Design & Innovation Project (DIP)

Project Report

<<Tele Driving of a Mobile Robot>>

Project Group: <<031>>

School of Electrical and Electronic Engineering

Academic Year 2023/24

Semester 1

Table of Contents

Acknowledgements

Before delving into the project, our team E031 (Tele Driving of a Mobile Robot) would like to express our gratitude and appreciation to the School of Electrical and Electronic Engineering for initiating the Design and Innovation Project. We would also like to express our gratitude to our mentors and supervisors, Associate Professor Dr. Su Rong, Li Jiangpeng, and Dr. Zhao Meiqi, for their guidance and help during these 13 weeks.

Our group is grateful for the opportunity to come together and work on such an interesting project. We all dedicated much time to work on the various challenges that we encountered and eventually overcame them with support from one another and our mentors.

1. Purpose/ Project Objectives

1.1 Main Objective

Our main task is to create a mobile robot vehicle that can be controlled with a steering wheel and pedal system with a digital twin during the 12 weeks of our project.

We decided to use existing technologies that support remote-controlled or teleoperated capabilities. Sensors such as an ESP32 camera allow for real-time feedback and communications to our robot, and an indoor antenna positioning system allows the x-y coordinates of our robot to be transmitted to our digital twin on Unity.

1.2 Inspiration For Project

Our primary motivation and goal lie in creating a prototype of a mobile robot that would be used in disaster rescue efforts.

The eventual aim is that our robot can safely and efficiently navigate small cracks and crevices in a collapsed building or disaster site to quickly locate individuals trapped by debris, such that an extraction plan can be quickly put in place once the trapped person is found. This would help various countries improve their disaster relief efforts by significantly reducing the risk taken by first responders.

2. Project Summary

For our robot to be functional in hazardous or inaccessible areas, we had to overcome various problems. Firstly, we had to find a way to remotely control our robot with an interface that is intuitive to use. A steering wheel and pedal set-up was implemented as it is an interface that many are familiar with. This allows for ease of training when learning to use the robot, should this be applied on a large scale.

Secondly, the user had to be able to get a good sense of the robot's surroundings, allowing for efficient navigation of otherwise inaccessible places. A camera, along with a virtual 3-Dimension (3D) environment, was integrated into the robot. The camera allows the user to see in real-time what obstacles they are facing, allowing them to traverse the obstacles. The virtual environment, paired with our local positioning system (LPS), gives the driver a way to see what they are expected to face. However, this 3D environment would be generated based on what the driver is predicted to face. Hence, the camera is needed to allow the driver to view their surroundings in real-time.

Lastly, but most importantly, to transmit both the output from the camera and the inputs to control the robot, we sent our information through an ESP32 wifi module that the camera has attached. This allowed us to communicate by setting up a client and server to send and receive various inputs, allowing us to effectively control our robot.

For our digital twin, the purpose was to create a system to aid in positional awareness of the robot. This can be helpful while navigating complex environments as we would always be able to obtain the x-y coordinates of the current location of our robot. Thus, we implemented an LPS that tracks the robot and displays its position on a 3D representation of the environment. The LPS would be an antenna positioning system, consisting of 4-6 antenna base stations acting as receivers and one antenna tag acting as a transmitter. The transmitter placed on the robot will transmit and display its location on Unity.

Our final challenge and main focus lay in integrating our digital twin and our steering-controlled vehicle, which we were able to achieve. In the process of integration, we created identical mazes in real life and on Unity for testing and determining the exact responsiveness and ideal speed for the vehicle's movements.

3. Scope

3.1 Building the Robot

Upon receiving the ELEGOO Smart Robot Car Kit V4.0 in our second week, we began the construction of the Robot Car immediately and completed it within the day. The Smart Robot Car Kit V4.0 came with various sensors, like line tracking and ultrasonic sensors that we decided against using due to the lack of implementation time during our project. We then familiarised ourselves with the functions and capabilities of the robot to better understand how we could use it to achieve our project objectives.

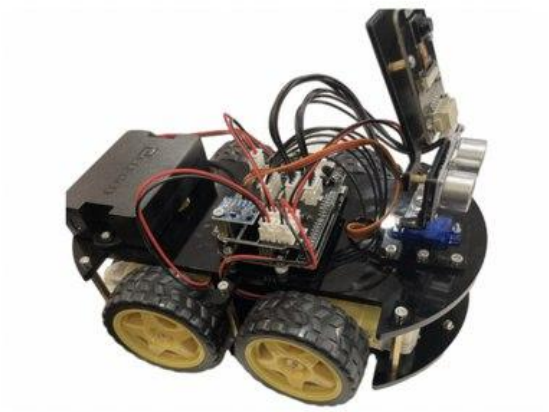

Figure 1: ELEGOO Smart Robot Car Kit V4.0

3.2 Setting Up Connection to ESP32

3.2.1 ESP32

To establish communication between the laptop and ESP32, we have chosen to make use of a Transmission Control Protocol (TCP) connection to deliver commands from the laptop to the ESP32.

We first started by establishing the ESP32 as a server within the TCP connection. This is done by making use of the <WiFi.h> library from Arduino. As seen from Figure 2 below, we define the ESP32 as a server with port 100.

Main.ino | WebServer.cpp | WebServer.h | app_httpd.cpp | camera_index.h | camera_pins.h

```
1   #include "WebServer.h"
2   #include <WiFi.h>
3   #include <SPI.h>
4
5   #define RXD2 33
6   #define TXD2 4
7
8   WebServer web_server;
9   WiFiServer server(100);
10  WiFiClient client;
```

Figure 2: ESP32 Server and Client Definition

Then, we define any potential connections to the port as a client for the ESP32 to listen to. Additionally, we define RXD2 and TXD2, which are the physical pins the ESP32 will use to transmit commands to the main Arduino UNO board, which will execute commands based on the inbuilt code.

Next, we had to set conditions for the ESP32 to receive commands. If no conditions are set, the ESP32 will listen to every bit that is transmitted to it by clients, including empty bits when nothing is transferred. This will disrupt the receiving of commands and cause some commands not to be executed. Hence, we set the condition for the ESP32 to only accept commands within the curly brackets "{" and "}". Thus, any characters or bits within the brackets will be stored within a string named "readBuff". This can be seen in Figure 3 below.

```
if(client.connected())
{
  while(client.available())
  {
    char c = client.read();
    if(c == '{')
    {
      printing = true;
      readBuff += c;
      //client.write(c);
    }
    if(c == '}')
    {
      printing = false;
      readBuff += c;
      Serial.println(readBuff);
      Serial2.print(readBuff);
      readBuff = "";
    }
    if(c != '{' && c != '}' && printing == true)
    {
      readBuff += c;
      //Serial.print(c);
    }
```

Figure 3: Setting conditions to accept commands

After the command has been fully received, signified by a "}", the ESP32 will send the commands to the main Arduino UNO board through the "Serial2.print" command, which will make use of the previously defined pins.

Lastly, we connect the ESP32 to a WiFi network, and it is ready to receive commands from clients.

3.2.2 Laptop Command Code

On the laptop side, our first objective was to establish a TCP connection to the ESP32 server. This was done by first configuring the laptop to be a client to connect to the Internet Protocol (IP) address and port of the ESP32. This can be seen in Figure 4 below.

```
void Robot::_connect()
{
    // Connect to robot
    cout << "Step 1 - Setting up DLL" << endl;

    int port = 100;
    WSADATA wsaData;
    int wsaerr;
    WORD wVersionRequested = MAKEWORD(2, 2);
    wsaerr = WSAStartup(wVersionRequested, &wsaData);
    if (wsaerr != 0)
    {
        cout << "The Winsock dll not found!" << endl;
    }
    else
    {
        cout << "The Winsock dll found!" << endl;
        cout << "The Status: " << wsaData.szSystemStatus << endl;
    }

    cout << "Step 2 - Set up Client Socket" << endl;
    clientSocket = INVALID_SOCKET;
    clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (clientSocket == INVALID_SOCKET)
    {
        cout << "Error At socket(): " << WSAGetLastError() << endl;

    }
    else
    {
        cout << "socket() is OK!" << endl;
    }

    cout << "Step 3 - Connect with Server Socket" << endl;

    sockaddr_in clientService;
    clientService.sin_family = AF_INET;
    InetPton(AF_INET, _T("192.168.59.246"), &clientService.sin_addr.s_addr);
    clientService.sin_port = htons(port);
```

Figure 4: Connecting code

After successfully establishing the connection, we established the commands to be sent based on the received inputs from the keyboard. The commands include:

- Moving forwards
- Moving backwards
- Turning left
- Turning right
- Turning the camera left
- Turning the camera right

Figure 5 below shows the forward command that we have written.:

```
if (instruct == "w")
{
    tell = "{\"N\":102,\"D1\":1,\"D2\":";
    actual_speed = ceil((test_speed / 9) * 155) + 100;
    string speed = to_string(actual_speed);
    speed += "}";
    tell.append(speed);
}
```

Figure 5: Movement command

We wrote the commands in a specific format: “N:102,D1:X,D2:X.” This format was required as the robot would only execute commands sent using this format. Table 1 below explains the parameters of the commands.

Parameter	Explanation
N	N is the type of command called by the robot; in this case, 102 relates to the movement of the robot.
D1	D1 is the direction of the movement; it is configured from 1-4: ● 1 is forward. ● 2 is backward. ● 3 is left. ● 4 is right.
D2	D2 is the speed of the movement; it is configured with a range of 0-255.

Table 1: Format Explanation

These commands are then sent to the ESP32 through the established TCP connection. This can be seen in Figure 6 below:

```
if (instruct == "d")
{
    tell = "{\"N\":102,\"D1\":4,\"D2\":";
    actual_speed = ceil((test_speed / 9) * 155) + 100;
    string speed = to_string(actual_speed);
    speed += "}";
    tell.append(speed);
}
else if (instruct == "p")
{
    tell = "{\"N\":100}";
}



char buffer[200];
strcpy_s(buffer, tell.c_str());


int byteCount = send(clientSocket, buffer, 200, 0);

if (byteCount > 0)
{
    cout << "Message sent: " << buffer << endl;
}
else
{
    WSACleanup();
}
```

Figure 6: Sending Commands Code

The commands are first stored in a character array. The command is then placed into an integer called “byteCount”. Whenever a command is called, the byte Count will be more than one, and hence, the command will be sent to the ESP32.

After the commands have been sent and the user is no longer using the robot, the user will be able to disconnect the laptop from the robot by calling the disconnect code that we have written. This tells the laptop to detach itself from the network socket it was connected to. This can be seen in Figure 7 below:

```
void Robot::_disconnect()
{
    cout << "Step 5 - Close Socket" << endl;

    WSACleanup();
}
```

Figure 7: Disconnect Code

With the connection established and the code for the movement commands written, we now have to integrate the keyboard inputs into the movement commands. This will be discussed in Section “3.2.4 Steering Wheel Integration.”

3.2.3 Laptop Camera Code

The second objective on the laptop side was to display a stream of the ESP32 camera, allowing the user to receive real-time visual information while driving the robot. We did this by writing C++ code using the openCV library, which allows us to display videos. After this, we connected the C++ code with the IP address of the camera to obtain the video stream of the camera. The code written can be seen in Figure 8 below:

```
#include <iostream>
#include <string>
#include <list>
#include <windows.devices.wifi.h>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace cv;

void main()
{
    VideoCapture cap;
    cap.open("http://192.168.0.23/video.mjpg", CAP_ANY); //Enter Camera IP Address
    if (!cap.isOpened())
    {
        std::cout << "Camera not linked.\n";
    }
    namedWindow("Robot View", WINDOW_AUTOSIZE);
    Mat frame;

    while (true)
    {
        bool check = cap.read(frame);
        if (!check)
        {
            std::cout << "No Image Connected.\n";
            break;
        }
        imshow("Robot View", frame);
        waitKey(1);
    }
}
```

Figure 8: Laptop Camera Code

With this code in place, the user will now be able to control the robot as well as obtain real-time visual information.

3.2.4 Steering Wheel Integration

To integrate the Logitech G29 Steering Wheel and Pedals with the Laptop Command Code in **Section 3.2.2**, input signals from the Logitech G29 Steering Wheel and Pedals must first be decoded so that the data from the input signals can be read and integrated into the main programme.

First, we explored the Logitech G29 Steering Wheel and Pedals documentation for any information on how the input signals are read [1]. We found that the input from the Logitech G29 Steering Wheel and Pedals utilised a data structure implemented in Microsoft DirectInput named DIJOYSTATE2, as shown in Figure 9 below.

```c++
typedef struct DIJOYSTATE2 {
    LONG lX;
    LONG lY;
    LONG lZ;
    LONG lRx;
    LONG lRy;
    LONG lRz;
    LONG rglSlider[2];
    DWORD rgdwPOV[4];
    BYTE rgbButtons[128];
    LONG lVX;
    LONG lVY;
    LONG lVZ;
    LONG lVRx;
    LONG lVRy;
    LONG lVRz;
    LONG rglVSlider[2];
    LONG lAX;
    LONG lAY;
    LONG lAZ;
    LONG lARx;
    LONG lARy;
    LONG lARz;
    LONG rglASlider[2];
    LONG lFX;
    LONG lFY;
    LONG lFZ;
    LONG lFRx;
    LONG lFRy;
    LONG lFRz;
    LONG rglFSlider[2];
} DIJOYSTATE2, *LPDIJOYSTATE2;
```

Figure 9: DIJOYSTATE2 Structure as defined in Microsoft DirectInput

Next, we identified which attribute under the DIJOYSTATE2 structure contributes to which input signal from the Logitech G29 Steering Wheel and Pedals, as well as the values of the input signal. To do this, we wrote a programme that displays the input from the Logitech G29 Steering Wheel and Pedals.

```
1   #include <conio.h>
2   #include <windows.h>
3
4   #include <cstdio>
5   #include <vector>
6
7   // DirectInput
8   #ifndef DIRECTINPUT_VERSION
9   #define DIRECTINPUT_VERSION 0x0800
10  #endif  // DIRECTINPUT_VERSION
11  #include <dinput.h>
12
13
14  using namespace std;
15  using Devices = std::vector<LPDIRECTINPUTDEVICE8>;
```

Figure 10: Header Files, Namespace and typedef

In Figure 10, these lines include necessary header files for console I/O, Windows API, standard input/output, and DirectInput, a set of APIs in Microsoft Windows for handling input from input devices like keyboards, mice, and game controllers, which includes the Logitech G29 Steering Wheel and Pedals. Line 15 introduces a type alias named Devices. The “std::vector” is a container class template provided by the C++ Standard Library that represents a dynamic array. Here, it is used to store pointers to objects of type LPDIRECTINPUTDEVICE8.

The LPDIRECTINPUTDEVICE8 is a Microsoft DirectX type representing a pointer to a “DirectInput " device in the Windows API. It serves as a handle or reference to the input device, the Logitech G29 Steering Wheel and Pedals.

```
58          //      // Print the wheel axis.
59          if (steeringWheelActivated)
60          {
61              printf("Wheel: %10d\n", input.lX);
62              //printf("Throttle: %10d\n", input.lY);
63              //printf("Brake: %10d\n", input.lRz);
64              //printf("Right Shifter Paddle: %10d\n", input.rgbButtons[4]);
65              //printf("Left Shifter Paddle: %10d\n", input.rgbButtons[5]);
66
67              // Find the index of the buttons
68              // int counter = 0;
69              // for (counter = 0; counter < 128; counter++){
70              //      if (input.rgbButtons[counter] == 128){
71              //          printf("Button no. : %10d\n", counter);
72              //      }
73              // }
74
75          }
```

Figure 11: Display Steering Wheel and Pedal Values

In Figure 11, we are then able to print the values of the Logitech G29 Steering Wheel and Pedals onto the console at regular intervals to determine which attributes in the DIJOYSTATE2 structure correspond to which input signal generated from the Logitech G29 Steering Wheel and Pedals, which can be seen in Table 2 below.

Attribute	Input	Range of Values
LONG lY	Throttle	65 535 to 0
LONG lRz	Brake Pedal	65 535 to 0
LONG lX	Steering Wheel	**Left Steer:** 0 to 32 767 (Center Point) **Right Steer:** 32 767 to 65 535
BYTE rgbButtons[4]	Right Shifter Paddle	**Unpressed:** 0 **Pressed:** 128
BYTE rgbButtons[5]	Left Shifter Paddle	**Unpressed:** 0 **Pressed:** 128

Table 2: Attributes with their corresponding inputs and range of values

Using the information in Table 2, we can now write a C++ programme designed to perform controlled actions such as movement of the car and camera using the established communication between the ESP32 and the laptop based on steering wheel and pedal inputs.

```
66      // ------------------------------
67      // Do some initial settings here, like connection, etc.
68      Robot::max_speed = 255;  // Set max speed
69      Robot robot = Robot("robot_1", robot_address);  // Create robot object
70      // ------------------------------
```

Figure 12: Creating the object 'robot'

The object 'robot' is created, which will then be called whenever a command is made based on the Logitech G29 Steering Wheel and Pedals as shown in Figure 12 above.

```
72      // ------------------------------
73      // This is the main loop.
74      bool steeringWheelActivated = false;
75
76      bool stop_critera = false;
77      bool moving = false;
78      while (stop_critera == false)
79      {
```

Figure 13: Flags introduced

Referring to Figure 13, the following flags are established:

- **steeringWheelActivated**: This flag is used to prevent any inputs from the Logitech G29 Steering Wheel and Pedals from executing a command at startup until the Logitech G29 Steering Wheel and Pedals are activated.
- **stop_critera:** This flag is used to exit the main WHILE loop when the stopping criteria is met.
- **moving:** This flag is used as an indicator of whether the robot car is stationary or mobile.

```
87          // Check if the steering wheel is activated
88          if (input.lX != 32767)
89          {
90              steeringWheelActivated = true;
91          }
```

Figure 14: Condition for Steering Wheel Activation

Since the value of the Logitech G29 Steering Wheel is fixed at 32 767 until an input is first established, we create a condition that ultimately tells the programme that the steering wheel is not yet activated, as seen in Figure 14. This prevents the robot car from executing any unintended commands.

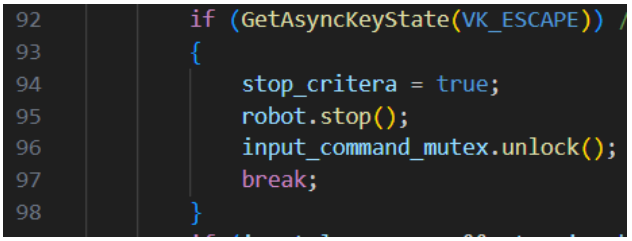

Figure 15: Condition for stopping the program

In Figure 15, we exit the programme when the ESC key is pressed on the laptop, ensuring that the robot first becomes stationary.

```
99          if (input.lY < 65535 && steeringWheelActivated == true)
100         {   // Throttle is pressed
101             moving = true;
102             input_command = 'W';
103             current_speed1 = ceil(((65535 - input.lY)/65535.0)*190) + 10;
104             robot.go_forward(current_speed1);
105         }
106         else if (input.lRz < 65535 && steeringWheelActivated == true)
107         {   // Brake (Reverse) is pressed
108             moving = true;
109             input_command = 'S';
110             current_speed1 = ceil(((65535 - input.lRz)/65535.0)*190) + 10;
111             robot.go_backward(current_speed1);
112         }
113         else if (input.lX <= 30567 && steeringWheelActivated == true)
114         {   // Steering wheel left
115             moving = true;
116             input_command = 'A';
117             current_speed1 = ceil(((30567 - input.lX)/30567.0)*155) + 50;
118             robot.turn_left(current_speed1);
119         }
120         else if (input.lX >= 34967 && steeringWheelActivated == true)
121         {   // Steering wheel right
122             moving = true;
123             input_command = 'D';
124             current_speed1 = ceil(((input.lX - 34967)/30568.0)*155) + 50;
125             robot.turn_right(current_speed1);
126         }
127         else if ((input.rgbButtons[5]) == 128)
128         {   // Camera turn left
129             input_command = 'N';
130             current_speed1 = 0;
131             robot.cam_turn_left();
132         }
133         else if ((input.rgbButtons[4]) == 128)
134         {   // Camera turn right
135             input_command = 'M';
136             current_speed1 = 0;
137             robot.cam_turn_right();
138         }
139         else //No command has been input
140         {
141             if (moving == true){
142                 moving = false;
143                 robot.stop();
144             }
145             input_command_mutex.unlock();
146             continue;
147         }
```

Figure 16: Robot Car commands

In Figure 16, IF statements were written to match input signals from the steering wheel and pedals to the appropriate commands of the robot car. The input signals from the throttle are used to control the forward movement of the robot car, where a variable speed ranging from 10 to 200 was decided. As for the reverse movement of the robot car, input signals from the brake pedal are used with the same variable speed range as the forward movement. We decided to vary the speed of turning in the range of 50 to 255 through input signals from the steering wheel. A threshold in the range of 30,567 to 34,967 was created to prevent the robot car from turning uncontrollably.

As for the movement of the camera, we decided to use the left and right shifter paddles as input signals. The left shifter paddle, when pressed, will execute the command to turn the camera on the robot car to the left, while the right shifter paddle will execute the command to turn the camera on the robot car to the right. Each press of the shifter paddle will move the camera by 1 unit up to 8 units on either side.

An ELSE condition is required to ensure that the robot car does not continue its last executed command continuously despite the absence of an input signal from the Logitech G29 steering wheel and pedals. Furthermore, the use of the moving flag prevents the Laptop from continuously sending stop commands to the robot car.

3.3 Setting Up Unity 3D Virtual Environment

3.3.1 Maze

A virtual maze was prototyped in Unity using in-engine 3D cuboid objects for the walls. A playspace of 2.45m by 2.45m was determined as the maximum limit for the demonstration. To ensure that the real-life car could realistically navigate the maze, the maze's hallways were made at least 35cm wide to accommodate its size. Additionally, we had to scale the virtual maze to reflect real-life measurements.

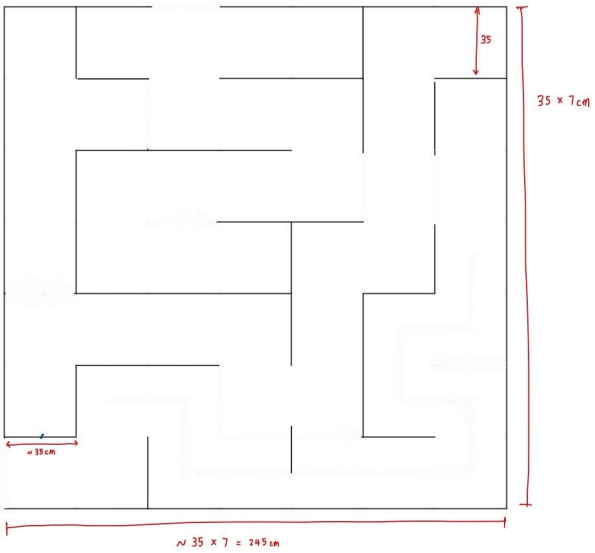

Figure 17: Maze Schematic/Drawing

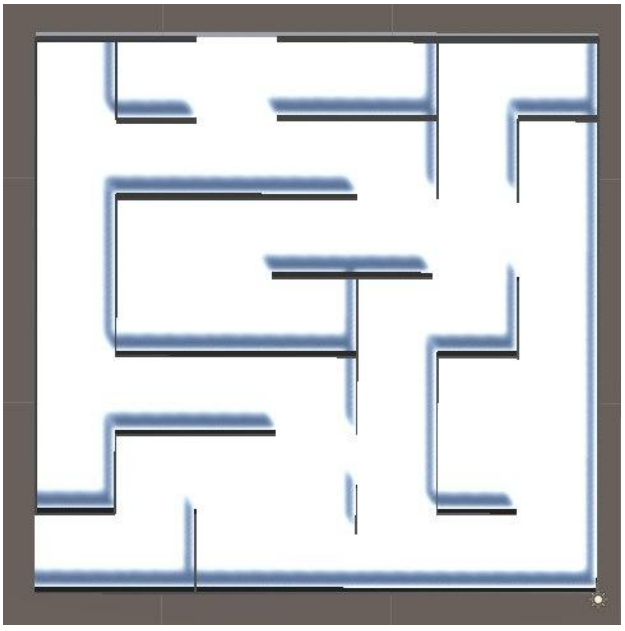

Figure 18: Maze Representation in Unity

3.3.2 Physical Maze

A physical maze was created to mimic the virtual maze generated in Unity. This allowed us to test the accuracy of our LPS and determine if there were any changes to be made. It also allowed us to test the precision of the vehicle and adjust the variable speeds accordingly to give the operator the most functional experience in handling the vehicle.

In our case, the physical maze represents a potentially inaccessible environment that humans are not able to enter. Hence, our UGV helps bridge this gap to assist in disaster relief.

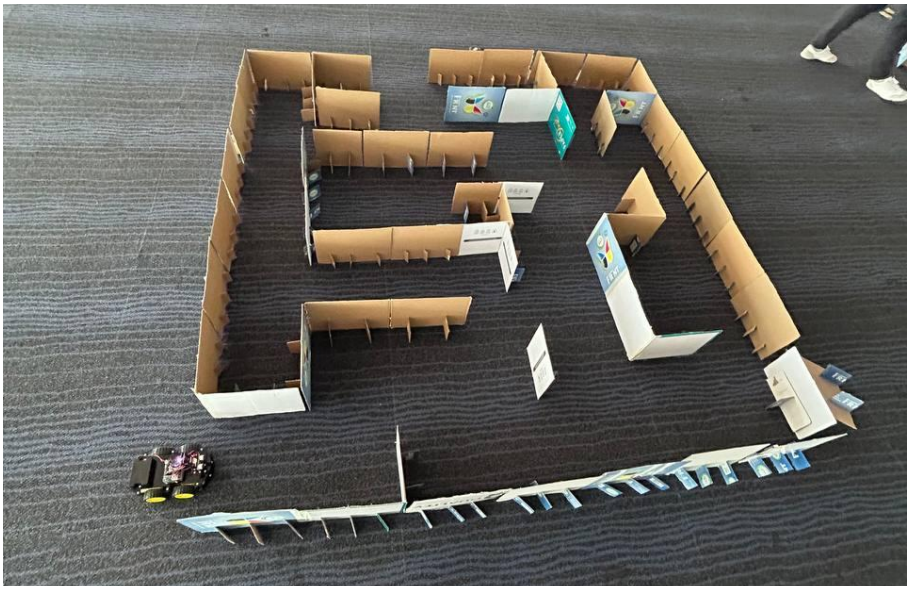

Figure 19: The Physical Maze

3.4 Integrating Local Positioning System

3.4.1 Communicating Coordinates to Unity

The LPS utilises an array of 4 antenna modules, one of which is connected to a computer and acts as the main anchor, with 3 other modules acting as secondary anchors. The 4 modules are arranged in a rectangular formation, the area of which comprises the play area of the smart robot car. A fifth transmitter module that acts as a tag is attached to the car and relays its position relative to the 4 modules to the main anchor. The output is a set of hexadecimal coordinates that require decoding before being used in Unity.

The following codes utilise the WebSocket and WebSocketSharp packages in C# programming to parse the hexadecimal coordinates from the antenna into Unity. In Figure 20, Unity establishes a WebSocket Server upon initialisation and awaits connection from the WebSocket Client.

```
// Start is called before the first frame update
void Start()
{
    if ((server == null) && (isConnected == false))
    {
        var server = new WebSocketServer("ws://127.0.0.1:8002");
        server.AddWebSocketService<MyWebSocketBehavior>("/");
        server.Start();

        isConnected = true;
        Debug.Log("Server Established");
        check_status();
    }

}
```

Figure 20: WebSocket Server initialised on the first frame rendered by Unity

The WebSocket client is then initialised as soon as the antenna array starts tracking, as seen in Figure 21. Importantly, the client can only be started after the server has been established beforehand.

```
4 references
public class WebSockectClient
{
    private static WebSocket ws;
    Uri uri = null;
    bool isUserClose = false;

    1 reference
    public static WebSocket Instance()
    {
        Console.WriteLine("client 11");
        if (ws == null)
        {
            ws = new WebSocket("ws://127.0.0.1:8002");
            ws.Connect();
            //ws.OnMessage += (sender, e) =>
            //{
            //Console.WriteLine("Received message:");
            //Console.WriteLine(e.Data);
            //};

            try
            {
                ws.Send("Client Received!");
            }
            catch (InvalidOperationException)
            {

            }
        }

        return ws;
    }

    3 references
    public static void Send(string message)
    {
        try
        {
            ws.SendAsync(message, (delegate (bool b) { Console.WriteLine("send : " + b); }));
        }

        catch (InvalidOperationException)
        {

        }
```

Figure 21: WebSocket Server Instance and Send Functions

Whenever a new coordinate is determined by the antenna array, it sends it via the WebSocket connection, which triggers the OnMessage function in Unity to signify a new set of coordinates has been sent and is ready for use. This OnMessage function and further functions that were called are represented in Figure 22.

```
public class MyWebSocketBehavior : WebSocketBehavior
{
    public string coord;
    public float HorizontalInput = 0f;
    public float VerticalInput = 0f;

    protected override void OnMessage(MessageEventArgs e)
    {
        Debug.Log($"Receive from client:{e.Data}");
        coord = e.Data;

        string file = @"C:\Users\zetin\Desktop\Intermediate2.txt";
        File.AppendAllText(file, e.Data);

        List<string> receivedHexValues = ReadSpecificHexValuesFromFile(coord, "01 03 30 CA DA 00 03 00 00 00 01");

        (int HorizontalInput, int VerticalInput) = ConvertHexToCoordinates(receivedHexValues);
        Debug.Log($"Converts: {HorizontalInput},{VerticalInput}");

    }
```

Figure 22: OnMessage function receiving message *e* from WebSocket

3.4.2 Decoding Coordinates for Unity

After parsing the hexadecimal coordinates into Unity, they need to be changed into decimal coordinates to be usable in the virtual environment. Figure 23 shows an example of the string of the hexadecimal values being processed.

```
Receive: 01 03 30 CA DA 00 03 00 00 00 01 00 0F 00
 00 2E 00 00 00 51 01 CC 02 4F 01 4E 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 50 54 [20:41:23 111]
```

Figure 23: Sample hexadecimal string

First, we determine which strings start with a pre-specified prefix. This prefix is only sent when the antenna array has an accurate coordinate reading with no errors, the rest of which we would like to filter out to avoid false movement of our virtual car. This is done by the ReadSpecificHexValuesFromFile function. The valid strings are then written into a list, where each hexadecimal value is its entry, and passed into the ConvertHexToCoordinates function. Here, we specify the index of hexadecimal values where the x and y coordinates are located and convert them into decimal values, which we can then use to move the virtual car. Note that the antenna may still transmit false values far outside the intended threshold, which is handled by making the x and y coordinates 100000 meant to signify a rejected value. These decimal coordinates are written into a text file with a specified path and name.

```
static List<string> ReadSpecificHexValuesFromFile(string line, string prefix)
{
    List<string> specificHexValues = new List<string>();

    if (line.StartsWith("Receive : ") && line.Contains(prefix))
    {
        Match match = Regex.Match(line, @"Receive : (.+?)\[\d{2}:\d{2}:\d{2} \d{3}\]");

        if (match.Success)
        {
            string hexValue = match.Groups[1].Value.Trim();
            specificHexValues = hexValue.Split(' ').ToList();
        }
    }

    return specificHexValues;
}

static (int, int) ConvertHexToCoordinates(List<string> hexValues)
{

    if (hexValues.Count >= 19)
    {
        int x = Convert.ToInt32(hexValues[13] + hexValues[14], 16);
        int y = Convert.ToInt32(hexValues[15] + hexValues[16], 16);
        int z = Convert.ToInt32(hexValues[17] + hexValues[18], 16);

        string file = @"C:\Users\zetin\Desktop\newCoords.txt";
        string CoordToWrite;

        if ((x > 30000) || (y > 30000))
        {
            x = 100000;
            y = 100000;
        }

        CoordToWrite = x.ToString() + " " + y.ToString();

        File.AppendAllText(file, CoordToWrite + Environment.NewLine);

        return (x, y);
    }

    else
    {
        return (100000, 100000);
    }

}
```

Figure 24: Converting a hexadecimal string into usable decimal coordinates

Finally, to move the car around in the virtual environment, the decimal values are read from the text file whenever Unity renders a new frame. As the rate of incoming coordinates is faster than the number of frames Unity renders, the last line of coordinates is read to ensure the most up-to-date position is used. We also implemented a filtering system that rejects the previously mentioned value of 100000 as well as any moment-to-moment fluctuations that could result in jerky movement. In this case, there needs to be a minimum 2 cm difference in any direction between the latest and previous coordinates to trigger any movement of the car.

```
// Update is called once per frame
void FixedUpdate()
{
    string[] lastLine = File.ReadLines(@"C:\Users\zetin\Desktop\newCoords.txt").Last().Split(" ");
    HorizontalInput = float.Parse(lastLine[0]);
    VerticalInput = float.Parse(lastLine[1]);
    Debug.Log($"Passed: {HorizontalInput}, {VerticalInput}");

    if ((HorizontalInput < 10000) && (VerticalInput < 10000))
    {
        float HoriDiff =  Math.Abs(oldHori - HorizontalInput);
        Debug.Log($"HoriDiff: {HoriDiff}");
        float VertiDiff = Math.Abs(oldVerti - VerticalInput);
        Debug.Log($"VertiDiff: {VertiDiff}");

        if ((HoriDiff > 2) || (VertiDiff > 2))
        {
            moveCar(HorizontalInput, VerticalInput);
        }
    }

    else
    {
        moveCar(oldHori, oldVerti);
    }

}


public void moveCar(float HorizontalInput, float VerticalInput)
{
    UnityEngine.Vector3 a = transform.position;
    moveDirection = new UnityEngine.Vector3(HorizontalInput, 5.49f, VerticalInput);

    transform.position = UnityEngine.Vector3.MoveTowards(a, moveDirection, movementSpeed * Time.deltaTime);
    Debug.Log($"Car moved to: {HorizontalInput},{VerticalInput}");
    oldHori = HorizontalInput;
    oldVerti = VerticalInput;
}
```

Figure 25: IF statements used to filter bad coordinates before calling the function to move virtual car

4. Schedule

PHASE	Planned Milestone Date	Actual Milestone Date
Initiating Phase	**Week 1-2** ● Basic Knowledge Learning: Learning C#, C++, and Arduino Programming ● Draft Charter Report: 1. Planning Milestones 2. Division into subgroups 3. Choosing a Leader & Treasurer 4. Financial Planning ● Arduino Keyboard Control: Initial connection between Arduino & laptop using ESP32 ● Unity Keyboard Control: "WASD" control of the robot model in Unity	**Week 1-2** Proceeded as planned
Planning Phase	**Week 3-4** ● Completing the Project Charter ● Beginning work on camera control ● Beginning work on steering wheel integration:	**Week 3-4** Proceeded as planned

	General familiarisation ● Beginning work on Unity coordinate control: instead of a keyboard, we input x-y coordinates for movement	
Execution Phase	**Week 5-6** ● Sending PC inputs to Arduino and the Camera through the ESP32 WiFi Module ● Controlling the 3D Unity model using coordinates ● Reading and translating the steering wheel control library to send information to the Arduino ● Receiving Images from the Camera **Week 7-8** ● Setting up a test venue using the antennas and measuring the dimensions of the test venue ● Reading communication ports in Unity and decoding information to real-time coordinates ● Adding the decoded coordinates into scripts to control the virtual robot **Weeks 9-10**	**Week 5-6** Proceeded as planned **Week 7-8** Proceeded as planned **Week 9-10** Proceeded as planned

	● Linking our various systems, debugging, and testing.	
Closing Phase	**Week 11-12** ● Linking subtasks into the final product: 1. Steering wheel-controlled robot 2. Unity transmits coordinates from antennas ● Testing of a robot in open space: 1. Building a physical maze 2. Testing the functionality of antenna positioning ● Debugging/Fixing errors: 1. Connectivity between the steering wheel & robot 2. Coordinates communication between Unity & Antennas ● Draft of slides & report	**Week 11-12** Proceeded as planned
Project End Date	**Week 13** ● Finishing the Final Report	**Week 13** Proceeded as Planned

	• Finishing Presentation Slides	

4.1 Differences Between Planned & Actual Schedules

Except for the execution phase, everything went smoothly. During execution, we experienced some challenges trying to control our robot with our laptop. The ELEGOO Smart Robot Car Kit V4 was the latest model in the ELEGOO Smart Robot Car series, yet its documentation was far less comprehensive. We had very little reference material to work with when figuring out the connection between the robot and our laptop. This caused some inconvenience, which thankfully did not hinder the projected timeline.

4.2 Benefits of Planning

Planning helped keep us on task and held us accountable for our work. It creates a system where we can quickly identify areas we are falling behind on, time-wise, such that manpower and efforts can be diverted to focus on areas at risk of falling behind first.

5. Cost

PHASE	Planned Costs	Actual Costs
Initiating Phase	$898.92 (2x ELEGOO Smart Car Kit + 1x Antenna Positioning system set)	$898.92
Planning Phase	0	0
Execution Phase	0	$35.16 (2x ESP32 cards)
Closing Phase	0	0
Project Total Costs	$898.92	$934.08

Table 3: Purchases Made

The reason there were additional incurred costs was due to a problem we faced during the execution phase: we could not send the commands we wanted to the ESP32 module that came with the ELEGOO Smart Robot Car.

Initially, we assumed it to be the ESP32 card, which came with the robot car, that was faulty and not able to receive our inputs. This resulted in us purchasing additional ESP32 cards. However, we found out later that the inputs we were referred to use by the user manual were inaccurate, and this was the reason why we could not send inputs to the ESP32 Module.

6. Outcomes / Benefits

6.1 Project Outcomes

6.1.1 Steering Wheel Integration

We successfully integrated a steering wheel and pedal control to simulate the action of driving a car with our robot. This similarity gives users a sense of familiarity and hence allows for new users to pick up the skill of manoeuvring our robot quickly. Our steering wheel has paddles located at the back, which we use to control the movement of the servo attached to the camera module. This allows the operator to rotate the camera left and right to get a better view of his surroundings and prevent the robot from colliding with obstacles.

We integrated a variable speed for our different movements, namely forward, backward, turning left, and turning right. The variable speed was integrated by taking the input values of the steering wheel output and mapping them to the respective motor speeds to turn the car left and right. The pedal also had a range of output values, which we mapped to the respective forward and backward speeds.

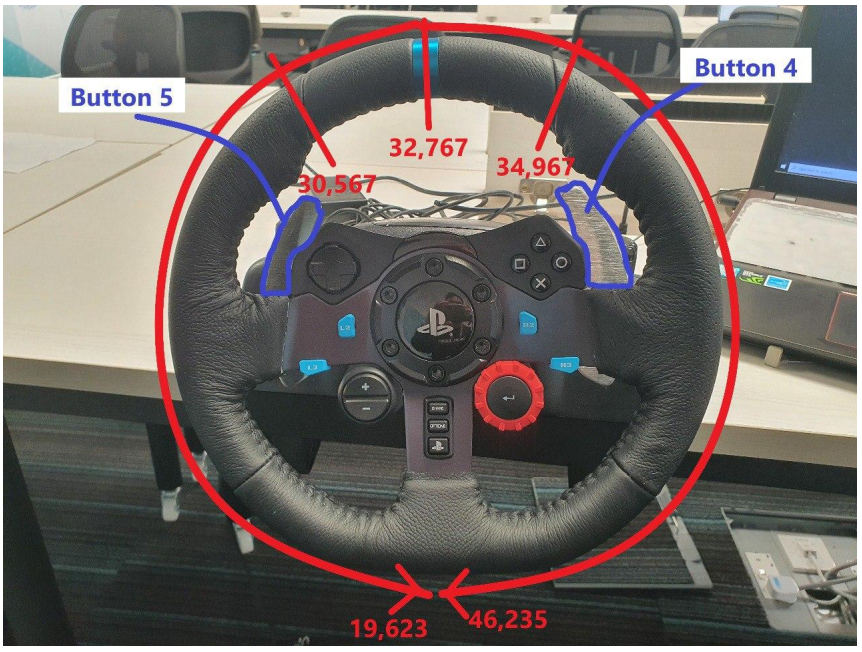

Figure 26: Range of steering wheel output values with Paddles

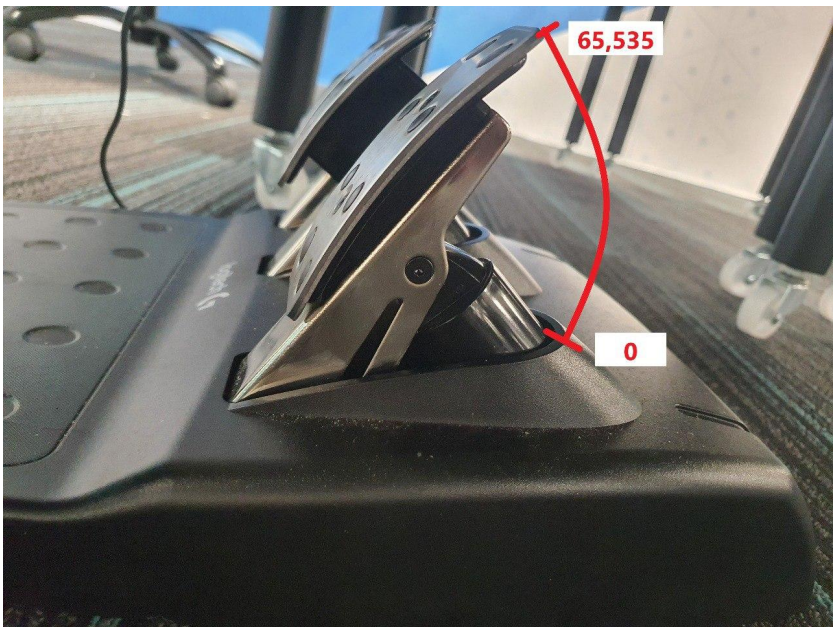

Figure 27: Range of Output from Pedals

6.1.2 Camera Integration

The camera was successfully integrated by initialising a WiFi connection with the attached ESP32 module on the camera chip itself. The camera integration allows for real-time situational awareness of the robot's surroundings.

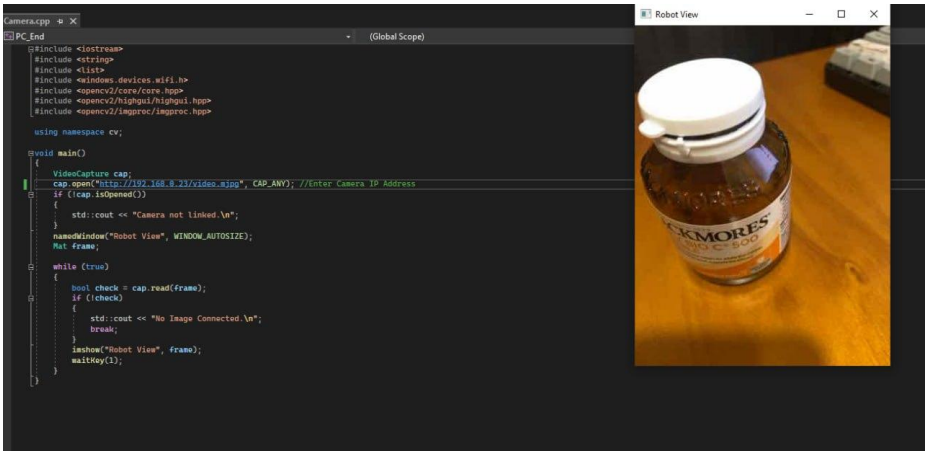

Figure 28: Successful Camera Initialisation from PC

6.1.3 Unity Virtual Environment

The virtual environment allows the operator to plan his manoeuvres ahead of time, making the movement of the robot efficient as large known obstacles can already be mapped out to scale in our virtual environment. The camera is meant to supplement the virtual environment because the real environment may deviate from what has been mapped out, especially considering the use case in a real-life disaster scenario.

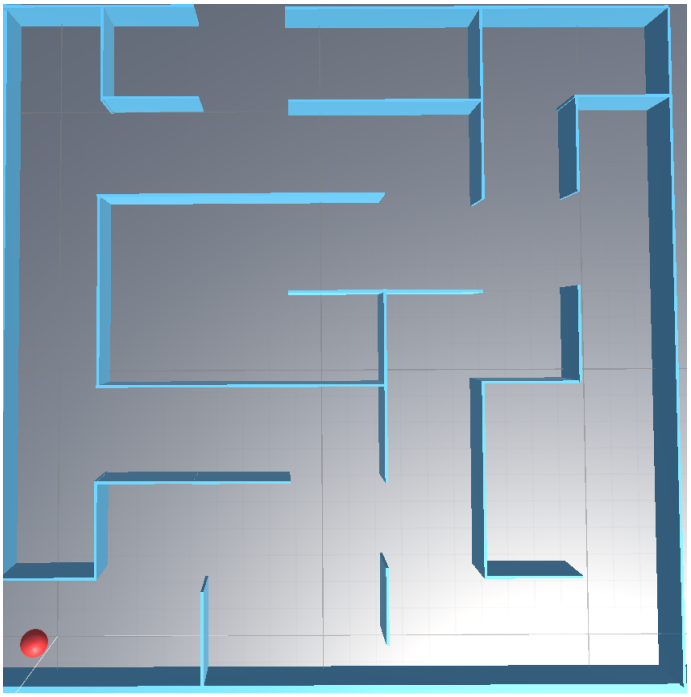

Figure 29: Unity Maze with Robot Position represented by red marker

6.2 Project Benefits

As a whole, our robot has met all the goals that we set out to achieve. Firstly, we successfully integrated a steering wheel and pedal set-up, allowing for a familiar feel for the controls of a robot. For future training, the familiar setup and feel would make training new operators quick and efficient.

We also successfully integrated the camera module into our user interface, providing the operator with real-time feedback on their surroundings. Lastly, combining these with our 3D virtual environment, we created a comprehensive prototype to demonstrate the potential that achieved our goal of creating a prototype of a robot that can access spaces inaccessible to humans.

7. Project Management Review

7.1 Project Initiation

The initial project objectives are specific, as we had a list of detailed tasks that each sub-group was delegated to complete in sequential order. These goals are measurable, as they are tangible goals that have a way of determining if we have achieved them. They are achievable as overarching goals are broken down into smaller milestones that can be achieved within each sub-group. With each sub-group working simultaneously, we can then combine all these achievements to work towards our overall goal of the project. The objectives are also relevant, as they help achieve the overall goal of creating a mobile robot that can be remotely controlled. Lastly, our objectives are also time-bound, as we set an overall deadline for the completion of our project, and within that deadline were multiple smaller deadlines to achieve various milestones.

7.2 Project Planning

The project was properly scheduled, as we followed our milestone targets strictly. Though we faced some problems in different areas, we had help from our supervisors and overcame them in a timely fashion. Roles and responsibilities were also clearly defined from the beginning, as we were immediately split into various subgroups to work on different aspects of our project.

7.3 Project Manager Role

7.3.1 Communication Strategies:

1. We established multiple telegram chat groups: a main group with our mentors, a subgroup without, and another subgroup for the people working on hardware since 5 out of 7 members of the group were primarily focused on hardware initially. This allows for more effective communication, and this system allows for the relevant messages to always reach the relevant people. To elaborate, the main chat is the group we use to consult and update our mentors, so if we need help from our mentors, we would primarily send messages to that group. The group without mentors exists for us to sort out work among ourselves, while the hardware chat group helps to allocate work between people working on hardware-specific tasks. This prevents any of us from being flooded with messages that are irrelevant to them.
2. Having weekly online meetings with people working on the same subtasks. This prevents anyone from doing redundant work that overlaps with another. Furthermore, this allows us to do testing effectively by putting together our work while everyone working on that subtask is present, such that any queries can be answered then and we would not have to spend time to first make sense of someone else's work.

7.3.2 Motivation Strategies/Monitoring & Control Activities for Progress:

Instead of any particular motivation strategies, we mainly use a system of reminders. People who are working on the same subtask will keep each other accountable by constantly updating and reminding each other of progress. This constant supervision within the group will encourage responsibility in each of us, as it reminds us we are part of a group and not keeping up with the workload affects more than just one individual.

Other than that, all of us primarily strive to follow our planned schedule and milestones to stay on track and ask for help within the group or from our mentors when we are faced with any challenges.

7.3.3 Cost Management

To manage our costs effectively and avoid overspending, we mainly came up with 3 measures:

1. We prepared a comprehensive budget plan at the very start of our project. We consulted our mentors on all the necessary parts we would need for the project and accounted for spares in case of breakdowns and the additional costs from shipping.
2. We decided to always have a group consultation before every purchase; we would only buy parts if everyone deemed it necessary. This is to ensure we do not waste any money on unnecessary parts.
3. We would always check for or find free alternatives before spending. Examples:
    a. Using free cardboard boxes obtained from Lee Wee Nam School Library for the physical maze.
    b. We are using Unity Personal instead of Unity Pro after finding out we do not need the paid version for our product.

7.3.4 Risk Management

To ensure our project can be completed smoothly, we established some measures to manage risk at the very beginning.

1. To mitigate against hardware component breakdown and failure during testing, we ensure that we always purchase hardware spares if possible. For example, buying two ELEGOO Smart Car Kits instead of one, and buying 2 more additional ESP32 boards on top of the ones in the kits. This is because the ESP32 is the most integral piece of hardware for integrating the smart car into everything else.
2. To mitigate against potential technical issues in the form of connectivity issues and software integration between the LPS and Unity, we would always conduct testing as early as possible or test at every step we could. This allows any errors to be identified and fixed early on. Examples: While attempting keyboard control of the robot, we first tested the connectivity between the laptop and robot by sending simple messages instead of jumping straight into programming the commands to control movement without first testing connectivity. In communicating the coordinates to Unity, we ensure that the script for decoding coordinates works outside of Unity, before putting in the script with everything else.
3. Since our project requires testing to be done in an open area and there is a camera connected to our robot, we have to mitigate the risks of invading the privacy of others.

To do so, we were careful in selecting remote areas devoid of human traffic to do testing.

8. Reflection

8.1 Engineering Knowledge

Arduino + Steering Wheel:

Proficiency in the C++ programming language was required to write the codes for the car's motor control, sensor reading, and camera control. C++ is also used to write codes to integrate the steering wheel by reading the input data and translating them into commands that the robot can understand and act accordingly. We also needed to integrate the translated commands into the robot's motor control system, which involves controlling the speed and direction based on the steering wheel input. We then calibrated the system to ensure that the steering wheel's range of motion aligns with the robot's movement capabilities. For example, if the input is too low, there will not be sufficient power to drive the robot.

Unity:

Unity Editor uses C# as the primary programming language. We need to have knowledge of realistic object movements and interactions, such as the "Rigid Body Dynamics" and the "Colliders" function. We applied physics principles to simulate the behaviour of the rigid body, which in our project refers to the robot. For example, when there is physics, a floating object will fall when 'play' is activated in the Unity environment.

Local Positioning System:

The LPS implements the localization algorithms to combine data from 4 antennas and determine the location, using coordinates, of the robot within the area of the 4 antennas.

8.2 Problem Analysis

ELEGOO Smart Robot:

Initially, the ESP32 module read both the commands sent by our laptop and empty bits. This led to problems in the execution of commands, as the commands were scrambled. At first, we tried to fix this by closing the connection after every command and opening a connection whenever we needed to send a command. However, that caused latency issues. In the end, we added conditions to the receiving of commands to read only the commands sent and ignore the empty bits, which solved our problems.

ESP32 Camera:

Initially, the camera web server code provided was outdated and streamed the live video using an old format that openCV was unable to display. Hence, we had to reconfigure the camera code of the publisher and integrate it back into the ESP32 code, updating the video format and configuring the size of the image. This solved the problem of the format and improved the latency of the video stream.

Maze:

As we had to build the physical maze, the dimensions of the maze had to be accurately calculated to ensure that the robot could drive through and manoeuvre each turn without colliding with the walls. We also have to ensure that the size is not too large, as it will be hard to construct and will require a lot of resources. Hence, we made necessary adjustments after constructing the physical maze according to the first draft of the design of the maze.

Local Positioning System:

We initially used 6 antennas each spaced 5 metres apart, and after testing, we realised that the position map produced too much interference. We then tried using 4 antennas spaced 10 metres apart and realised that it produced higher accuracy in the position. However, as the LPS updates in real-time and with the presence of a slight disturbance, when sending the coordinates back to Unity, the accuracy was also affected due to latency. Hence, we would have to calculate the average value for every 10 coordinates or set a threshold to minimise the error.

Communicating Coordinates:

The last big difficulty we faced was trying to communicate coordinates from the LPS to Unity. Initially, we tried to use a text file that both the LPS application could write to and the Unity side could read from, but that failed as only one application could access the text file at a time. Both applications couldn't access the file. Then we moved on to using named pipes and shared memory. For named pipes, we had to create a named pipe on both Unity and the LPS to allow for bi-directional communication of the coordinate data. However, named pipes did not work as the named pipe on Unity used the same thread as Unity's render pipeline, and as such, we could not use it to communicate the data. For shared memory, we mainly used "System.IO.MemoryMappedFiles" to try to create a shared memory segment that both Unity and the LPS application could access. However, while using shared memory, for some reason, the Unity side freezes and crashes whenever we try to run the virtual environment. In the end, we had to use websockets and get aid from our mentor to solve it.

8.3 Individual and Team Work

Our project mainly consisted of the culmination and integration of various, distinctly separate parts, which we roughly classified as hardware and software. Individual work was important for us to get up to speed on unfamiliar topics such as Unity, C++/C# programming and Arduino so that we had the necessary knowledge and preparation needed for meaningful progress during group meetings.

Teamwork, however, was also crucial, especially during the final weeks of implementation, when effective communication between team members was necessary to successfully integrate all the separate parts of the project. The team also did not hesitate to help each other when anyone encountered problems or had busy periods anytime during the 13 weeks, ensuring that progress was smooth and any issues encountered were resolved promptly.

8.4 Future Recommendations

Some future research that could be done may include autonomous navigation. As our robot is designed to be used in disaster rescue efforts, future research can focus on improving obstacle detection and avoidance capabilities. For example, in an earthquake environment, the robot would be able to automatically sense and avoid obstacles that may fall. Autonomous navigation also allows the robot to navigate complex environments without human control, which might improve its precision. In the future, machine learning models can also be trained and incorporated into the motion control of the robot, which could train the robot to be able to plan its path and find the shortest path.

In terms of hardware, the design of the robot should be improved to drive through challenging terrain, such as uneven surfaces or even underwater environments. Robotic arms can also be installed to bring food and water to the rescuers when additional time is required to clear the obstacles.

References

[1] M. A. Han, “logitech-wheel-SDK” [Online], October 11 2020. Available: https://gitlab.com/modanhan/logitech-wheel-sdk

Appendix A - Project Members Information

	Name	Project contributions	Report Contribution
1	Apichart Yapakdee	Group Leader, Initial Arduino Connection, Communicating Coordinates, Charter, Maze Building, or Storyboard Video Creation, Group Report, Final Presentation	1. Purpose/Project Objectives 2. Project Summary 3.4.2 Decoding Coordinates for Unity 4. Schedule 7.3 Project manager role 8.2 Problem Analysis (Communicating Coordinates)
2	Ibrahim Bin Mohamed Farid	Treasurer, Steering Wheel Integration, Project Charter, Building Smart Car, Storyboard Video Creation, Building Physical Maze, Group Report, Final Presentation	3.2.4 Steering Wheel Integration
3	Tok Jun Wei	Initial Arduino Connection, Project Charter, Arduino to PC connection, Arduino Camera, PC Command, PC Camera Display, Storyboard, Video Creation, Building Physical Maze, Group Report, Final Presentation	3.2.1 ESP32 3.2.2 Laptop Command Code 3.2.3 Laptop Camera Code
4	Dylan Teo Wen Jun	Building Smart Car, Initial Arduino Functions, Arduino Camera Code, Building Physical Maze, Storyboard Video Creation, Group Report, Final Presentation,	Acknowledgements 1. Purpose/ Project Objectives 2. Project Summary 3.1 Building the Robot 4. Schedule 5. Cost 6. Outcomes/Benefits 7.1 Project initiation

			7.2 Project planning
5	Agnes Ang Jia Hui	Unity Robot Movement, Design Unity Maze, Project Charter, Building Physical Maze, Storyboard Video Creation, Group Report, Final Presentation	3.3.1 Maze 8.1 Engineering Knowledge 8.2 Problem Analysis 8.4 Future Recommendations
6	Aw Zeting	Antenna to Unity Connection, Unity Robot Movement, Managing Antenna Array, Project Charter, Storyboard Video Creation, Group Report, Final Presentation	3.4.1 Communicating Coordinates to Unity 3.4.2 Decoding Coordinates for Unity 6.1.3 Unity Virtual Environment 8.3 Individual and Team Work
7	Rifayah Zarir	Charter, Maze Building, Storyboard Video Creation, Group Report, Final Presentation	1. Purpose/Project Objectives 2. Project Summary

Appendix B – Codes Used

```
#include "WebServer.h"
#include <WiFi.h>
#include <SPI.h>

#define RXD2 33
#define TXD2 4

WebServer web_server;
WiFiServer server(100);
WiFiClient client;

bool connected = false;

boolean alreadyConnected = false; // whether or not the client was connected previously

char command_move[2];
char command_look[1];
bool movement = false;
bool camera = false;
bool speed_recv =  false;
bool printing = false;
String readBuff = "";
String sendBuff = "";

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Serial2.begin(9600, SERIAL_8N1, RXD2, TXD2);
  web_server._start();
  server.begin();
  printWifiStatus();
}

void loop() {
  // put your main code here, to run repeatedly:
  while(true)
  {
    command();
  }

}
```

Figure 30: Arduino Code Part 1

```
43  void command()
44  {
45    if (!connected)
46    {
47      client = server.available();
48      if(client)
49      {
50        Serial.println("Got a client!");
51        if(client.connected())
52        {
53          Serial.println("and it's connected!");
54          connected = true;
55        }
56        else
57        {
58          Serial.println("but it's not connected!");
59          client.stop();
60        }
61      }
62    }
63    else
64    {
65      if(client.connected())
66      {
67        while(client.available())
68        {
69          char c = client.read();
70          if(c == '{')
71          {
72            printing = true;
73            readBuff += c;
74            //client.write(c);
75          }
76          if(c == '}')
77          {
78            printing = false;
79            readBuff += c;
80            Serial.println(readBuff);
81            Serial2.print(readBuff);
82            readBuff = "";
83          }
84          if(c != '{' && c != '}' && printing == true)
85          {
86            readBuff += c;
```

Figure 31: Arduino Code Part 2

```
76          if(c == '}')
77          {
78            printing = false;
79            readBuff += c;
80            Serial.println(readBuff);
81            Serial2.print(readBuff);
82            readBuff = "";
83          }
84          if(c != '{' && c != '}' && printing == true)
85          {
86            readBuff += c;
87            //Serial.print(c);
88          }
```

Figure 32: Arduino Code Part 3

```
104       else
105       {
106         Serial.println("Client is gone.");
107         client.stop();
108         connected = false;
109       }
110     }
111     delay(100);
112   }
113
114
115   void printWifiStatus() {
116     // print the SSID of the network you're attached to:
117     Serial.print("SSID: ");
118     Serial.println(WiFi.SSID());
119
120     // print your WiFi shield's IP address:
121     IPAddress ip = WiFi.localIP();
122     Serial.print("IP Address: ");
123     Serial.println(ip);
124
125     // print the received signal strength:
126     long rssi = WiFi.RSSI();
127     Serial.print("signal strength (RSSI):");
128     Serial.print(rssi);
129     Serial.println(" dBm");
130   }
```

Figure 33: Arduino Code Part 4

```
class Robot
{
public: // Static Class Attributes
    static float max_speed;

public:  // Instance Public Attributes
    std::string name;

public:  // Instance Public Methods
    Robot(std::string iname, std::string iaddress);
    void stop();
    void go_forward(int speed);
    void go_backward(int speed);
    void turn_left(int speed);
    void turn_right(int speed);
    void cam_turn_left();
    void cam_turn_right();
    void step();
    void _read_sensor();
    void send_int(std::string instruct);
    void send_cam(std::string instruct);
    void _disconnect();


private:  // Instance Private Attributes
    std::string _address;
    std::list<command> _actuators_to_write;  // [struct command 1, struct command 2, ...]
    std::string _sensors_to_read;  // Not decided yet.


private:  // Instance Private Methods
    void _send(command_type type, float variable_1);  // Send out information that only has one variable.
    void _send(command_type type, float variable_1, float variable_2);  // Send out information that has two variables.
    void _receive();
    void _write_actuator();
    void _connect();


};
```

Figure 34: Robot Code