CG4002 Computer Engineering Capstone Project
2022/2023 Semester 1

"Laser Tag++"
Design Report

Group **B16**	Name	Student #	Specific Contribution
Member #1	Alexander Tan Jun An	A0199267J	1, 2, Hw Sensors
Member #2	Pranav Venkatram	A0220015E	1, 2.2, 2.3, Hw AI
Member #3	Braden Teo Wei Ren	A0218202W	1, 2.1, Comms Internal
Member #4	Danzel Ong Jing Hern	A0199331Y	1, 2, Comms External/Internal
Member #5	Hu Xuefei	A0220693H	1, 2.3, Sw Visualizer

Table of Contents

Section 1 System Functionalities

This section will list the system functionalities for the laser tag game, through user stories.

As a…	I want…	So that…
FPS gamer	To be able to fire shots at another player	I will be able to do damage to the opponent
AR game enthusiast	To be able to use motion controls to execute actions such as reloading, throwing a grenade, activating a shield and logging out of the system	I can maximise immersion, by not requiring the need to press buttons
AR game enthusiast	To be able to get visual feedback of my actions on the visualiser, such as when firing my gun or activating a motion control such as reload, grenade of shield	I will be able to understand what is happening in the AR game
First time AR game player	The hardware to be easy and comfortable to use	It will not be too overwhelming for a first timer
FPS Gamer	To see my HP at all times	I know how well I am doing
FPS Gamer	To see the HP of my current shield, and when it will be available for activation again	I can plan strategies ahead
FPS Gamer	To see how many bullets are left in the magazine	I know when to reload
FPS Gamer	To see how many grenades and shields I have left	I can plan strategies
Avid gamer	Every shot I made to be successfully registered, where shots do not “disappear” after firing	I can have an enjoyable and fair experience
First time AR game player	Every action I make to be registered by both me and my opponent’s visualiser with low latency	I can be fully immersed in a seamless AR game
AR game enthusiast	To not be randomly disconnected from the game, or be able to easily reconnect and continue playing	The gameplay experience is not compromised

Section 2 Overall System Architecture

This section of the report will provide details on the planned system architecture of the project, the system’s intended final form, as well as the main algorithm for the game.

2.1 Overview of System Architecture

The following diagram illustrates the high-level system architecture of the system.

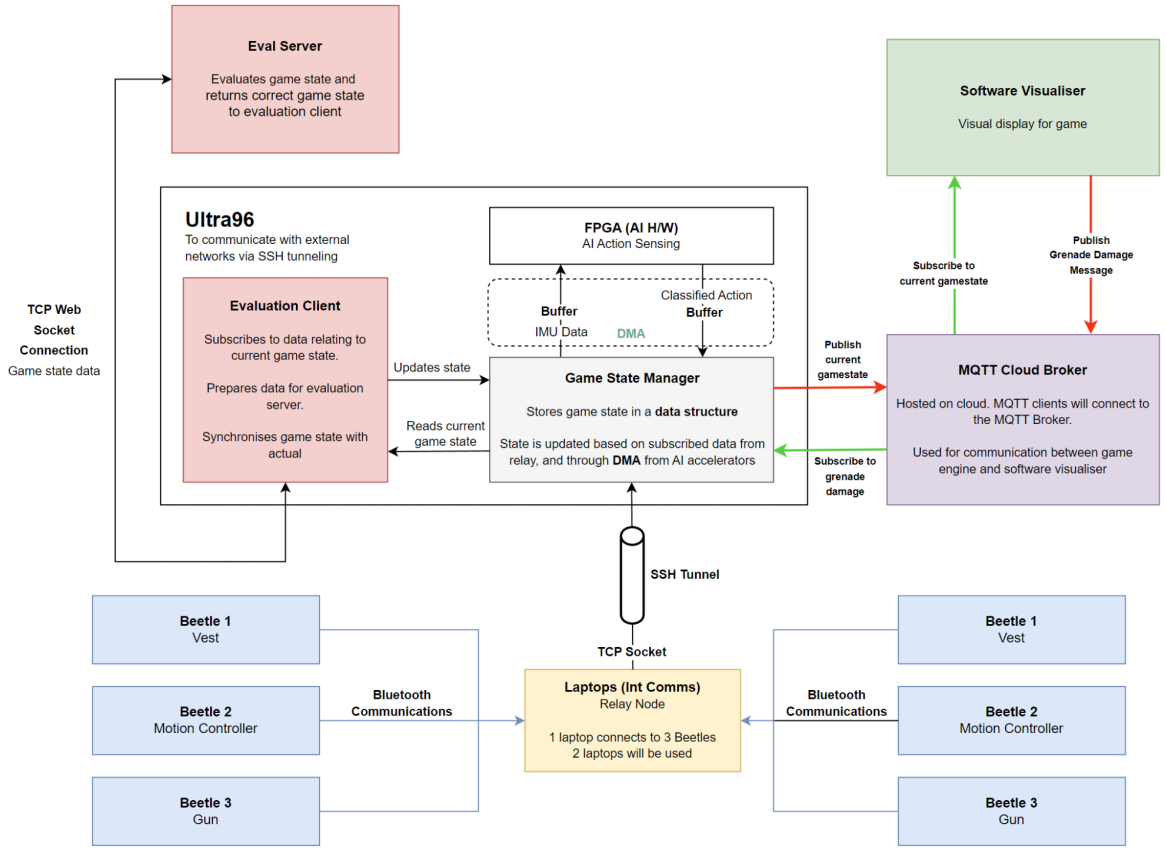


Figure 2.1: Laser Tag Game System Architecture

2.1.1 Beetle and Ultra96 Implementations

For the implementation, each beetle will serve a different role in the game, receiving raw data from the different sensors such as our IR sensors, IR emitters, and IMUs. As bluetooth connections are low bandwidth and unstable, each beetle will also carry out onset move detection, reducing the number of packets needed to be transferred over bluetooth connection to the relay node laptop, improving latency.

The Ultra96 board will also run both the game state manager, a custom built game engine, as well as our evaluation client, on top of AI processing. The game state management system implemented helps manage our separate game states, for both evaluation server, as well as the software visualiser. By separating game states, it allows us to introduce additional actions required for the software visualiser, such as ***shieldDown*** to lower the shield after activation, or ***bluetoothDisconnect/Connect*** messages to display the connection status of our bluetooth devices. The game state is stored in a local data structure, similar to that used in the evaluation server, as seen in figure 6.4 in the External comms section of this report**.** The evaluation client, on the other hand, prepares evaluation packets for the evaluation server, and will communicate directly with the evaluation server, and game state manager.

2.1.2 Communications and Software Interfaces

Communication between systems are mainly done either through TCP socket communications, or through MQTT protocol, where each component will subscribe or publish to the relevant topics on the MQTT broker, hosted on the cloud, allowing different components on different networks to communicate with each other reliably, as shown in Figure 2.1 above.

To communicate and send motion sensor, gun and vest data for the laser tag onto the game

state manager running on our Ultra96, connected to a different network, a SSH tunnel is set up, to allow the creation of the TCP socket for data transfer. Communications between the Ultra96's evaluation client and the evaluation server are also done through the use of TCP sockets, where messages are AES encrypted before transfer through the socket.

As the AI accelerator lies on a separate hardware component on the Ultra96, motion data sent from the Beetles, received via the TCP server have to be passed onto the on-board memory, via DMA (Direct Memory Access) on the Ultra96, to allow for classification of data. The outputs of the classification are similarly passed back to the game state manager via DMA.

Finally, to communicate reliably and with low latency to the software visualiser, MQTT protocol is used. A modified version of the gamestate, the visualiser state, will constantly be published onto the cloud MQTT broker, and subscribed by the software visualisers Unity applications. The visualiser also publishes data back to the game state manager, stating if players have taken damage when an in-game grenade has been thrown. More detailed information on communications, tunnelling, protocols used, and software interfaces for the system can be found in *Section 6: External Communications* of this report.

2.2 System Final Form

The following figure shows the system's intended final form. It will be composed of a glove, vest, and a shooting apparatus.

Figure 2.2: Diagram for system final form

2.3 System Main Algorithm

The following figure illustrates the high level overview of the main 6-step algorithm for the laser tag game, utilised by the game state manager on the Ultra96.

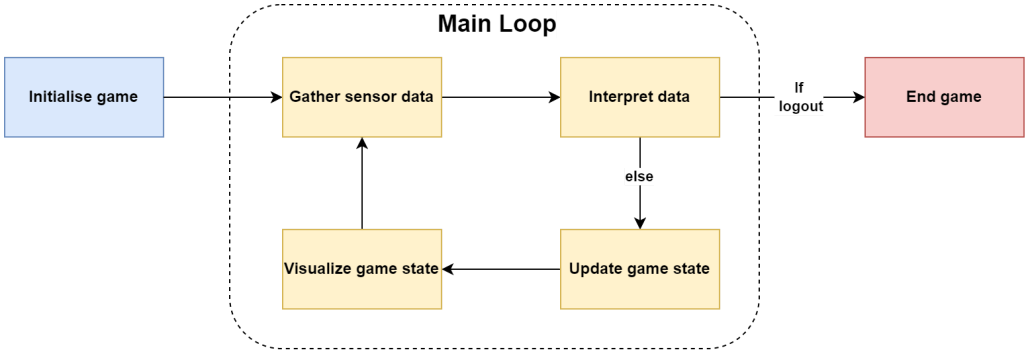


Figure 2.3: Main Game Algorithm

2.3.1 Initialise Game

In this first step, the game state manager is initialised, where the game state for all players are set to their default, specified values. MQTT communications from client to the cloud broker on the Ultra96, as well as the software visualisers are set up. If required, the TCP connection from Ultra96 to the evaluation server is set up as well. Next, bluetooth communications between each beetle and the relay node, and the final TCP connection from the relay node to the Ultra96, along with the required SSH tunnel, are set up.

2.3.2 Main Algorithm Loop

After initialisation, the system transits into the main loop, where gameplay occurs. This main loop consists of 4 separate steps. The first step, "Gather Sensor Data" happens when the IMU and IR sensor data is gathered on the beetles. The raw data from the sensors are cleaned, before being sent over bluetooth to the relay node, for the next step, "Interpret Data".

In this step, data from the relay node is transmitted to the Ultra96 which hosts our game state manager and AI classifier, through a TCP socket. IMU data is fed to the AI model to classify the action type. The AI model will have been trained on IMU sensor data such as acceleration and angular velocity along the x, y and z axes. Further, features will be extracted before passing data to the model using Spectral Analysis. This reduces model complexity and enables better accuracy. While all subsystems were being developed, we used sample IMU data obtained from Kaggle [17] for training, validation and testing. Once the hardware sensor subsystem was complete, we re-trained, re-validated and re-tested the model using our own IMU sensor data.

The third step in the loop is to "Update Game State". After interpreting the players actions, along with other data types received to denote a shot being fired, or getting hit, will be used by the game state manager to update the game state. More information on the data packet types and format can be found in the External Comms section of this report.

Finally, in the "Visualise Game State" step, the software visualiser receives the updated game state for all players to be displayed from the game state manager, by subscribing to the MQTT topics published by the game state manager, before looping back to the first step of the loop again. This 4-step loop continues infinitely, as long as gameplay continues.

2.3.3 End Game

When the "logout" action is interpreted for all players, the system enters the "End Game" part of the main algorithm. At this stage, all gameplay is stopped, and the game state is frozen by the game state manager. Final scores will be displayed on the software visualiser. A new game can be started by re-initialising the system.

2.3.4 Game States

The current game state of each player is maintained by our ***Player*** classes, in the *player_state* class variable, a dictionary with fields similar to that expected by the evaluation server for a single player. The player state is only modified through methods in the Player class, such as deducting health, and reducing shield timers. Overall game state for both players however, is stored as a Python dictionary with 2 fields, "p1" and "p2", where each field contains the dictionary variable *player_state* of each player. This game state, stored as the state variable *GAME_STATE* in the GamestateManager class is what is sent to the evaluation server.

As not all actions need to be sent to the evaluation server, such as getting hit, as well as to allow for more types of data to be viewed on the software visualiser, a separate game state, known as the visualiser state, is created via deep copy of the actual game state, and modified

with custom actions to be sent to the software visualiser, to carry out additional actions such as lowering of shield, and to indicate the connection status of bluetooth devices.

2.3.5 Damage Logic- Grenade, Shoot and Respawn

Upon receiving a 'hit' data packet from the Arduino beetles through the TCP socket, or from the software visualiser via MQTT 'gamestate/damageresult' topic by the game state manager, it is first verified against the *damage_tracker* state variable, which is set to and remains 'True' for 1 second after a player has done a damaging move, such as throwing a grenade or firing their gun, and set to 'False' after the damage from the opposing player has been taken. This tracker is put in place to ensure players do not take double hits from duplicate trigger packets, as well as to timeout missed shots or when players are behind cover.

Damage taken by players is calculated in the *player_hit()* method in the Player class, which handles deduction of player shield and actual health, as well as the full reset of the player and updating of score when a player's health reaches 0.

2.3.6 Shield Timers

Shield timers are handled on the Ultra96 in the game state manager directly. Upon classification of a valid shield action as per the game rules, a temporary thread is started on the Ultra96. This thread calls the *reduce_shield_cooldown()* method in the Player class once every second, reducing the shield cooldown by 1 every time. The thread is terminated once the shield cooldown reaches 0. Handling the shield cooldowns on a separate thread enables our game state manager to continue processing other player actions during the game during the 10 second duration, ensuring the quality of the game play.

2.3.7 Bluetooth Connection Handling

To improve user experience and game fairness, upon the disconnection of any of the player's bluetooth devices, the game state manager rejects all incoming data packets related to game play, and stops updating the game state. Only upon connection/reconnection of all devices will gameplay be allowed to continue. This feature can be toggled on and off at any time.

The connection statuses of the individual bluetooth devices are sent over from the laptop relay nodes over TCP socket, where each laptop is connected to 3 separate bluetooth devices, servicing the different hardware components required for the game. Upon bluetooth device disconnection, an error is thrown, and the relay node client sends a bluetooth disconnect message to the Ultra96, which is processed by the game state manager. Similarly, only upon receiving the handshake acknowledgement from the beetle will the bluetooth connected message be sent to the Ultra96 to update the state for connected devices.

2.3.8 Admin Client

The admin client script allows for managing of game server configuration concurrently and state remotely, by connecting to the game engine via TCP socket, and MQTT broker. The admin client allows admin users to toggle 'Evaluation Mode' on the Ultra96, enabling and disabling the sending of game state to the evaluation server, resetting the game entirely without the need for players to disconnect, as well as toggling 'Bluetooth Enforcement', which prevents the updating of game state, if any bluetooth device has disconnected.

2.3.9 Game Engine Modes

The game engine can be started in 2 modes during setup - evaluation and free play mode. This can be changed at any time with the use of the admin client, by toggling 'Evaluation mode'. When running in evaluation mode, the game engine runs in a turn based style, recording if each player has made moves, before sending the game state to the evaluation client, while free play mode allows a normal, unrestricted game of laser tag to be played.

Section 3 Hardware Sensors [Alexander Tan Jun An]

This section will provide details on the various devices/components that will be used in the overall system.

Section 3.1 Hardware Devices

In this section, the devices required for our wearable vest and gun will be listed. For each device, the relevant supporting components will be described, as well as the datasheet of the device itself.

1. **Beetle BLE:**
   The Bluno Beetle is a board based on Arduino Uno with Bluetooth 4.0 (BLE). The main role of Beetle BLE will be to retrieve the data generated by the connected sensors, do some pre-processing, and then feed the cleaned data to the machine learning models running on the Ultra96. Bluetooth will be used for communication between the Beetle BLE and Ultra96.

Datasheet:

- https://wiki.dfrobot.com/Bluno_Beetle_SKU_DFR0339
- https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ProductDocuments/DataSheets/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061B.pdf

2. **MPU-6050 IMU:**
   The MPU-6050 IMU will be worn on the player's left hand. Being able to measure velocity, orientation, acceleration, displacement, and other motion like features will allow us to identify the actions performed by the player.

Datasheet:

- https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf
- https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf

3. **IR Transmitter TSAL6200:**
   The IR transmitter will be present on the guns of both players. Upon triggering, a pulse will be fired from the transmitter. The pulse released will be between 940-950nm. A resistor will be connected in series with the transmitter to limit the current going through it.

Datasheet:

- https://www.farnell.com/datasheets/1864140.pdf

4. **IR Receiver Diode TSOP38238:**
   The IR receiver diode will be present on the vest of both players. It will be able to detect IR pulses from the IR transmitter of the other player.

Datasheet:

- https://www.farnell.com/datasheets/2049183.pdf

5. **Red LED HUR334d:**
   This red LED will be the feedback unit found on both the vests. Upon receiving a hit from the opposing player, the LED will flash.

Datasheet:

- http://www.sgbotic.com/products/datasheets/display/5MM_LED_colors.pdf

6. **Piezo Buzzer:**
   Piezo buzzers will be present on the shooting apparatus of both players. This is to indicate that a shot has been made by the player.

Datasheet:

- https://components101.com/sites/default/files/component_datasheet/Buzzer%20Datasheet.pdf

Section 3.2 Pin Table and Final Design

In this section, the pinout diagram of each individual component will be provided. The individual connections between components will also be provided.

1. Player's glove/Beetle BLE to MPU-6050 IMU:

The following figures show the pinout diagrams for both the Beetle BLE and MPU-6050 IMU respectively.

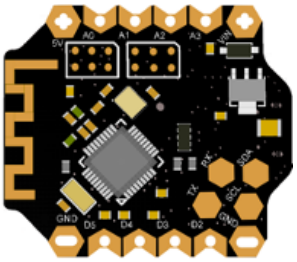


Figure 3.1: Pinout diagram of Beetle BLE

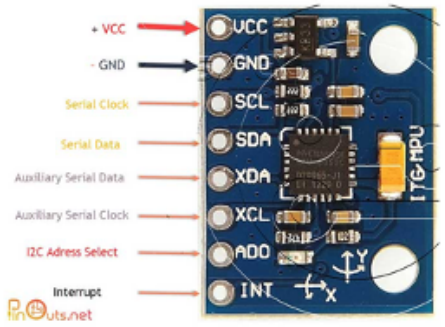


Figure 3.2: Pinout diagram of MPU-6050 IMU

For clarity, the following pin table will show the connections across Beetle BLE and MPU-6050 IMU.

Beetle BLE	**MPU-6050 IMU**
5V	VCC
GND	GND
SCL	SCL
SDA	SDA
D2	INT

2. Player's gun/Beetle BLE to IR Transmitter

The following is a pinout diagram of the IR transmitter.

Figure 3.3: Pinout diagram of IR Transmitter

For clarity, the following pin table shows the connections across Beetle BLE and the IR Transmitter.

Beetle BLE	**IR Transmitter TSAL6200**
D3 (PWM Channel 3)	+
GND	-

3. **Player's vest/Beetle BLE to IR Receiver**

The following is a pinout diagram of the IR receiver.

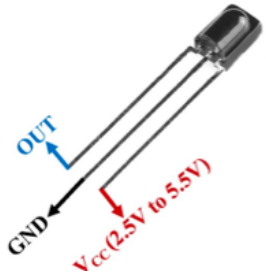
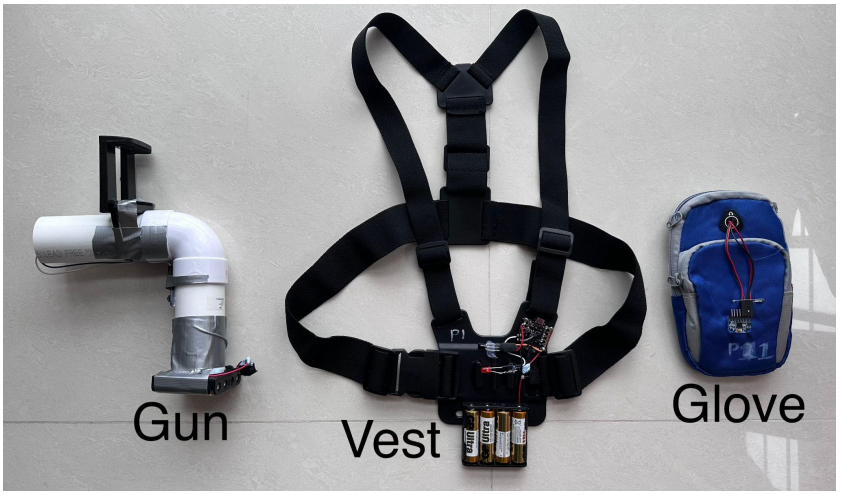


Figure 3.4: Pinout diagram of IR Receiver

For clarity, the following pin table shows the connections across Beetle BLE and the IR Receiver Diode.

Beetle BLE	**IR Receiver Diode TSOP38238**
5V	VCC
GND	GND
D2	OUT

The following picture shows how the final set for a single player looks like.



The gun is constructed from PVC pipes and a right angled pipe connector. The vest is a adapted from a action camera chest mount and the glove is made from a sports wrist pouch. Since the PVC pipes and pouch are hollow, it allowed us to store the Beetles, wires and other components inside, increasing durability.

As the schematics did not change much from the initial report, the section has been removed from the final report. Please refer to the initial report if needed.

Section 3.3 Operating Voltage and Current

This section describes the operational voltage level and current drawn by each component/device.

1. **Beetle BLE**
   - Operational Voltage: 5V DC

- Operational Current: 10 mA
- Power draw = 5V x 10mA = 50mW

2. MPU-6050 IMU

- Operational Voltage: 2.375V - 3.46V
- Operational Current: 3.6mA - 3.9mA
- Power draw = 3V x 3.8mA = 11.4mW

3. IR Transmitter TSAL6200

- Operational Voltage: 1.35V
- Operational Current: 100mA
- Power draw = 1.35V x 100mA = 125mW

Powering off a 5V pin, $R = \frac{V}{I} = \frac{5V - 1.35V}{0.1} = 36.5\Omega$. Therefore, to prevent high current from damaging the transmitter, resistance of more than 36.5 Ω should be used.

4. IR Receiver Diode TSOP38238

- Operational Voltage: 2.5V - 5V
- Operational Current: 0.27mA - 0.45mA
- Power draw = 5V x 0.35mA = 1.75mW

5. Red LED HUR334d

- Operational Voltage: 2V
- Operational Current: 20mA
- Power draw = 2V x 20mA = 40mW

6. Piezo Buzzer

- Operational Voltage: 4V - 8V
- Operational Current: 20mA
- Power draw = 5V x 20mA = 100mW

The estimated power for each major component of the system is in the table below.

Component/Device	Estimated Power Draw (mW)	Estimated Load Current
Glove (Beetle BLE + IMU)	61.4mW	13.9mA
Gun (Beetle BLE + IR Transmitter + Piezo Buzzer)	275mW (When the gun is triggered)	130mA
Vest (Beetle BLE + IR Receiver + Red LED)	91.75mW (When the vest is hit)	30.45mA

As each Beetle oversees one component, the battery capacity required will not be extremely high. Since the devices will be donned by the player, weight is a major concern. Batteries that are light in weight will be ideal for this use case. In our final design, each Beetle was powered with 4xAA battery packs that allowed us to go through multiple rounds of games and testing before it ran out.

Previously, our group planned to invest in rechargeable batteries. However, we found that it was more cost effective for us to use disposable batteries. Throughout the duration of our project, we only had to replace the batteries a few times. The total cost of this was less than purchasing rechargeable batteries for all systems.

Section 3.4 Algorithms and Libraries

This section will describe the algorithms and libraries we plan to use on Beetle BLE to retrieve data from the sensors.

1. **MPU-6050 IMU:**
   As I2C communication will be employed, the main library used will be I2CDevLib by Jeff Rowberg. This library is found at https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050. The two main portions of the library used will be I2CDev and MPU6050.

2. **IR transmitting and receiving:**
   For transmitting and receiving IR signals between guns and vests, we will be adapting and modifying the Arduino-IRremote library found at https://github.com/Arduino-IRremote/Arduino-IRremote. The NEC protocol was used as it is most commonly employed in Arduino projects, allowing easier debugging.

3. **Onset Detection**
   As a form of data pre-processing, onset detection is done on the Beetle BLE. This allows us to transmit data that the algorithm thinks might be an action. Onset detection is accomplished with a sliding window that compares the current windows average acceleration with the previous windows average acceleration. If the threshold is exceeded, the Beetle then sends the IMU data through bluetooth.

Section 3.5 Challenges & Progression of the subsystem

Initially, the system was rudimentary and we only verified that the IR transmitter and receiver were working together. Transitioning to the single player game, a button was added to act as a trigger to activate the IR transmitter and an LED was added to act as a hit indicator on the vest. For the two player game, the hardware subsystem saw no major redesigns but we had to pay extra attention to player identification so that a player cannot shoot himself. To cope with the physical stress of data collection for unseen player game, the connections on the glove had to be resoldered and strengthened as wires breaking were a common occurrence.

Throughout the process of building the hardware required for the game, several challenges were faced.

1. **Corrupted Bootloader**
   As part of system integration and testing, the Beetles were succumbed to multiple code uploads. The serial port was also subjected to countless plugging and unplugging. This resulted in the serial port dying, either to physical wear and tear or a corrupted bootloader. As such, sets like the glove had to be resoldered entirely from the ground up, which added time to our system integration.

2. **Bluetooth Module**
   Towards the end of the project, we faced bluetooth issues. One of the Beetle's bluetooth was unable to connect and could not be found by a bluetooth scan as well. Our team concluded that the bluetooth module might have died. This resulted in us having to resolder a backup Beetle.

Section 4: Hardware AI [Pranav Venkatram]

In this section, we discuss the functioning of the Hardware AI subsystem. This component is meant to identify hand actions performed by players to augment the gameplay. We achieve this by using IMUs sensors to sense hand actions and classify them using a Multi-Layer Perceptron Neural Network (MLP). We followed the following process flow:

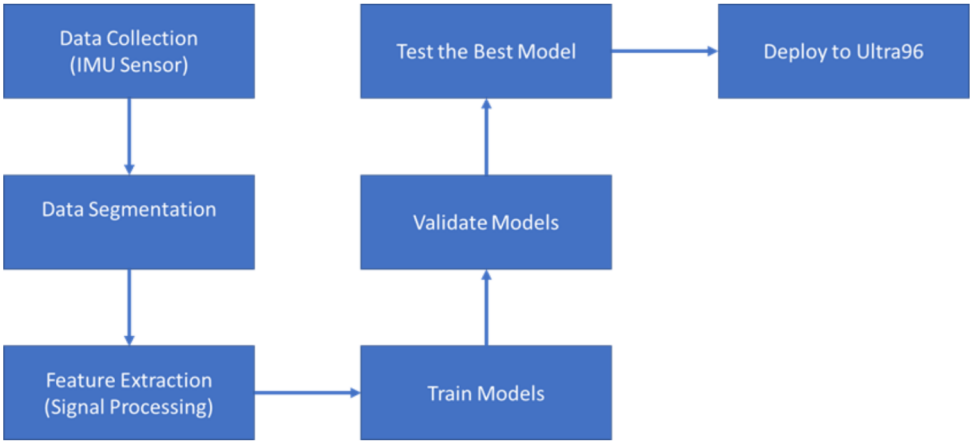


Figure 4.1: HW AI Development Process Flow

In the following subsections, we explore these steps in further detail.

4.1 Data Collection & Segmentation

Each player can perform 4 actions: Shield, Reload, Throw Grenade and Exit. Since all actions are executed by arm movements, we **mounted an IMU to each player's wrist.** This captures movement of the shoulder, elbow and wrist. Thus, we obtained **6 sequences of raw data** - acceleration and angular velocity along the x, y, and z axes.

To collect IMU data for each action, we used onset detection to collect data for the time that a significant change in acceleration or angular velocity was encountered. Hence, **each sample's duration is variable.**

Further, the data was collected while varying the speed and gait of the action. Since users may perform actions with differing speeds and posture, it is essential that the data contains variation such that the model will be accurate. Intuitively, players are likely to perform actions fast as they are looking to defeat their opponents.

4.2 Feature Extraction & Data Pre-processing

4.2.1 Data Pre-processing

Based on inspection of IMU data's frequency spectrum, there is **high frequency noise** in the IMU sensor data. Hence, a **Low-Pass Filter (LPF)** was used to pre-process the data.

Design Requirement for LPF:

1. Must have flat passband (so filter will not induce additional noise)
2. Must have a good roll-off i.e. steepness of attenuation for increase in order of filter

The **Butterworth filter** satisfies the above conditions [73] and was thus adopted for our use case. The following parameters were chosen for the LPF based on visual inspection of the time domain signal before and after filtering:

1. Order (Poles) - 6
2. Cut-off Frequency - 3Hz (for 0dB roll-off)

Figure 4.2: Before LPF (Z Axis Angular Velocity)

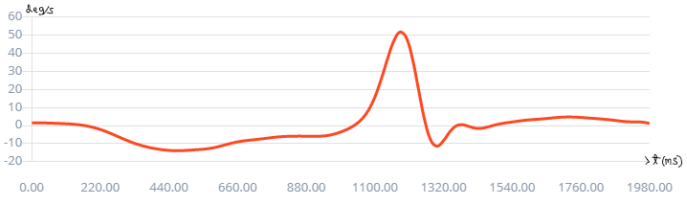


Figure 4.3: After LPF (Z Axis Angular Velocity)

4.2.2 Feature Extraction

Since a single action can be performed slightly differently, the raw time series sensor data for the same action will be different, Hence, we needed to extract features that are common to all repetitions of actions such that patterns are more apparent for the model to learn. This was critical to ensuring good model accuracy as well as reducing model complexity.

We chose to extract features on the Ultra96 as data manipulation is more seamless using Python as opposed to working with Arduino code on the Beetle.

Based on literature by Edge Impulse[24], feature extraction on IMU data can be done using **Spectral Analysis.** We have adopted Edge Impulse's library to apply this feature extraction [57]. The following features were extracted for each axis from gyroscope and accelerometer data, giving 30 features per segment of data (3 axes x 2 sensors x 5 features = 30 features):

<u>Time Domain Features:</u>
Since the system will receive a statistically significant number of signal snapshots, all **snapshots representing each action can be modelled by a Normal distribution** (according to Central Limit Theorem). To characterise this distribution, the following time domain features are extracted from each signal:

a. Root Mean Square (X_{RMS}) - Average of a discrete time signal x(t)

$$X_{RMS} = \sqrt{\frac{1}{N}(x[1]^2 + x[2]^2 + \ldots + x[N]^2)}$$

Equation 4.1: RMS formula for Discrete data [48]

b. Skewness (g_1) – Degree of symmetry with respect to a normal distribution [49]

$$g_1 = \frac{m_3}{m_2^{3/2}} \quad \text{where } m_i = \frac{1}{N}\sum_{n=1}^{N}(x[n] - \bar{x})^i \quad \text{and x is a discrete time signal}$$

Equation 4.2: Skewness formula [50]

c. Kurtosis – Probability of outliers in a distribution i.e. far from the mean [49]

$$\text{Kurtosis} = \frac{\sum(x_i - \bar{x})^4}{n\sigma^4}$$

$x_i - i^{th}$ sample
$\bar{x}$ – average of sample
n – number of samples
σ – standard deviation

Equation 4.2: Kurtosis formula [51]

Frequency Domain Features:

a. Log of Peak Spectral Power of each FFT bin of width 1.5 Hz

FFT is an algorithm to perform Fourier Transform. Since our sampled data is discrete, the FFT applies **Discrete Fourier Transform**, resulting in an euler representation of the signal in terms of magnitude and phase. The **log of the magnitude squared** is computed and the **maximum value for each bin of interest** is extracted.

The purpose of **log is to limit the range of values** so the model will not have saturated weights exceeding the supported precision.

The number of extracted peaks depends on the cutoff frequency. For our cutoff frequency of 3Hz (0 dB roll-off), 2 bins are considered (0Hz-1.5Hz, 1.5Hz-3Hz).

4.3 Model selection and optimization

We explored an MLP model for classifying IMU data.

4.3.1 Multi-Layer Perceptron (MLP) Architecture

An MLP consists of a set of layers, each consisting of neurons. These layers are linked sequentially such that the output of one layer is passed to the next. We used the following architecture:

1. Input Layer - 30 Neurons (30 Extracted Features from window of data)
2. Dense Layer - 50 Neurons & ReLU Activation
3. Dense Layer - 40 Neurons & ReLU Activation
4. Output Layer - 4 Neurons & Softmax Activation (Classification as 1 of 4 actions)

Neuron Architecture

Each Neuron performs a matrix multiplication between a weight matrix for a particular layer and the input vector of data as follows:

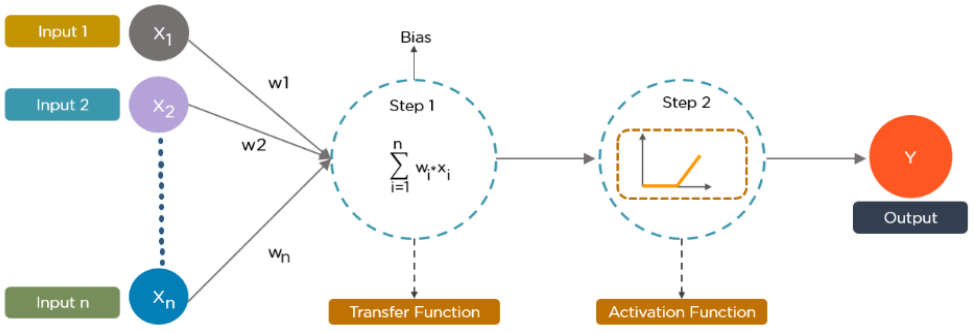


Figure 4.4: Neuron Architecture[53]

The result is then broadcast to every neuron in the following layer for further processing.

Activation Functions

After a neuron performs a multiplication, the result is passed through an activation function before it is broadcasted to the next layer. An activation function is a nonlinear function which prevents the neural network from making purely linear predictions.

For our model we have chosen the following activation functions:

1. Rectified Linear Unit (ReLU) - Used in hidden layers as it is simple and fast

$$\mathrm{ReLU}(x) = (x)^+ = \max(0, x)$$

Where x is the output of a neuron

Figure 4.5: Formula to compute ReLU [71]

2. Softmax - Function that converts real numbers given by the output layer into probability values signifying the likelihood of the input belonging to certain classes [52]

$$\mathrm{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Where x_i is the output of a particular neuron

x_j is the output of each neuron in the output layer

Softmax(x_i) is the probability that x_i belongs to class i

Figure 4.6: Formula to compute Softmax [72]

To arrive at the above architecture, the following **parameters** were tuned:

1. Number of layers
2. Number of neurons

The models' accuracy on the validation dataset was compared for combinations of the above parameters (key comparison are listed below):

Number of hidden layers	Neurons per hidden layer	Accuracy (%)
1	50	87.5
2	50, 40	93.75
1	20	40.6
2	20, 20	56.2

This evaluation indicates how the model accuracy changes with respect to its architecture. Based on this data, we chose to implement an MLP with 2 hidden layers, with 50 and 40 neurons respectively.

4.3.1 MLP Optimization

When implementing the model in HLS, **pipelining** and **loop unrolling** were applied to outer loops and inner loops respectively to speed up inference. While these optimizations required significant hardware utilisation, the average inference time was sped up from **0.424 ms** to **0.388 ms (8.49 % speed up).**

4.4 Model Training

The data is split using the following proportions:

1. Training Data - 50%
2. Validation Data - 25%
3. Testing Data - 25%.

Since the model is only deployed on the Ultra96, it was trained and tested using the Python stack (**TensorFlow**). The training data undergoes **feature extraction using Spectral Analysis** as mentioned earlier and is provided to the classifier for training. The trained model was then converted to HLS for deployment on the Ultra96 FPGA.

4.5 Model Validation & Testing

One drawback of using an MLP for gesture classification, is **limited training data**. Typically, neural networks require large amounts of data to learn from. Since this data is generated manually, there is a **high risk of the model overfitting** to such a small dataset. This was noticed when training the same model on random splits of the data and observing the **variance of the validation accuracy.** This variance was initially high for small datasets, but as more data was provided, the model variance began to decrease. Eventually, when the model was deployed to the FPGA for real time testing, it was trained on all the collected data.

4.6 Ultra96 Synthesis & Simulation

As stated earlier, the MLP is implemented in HLS. The following steps were run in Vitis to generate an IP block [34]:

a. C simulation – Pre-synthesis validation that the C program correctly implements the Keras model

b. C/RTL synthesis

c. C/RTL co-simulation – Post-synthesis validation that the RTL functionally matches the C code

d. Vivado synthesis – Transform RTL into gate level design

These processes make certain that the bitstream running on the FPGA will function the same as the Keras model.

4.7 Realising the Neural Network on FPGA

Once the MLP is trained, validated and tested on the Python stack, it is ready to be deployed to the FPGA. This was done by inspecting the weights of the model and implementing the model logic in C++ HLS code. The implementation of dense layers and softmax activations were derived from literature on frameworks for developing neural networks [63].

After generating the IP block, it is integrated with a block diagram in Vivado containing key blocks such as Ultrascale+ MPSoC and DMA such that the model can communicate with external subsystems:

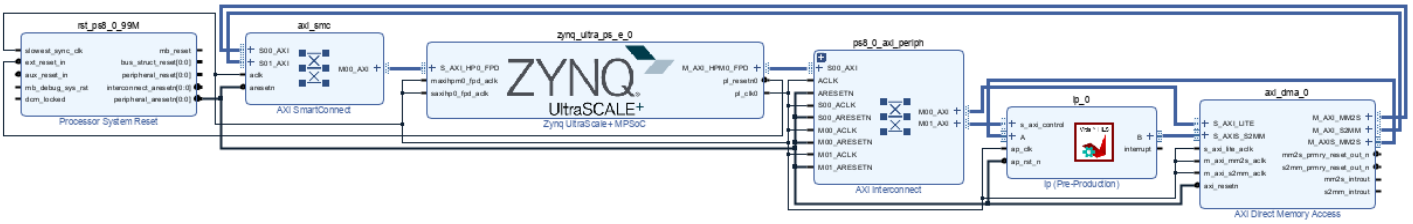


Figure 4.7 – Block Diagram in Vivado

4.8 Design Evaluation – Timing, Area

When Vitis performs synthesis, it generates a HLS report, detailing aspects such as resource usage, latency etc. The reports below show these metrics for the optimised and unoptimized models:

Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
ip		-	856	8.560E3	-	857	-	no	19	70	22	35	0

Figure 4.8 – C Synthesis Report (with pipelining & loop unrolling)

Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
ip		-	30307	3.030E5	-	30308	-	no	3	1	1	7	0

Figure 4.9 – C Synthesis Report (without optimizations)

Based on the reports above, the design's timing is measured by the Latency metric, and the area is determined by the number of Flip Flops (FF), Look Up Tables (LUT), Block RAM (BRAM), Digital Signal Processing blocks (DSP), used.

As mentioned earlier, applying loop optimizations speeds up real time inference by 8.49%. However, this performance requires 70% of the Ultra96's DSPs. Since the model computes weighted sums, it is expected that unrolling and pipelining require a large number of multiplier blocks. Since the FPGA is used exclusively for gesture classification, it is justifiable to make full use of its hardware. Hence, we prioritise model speed to reduce response time.

4.9 Progression of the subsystem

A basic model was initially implemented on sample IMU data from Kaggle [17]. Once our sensors were ready and we collected our own data, the feature extraction parameters and model architecture needed to be refined to suit our data. We found that tuning the cut-off frequency of the low pass filter during feature extraction, smoothed out our sensor readings better and subsequently improved model accuracy. Further, our MLP only had a single hidden layer for the Kaggle data. To ensure good accuracy on our own data, increasing the number of hidden layers was necessary. Once this tuning was complete, a considerable amount of time was spent gathering data to train the model from the one player game to the two player game.

Section 5 Internal Communications [Braden Teo Wei Ren]

In this section, we discuss how the laptop acts as a relay node between the Beetle BLEs and the Ultra96 server. Each laptop communicates with three Beetles via bluetooth low energy (BLE).

In BLE communications, the Generic Access Profile (GAP) controls bluetooth connections and advertising, allowing for a device to be visible to other devices [40]. The peripheral (Beetle) would transmit an advertising packet outwards to inform the central device (relay node laptop) of its presence. The laptop would continuously scan for those packets and then establish a dedicated connection between the two devices. Once a dedicated connection is established, the Generic ATTribute Profile (GATT) comes into play. The Beetle would stop advertising itself, until the connection between the Beetle and laptop is broken [43]. GATT uses the concept of Services and Characteristics to transfer data between devices. For the Bluno Beetle v1.1 that we use in this project, BLE communication occurs when the Serial Port characteristic of the Beetle is written with new data or data is being read from it.

5.1.1 Managing Tasks on Beetles

Each of the three Beetles is in charge of different sensors – IR receiver, IR Emitter, and IMU. Each time a signal is received from its sensor, the Beetle would have to transmit the data to the relay node laptop via BLE communication. However, the laptop might send over a packet of data to the Beetle via BLE communication at the exact same time. Due to the fact that the Beetle only allows BLE communication via reading from, and writing to the single Serial Port characteristic in the Arduino board, we cannot handle simultaneous sending and receiving of data, as that would result in the collision of incoming and outgoing data. Thus, our Arduino code on the Beetle would prioritise the receiving of data first, and if no data is being received from the laptop, the Beetle is able to send out the data it received from the Beetle's sensor. Otherwise, the Beetle waits for the next loop to send its data.

5.1.2 Managing Threads on Relay Node Laptop

Multithreading is required to maintain concurrency of internal communications. 3 threads run on each relay node laptop – each thread receives and sends data from/to each of the 3 beetles separately, concurrently. These threads are implemented using Python's *threading* module. A point to note is that due to the Global Interpreter Lock (GIL) in Python, only one thread holds control of the Python interpreter at any one time, preventing threads from truly running in parallel. However, this would likely not be an issue for all threads, as the rate of transmission for IR sensors and emitters would only receive occasional data, at a relatively low frequency. The Beetle attached to the IMU on the other hand, utilises onset movement detection, and thus only transfers data when motion is detected, compared to a constant stream, allowing threads to be constantly switched out. During testing, no data loss or latency issues were faced when utilising threads, and as such, multiprocessing was not implemented.

In order to support data transmission from the Beetles to the game state manager on the Ultra96, a TCP socket is set up from the relay node laptop to the Ultra96. Each set of Beetles along with their threads corresponding to a single player utilises an instance of the TCP socket client created via the *TcpClient* class, which includes methods to format raw data received into the required formats by the game state manager.

To ensure data integrity as 3 threads are simultaneously sharing a single instance of the socket client, thread locking was implemented using the threading lock found in Python's *threading* module as well. Each time a Beetle thread wants to send data, it would have to acquire the lock to access the TCP socket client, and would only release it after the acknowledgement message from the Ultra96 is received.

5.2 Setting up and Configuring BLE interfaces

The following figure illustrates what our BLE interface looks like.

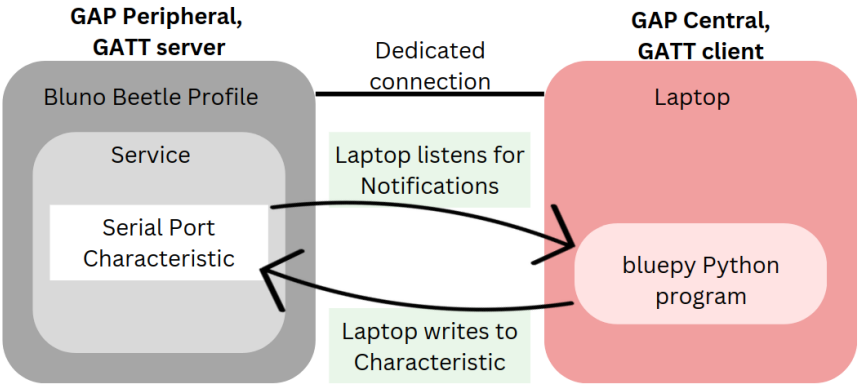


Figure 5.1: BLE interface

5.2.1 Bluno Beetle BLE Configuration

Even though the Beetles have pre-set configurations for BLE communication, we nonetheless re-configure them in order to reduce the probability of bugs occurring in our program, should the Beetles be incorrectly configured in the factories. Configuration can be done in the Arduino IDE. After connecting a Beetle to a relay node laptop and opening the serial monitor, we enter AT (ATtention) commands that assign the Beetle the role of BLE peripheral device and set the baud rate of UART to 115200. We also use an AT command to get the MAC address of the Beetle.

5.2.2 Relay Node Laptop BLE Configuration

The relay node laptop that we used, originally running only Windows, had to be dual-booted with Linux. We also used the *bluepy* Python package to allow for BLE communications with the peripherals. The laptop is by default a central BLE device, so no other configuration has to be done.

Originally, we wanted to use the *Bleak* Python package instead of *bluepy*, as *Bleak* can run on Windows OS. However, we found out that certain BLE functionalities did not work. We also tried running both *Bleak* and *bluepy* on a Virtual Machine, but it still did not work for the laptop we tried, possibly due to certain firmware settings in the laptop. Thus, in the end, we set up *bluepy* on a native Linux environment, which has been proven to work for the other project groups.

5.3 Detailed Communication Protocol

5.3.1 Packet Type

The following table illustrates the packet types and descriptions we would be transmitting via BLE. Even though packet types are denoted as characters for easier reference, they are translated into their equivalent ASCII codes when being transmitted over bluetooth.

Packet Type	Character Representation	ASCII Code	Description
HANDSHAKE	H	72	Sent from laptop to initiate handshake with Beetle
HANDSHAKE_ACK	A	65	Acknowledge handshake request

RESET	R	82	Sent to initiate Beetle reset request
DATA	D	68	Contains IR emitter/receiver data
DATA_ACK	K	75	Acknowledgement for receiving DATA packet
IMU_DATA	I	73	Contains raw IMU data at 3 dp
EXT_IMU_DATA	X	88	Special data packet with raw IMU data up to 5 dp
IMU_END	E	69	Signals end of IMU data period - Move ended
EOF_ACK	F	70	Acknowledgement for receiving IMU_END packet

5.3.2 Packet Format

The general data packet structure can be seen in the figure below:

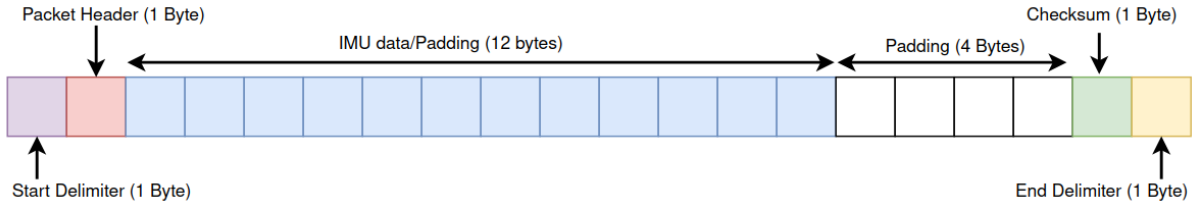


Figure 5.2: Data packet structure

The Maximum Transmission Unit (MTU) of the Bluno Beetle is 23 bytes. Disregarding the 3 bytes used for overhead, only 20 bytes can be transmitted by the Beetle. To reduce complexity, all data packets sent over BLE will be a fixed 20 bytes long regardless, and the checksum will always be on the 20th byte of the frame.

The 12 bytes used to store IMU data are otherwise padded with 0's when not used to transmit IMU data. Meanwhile, the Padding section is permanently filled with 0's.

As each IMU reading takes up to 6 values along the xyz axis, 2 bytes are reserved for each value in the IMU data field in the data packet. However, raw IMU readings are presented as float32, which are up to 32 bits size. In order to reduce the size of the data for transmission, while retaining a decent float value as well as the signedness of the reading without the need to send multiple data packets for 1 IMU reading, the original float values are multiplied by a factor of 1000 to preserve up to 3 decimal places. The scaled values are then truncated and stored as *int16* data type in the IMU data field of the packet, preserving the signedness. When received on the relay node laptops, these values are divided by 1000 before sending to Ultra96.

For all other packet types, just the packet header contains enough information for the beetle and relay node laptops to carry out functions such as handshaking, resetting the Beetle, and recording of IR receiver or emitter data.

5.3.3 Baud Rate

The default baud rate used by the Beetles is 115200 bits per second. We chose to use this baud rate for BLE communication as it allows for fast yet reliable transfer of data.

5.3.4 Handshaking

Immediately after a Beetle has established BLE connection with the relay node laptop, the laptop will initiate a three-way handshake with the Beetle. The laptop will send a

HANDSHAKE packet to the Beetle, which acknowledges it and sends back a *HANDSHAKE_ACK* packet. The laptop waits for this acknowledgement and likewise, sends a *HANDSHAKE_ACK* packet back to the Beetle, signalling that both devices are ready to transmit data. This three-way handshake ensures that both devices are able to send and receive data packets without any issues.

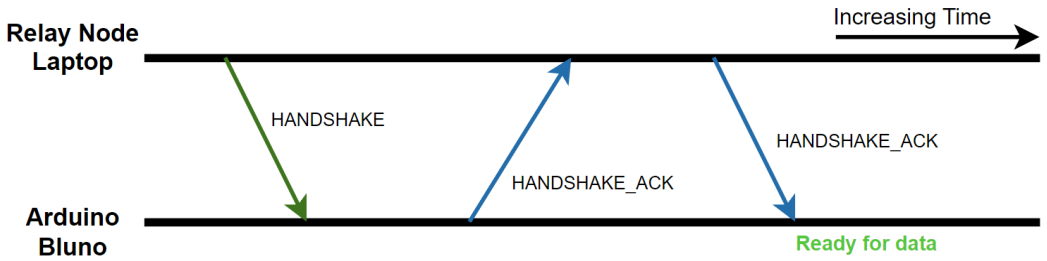


Figure 5.3: 3-way Handshake Protocol

As shown in the figure below, if either the relay node laptop or the Beetle times out, the same packet will be resent. If either side receives a corrupted packet, the corrupted packet would be dropped. Duplicate packets would also be dropped.

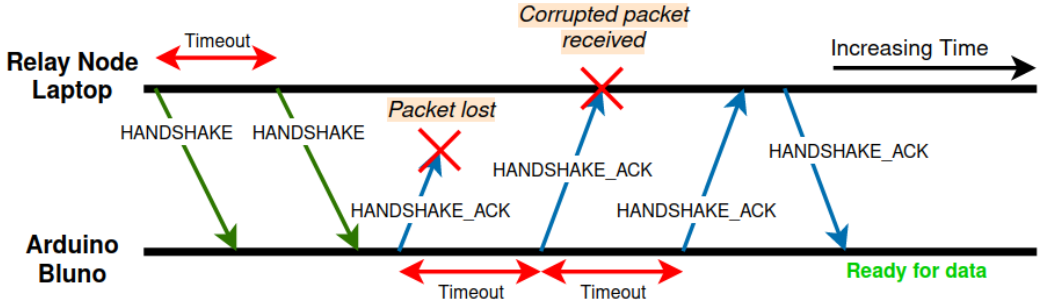


Figure 5.4: 3-way Handshake Protocol with Packet Loss and Corruption

Additionally, if the relay node laptop has already sent a specific number of *HANDSHAKE* packets with no response, it will send the *RESET* packet to the Beetle, signalling the Beetle to reset. The Beetle would start executing the instructions in its Arduino sketch from the beginning (i.e. doing reconnection and handshake again).

5.3.5 IR Receiver/Emitter Data Sending (Stop and Wait)

As IR receiver and emitter data are sent at a low volume, and guaranteed delivery is required to maximise user experience, stop and wait was implemented, as seen below.

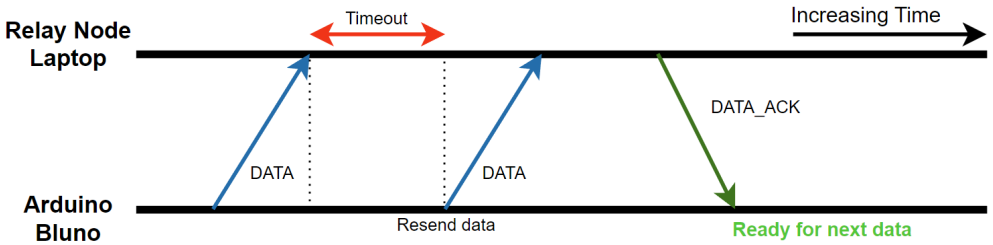


Figure 5.5: Stop-and-wait protocol for IR Receiver/Emitter

Every data packet sent to the relay node laptop has to be acknowledged, before the next data packet can be sent. In the event that the acknowledgement was not within the timeout time frame, the data packet would be resent again to ensure that the data was received.

5.3.6 IMU Data Sending

In order to reduce delay and maximise the number of readings when a move is detected, we send IMU data over to the relay node laptop following the protocol below.

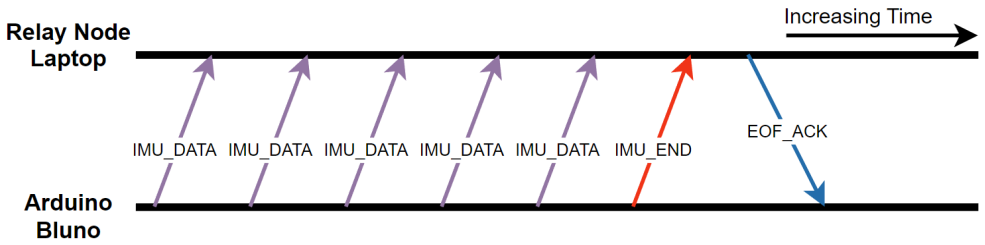


Figure 5.6: IMU Data Protocol

Upon the completion of handshake with the laptop, and at the detection of a move, the Beetle connected to the IMU will begin to continuously send data packets containing IMU readings at 27ms intervals. This continues indefinitely until the move is detected to have ended. When this happens, the *IMU_END* packet is sent to the relay node laptop, indicating that all IMU data for this move has been sent. The relay node laptop, on receiving the *IMU_END* data packet, replies with the *EOF_ACK* packet to the Beetle. On the Beetle, if an *EOF_ACK* packet is not received within the set timeout of 300ms, the *IMU_END* packet will be continually sent until acknowledgement is finally received. Only after acknowledgement will the Beetle be able to start sending for the next move.

On the relay node laptop, the frames of data received from the Beetle are unpacked and converted back into float values as they arrive, and buffered into a 1 dimensional Python list, appending new values to the back. After receiving the *IMU_END* frame, the frame is acknowledged, and the entire list containing all the IMU data received is sent over TCP to the Ultra96, and the buffer is reset, ready for the next move.

We had several considerations for choosing this implementation of the communication protocol. First, we had tried using stop-and-wait, but the data was being sent at too slow of a rate; it took approximately 100ms between the sending of a packet to the sending of the next packet. This led to large volumes of data being lost as the sending of data to the relay node laptop was lagging behind the sampling being done by the IMU, and the buffers on the Beetle were being overflowed with too much IMU data. Thus, we tried implementing a sliding window protocol next. A sliding window protocol can be much faster than stop-and-wait as acknowledgements do not need to be sent after every packet is received at the relay node side. While we did manage to implement the protocol for the IR Receiver/Emitter Beetles as a test, when it came to the IMU Beetle, many bugs surfaced. We spent a lot of time debugging the code but in vain, so we decided to cut our losses and implement the protocol in Figure 5.6 that does not have any acknowledgements for the data packets. That said, the probability that a packet sent getting corrupted was very low. We tested this by printing out a message on the terminal every time the checksum calculated fails, and saw very few of such messages being printed out. Moreover, when integrated with Hardware AI, our hand actions (i.e. *grenade, shield, reload, logout*) could be reliably detected and interpreted.

5.3.7 Packet Checksum (CRC8)

In order to maintain data packet integrity when sending data over BLE which is prone to noise and hence packet corruption, the ***CRC8.h*** library was used on the Arduino side, and ***CRCCheck*** library on the relay node side was used to verify the received data packets. CRC8 was chosen as it is 8 bits long, which fit our data packet structure, while providing sufficient error detection. On detecting faulty packet data, the entire packet is discarded, and no acknowledgement is sent. The source would eventually time out, and resend the data packet,

until acknowledgement is received.

5.4 Handling Reliability Issues

5.4.1 Packet Fragmentation

Packet fragmentation occurs when a data packet reaches the relay node laptop in chunks. This could be due to interference from other devices. To tackle this problem, start and stop delimiters of length one byte each are added into the packet format to denote the start and end of a data packet. Upon receiving a start delimiter, the relay node laptop appends the subsequent incoming data to a buffer. Further chunks of fragmented data continue to be appended to the buffer until two conditions are met – the stop delimiter is received and the total length of the received data is 20 bytes. If the two conditions are met, the packet is further processed by the rest of the Python program. However, if the stop delimiter is not the last byte of the 20 bytes of data, the data stored in the buffer is dropped. The subsequent data that is still being sent to the relay node laptop would be ignored, as the single condition for appending data to the buffer is the presence of the start delimiter.

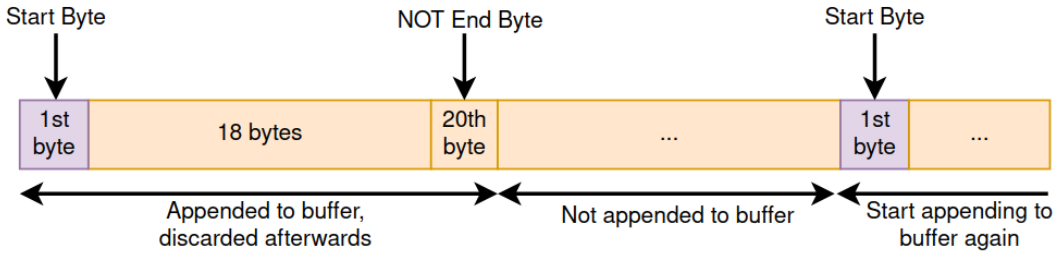


Figure 5.7: Data fragmentation handling

5.4.2 Connection drops

The BLE connection might break due to the devices being out of range, interference from other devices, low battery, or various other reasons. When that occurs, the relay node laptop will detect the disconnection and will continuously try to reconnect to the Beetle using the Beetle's MAC address. When connection is established, the laptop will initiate the three-way handshake again, and once the handshake is complete, the Beetle is able to send data to the relay node laptop.

Section 6 External Communications [Danzel Ong Jing Hern]

This section will explain in further detail how external communications between different subsystems are set up, the protocols used, message formats, software interfaces, how secure communications are ensured, as well their justifications.

6.1 MQTT (MQ Telemetry Transport)

MQTT was the main chosen communications protocol for external communications. MQTT is a lightweight protocol, using small fixed size headers, suitable for resource constrained, minimal bandwidth, or unreliable connection environments. [2] The publish/subscribe architecture of MQTT consists of 2 parts - the first, clients, which would consist of our relay node laptops, Ultra96 and software visualiser, and the second - brokers, which acts as a server and "post-office" for our clients to connect to, to receive messages which they are subscribed to, called topics. [1] This architecture makes message management simple across multiple devices, and streamlines complex processes making debugging easier.

Bi-directional communications where clients can both subscribe, and publish to certain topics, as well as being able to ensure secure, reliable message delivery, are features which MQTT provide, which are vital in a laser-game system. These features maximise the user experience in game by preventing packets being dropped which may invalidate actions performed by the player such as shooting, resulting in the feeling of a buggy, unpolished game for the player. Finally, the lightweightness and efficiency for running MQTT clients [1] ensure that more computing resources can be reserved to run computationally expensive tasks, such as AR visuals, enabling inter-device communications without affecting gameplay performance. Thus, MQTT was the main communications protocol chosen for the project.

For the project, **HiveMQ** was chosen as the main MQTT broker [3]. While other MQTT brokers are available, HiveMQ provides a free, cloud-based MQTT broker which is able to connect up to 100 devices. Furthermore, the HiveMQ broker is able to communicate effectively with a large number of client libraries spanning multiple languages such as Python, C++ and Arduino, which is required to provide reliable software interfaces between our devices, and the broker. [3] The MQTT client libraries supported by HiveMQ which will be used are: **Paho Python,** the Python MQTT client library [5] to allow communications between Ultra96 and the cloud MQTT broker, as well as **M2MqttUnity** [56] to receive MQTT messages on the software visualiser Unity client.

Being a free service also reduces cost for the project which can be allocated elsewhere, while being a cloud-based broker makes setting up connections simple, and accessible, and makes real-time data monitoring and debugging simpler. As such, HiveMQ was chosen as the primary broker for our external communications, between components.

6.2 SSH Tunnelling

As the Ultra96 sits on a separate network behind the NUS SoC Firewall, the device will have to be port-forwarded to enable devices on other networks to communicate with the Ultra96. SSH tunnelling was chosen as it allows secure, remote web access for devices without exposing local ports onto the internet [6].

To achieve this, a SSH Tunnel will be created on the Ultra96 to enable connections from other devices. After logging into the Ultra96 via SoC Sunfire SSH client, a bash script on the terminal can be run on the Ultra96 to port-forward, and expose a port for external data to reach the Ultra96 as follows:

ssh -L <Source IP>:<Source Port>:<Destination IP>:<Destination Port> <SSH Server>

Figure 6.1: Bash Script to Enable Port Forwarding

Running the script will enable tunnelling into the Ultra96 until logged out. To automate setting up of an SSH tunnel via Python, the **sshtunnel** library [7] library was used on our relay node laptops, used by the internal comms, to automate the setting up of tunnels from the relay node laptop, onto the Ultra96 to enable TCP connection.

Setting up of SSH tunnels via the library is conceptually similar to manual setup, where in the code, the *remote_bind_address* is where we input our target IP and port, while *local_bind_address* is where we input our source IP, as well as source port.

6.3 Concurrency Handling

Socket communications are typically blocking, “holding” up the system until a message is received. In our system, multiple communication connections may have to be made. For example, the 2 relay laptops would have to connect with up to 3 beetles each, receiving data

via bluetooth, and sending this data over to the Ultra96 via TCP socket. In order to allow concurrent communications with multiple connected client sockets, **multithreading** was implemented, where each connected TCP socket client has its own dedicated thread spawned to handle sending/receiving of data. Multithreading will be implemented via the default Python **threading** library [8].

Apart from using threads to handle multiple TCP clients to receive data simultaneously, threads are also created to allow our game state manager to update the game state concurrently while receiving data, improving latency. Temporary threads, such as those seen below in Figure 6.1, threads 3/4/5/6, are created by the game state manager when certain actions such as *shield, grenade* and *shoot* are performed. These temporary spawned threads are used to update the game state for players' shield timers at a steady 1 second interval, as well as to timeout damage dealing actions if no hit was registered.

Multiprocessing, involving the use of multiple processors on the Ultra96 was initially considered as an alternative to multithreading, to allow for faster computation and processing. However, as the computation tasks for handling processing of game state data were computationally light, AI classification was done on dedicated hardware, and our data was mostly bound to the network, multithreading of tasks was decided as sufficient for our use.

6.3.1 Concurrency Data Safety

To prevent race conditions where multiple threads write to the game state and override each other resulting in incorrect game state, as well as prevent data corruption when sending over networks, thread locking from the Python **threading lock** [54] library was used.

On the relay nodes running the TCP clients which received data from 3 separate beetles as seen in the figure below, each client had a dedicated lock, which had to be acquired before data could be sent over the TCP socket. This lock is released after the reply from the TCP server has been successfully received, allowing other data to be transmitted.

To ensure integrity of game state on the Ultra96, as well as to improve data processing speed, a Python **Queue** [55] was used. Due to the thread safe nature of queues, it allows our 2 TCP client handler threads to enqueue data safely into the queue. This data is dequeued sequentially by the game state manager, running on a separate thread. Similar to the relay node TCP clients, the game state manager has a single lock object, which has to be acquired by any thread which has to update the game state, to prevent race conditions.

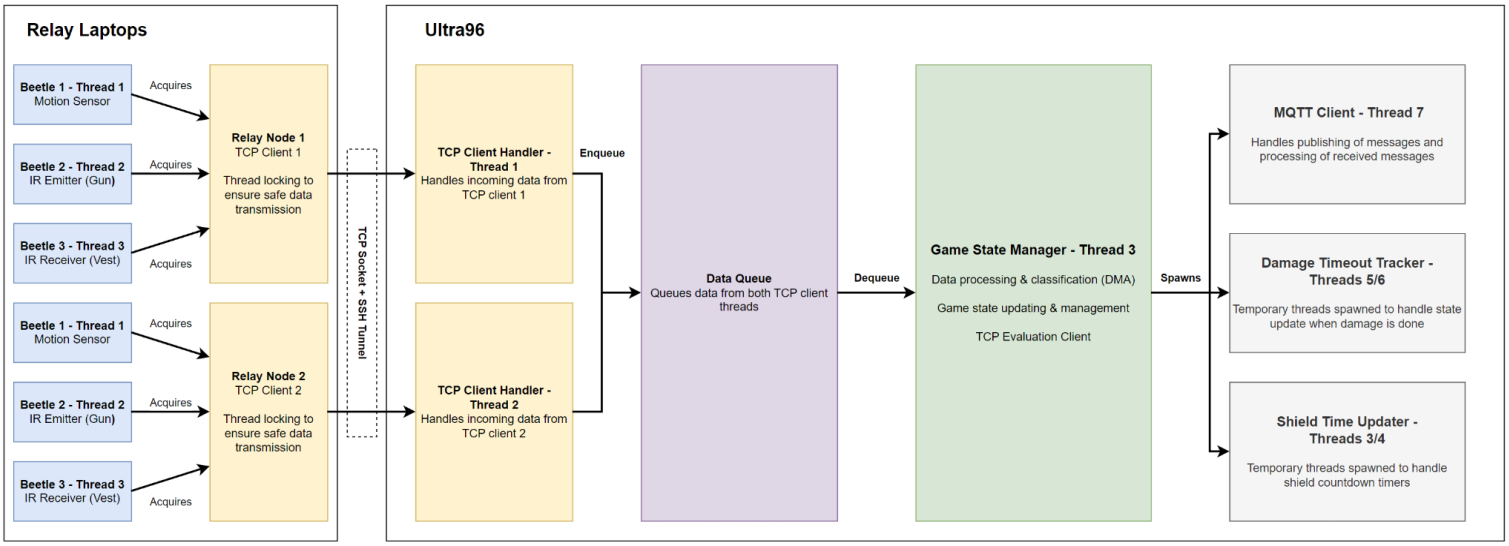


Figure 6.1: Map of Threads Implemented, as well as their interactions

6.4 Communications Between Sub-Systems

This section of the report will go into detail on how each sub-system (Evaluation server, Relay Nodes, Software Visualiser and Ultra96), communicate with each other. The diagram below illustrates high-level communications between these sub-systems.

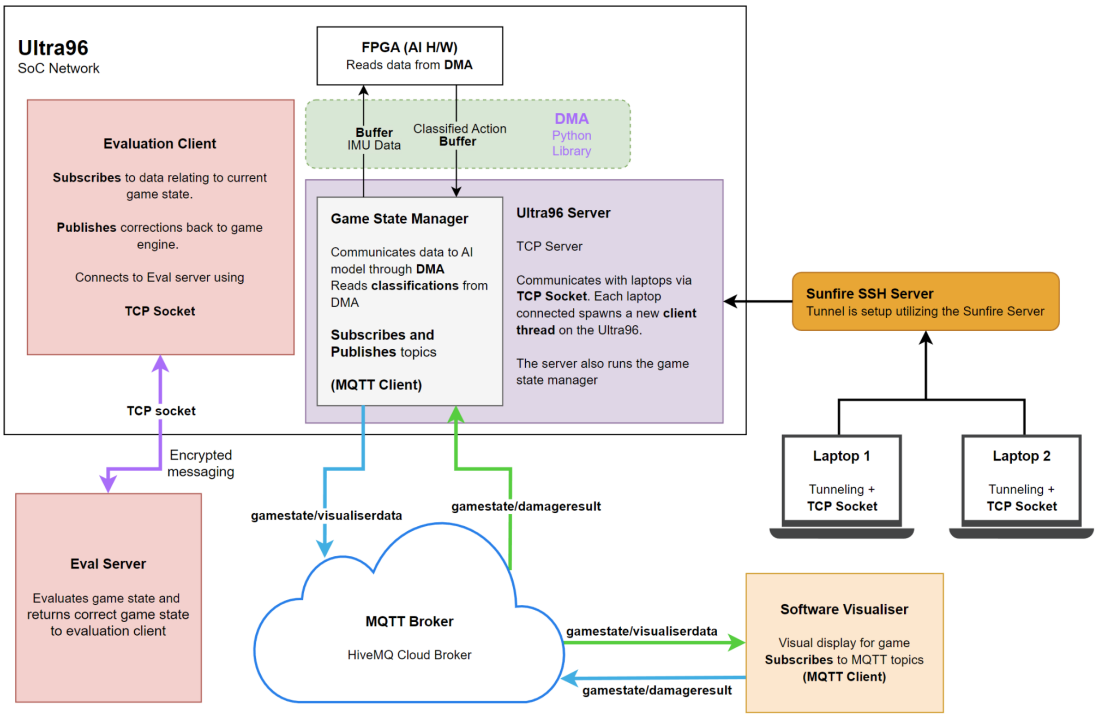


Figure 6.2: Communications Overview

MQTT communications are labelled using green and blue arrows, where green arrows represents **subscription** to messages, and blue arrows represent **publishing** of messages. In MQTT, a topic is a UTF-8 encoded string which the broker uses to filter messages to the respective clients which have subscribed to the corresponding topic [9].

6.4.1 Secure Communications

Data packets from our application will be sent over the network, and as such, it is important to encrypt our data packets to prevent information hijacking, and act as a way to authenticate if received data packets do belong to us.

As such, data will not be sent over in clear text, but instead, encrypted with the AES (Advanced Encryption Standard), symmetric block cipher, in the CBC (Ciphertext Block Chaining) mode, where a plaintext block gets XOR-ed with the previous ciphertext block before encryption. Encryption of clear text data will be done following the steps as shown in the diagram below.

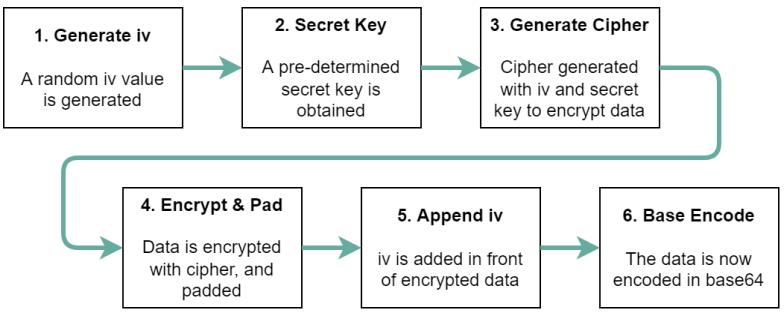


Figure 6.3: Data encryption procedure

The iv (Initialization Vector) will be of block size 16 bytes as default for AES, and the predetermined secret key will be of either 16, 24 or 32 bytes, where 1 byte = 1 character. For AES encryption, the **pycryptodome** library [10] will be utilised. Base64 encoding will be done using the default **base64** Python library [11].

As for decryption, the raw cipher text will first be base64 decoded. The iv value will then be extracted from the first 16 bytes of the decoded string. Similar to encryption, the AES cipher will then be created using the secret key, and iv. Finally, the encrypted data will be decrypted, and unpadded to return the clear text data for use. This encryption and decryption process will be used in the TCP socket communications between the evaluation client running on the Ultra96 to the evaluation server.

While AES is a moderately resource intensive encryption algorithm [12], the data packets going from the relay node to the Ultra96 via TCP socket will not be encrypted, due to the expected high volume of data being transferred between the 2 devices. Having to decrypt and encrypt the raw data from the sensors may introduce unnecessary overheads in packet size due to padding, as well affect performance, increasing latency.

6.4.2 TCP

TCP (Transmission Control Protocol), was the main protocol chosen for communications between all our devices. While UDP (User Datagram Protocol) is simpler and faster due to smaller overheads in header size, which benefits streaming large amounts of data, it is an unreliable service with no guaranteed delivery when compared to TCP, which is connection oriented, and is able to guarantee ordered, data delivery [13].

In the context of a game, while latency is important, the importance of ensuring guaranteed delivery is far greater, as having lost packets could mean player shots or actions not being registered, leading to a bad user experience. Thus, the benefits of guaranteed delivery provided by TCP outweigh that of UDP, and as such, TCP was chosen.

6.4.3 Message Formats

Messages and data will be set as stringified JSON objects, which will then be encrypted in the format as mentioned in *Section 6.4.1 Secure Communications*.

Apart from the payload data, the data packet will be prefixed with a 'header', which will contain the size of the expected payload data in bytes. A single '_' underscore denotes the end of the header, and the start of the encrypted payload of size as stated in the header. The entire data packet will then be encoded in base64, in a utf-8 format before being sent.

6.4.4 Ultra96 and Eval Server Communications

Communications between the Ultra96 and the evaluation server will be done via TCP socket connection. To achieve this, the Ultra96 board will run a Python program called the "Evaluation Client". The evaluation server and client will run the Python **socket** library [14] to communicate with each other via TCP sockets, to send and receive data.

On the Ultra96, the evaluation client is initialised together with the custom game state manager, *'QueuedGameManager.py'* when started in evaluation mode. The TCP client object is stored as a class variable, and called when the gamestate is ready to be sent to the evaluation server. The evaluation client helps prepare the packets for evaluation, including the encryption of packets, according to the steps mentioned in *6.4.1 Secure Communications*, as well as formatting the message in the correct message format as mentioned in *6.4.3 Message Format*.

The game state manager, whose role is to maintain, update and act as a source of truth for the current game state, upon receiving the evaluation server reply message after a game state is sent, will override its current game state with that sent from the eval server.

The data format sent from the eval client to the eval server is illustrated in figure 6.4 below. The message format below representing the current game state is also what is published onto the MQTT broker and subscribed by the software visualiser to update game state.

```
{
  "p1": { 'hp': 40, 'action': 'shoot', 'bullets': 3, 'grenades': 1, 'shield_time': 3, 'shield_health': 10, 'num_deaths': 1, 'num_shield': 2 }
  "p2": { 'hp': 80, 'action': 'shield', 'bullets': 0, 'grenades': 2, 'shield_time': 10, 'shield_health': 30, 'num_deaths': 1, 'num_shield': 1 }
}
```

Figure 6.4: Gamestate Data JSON Format

6.4.5 Ultra96 and Relay Node Communications

In total, 2 relay node laptops will be used, each connected to 3 separate beetles via bluetooth. Each of these laptops will set up communications with the Ultra96 to transfer data.

External communications from the relay node laptops to the Ultra96 will be done by first tunnelling onto the Ultra96 from the relay laptops, and setting up a TCP socket. The messages sent from the relay node onto the Ultra96 game state manager will not be encrypted due to the high expected volume of data, thus reducing latency incurred from the encryption and decryption process.

Data being sent from the relay nodes to the Ultra96 game state manager will follow the 'Message Format' as shown in figure 6.5 below. The 'player' field will change according to whether the data came from player 1 or 2, while the 'data_type' field is used to quickly sort the type of data packet is received, to be processed accordingly. The types of valid data types are also shown below. For example, a data packet with type 'shoot' would result in the game state manager directly updating that the corresponding player has fired a shot.

Upon receiving a data packet of type 'motion', the raw motion data contained within the 'data' field of the data packet, which is a standard 1-Dimensional Python list of float values up to 3 decimal place precision, of unfixed length, will be passed directly to the AI classifier, via DMA (Direct Memory Access), using the Python Pynq library for DMA [15]. The classified action will also be returned to the game state manager correspondingly, via DMA.

Finally, to track connection statuses of the bluetooth devices for both players, the relay node laptop sends data packets of data type 'BLE_DC', with the type of device disconnected in the 'data' field, as a list with a single string, (gun, vest or imu), to indicate that that player's device has disconnected. The process is the same for device connection, but with a packet data type of 'BLE_C' instead, to indicate the device has connected.

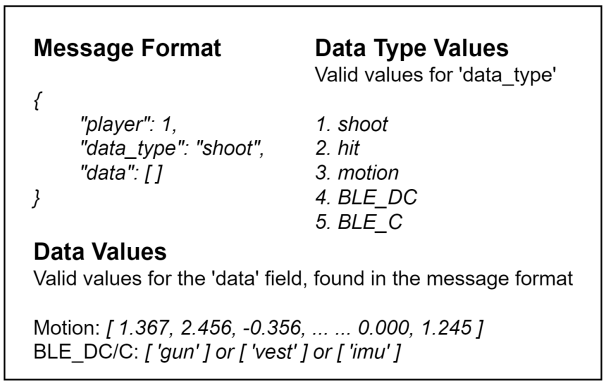


Figure 6.5: Relay Node to Ultra96 Data Format

6.4.6 Ultra96 and Visualiser Communications

The software visualiser will provide the player with important gameplay information, visually. The data relating to the current game state will be communicated to the visualiser from the game state manager running on the Ultra96, via MQTT broker.

A MQTT client, utilising the M2MqttUnity library will run on the software visualiser, which subscribes to the "gamestate/visualiserdata" topic published by the game state manager. Subscribing to the topic would allow the software visualiser to continuously listen for game state updates, and update the UI elements on the fly quickly, when the state is altered. In order to update the game state if a player has taken grenade damage, the MQTT client on the software visualiser publishes messages with topic "gamestate/damageresult", with message format as seen in **figure 6.X below**. Each field for 'p1' and 'p2' is a boolean value, set to True if the said player has taken grenade damage.

```
{ "p1": False, "p2": True }
```

Figure 6.6: Grenade damage message format

Receiving of MQTT messages on the Ultra96 is done in the *on_message()* callback function. As receiving of messages happens asynchronously, the function has to acquire the threading lock, before updating the player healths based on the message sent from the visualiser.

Game state data sent to the visualiser would follow the same JSON object form as shown in figure 6.4 above. Different from the game state however, is that the software visualiser receives an expanded set of valid actions for each player.

This includes actions such as *'hit'*, for the visualiser to display visual effects for getting hit, *'shield_down'* for the visualiser to stop displaying the shield for players, and finally, *'BLE_DC'* and *'BLE_C'* for the visualiser to update visually the connection status of all the bluetooth devices for each player.

6.5 System Evolution - External Comms and Game State Manager

Over the course of the project, the external comms and game state manager have evolved, as product needs were better understood, and better solutions were discovered.

Initially, communications between the game state manager and evaluation client were done through MQTT protocol. This resulted in a slight increase in delay, as game state must now be communicated through the cloud MQTT broker in both directions, increasing overheads such as formatting and parsing of data from text to string. This was eventually changed, and the original MQTT evaluation client was created to receive game state data directly from the game state manager.

The way the game state manager receives data also changed, when moving from the 1-player, to the 2-player game. Originally, when receiving player data through each of the 3 TCP client threads, each thread would acquire the game state for the entire duration of the processing of data, before releasing it to the other threads. This increased delay on both the relay laptops, as data was being held up on the relay node laptops, unable to send their data to the Ultra96. To combat this, a queue was implemented on the Ultra96 server. Now, all data received on the relay node laptop threads will be enqueued immediately onto the shared queue, while a separate thread feeds the incoming data into the game state manager concurrently. This frees up the TCP client socket for a much greater amount of time, allowing much more data from the players to be sent to the Ultra96.

Section 7 Software Visualizer [Hu Xuefei]

In our project, the laser tag game will be visualised on the player's phone to give the players an image of the game, and hence a more immersive experience. In this section, the design and implementation of the visualiser will be discussed.

7.1 User Survey

Before designing the visualizer, we conducted a survey where around 95% participants are active, avid FPS game players.

7.1.1 Setup related questions

The following are the questions we asked related to the base requirements of the design.

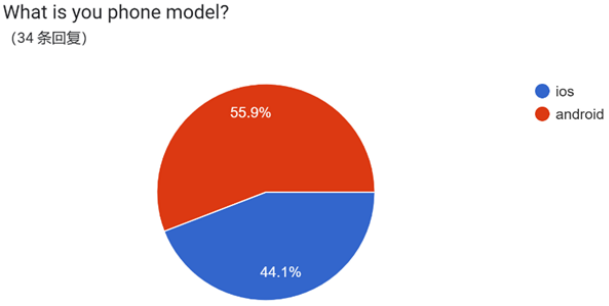


Figure 7.1: Survey question 1

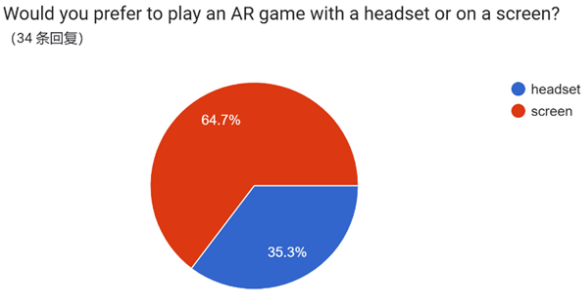


Figure 7.2: Survey question 2

From the results, we decided to develop an Android phone app, where the player's phone will be mounted on the gun and the phone to be used as a visualiser for the game.

7.1.2 Game scene related questions

The following are the questions we asked related to the game scene design.

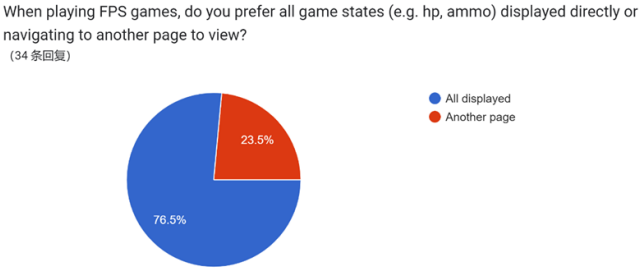


Figure 7.3: Survey question 3

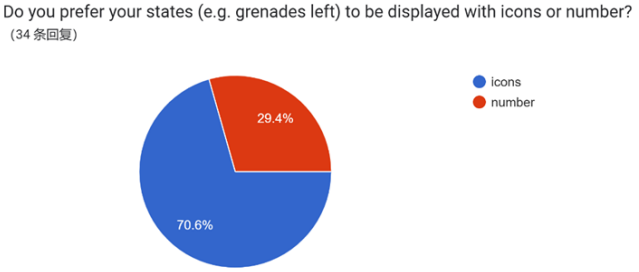


Figure 7.4: Survey question 4

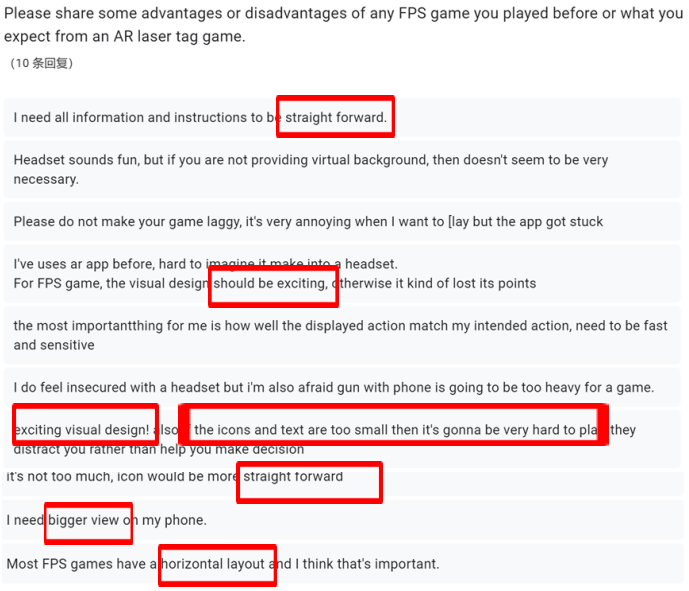


Figure 7.5: Survey question 5

From the survey results, we decided to display the ammo remaining for both the player, and opponent, represented by the number of icons on screen. The HP of each player, along with their shield, is displayed with a dynamic HP bar, which depletes proportionally with the remaining HP, as well as the numerical value to make it more straight forward.

Next, we noticed that 'excitement' is an element mentioned in multiple answers. So apart from the AR shield of players, blood splatter, shooting effects, grenade projection and explosion effects, as well as sound effects were added to enhance the game experience. Since the result of players' actions are displayed as different kinds of effects, the player's hand and opponent figure used to display actions in the initial design were taken off.

To provide the players with a bigger view, the game is designed to have horizontal layout instead of a vertical layout in the initial design. The health bar and countdown for shields are also only displayed when a shield is activated instead of staying on the screen all the time. Since the layout of the game has changed compared to the initial design, the location of the phone has also changed to be placed on top of the gun, instead of on the side.

7.2 Visualizer Design

Assets from Unity asset store are used to improve the game styling and UI Design.

7.2.1 Menu

Figure 7.6: Mode Selection Page

Figure 7.7: Player Selection Page

As shown in figure 7.6, there are two modes to choose. If "Single Player Test" is chosen, the main game play scene is entered, and the player is set to Player1 by default. This is a mode to help users to test if their actions can be correctly detected, and to get familiarised with the game. If "Two Player Mode" is selected, then the scene shown in figure 7.7 above is displayed. Once a player is selected, the main game scene is entered.

7.2.2 Gameplay

The picture below is the design of the main game. Elements at the bottom are associated with the user, while elements at the top are associated with the opponent.

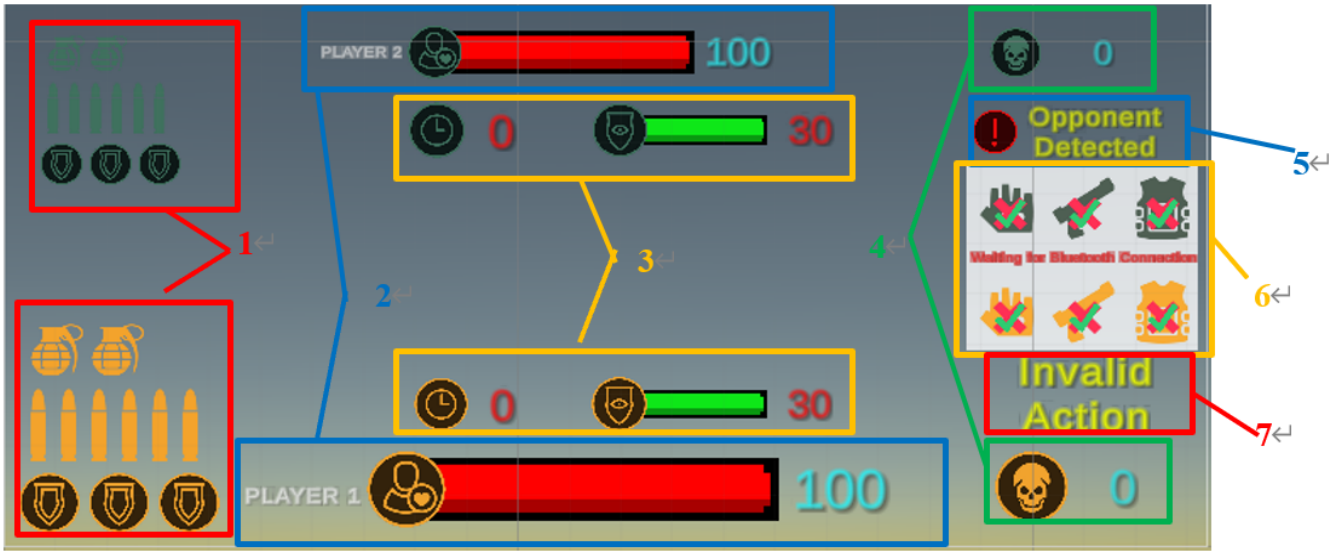


Figure 7.8: Main Gameplay Scene

Part 1: Player's consumables – displays the number of grenades, shields and bullets both players have. For example, each grenade icon symbolises a grenade remaining for the player.

Part 2: Player's status – displays the player number, and their respective remaining HP.

Part 3: Shield information – will only appear if the shield of either player is activated. This part includes a countdown in seconds for shield cooldown, and the HP of the current shield.

Part 4: Player score – shows the number of deaths for the corresponding player.

Part 5: Opponent detection alert – only appears when the opponent is detected on screen.

Part 6: Bluetooth information panel – appears whenever the bluetooth of a device disconnects. For each player device, if it is connected, a green tick is shown. Otherwise, a red cross is shown. When all 6 devices are connected, this panel will disappear. The panel is put at the side of the screen so when any device disconnects, the game can still continue.

Part 7: Invalid action alert – appears for 2 seconds when the user does an invalid action. For example, shooting when there are no bullets left, or trying to activate another shield before the shield cooldown timer reaches 0.

7.2.3 Scoreboard

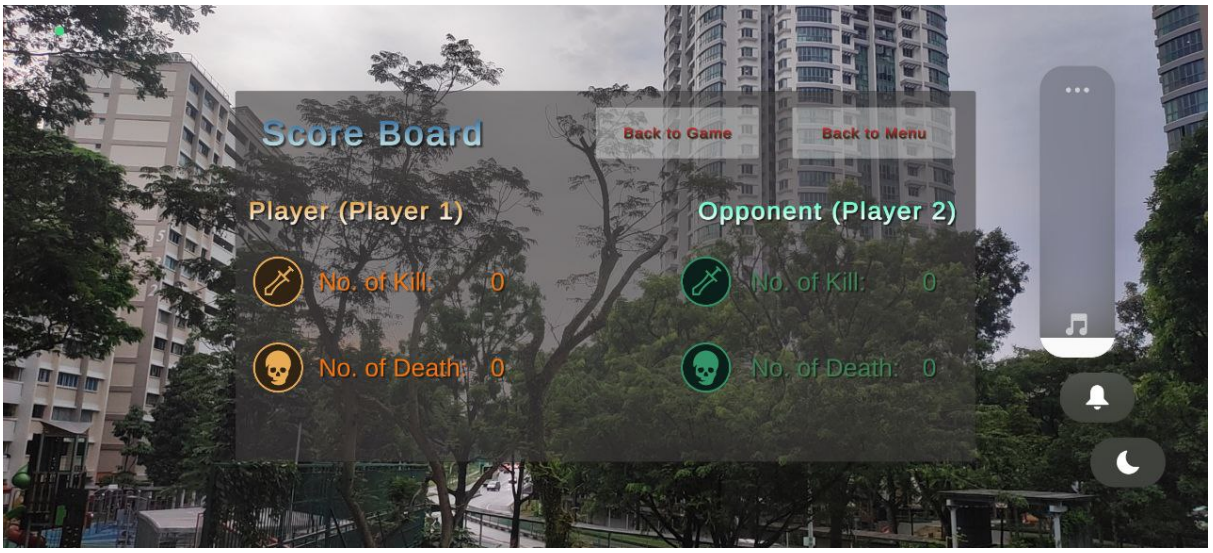


Figure 7.9: Scoreboard Page

As seen in figure 7.9, after the player logs out, the app will automatically switch to the Scoreboard scene that displays the number of opponents killed and the number of deaths for both players. The “Back to Game” button can be used when a player accidentally logs out and wants to go back to the same game The “Back to Menu” button is for the player to start a new game without closing and launching the app again.

7.3 Visualizer software architecture

7.3.1 Frameworks and Libraries

The main platform used for developing the visualizer is Unity 3D and the Unity entity-component system is adopted. All animation and sound effects are achieved by Unity built-in features. Unity built-in TextMesh Pro package is imported for better UI design.

To make use of the phone camera and enable all the AR effects, Vuforia was imported, and AR anchoring and opponent detection are handled by Vuforia engine.

For connection with External Comms, the M2MQTTUnity library is adopted to create a MQTT client on the visualiser, which connects to the MQTT broker, and subscribes to the “cg4002/b16/gamestate/visualiserdata” topic, for receiving game state information, and publishes the topic “cg4002/b16/gamestate/damageresult” for publishing information back to the game engine on whether a player has taken grenade damage from a thrown grenade.

Finally, the Newtonsoft JSON package was imported for deserialization and serialisation of messages received, and published to the MQTT broker.

7.3.2 Architecture Design

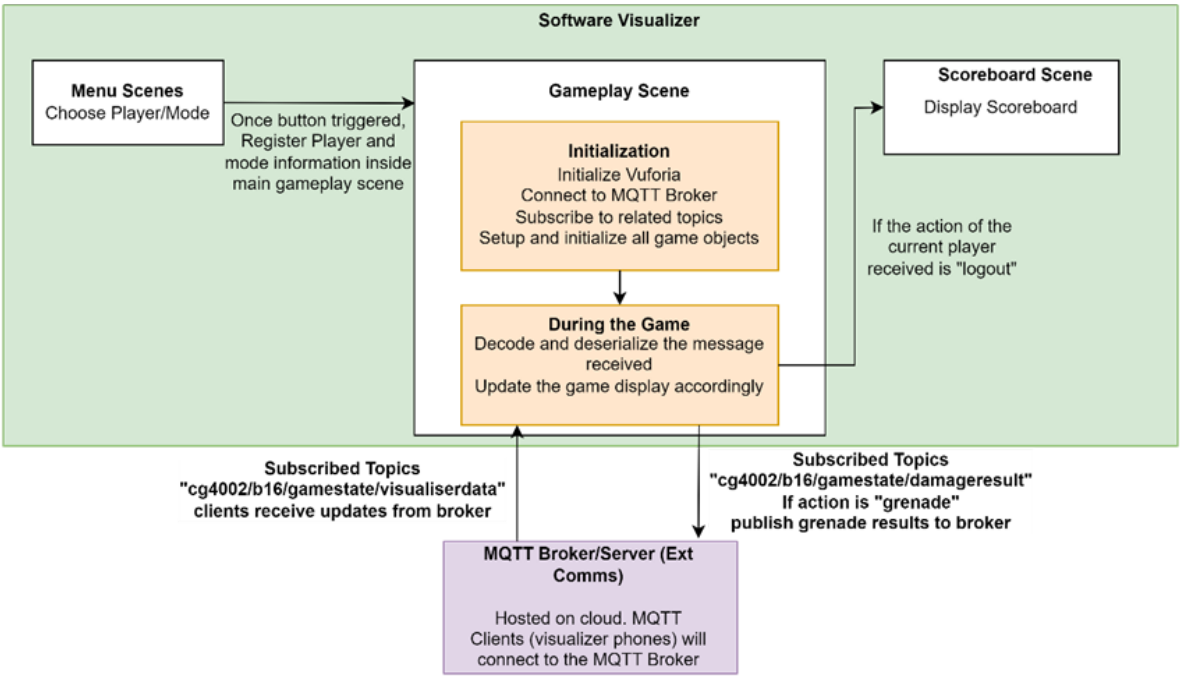


Figure 7.10: Visualizer Architecture Design

As shown in the figure above, the user will press the buttons in the menu scenes and related information is then stored in the main script associated with the main gameplay scene.

In the Gameplay scene, after initialization, the clients which are the phones used for visualizers will constantly wait for messages published onto the MQTT broker by the external comms. When receiving a message from the subscribed topic, each player's visualiser will then update accordingly based on the received game state.

Two special cases are grenade and logout. Whenever an opponent is detected by the camera, it will be registered in a Boolean flag "opponentIsDetected". If a grenade action of the current player is received, the detection status of whether the opponent is detected on camera, is checked through the flag "opponentIsDetected". If it is true, a hit is registered, else, no hit is registered. The hit detection message will then be serialised and published to the "cg4002/b16/gamestate/damageresult" topic, back to the Ultra96 to update the game state. If a "logout" action is received, the current scene exits, and enters the Scoreboard scene.

7.4 Sensors

The only sensor used is the phone camera which provides the visual background, and detects the opponents image target. The camera is accessed by the Unity ARCamera object provided by Vuforia Engine.

7.5 UI overlay

All UI designs are implemented with Unity UI and TextMesh Pro. Canvas, a Unity built in object is used. UI elements are then placed on the Canvas. Since the Canvas is an object always placed directly in front of the camera, all elements on the Canvas are overlaid on the background camera feed.

Each part of the UI design has a script associated with them to modify them, and all the functions in the scripts are called in the main "mqttReceiver.cs" script.

The following parts refer to the ones labelled in figure 7.8.

Part 1: The number of grenade, shield and bullet icons displayed for both players are controlled by grenade, shield and bullet scripts respectively. In the main script, the number of these counts received from the MQTT Broker are input to their respective function which updates the number of icons displayed.

Part 2: The player number of each player is updated when the user chooses player one or two in the menu scene, and the health bar and number is controlled by health bar script. Max HP is inputted during initialization and Hp state received from MQTT Broker is used to update the HP value and health bar display.

Part 3: The countdown is controlled by the timer script. It can control whether the clock icon and number is enabled, hence displaying and updating the value of seconds left. The shield health bar is associated with the same script as the player health bar, which can update the HP state according to max HP and input HP and control if the health bar and HP value is enabled and displayed.

Part 4: The number of deaths is controlled by the death script which updates the number according to number input. So when a new game state is published to the broker, the main script receives it and inputs the new death number to update.

Part 5: The icon and text are controlled by the opponentDetection script. With the built-in feature that Vuforia Image Target provides, when an opponent is detected, the *isDetected* method in opponentDetection is called to enable the warning sign and text. When the opponent is lost, the isLost method in opponentDetection is called to disable the sign and text.

Part 6: On initialization, all the bluetooth statuses are initialised to disconnected. Whenever a device connects or disconnects, a message will be published to the MQTT broker and the Bluetooth script associated with the icon representing the corresponding device will be called to choose between a tick or a cross. Each device has a Boolean flag in the main script indicating their Bluetooth connection states. If all flags are set to true, all devices are connected, and the parent empty object containing this panel will be deactivated, causing the whole panel to disappear.

Part 7: When a gamestate with invalid action is received, the invalid action text is enabled, and a flag count is set to 2f. Each frame count decreases the flag by one Time.deltatime. The count takes 2 seconds to reach 0, disabling the invalid action text. Hence, the invalid action text is only displayed for 2 seconds upon activation.

7.6 AR effects

7.6.1 Grenade

A GrenadeGo script is used to spawn each 3D rigidbody grenade object, despawn it, play the explosion animation after 2 seconds, and then finally destroy the object. In the GrenadeThrower script, a force is set to act on the rigidbody grenade object. The grenade is now projected forward before it explodes. In order to differentiate from whom the grenade is thrown, (Player vs opponent), 2 methods are used, forwardThrow and backThrow, which applies the acting force on the grenade from opposite directions. Along with setting different grenade starting positions with respect to the camera for both the player and the opponent, this helps create the visual effect of either the player throwing a grenade, or a grenade from

the opponent being thrown at the player, establishing different grenade throwing effects for both players.

7.6.2 Shield

The player shield is controlled by a playerShield script, in which a sphere with designed material is activated or deactivated through the Unity built in SetActive method. The AR camera is placed inside the sphere, so when the sphere is activated or deactivated, a protector-like effect will either appear or disappear.

The opponent's shield is controlled by a detected script, the opponent shield will be activated or deactivated through the script.

7.6.3 Shoot and Blood Splatter effect

When a player or opponent shoots, a muzzle flash animation is played through a DisplayShoot script. When players take bullet damage, the screen will flash red. This is achieved by the FlashRed script which uses Coroutine and IEnumerator to measure the time for the Red panel to fade in and fade out. When an opponent takes bullet damage, a blood splatter animation is played through a DisplayBlood script.

7.6.4 Object Anchoring

The opponent's shield is anchored to the QR Code, i.e. the image target used for opponent detection. In this way, the shield will move and rotate around with the opponent.

The rest of the assets are anchored to the player camera. The 3D objects for these effects move around with the camera. In this way, all these effects are fixed at a certain position on the screen.

Section 8 Societal and Ethical Impact

This section will weigh the pros and cons that surround the societal and ethical impact and the potential of such systems. Going wild, the use of motion control and AR systems can be applied to the field of criminal forensics. Crime scenes are not static, and change/deteriorate with time. However, with modern technologies such as 3D scanning, which are able to scan and record the positions of objects with accuracies up to the millimetres, we will be able to preserve these crime scenes along with their perishable evidence, digitally, forever. In the future, AR systems like that in our laser tag game, along with motion controls, will allow witnesses, the persecuted, and lawyers to revisit original crime scenes, augmented with the digital recreations of the preserved evidence, to view the scene at time of crime, re-enact events, and use motion controls to alter these objects in the digital environment.

From our research, there is a world of opportunity for technology to benefit humanity. However, there will always exist individuals with ill-intent that want to exploit technology for malicious gain. The privacy concerns related to AR wearables are due to the fact that they overlay graphics on live camera feeds, and the data from what the user is viewing, where he is, can be collected. To prevent ethical issues and protect privacy, the protection of personal data collected from AR wearables need to be of utmost importance. Data collected has to be anonymized and all traces to individuals must be either removed or encrypted. Additionally, data usage by stakeholders must also be regulated and consented by users.

References

1. "MQTT: The Standard for IOT Messaging", mqtt.org, https://mqtt.org/ [Accessed: 18 August 2022]
2. Corinne Bernstein, Kate Brush, Alexander S. Gillis, "MQTT (MQ Telemetry Transport)", TechTarget https://www.techtarget.com/iotagenda/definition/MQTT-MQ-Telemetry-Transport [Accessed: 18 August 2022]
3. "Reliable Data Movement For Connected Devices", HiveMQ. https://www.hivemq.com/ [Accessed: 21 August 2022]
4. Nick O'Leary, "Arduino PubSubClient - MQTT Client Library Encyclopedia", HiveMQ. https://www.hivemq.com/blog/mqtt-client-library-encyclopedia-arduino-pubsubclient/ [Accessed: 18 August 2022]
5. Roger Light, "Paho Python - MQTT Client Library Encyclopedia", HiveMq. https://www.hivemq.com/blog/mqtt-client-library-paho-python/ [Accessed: 18 August 2022]
6. Sakshyam Shah, "SSH Tunneling Explained", Teleport. https://goteleport.com/blog/ssh-tunneling-explained/ [Accessed: 18 August 2022]
7. Pahaz, "sshtunnel 0.4.0", GitHub https://github.com/pahaz/sshtunnel/ [Accessed: 18 August 2022]
8. "Threading – Thread-based parallelism", Python https://docs.python.org/3/library/threading.html [Accessed: 18 August 2022]
9. HiveMQ Team, "MQTT Topics, Wildcards, & best Practices - MQTT Essentials: Part 5", HiveMQ. https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/ [Accessed: 21 August 2022]
10. "AES", PyCryptodome. https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html [Accessed: 18 August 2022]
11. "Base64 - Base16, Base32, Base64, Base85 Data Encodings", Python. https://docs.python.org/3/library/base64.html [Accessed: 18 August 2022]
12. Ron Franklin, "AES vs RSA Encryption: What Are the Differences?", precisely. https://www.precisely.com/blog/data-security/aes-vs-rsa-encryption-differences [Accessed: 21 August 2022]
13. Matt Cook, "TCP vs. UDP: What's the Difference?", Lifesize. https://www.lifesize.com/en/blog/tcp-vs-udp/ [Accessed: 18 August 2022]
14. "Socket – Low-Level networking interface", Python. https://docs.python.org/3/library/socket.html [Accessed: 21 August 2022]
15. "DMA", Python productivity for Qynz (Pynq). https://pynq.readthedocs.io/en/v2.1/pynq_libraries/dma.html#:~:text=In%20the%20Python%20code%2C%20two,other%20for%20output%20are%20allocated.&text=Transfer%20the%20input_buffer%20to%20the,DMA%20to%20the%20output%20buffer. [Accessed: 21 August 2022]
16. S. Sasidhar, "CG4002: Computer Engineering Capstone Project Hardware AI", *NUS* [Online] [Accessed: 19 August 2022]
17. Dasmehdixtr, "Human action recognition dataset," *Kaggle*, 25-Dec-2019. [Online]. Available: https://www.kaggle.com/datasets/dasmehdixtr/human-action-recognition-dataset [Accessed: 19 August 2022]
18. P. P. Ippolito, "Feature extraction techniques," *Medium*, 11-Oct-2019. [Online]. Available: https://towardsdatascience.com/feature-extraction-techniques-d619b56e31be. [Accessed: 19 August 2022]

19. “hls4ml Quick start,” *Quick Start · GitBook*. [Online]. Available: https://fastmachinelearning.org/hls4ml/setup/QUICKSTART.html. [Accessed: 19 August 2022]
20. *Continuous motion recognition - edge impulse documentation*. [Online]. Available: https://docs.edgeimpulse.com/docs/tutorials/continuous-motion-recognition. [Accessed: 19 August 2022]
21. “Welcome to HLS4ML's documentation!,” *Welcome to hls4ml's documentation! - hls4ml 0.5.1 documentation*. [Online]. Available: https://fastmachinelearning.org/hls4ml/index.html [Accessed: 19 August 2022]
22. “Hls4ml tutorial,” *Google Slides*. [Online]. Available: https://docs.google.com/presentation/d/1c4LvEc6yMByx2HJs8zUP5oxLtY6ACSizQdKvw5cg5Ck/edit#slide=id.ge9c66f87b4_6_1. [Accessed: 19 August 2022]
23. A. Taylor, “Accelerating your ultra96 developments!,” *Hackster.io*, 01-Oct-2018. [Online]. Available: https://www.hackster.io/adam-taylor/accelerating-your-ultra96-developments-806a72. [Accessed: 19 August 2022]
24. *Spectral features - edge impulse documentation*. [Online]. Available: https://docs.edgeimpulse.com/docs/edge-impulse-studio/processing-blocks/spectral-features. [Accessed: 19 August 2022]
25. S. Turney, “Skewness: Definition, Examples & Formula,” *Scribbr*, 12-Jul-2022. [Online]. Available: https://www.scribbr.com/statistics/skewness/. [Accessed: 19 August 2022]
26. S. Turney, “What is kurtosis?: Definition, Examples & Formula,” *Scribbr*, 18-Jul-2022. [Online]. Available: https://www.scribbr.com/statistics/kurtosis/. [Accessed: 19 August 2022]
27. “Neural network settings,” *Classification (Keras) - Edge Impulse Documentation*. [Online]. Available: https://docs.edgeimpulse.com/docs/edge-impulse-studio/learning-blocks/classification. [Accessed: 19 August 2022]
28. T. Yiu, “Understanding random forest,” *Medium*, 29-Sep-2021. [Online]. Available: https://towardsdatascience.com/understanding-random-forest-58381e0602d2. [Accessed: 19 August 2022]
29. “DMA,” *DMA - Python productivity for Zynq (Pynq) v1.0*. [Online]. Available: https://pynq.readthedocs.io/en/v2.4/pynq_libraries/dma.html. [Accessed: 19 August 2022]
30. Cathalmccabe, “Tutorial: Using a HLS stream IP with DMA (Part 1: HLS design),” *PYNQ*, 25-Nov-2021. [Online]. Available: https://discuss.pynq.io/t/tutorial-using-a-hls-stream-ip-with-dma-part-1-hls-design/3344. [Accessed: 19 August 2022]
31. Cathalmccabe, “Tutorial: Using a HLS stream IP with DMA (Part 3: Using the HLS IP from PYNQ),” *PYNQ*, 25-Nov-2021. [Online]. Available: https://discuss.pynq.io/t/tutorial-using-a-hls-stream-ip-with-dma-part-3-using-the-hls-ip-from-pynq/3346. [Accessed: 19 August 2022]
32. “Fast machine learning,” *Indico*. [Online]. Available: https://indico.cern.ch/event/822126/contributions/3482454/. [Accessed: 19 August 2022]
33. “Xilinx documentation portal,” *Documentation Portal*. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Area-Information. [Accessed: 19 August 2022]
34. “Xilinx documentation portal,” *Documentation Portal*. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Using-the-Flow-Navigator. [Accessed: 19 August 2022]

35. “Xilinx documentation portal,” *Documentation Portal*. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Generating-and-Opening-the-HLS-Report. [Accessed: 19 August 2022]
36. M. Dealessandri, “What is the Best Game Engine: Is unity right for you?,” GamesIndustry.biz, 16-Jan-2020. [Online]. Available: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you . [Accessed: 21-Aug-2022]
37. GitHub. 2022. GitHub - eclipse/paho.mqtt.m2mqtt. [online] Available at: <https://github.com/eclipse/paho.mqtt.m2mqtt.git > [Accessed 21 August 2022].
38. Docs.unity3d.com. 2022. About AR Foundation | AR Foundation | 4.2.3. [online] Available at: <https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@4.2/manual/index.html > [Accessed 21 August 2022].
39. GitHub. 2022. Body-Tracking-ARKit/HumanBodyTracking.cs at master · LightBuzz/Body-Tracking-ARKit. [online] Available at: <https://github.com/LightBuzz/Body-Tracking-ARKit/blob/master/body-tracking-arkit/Assets/Scripts/HumanBodyTracking.cs > [Accessed 21 August 2022].
40. K. Townsend, “Introduction to bluetooth low energy,” Adafruit Learning System. [Online]. Available: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gap. [Accessed: 21-Aug-2022].
41. GitHub. 2022. GitHub - homuler/MediaPipeUnityPlugin: Unity plugin to run MediaPipe graphs. [online] Available at: <https://github.com/homuler/MediaPipeUnityPlugin.git > [Accessed 21 August 2022].
42. “CC2540,” CC2540 data sheet, product information and support | TI.com. [Online]. Available: https://www.ti.com/product/CC2540#features. [Accessed: 21-Aug-2022].
43. K. Townsend, “Introduction to bluetooth low energy,” Adafruit Learning System. [Online]. Available: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt. [Accessed: 21-Aug-2022].
44. “Bluno_sku_dfr0267,” DFRobot. [Online]. Available: https://wiki.dfrobot.com/Bluno_SKU_DFR0267. [Accessed: 21-Aug-2022].
45. “Interfaces, exceptions and utils,” Interfaces, exceptions and utils - bleak 0.16.0a1 documentation. [Online]. Available: https://bleak.readthedocs.io/en/latest/api.html?highlight=mac+address#module-bleak.backends.bluezdbus.client. [Accessed: 21-Aug-2022].
46. “Developer help,” Bluetooth® Low Energy Packet Types - Developer Help. [Online]. Available: https://microchipdeveloper.com/wireless:ble-link-layer-packet-types. [Accessed: 21-Aug-2022].
47. Cevinius, “Serial communication with the Bluno using Bluetooth Le,” cevinius, 17-Aug-2016. [Online]. Available: https://www.cevinius.com/2016/08/17/serial-communication-with-the-bluno-using-bluetooth-le/. [Accessed: 21-Aug-2022].

48. “How standard deviation relates to root-mean-square values - technical articles,” *All About Circuits*. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/how-standard-deviations-relates-rms-values/. [Accessed: 31-Aug-2022].

49. “Understanding parametric tests, skewness, and kurtosis - technical articles,” *All About Circuits*. [Online]. Available:

https://www.allaboutcircuits.com/technical-articles/understanding-the-normal-distribution-parametric-tests-skewness-and-kurtosis/. [Accessed: 31-Aug-2022].

50. "Scipy.stats.skew," *scipy.stats.skew - SciPy v1.9.3 Manual*. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.skew.html. [Accessed: 10-Nov-2022].

51. Admin, "Examples of kurtosis calculation (step by step)," *All Things Statistics*, 08-Jan-2022. [Online]. Available: https://allthingsstatistics.com/descriptive-statistics/kurtosis-calculation-example/#:~:text=The%20Kurtosis%20of%20a%20given,%CF%83%20denotes%20the%20standard%20deviation. [Accessed: 31-Aug-2022].

52. "Softmax function," *Wikipedia*, 05-Aug-2022. [Online]. Available: https://en.wikipedia.org/wiki/Softmax_function. [Accessed: 31-Aug-2022].

53. A. Biswal, "Top 10 deep learning algorithms you should know in 2022," *Simplilearn.com*, 19-Jul-2022. [Online]. Available: https://www.simplilearn.com/tutorials/deep-learning-tutorial/deep-learning-algorithm. [Accessed: 31-Aug-2022].

54. "Threading – Thread-based parallelism: Lock-Objects", Python. [Online]. Available: https://docs.python.org/3/library/threading.html#lock-objects [Accessed: 6-November-2022]

55. "Queue- A synchronized queue class", Python. [Online]. Available: https://docs.python.org/3/library/queue.html [Accessed: 6-November-2022]

56. "M2MqttUnity - M2MQTT for Unity", CE-SDV-Unity [Online Repository]. Available: https://github.com/CE-SDV-Unity/M2MqttUnity [Accessed: 11-November-2022]

57. Edgeimpulse, "Processing-blocks/spectral-analysis," *GitHub*. [Online]. Available: https://github.com/edgeimpulse/processing-blocks/tree/master/spectral-analysis. [Accessed: 11-Nov-2022].

58. "DMA," *DMA - Python productivity for Zynq (Pynq)*. [Online]. Available: https://pynq.readthedocs.io/en/v3.0.0/pynq_libraries/dma.html. [Accessed: 11-Nov-2022].

59. "Export IP Invalid Argument / Revision Number Overflow Issue (Y2K22)," *Xilinx Customer Community*. [Online]. Available: https://support.xilinx.com/s/article/76960?language=en_US. [Accessed: 11-Nov-2022].

60. C. Hongzheng, "Vivado HLS in a Nutshell," *Vivado HLS in a Nutshell · Hongzheng Chen*, 11-Mar-2020. [Online]. Available: https://chhzh123.github.io/blogs/2020-03-11-vivado-hls/. [Accessed: 11-Nov-2022].

61. C. Hongzheng, "Pynq & Zynq Soc Tutorial," *Pynq & Zynq SoC Tutorial · Hongzheng Chen*, 19-Jan-2021. [Online]. Available: https://chhzh123.github.io/blogs/2021-01-19-fpga-soc/. [Accessed: 11-Nov-2022].

62. W. Knitter, "Introduction to using AXI DMA in embedded linux," *Digilent Projects*, 29-Jul-2021. [Online]. Available:

https://projects.digilentinc.com/whitney-knitter/introduction-to-using-axi-dma-in-embedded-linux-5264ec. [Accessed: 11-Nov-2022].

63. N. Roussos and G. Evangelou, “A framework for developing neural networks in hardware accelerators,” *GitHub*, 10-Apr-2021. [Online]. Available: https://github.com/n-roussos/A-framework-for-developing-Neural-Networks-in-hardware-accelerators. [Accessed: 11-Nov-2022].

64. “HLS Pragmas,” *Xilinx Documentation Portal*, 19-Oct-2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas. [Accessed: 11-Nov-2022].

65. R. C. Panicker, “EE4218 Labs,” *Wiki.nus*, 25-Jan-2022. [Online]. Available: https://wiki.nus.edu.sg/display/ee4218. [Accessed: 11-Nov-2022].

66. “Lab: Axistream single DMA (axis),” *Lab: Axistream Single DMA (axis) - pp4fpgas 0.0.1 documentation*, 2019. [Online]. Available: https://pp4fpgas.readthedocs.io/en/latest/axidma.html. [Accessed: 11-Nov-2022].

67. C. McCabe, “PYNQ axi DMA Example,” *YouTube*, 21-Oct-2021. [Online]. Available: https://www.youtube.com/watch?v=K4OkNH17hiA. [Accessed: 11-Nov-2022].

68. B. H. Fletcher, “Ultra96-V2 Vivado 2020.2 basic hardware platform,” *Hackster.io*, 26-Apr-2021. [Online]. Available: https://www.hackster.io/BryanF/ultra96-v2-vivado-2020-2-basic-hardware-platform-6b32b8. [Accessed: 11-Nov-2022].

69. “Streaming floats with TLAST,” *Xilinx Customer Community*, 06-Jun-2022. [Online]. Available: https://support.xilinx.com/s/question/0D52E00007DnHxuSAF/streaming-floats-with-tlast?language=en_US. [Accessed: 11-Nov-2022].

70. P. Kazarinoff, “Using python and an Arduino to read a sensor,” *Python for Undergraduate Engineers*, 11-Mar-2021. [Online]. Available: https://pythonforundergradengineers.com/python-arduino-potentiometer.html. [Accessed: 11-Nov-2022].

71. “Relu,” *ReLU - PyTorch 1.13 documentation*. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html. [Accessed: 12-Nov-2022].

72. “Softmax,” *Softmax - PyTorch 1.13 documentation*. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html. [Accessed: 12-Nov-2022].

73. Electrical4U, “Butterworth Filter: What is it? (Design & Applications),” *Electrical4U*, 16-Apr-2021. [Online]. Available: https://www.electrical4u.com/butterworth-filter/. [Accessed: 12-Nov-2022].