the laptop will send an 'R' packet to command the Beetle to reset itself as seen below. This causes a forced disconnect and calls the connection method to start again.

```
def reset(self):
    padding = (0,) * 19
    packetFormat = 'c' + 19 * 'B'  # 1 char & 19 unsigned char
    resetByte = bytes('R', 'utf-8')
    packet = struct.pack(packetFormat, resetByte, *padding)
    self.serial_chars.write(packet, withResponse=False)
    BTL_RESET_STATUS[self.peripheral_obj_beetle.addr] = False
    self.connect()
```

In order to minimise disconnections due to range, we would position the laptop strategically outside the arena perimeter.

5.4 Developments, Issues Faced and Solutions

There were three major changes to the internal communications between the initial subsystem and 1-player game. Firstly, integration had to be done with external communications and hardware sensors. The former was simply merging the external and internal comms code into a single Python file. The latter was more rigorous as the transmission and receiving of sensor data had to be real and correct. This involved a great amount of time spent testing and debugging.

Secondly, we implemented the sending of updated game state to Beetles to update their hardware visual indicators - seven-segment display (bullets count), white LED (shield activation) and red & blue LED strip (player health). This added more time spent on testing and debugging of hardware-internal communications.

Lastly, a cleaner and more efficient way to collect motion data for training was made. Previously, motion data was copied from the serial monitor of Arduino IDE running the GLOVE Beetle code (which required wired connection to the laptop) into a CSV file. In place of this, we made use of the relay laptop code. When the code is run, the GLOVE can be connected to the relay via Bluetooth (thus not needing a wired connection). It also creates a CSV file and writes motion data into it when packets from the GLOVE are received and unpacked. The data collection process was thus significantly improved.

Between 1-player game to 2-player game with unseen players, there was no structural change made to the internal communications of our system.

Section 6 External Communications

This section on External Communications describes how communication between the various subsystems (Ultra96, Evaluation Server, Relay Laptops) was achieved. In particular, it details the communication protocol, encryption protocol, message formats and library APIs that were used. Concurrency on the Relay Laptops and Ultra96 is also discussed.

6.1 Communication between Ultra96 and Evaluation Server

Communication between the Ultra96 and the Evaluation Server was established through the Transmission Control Protocol (TCP). The communication was also secured by encrypting the messages using the Advanced Encryption Standard (AES).

6.1.1 Transmission Control Protocol (TCP)

TCP is one of the main protocols in the Internet Protocol Suite, providing a connection-oriented, reliable and ordered delivery of a stream of bytes between processes running on hosts communicating via an IP network [25]. It also provides Flow Control and Congestion Control. While the alternative User Datagram Protocol (UDP) has a smaller overhead and lower latency due to being a connectionless protocol, it sacrifices reliability and the guarantee of in-order data delivery. In the context of our Laser Tag game, whilst latency can improve user experience, ensuring that all player actions are registered and that the game state is updated correctly should have utmost priority. Hence, TCP was chosen over UDP.

TCP can be implemented using the **socket** module in the Python Standard Library, as shown in the following sequence of API calls:

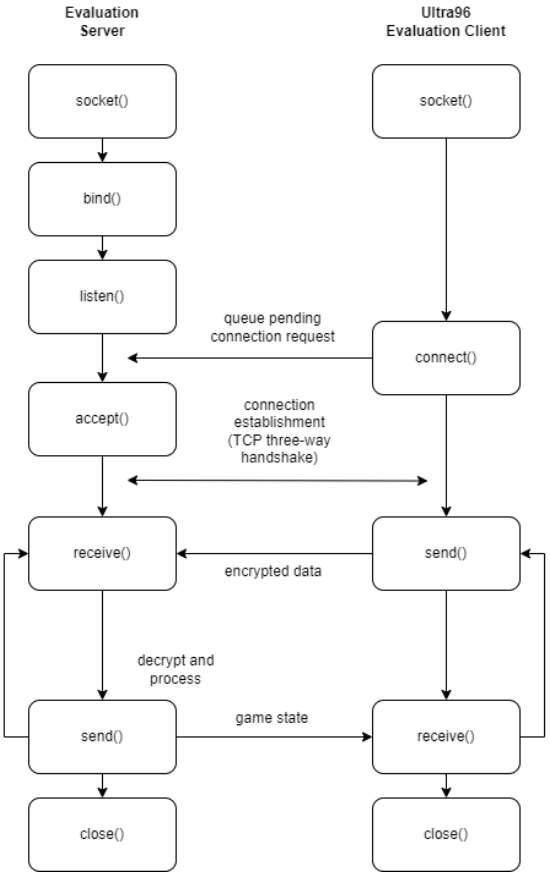


Figure 6.1: Sequence of API calls and Data Flow for TCP

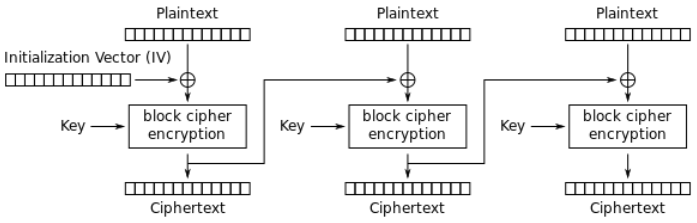
As shown in the TCP connection flow, the Evaluation Server and Ultra96 Client first both create a stream socket by calling socket(). The server then binds its socket to a local address and listens to wait for any connection request from the client side. The listen() call converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. The client tries to establish a connection to the server by calling connect() and initiating TCP's three-way handshake. The connection request is stored in a connection queue on the server side while the server calls accept() to extract the first pending connection request from the connection request queue of the listening socket [26]. After the corresponding request from the client has been accepted, the server creates and returns a new connection socket where data is exchanged between the Ultra96 and Evaluation Server using send() and receive() calls.

During each turn, the Ultra96 Client waits for the Game Engine to commit before sending the game state to the Evaluation Server. Meanwhile, the Evaluation Server performs a blocking recv() for data from the Ultra96. After receiving and decrypting data from the Ultra96, the Evaluation Server sends back the correct game state before setting up the next turn. The Ultra96 Client then receives this correct game state and updates its local game state if necessary. After the Laser Tag game ends, the Ultra96 Client and Evaluation Server shut down and close() their respective sockets.

6.1.2 Encryption using Advanced Encryption Standard (AES)

To ensure secure communication with data packets being sent over the network, messages between the Ultra96 and Evaluation Server should be encrypted so as to preserve confidentiality. Advanced Encryption Standard (AES), a symmetric-key block cipher in Cipher Block Chaining (CBC) mode can be used to encrypt the messages. AES is supported by the **PyCryptodome** library. The steps taken to encrypt and decrypt messages are detailed in the following pseudocode:

```
def encrypt_message(self, game_state_dict):
    secret_key = bytes(str(self.secret_key), encoding="utf-8")
    plaintext = json.dumps(game_state_dict).encode("utf-8")
    plaintext = pad(plaintext, AES.block_size)
    cipher = AES.new(secret_key, AES.MODE_CBC)
    iv = cipher.iv
    encrypted_message = cipher.encrypt(plaintext)
    cipher_text = base64.b64encode(iv + encrypted_message)
    return cipher_text
```

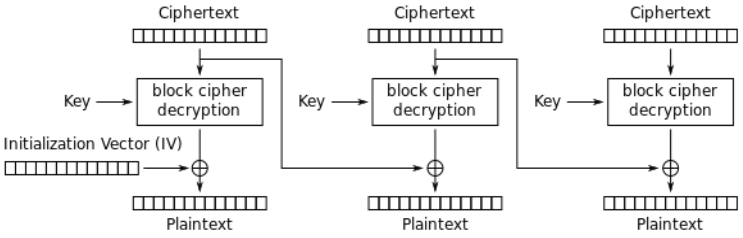


$$C_i = E_K(P_i \oplus C_{i-1}),$$
$$C_0 = IV,$$

Figure 6.2: AES CBC mode Encryption

For encryption in CBC mode, each block of plaintext P_i is XORed with the previous ciphertext block C_{i-1} before being encrypted with the symmetric secret key K. Thus, each ciphertext block depends on all plaintext blocks processed up to that point. An unpredictable Initialization Vector (IV) must also be used to make each message unique and prevent attacks on AES CBC mode such as the BEAST Attack [27]. The IV will be of AES block size (16 bytes) and appended to the front of the encrypted message as C_0 before the entire ciphertext is encoded in base64 and sent to the Evaluation Server. As AES is a block cipher, the plaintext also has to be padded to become a multiple of block size before encryption.

```python
def decrypt_message(self, cipher_text):
    decoded_message = base64.b64decode(cipher_text)
    iv = decoded_message[:AES.block_size]
    secret_key = bytes(str(self.secret_key), encoding="utf8")
    cipher = AES.new(secret_key, AES.MODE_CBC, iv)
    decrypted_message = cipher.decrypt(decoded_message[AES.block_size:])
    decrypted_message = unpad(decrypted_message, AES.block_size)
    decrypted_message = decrypted_message.decode('utf8')
```



Cipher Block Chaining (CBC) mode decryption

$$P_i = D_K(C_i) \oplus C_{i-1},$$
$$C_0 = IV.$$

Figure 6.3: AES CBC mode Decryption

When the ciphertext from the Ultra96 Client is received, it has to be decoded from base64 to bytes before extracting the IV from the first 16 bytes. For decryption in CBC mode, each block of ciphertext C_i is decrypted with the symmetric secret key K first to produce P_i XOR C_{i-1} (reverse of encryption) before being XORed with the previous ciphertext block C_{i-1} to get back plaintext block P_i. The original message sent from the Ultra96 can then be recovered by unpadding from the last block.

6.1.3 Message Format between Ultra96 and Evaluation Server

Messages sent between the Ultra96 and Evaluation Server are stringified JSON objects with the following format:

JSON Format:
{
"p1": player_1_state_dict,
"p2": player_2_state_dict
}

player_X_state_dict:
{

"hp":	integer value of current player health,
"action":	string representing the current action performed by the player taking values from "grenade, reload, shoot, logout, shield",
"bullets":	integer value of number of bullets left in the magazine,
"grenades":	integer value of number of grenades left,
"shield_time":	integer value of number of seconds remaining in the shield,
"shield_health":	integer value of amount damage the shield can take,
"num_deaths":	integer value of number of times the player died,
"num_shield":	integer value of number of shield left

}

The Packet Format expected by the Evaluation Server is in the form Len_crypt(JSON) where:
1) Len is the size of crypt(JSON) in bytes
2) Len and crypt(JSON) are separated by an underscore "_"
3) crypt(JSON) is the game state JSON encrypted using AES CBC mode encoded in base64, in a utf-8 format

After receiving and decrypting the game state JSON from the Ultra96 Client, the Evaluation Server sends back the correct game state JSON so the game state on the Ultra96 can be updated if necessary.

The Packet Format sent by the Evaluation Server is in the form Len_JSON where:
1) Len is the size of JSON in bytes
2) Len and JSON are separated by an underscore "_"
3) JSON is the correct game state encoded in a utf-8 format

6.2 Communication between Relay Laptops and Ultra96

Communication between the Relay Laptops and the Ultra96 was also established through TCP as discussed in Section 6.1.1. However, as the Ultra96 sits within the NUS firewall, SSH Tunneling was required to bypass the firewall to allow the laptops to communicate with the Ultra96.

6.2.1 SSH Tunneling

SSH Tunneling allows us to bypass firewalls by port forwarding. With local port forwarding, connections from an SSH client are forwarded via an SSH server to a destination server. The SSH client listens on a configured port and directs all data sent to that port through a secure tunnel to the SSH server [28]. The server then redirects it to the destination host and port.

As the Ultra96 sits within the NUS network and can only be accessed through its port 22, two SSH tunnels are required. One to ssh into stu.comp.nus.edu.sg to bypass the NUS firewall and another to ssh onto the Ultra96. The first SSH tunnel is established from the laptop to port 22 on the Ultra96 using stu.comp.nus.edu.sg as the remote gateway node, while the second SSH tunnel is established from the laptop to the destination port on the Ultra96 via the Ultra96 port 22 through the first SSH tunnel [29].
SSH Tunneling can be implemented using the **sshtunnel** Python library, as shown in the following pseudocode:

```
def open_tunnel(self):
    tunnel_one = sshtunnel.open_tunnel(
        ssh_address_or_host=('stu.comp.nus.edu.sg', 22),
        remote_bind_address=(self.ultra96_ip_address, 22),
        ssh_username=self.stu_username,
        ssh_password=self.stu_password,
        block_on_close=False
    )
    tunnel_one.start()

    tunnel_two = sshtunnel.open_tunnel(
        ssh_address_or_host=('localhost', tunnel_one.local_bind_port),
        remote_bind_address=('localhost', self.ultra96_port_num),
        ssh_username='xilinx',
        ssh_password='whoneedsvisualizer',
        block_on_close=False
    )
    tunnel_two.start()
```

Figure 6.4: SSH Tunneling to Ultra96

6.2.2 Message Format between Relay Laptops and Ultra96

Messages sent between Relay Laptops and Ultra96 are stringified JSON objects with the following format:

JSON Format:
{
"player_num": integer value of player number (1 or 2),
"data_type": string representation of type of data being sent
taking values from “IMU, GUN, VEST, TIMEOUT”,
"data_value": array of 6 float values if data_type is "IMU"
}

If data_type is "IMU", the data_value field will contain the raw data collected from the MPU sensor mounted on the glove of the player. The data would be in the form of an array with 6 float values, representing the accelerometer and gyroscope values in three axes.

The Packet Format sent by the Relay Laptops is in the form Len_JSON where:
1) Len is the size of JSON in bytes
2) Len and JSON are separated by an underscore “_”
3) JSON is encoded in a utf-8 format

6.3 Concurrency on Relay Laptops and Ultra96

In our Laser Tag game, multiple communication channels have to be established. For each player, the Relay Laptop has to establish Bluetooth connections with 3 Beetles each. The Ultra96 also has to receive data from each Relay Laptop through TCP sockets. Thus, multithreading is necessary to allow concurrent communication [31]. Multithreading is supported by the **threading** library in Python.

On the Relay Laptop, each Beetle has its own thread which handles both sending and receiving of data. Hence there are 3 threads, 1 for each of the 3 Beetles on the Gun, Vest and Glove respectively. However, for the TCP connection with the Ultra96, there are 2 separate threads for sending and receiving data. The rationale behind separating send and receive into 2 threads will be detailed in the next section 6.4. Whenever a Beetle thread receives sensor data from its respective Beetle and hardware, it would put that data onto the sensor_data_queue to Ultra96 in the format specified in Section 6.2.2. In send thread on the Ultra96 Client, if the sensor_data_queue is not empty, the sensor_data_dict is popped from the queue and sent to the Ultra96. In the receive thread on the Ultra96 Client, whenever it receives an updated_game_state propagated from the Eval Server, the updated_game_state is put onto the game_state_queue to be sent to the Beetles.

On the Ultra96, each TCP connection from the Relay Laptops has its own dedicated thread to handle both sending and receiving of data. The TCP connection with the Eval Server also has its own dedicated thread to handle both sending and receiving of data. Data received from the Relay Laptops is checked for its data_type before being pushed onto its respective queue or sent to FPGA for predictions. When the data is processed and the game state is committed, the Eval Client thread sends this game state to the Eval Server and waits to receive the updated game state. The updated game state is pushed onto a queue to be sent back to the Relay Laptops. Queues are used as they are thread-safe and can ensure that data is processed sequentially and that the game state is updated correctly. Other threads on the Ultra96 for the FPGA and Game Engine were also considered and will be discussed later in Section 7.2.

6.4 Developments, Issues Faced and Solutions

Overall, communications between the various subsystems and their message formats did not undergo significant developments from the individual subsystem to the single-player game to the two-player games. However, changes to threading were made to improve latency.

Initially, for two-way communication between the Relay Laptops and the Ultra96, send() and recv() were kept in the same thread on the Relay Laptop Client. This follows the same send() then recv(), recv() then send() strategy used between the Ultra96 and Eval Server. However, this meant that whenever a data packet is received on the Ultra96 Server thread, a game state has to be sent back, otherwise recv() would be blocking on the Relay Laptop Client thread and subsequent data cannot be sent. The solution was to add an additional boolean "updated" field to game states sent back. If the updated_game_state queue from the Eval Server is not empty, send back the updated_game_state with "updated"=True, else send "updated"=False. On the Relay Laptop Client, ignore data packets with "updated"=False and only put game states with "updated"=True onto the queue to be sent back to the Beetles. However, sending back the game state for each data packet received increases latency, especially when sending large streams of IMU data.

Instead, for the single-player game onwards, game states are only sent back to the Relay Laptops when the updated game state is received from the Eval Server. Since recv() is blocking, separate threads are used for send() and recv() on the Relay Laptop Client. But another issue arises when the Relay Laptop does not send data continuously (no Shoot, Hit or IMU values does not exceed threshold on hardware between turns). Updated game states can only be sent back to the Relay Laptops and Hardware after the Ultra96 receives the next data packet, which might result in considerable delay for the players, especially since our group does not have a Visualizer. If send() and recv() are also separated on two threads on the Ultra96, polling on the receive thread to check if the updated_game_state queue is empty will increase latency of other threads. An easier solution would be to send a Timeout Packet from the Relay Laptop if nothing is sent within a certain time. This can be implemented using threading.Timer() as shown below:

```
def run(self):
    recv_thread = threading.Thread(target=self.receive_updated_game_state)
    recv_thread.start()
    while not (global_shutdown.is_set() or self.shutdown.is_set()):
        try:
            self.send_wait_timer = threading.Timer(
                self.send_wait_timeout, self.add_timeout_data_dict)
            self.send_wait_timer.start()
            sensor_data_dict = None
            while sensor_data_dict is None:
                if not sensor_data_queue_from_beetle.empty():
                    sensor_data_dict = sensor_data_queue_from_beetle.get()
                    if self.send_wait_timer:
                        self.send_wait_timer.cancel()
                        self.send_wait_timer = None

                    if not self.send_sensor_data(sensor_data_dict):
                        self.stop()
```

Figure 6.5: Separate Threads for recv() and send() on Relay Laptop

Section 7 Game Engine

This section on Game Engine describes the design and implementation of the game engine on the Ultra96. In particular, it details how the game state is initialised, how actions are predicted and committed each turn, and how correctness is ensured using updates from the Eval Server. Details of thread management are also discussed.

7.1 Game Engine Design

7.1.1 Initialisation

The Game Engine on the Ultra96 maintains the game state initialised using the same GameState() class on the Eval Server. The GameState() class also initialises the player state for each player, including the player's hp, action, number of bullets, number of grenades, shield time, shield health, number of shields and number of deaths.

The Game Engine also keeps track of the current actions predicted for each player, initialised to Actions.no at the start of each new turn. There are 5 possible actions that each player can take each turn. Defined in the Actions() class in Helper.py, they are Shoot, Shield, Grenade, Reload and Logout. For predictions made by the FPGA, there is also Actions.no representing idle movement detected by the IMU.

7.1.2 Wait for Actions

During each turn, the Eval Client on the Ultra96 waits for actions from both players to be predicted and committed before sending the game state to the Eval Server. Actions such as Shield, Grenade, Reload and Logout are pushed onto the action_queue after enough IMU data is received and predicted by the FPGA while Shoot is immediately pushed onto the action_queue if data_type = "GUN" for an incoming packet from Relay Laptop.

```
def wait_for_actions(self):
    while (self.player_1_action == Actions.no or self.player_2_action == Actions.no):
        if self.player_1_action == Actions.no and not action_queue_player_one.empty():
            self.player_1_action = action_queue_player_one.get()

        if self.player_2_action == Actions.no and not action_queue_player_two.empty():
            self.player_2_action = action_queue_player_two.get()

    pos_p1 = 1
    pos_p2 = 1
    # no visualizer means grenade always hit
    if self.player_1_action == Actions.shoot or self.player_2_action == Actions.shoot:
        pos_p2 = 4
        print('[Game Engine] Resolving HIT or MISS')
        self.hit_timer = threading.Timer(
            self.hit_timeout, self.stop_timer)
        self.hit_timer.start()
        while self.hit_timer:
            if not VEST_queue_player_one.empty() or not VEST_queue_player_two.empty():
                pos_p2 = 1
                print(
                    '[Game Engine] HIT: Players in Line of Sight')
                if self.hit_timer:
                    self.hit_timer.cancel()
                    self.hit_timer = None

    self.commit_action(pos_p1, pos_p2)

    return self.game_state.get_dict()
```

Figure 7.1: Game Engine Wait for Actions

As shown in Figure 7.1, during each turn, only the first action predicted and pushed onto the action_queue is committed for each player. This is ensured by only setting the player_action to the first action in the queue if player_action = Actions.no and the action_queue for that player is not empty.

After both player_actions are set, they are put in default position 1 (line of sight). If either player_action was a Shoot, the Game Engine will try to resolve if the Shoot was a hit or a miss. This is implemented by first setting player 2's position to 4 (behind a barrier). If a Hit Packet is pushed onto the vest_queue before the timer expires, player 2 would be put back into position 1 before committing the actions. All other actions (including Grenade since our group does not have a visualizer) are assumed to always happen/hit.

7.1.3 Commit Actions

Commit Action takes in both player positions determined in Wait for Actions and follows the same logic as the move_one_step() function in MoveEngine.py on the Eval Server.

```
def commit_action(self, pos_p1, pos_p2):

    action_p1_is_valid = self.player_1.action_is_valid(self.player_1_action)
    action_p2_is_valid = self.player_2.action_is_valid(self.player_2_action)

    self.player_1.update(pos_p1, pos_p2, self.player_1_action,self.player_2_action, action_p2_is_valid)
    self.player_2.update(pos_p2, pos_p1, self.player_2_action,self.player_1_action, action_p1_is_valid)
```

Figure 7.2: Game Engine Commit Action

As shown in Figure 7.2, actions from both players are first checked for validity (e.g Shoot valid only if num_bullets > 0 or Shield valid only if shield_time <= 0) before calling the update() function found in StateStaff.py with the arguments pos_self, pos_opponent, action_self, action_opponent, action_is_valid. After both player states are updated, the game state is considered committed and ready to be sent to the Eval Server.

7.1.3 Update Game State

After sending the game state to the Eval Server and receiving the updated game state, the local game state on the Game Engine is also updated to maintain a consistent game state with the Eval Server. This is implemented by calling initialize_from_dict() from PlayerState.py to override the current game state with the updated game state received from the Eval Server.

A new turn is signalled by setting both player_actions back to Actions.no and emptying the action_queue.

```
def update_game_engine(self, updated_game_state):

    self.player_1.initialize_from_dict(updated_game_state['p1'])
    self.player_2.initialize_from_dict(updated_game_state['p2'])

    self.player_1_action = Actions.no
    self.player_2_action = Actions.no
```

Figure 7.3: Game Engine Update Game State

7.2 Thread Management

Initially, the Ultra96 had 5 threads running concurrently, 2 threads for TCP connections with both Relay Laptops, 1 thread for TCP connection with the Eval Server, 1 thread for the FPGA and 1 thread for the Game Engine. Each thread and its purpose are summarised in the following table:

Thread	Use / Purpose
Ultra96 Server 1 & 2	Receives data from Relay Laptops. Checks for data_type and pushes data_values of different types to respective queues. Sends back updated game states to Relay Laptops and Hardware for two-way communication
FPGA	Polls IMU_queue for IMU data. Checks for start-of-move and exceeding of threshold before sending input data of correct size to FPGA for prediction. Push predicted action onto action_queue
Game Engine	Waits for Actions, Commits Actions and Updates Game State as described in Section 7.1
Eval Client	Sends game state to Eval Server when ready Receives updated game state from Eval Server

Figure 7.4: Threads on Ultra96

In order to improve latency, threads that perform polling were integrated with other threads. The FPGA thread was integrated with the Ultra96 Server threads while the Game Engine thread was integrated with the Eval Client thread. Details of these developments are discussed in Section 7.3.

7.3 Developments, Issues Faced and Solutions

Overall, the Game Engine logic did not undergo significant developments from the individual subsystem to the single-player game to the two-player and two-player unrestricted games.

The major difference between the game modes is when and what actions are committed. For the single-player game, actions are committed immediately each turn when an action for player 1 is popped from the action queue. Both player positions are set to 1 and the game state is updated with player 1's action and player 2's action set to Actions.no. For the two-player game, as described in Section 7.1, actions are only committed after the first action for both players are popped from their respective action queues. The player positions are based on whether a Hit Packet is received for Shoot actions only. For the two-player unrestricted game, actions for each player are committed immediately and the game state is updated with the player's action and Actions.no for the other player. The player positions are still based on whether a Hit Packet is received for Shoot actions only.

One of the major issues faced when integrating the Game Engine with the FPGA was sending the right set of input data for the ML model on the FPGA. To overcome this, a start-of-move frame had to be sent from the Beetle to the Relay Laptop to the Ultra96 when the threshold is

exceeded on the Beetle. However, there were cases where the threshold on the Beetle was exceeded multiple times for each action performed, resulting in multiple sets of input data being sent. Thus, a secondary threshold was also checked on the Ultra96 using the sum of differences between each IMU data sent. Only when the secondary thresholding is exceeded will the input data be sent to the FPGA for prediction.

In terms of thread management, changes were made between the one-player game and the two-player game to improve latency. As described in Section 7.2, originally there were 5 threads running concurrently on the Ultra96. In order to improve latency, threads such as the FPGA thread and Game Engine thread that perform polling were integrated with other threads. The FPGA thread was rewritten as an FPGA class and integrated into the Ultra96 Server threads. When data_type = IMU is received from the Relay Laptops, the FPGA class calls process_IMU() and extends the input data to be sent to the FPGA if start-of-move is True. Once the input data is of the correct size, a secondary thresholding is performed and data is only sent to the FPGA for prediction if the threshold is exceeded. Latency is improved as IMU data is no longer polled from the IMU_queues. The Game Engine thread was also rewritten as a Game Engine class and integrated into the Eval Client thread. The Eval Client thread now calls the functions wait_for_action(), commit_action() and update_game_engine() instead. Overall, the number of threads on the Ultra96 was reduced to 3 and polling is reduced, significantly improving latency.

Section 8 Societal and Ethical Impact

There are many aspects of our AR laser tag system that can be generalised to other wearable systems. In particular, we can implement gesture recognition based on the same idea of collecting IMU data and using machine learning to predict actions.

Gesture recognition has a wide range of applications that can benefit society. For example, gesture recognition is used in Healthcare and as a Smart Assistive Technology to help patients with disabilities better communicate with their caregivers [32]. It can also improve their quality of life by helping to overcome a barrier related to their disability or by automating certain tasks using gestures. Gesture recognition is also widely used for Human-Computer Interactions. Complementing an AR/VR system, using gestures can help make interactions within AR/VR more efficient, natural and immersive.

Despite the wide range of applications and benefits that gesture recognition brings, there are still ethical concerns such as privacy, confidentiality, accuracy and ownership. Studies have shown that motion data collected in AR/VR or using wearables can be used to uniquely identify individuals [33]. Other personal information can also be inferred from motion data. In extreme cases, gesture recognition systems can even be used for the surveillance and monitoring of individuals, which leads to the violation of privacy rights especially if full consent was not given. Furthermore, data collected and stored are also vulnerable to cyber-attacks and data breaches, further raising privacy and confidentiality concerns. In terms of accuracy, incorrect predictions of gestures might also result in inconveniences or in serious cases, accidents. This raises questions such as who should be held accountable during such events, the person who performed the action or the developers/company. The lines are further blurred if the algorithm or system used is non-proprietary.

To overcome these ethical issues, informed consent has to be given from individuals before such motion data can be collected. Gesture recognition systems should also be implemented with privacy-enhancing measures such as data minimisation and encryption. Data protection and management guidelines and laws such as PDPA must also be followed [34].

References

[1] DFRobot, “Bluno_beetle_sku_dfr0339,” DFRobot. [Online]. Available: https://wiki.dfrobot.com/Bluno_Beetle_SKU_DFR0339. [Accessed: 14-Apr-2023].

[2] DFRobot, “Digital_IR_Receiver_Module__sku_dfr0094_,” DFRobot. [Online]. Available: https://wiki.dfrobot.com/Digital_IR_Receiver_Module__SKU_DFR0094_. [Accessed: 14-Apr-2023].

[3] “Rainbow led 1x5 module WS2812,” kuriosity. [Online]. Available: https://www.kuriosity.sg/rainbow-led-1x5-module/. [Accessed: 14-Apr-2023].

[4] “DC5V 2835SMD 600leds flexible single color led strip light - waterproof optional,” SuperLightingLED.com Online Store. [Online]. Available: https://www.superlightingled.com/dc5v-2835smd-600leds-flexible-single-color-led-strip-light-waterproof-optional-p-2998.html. [Accessed: 14-Apr-2023].

[5] InvenSense Inc., “Components101 - electronic components pinouts, details & datasheets,” components101. [Online]. Available: https://components101.com/sites/default/files/component_datasheet/MPU6050-DataSheet.pdf. [Accessed: 14-Apr-2023].

[6] DFRobot, “DIGITAL_IR_Transmitter_Module__sku_dfr0095_,” DFRobot. [Online]. Available: https://wiki.dfrobot.com/DIGITAL_IR_Transmitter_Module__SKU_DFR0095_. [Accessed: 14-Apr-2023].

[7] CUI Devices, “CMI-1295ic-0585T cui devices | mouser Singapore,” Mouser Electronics. [Online]. Available: https://www.mouser.sg/ProductDetail/CUI-Devices/CMI-1295IC-0585T?qs=OlC7AqGiEDkuskN%2Fok3iNw%3D%3D. [Accessed: 14-Apr-2023].

[8] “Types of seven segment displays and controlling methods,” ElProCus. [Online]. Available: https://www.elprocus.com/types-of-7-segment-displays-and-controlling-ways/. [Accessed: 14-Apr-2023].

[9] Texas Instruments, “Analog | Embedded Processing | Semiconductor Company | ti.com,” Texas Instruments. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74ls47.pdf. [Accessed: 14-Apr-2023].

[10] ISINWEI, Shopee.sg. [Online]. Available: https://shopee.sg/product/189216177/3305880447?gclid=CjwKCAjw8-OhBhB5EiwADyoY1S20oPT75R0W858ZJUd0ovVda_3QjsHSluWWNkwoV1l0amtK2PTbABoCHQIQAvD_BwE. [Accessed: 14-Apr-2023].

[11] bdaouas and cliffgi, “Bluno beetle ble power consumption,” RobotShop Community, 20-Dec-2016. [Online]. Available: https://community.robotshop.com/forum/t/bluno-beetle-ble-power-consumption/27208. [Accessed: 14-Apr-2023].

[12] “KY-022 infrared receiver module,” MakerSupplies Singapore. [Online]. Available: https://www.makersupplies.sg/products/ky-022-infrared-receiver-module. [Accessed: 14-Apr-2023].

[13] “Sensor Wiki: KY-005 infrared transmitter (IR),” The Geek Pub. [Online]. Available: https://www.thegeekpub.com/wiki/sensor-wiki-ky-005-infrared-ir-transmitter/. [Accessed: 14-Apr-2023].

[14] cyborg5, “Cyborg5/Irlib2: Library for receiving, decoding, and sending infrared signals using Arduino,” GitHub. [Online]. Available: https://github.com/cyborg5/IRLib2. [Accessed: 14-Apr-2023].

[15] FastLED, “FastLED/fastled: The fastled library for colored led animation on Arduino.,” GitHub. [Online]. Available: https://github.com/FastLED/FastLED. [Accessed: 14-Apr-2023].

[16] jrowberg, GitHub. [Online]. Available: https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050. [Accessed: 14-Apr-2023].

[17] jrowberg, GitHub. [Online]. Available: https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/I2Cdev. [Accessed: 14-Apr-2023].

[18] B. Or, “What is IMU?,” *Medium*, 19-Nov-2022. [Online]. Available: https://towardsdatascience.com/what-is-imu-9565e55b44c. [Accessed: 20-Jan-2023].

[19] M. Stewart, “Guide to classification on Imbalanced Datasets,” *Medium*, 29-Jul-2020. [Online]. Available: https://towardsdatascience.com/guide-to-classification-on-imbalanced-datasets-d6653aa5fa23. [Accessed: 20-Jan-2023].

[20] H. Patel, “What is feature engineering - importance, tools and techniques for machine learning,” *Medium*, 02-Sep-2021. [Online]. Available: https://towardsdatascience.com/what-is-feature-engineering-importance-tools-and-techniques-for-machine-learning-2080b0269f10. [Accessed: 20-Jan-2023].

[21] “Timing summary report,” *Xilinx*. [Online]. Available: https://www.xilinx.com/video/hardware/timing-summary-report.html. [Accessed: 21-Jan-2023].

[22] “Power estimation and analysis using Vivado,” *Xilinx*. [Online]. Available: https://www.xilinx.com/video/hardware/power-estimation-analysis-using-vivado.html. [Accessed: 21-Jan-2023].

[23] M. Afaneh, “Bluetooth 5 speed: How to achieve maximum throughput for your ble application,” *Novel Bits*, 12-Jul-2022. [Online]. Available: https://novelbits.io/bluetooth-5-speed-maximum-throughput/. [Accessed: 21-Jan-2023].

[24] Auris, Inc, “What causes Bluetooth to Disconnect?,” *Auris, Inc*. [Online]. Available: https://theauris.com/blogs/blog/why-does-my-bluetooth-keep-disconnecting. [Accessed: 21-Jan-2023].

[25] “Transmission control protocol,” *Wikipedia*, 19-Jan-2023. [Online]. Available: https://en.wikipedia.org/wiki/Transmission_Control_Protocol. [Accessed: 21-Jan-2023].

[26] Real Python, “Socket programming in python (guide),” *Real Python*, 21-Feb-2022. [Online]. Available: https://realpython.com/python-sockets/. [Accessed: 21-Jan-2023].

[27] “Block cipher mode of Operation,” *Wikipedia*, 08-Jan-2023. [Online]. Available: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation. [Accessed: 21-Jan-2023].

[28] Admin, “SSH tunneling: Client Command & Server Configuration,” *SSH Tunneling: Client Command & Server Configuration*, 03-Jan-2023. [Online]. Available: https://www.ssh.com/academy/ssh/tunneling-example. [Accessed: 21-Jan-2023].

[29] “Welcome to SSHTUNNEL's documentation!¶,” *Welcome to sshtunnel's documentation! - sshtunnel 0.4.0 documentation*. [Online]. Available: https://sshtunnel.readthedocs.io/en/latest/. [Accessed: 21-Jan-2023].

[30] “Product documentation,” *NI*. [Online]. Available: https://www.ni.com/docs/en-US/bundle/labview-fpga-module/page/lvfpgaconcepts/fpga_dma_communication.html. [Accessed: 21-Jan-2023].

[31] J. Brownlee, “Python threading: The Complete Guide,” *Super Fast Python*, 14-Dec-2022. [Online]. Available: https://superfastpython.com/threading-in-python/. [Accessed: 21-Jan-2023].

[32] T. Kanokoda, Y. Kushitani, M. Shimada, and J.-I. Shirakashi, “Gesture prediction using wearable sensing systems with neural networks for temporal data analysis,” *Sensors (Basel, Switzerland)*, 09-Feb-2019. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6386881/. [Accessed: 10-Apr-2023].

[33] L. Redins, “New Study reveals motion data can identify people in VR: Biometric Update,” *Biometric Update* |, 23-Feb-2023. [Online]. Available: https://www.biometricupdate.com/202302/new-study-reveals-motion-data-can-identify-people-in-vr#:~:text=A%20new%20study%20has%20revealed,application%20can%20identify%20them%20uniquely. [Accessed: 10-Apr-2023].

[34] “PDPC: PDPA Overview,” *Personal Data Protection Commission*. [Online]. Available: https://www.pdpc.gov.sg/overview-of-pdpa/the-legislation/personal-data-protection-act#:~:text=What%20is%20the%20PDPA%3F,Banking%20Act%20and%20Insurance%20Act. [Accessed: 10-Apr-2023].