“I am Mr. Robot”: A Virtual Reality Interface for the Nao Robot Platform

Alex Gabriel Majka
20429324

Final Year Project – **2024**
B.Sc. Single/Double Honours in
Computer Science and Software Engineering

Department of Computer Science
Maynooth University
Maynooth, Co. Kildare
Ireland

A thesis submitted in partial fulfilment of the requirements for the B.Sc. Single/Double Honours in Computer Science and Software Engineering.

Supervisor: **Ralf Bierig**

Contents

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of **BSc Computer Science & Software Engineering** qualification, is *entirely* my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

I hereby acknowledge and accept that this thesis may be distributed to future final year students, as an example of the standard expected of final year projects.

Signed: Alex Gabriel Majka Date: 20.03.2024

Acknowledgements

I would like to thank Ralf Bierig, Rudi Villing and the RoboÉireann team for their continuous support and help during the creation of this project. Without their guidance and technical knowledge this project wouldn’t have been possible.

Abstract

This project investigates the creation of a software interface to facilitate Human-Robot interaction, specifically enabling human control over a Nao Robot's actions in a football playing scenario through a Virtual Reality (VR) application. Utilizing the existing B-Human codebase employed by the RoboÉireann team, the initiative required navigating a complex codebase, establishing effective communication channels, and sharing camera feeds for real-time visual feedback. Challenges encountered included integrating with the Debugging architecture and optimizing the SimRobot environment, necessitating a shift in hardware from a modern Asus Tuf A15 to an older Lenovo Yogabook for compatibility. The project underscores the importance of adaptability and technical insight in developing interactive robotics solutions. While the solution remained theoretical due to incomplete development and testing phases, it proposes a comprehensive testing framework and highlights future work opportunities. These include empirical validation against upcoming RoboCup challenges and further exploration of the Debugging architecture with enhanced robotics and programming proficiency. The project serves as a foundational endeavour in Human-Robot interaction, offering insights and a detailed approach for subsequent innovation in the field.
Keywords: Human-Robot Interaction, Virtual Reality, Nao Robot, RoboCup.

List of Figures

Chapter one: Introduction

Summary

Chapter 1 provides a comprehensive overview of the project's subject matter and delineates the underlying motivations for its inception. Furthermore, this chapter conducts a thorough investigation into the challenges inherent to the project and outlines the methodology adopted to devise a viable solution. Additionally, it encompasses a detailed examination of the solution's efficacy and enumerates the accomplishments realized throughout the project's lifecycle.

1.1 Topic addressed in this project

This project delves into the dynamics of human-robot interaction, specifically focusing on utilizing a sophisticated interface and software framework to facilitate human control over a robotic entity designed for playing football. The robot model employed for this purpose is the Nao Robot, a fully autonomous, programmable humanoid robot renowned for its participation in the RoboCup Standard Platform League (SPL). The SPL is recognized as a prestigious annual international competition that showcases the capabilities of robot soccer teams.

1.2 Motivation

The initial phase of this project was focused on exploring the potential uses of a robot and the software already in place for the RoboÉireann RoboCup team. As the project was getting off the ground, the RoboCup Administration announced a challenge that was directly relevant to our project's goals, offering an opportunity for all teams to participate.

The aim here is to investigate the possibility of creating software that enables humans to interact with and control a robot during a football game. This involves examining how Human-Robot interactions can be enhanced using Virtual Reality, while also considering the limitations and challenges of developing such software with the technology that is currently available.

1.3 Problem statement

The core technical challenge of this project involves developing software capable of integrating the existing B-Human based software, currently utilized on the Nao Robots by the RoboÉireann team, with a Virtual Reality application. This integration aims to leverage the robot's camera feed and enable the sending of action requests to the robot. Key challenges include navigating and understanding a codebase exceeding 50,000 lines and establishing a reliable communication link between this extensive codebase and a Virtual Reality application. Subsequent considerations will focus on the development of the user interface, addressing potential latency issues, and enhancing the overall user experience of the application.

1.4 Approach

The analysis of the problem was approached by deconstructing it into its fundamental components necessary for crafting a solution. These components included establishing a communication link between the existing Nao robots and an external application, as well as developing a front-end application capable of displaying camera feeds and facilitating the timely dispatch of commands.

To construct these components, it was imperative to grasp the communication protocols within the RoboÉireann codebase and understand how camera feeds are processed within the software. Concurrently, acquiring skills in front-end application development was necessary to ensure the application could receive and display camera feeds and transmit commands to the robot effectively.

The initial step in addressing this challenge involved participating in weekly meetings with the RoboÉireann team. This engagement provided a deeper insight into the software and methodologies employed by the team in enhancing the Nao Robot's capabilities. Through these interactions, a foundational understanding of the codebase was established, laying the groundwork for devising a solution strategy.

Early discussions with the team revealed the potential for controlling the robot by delving into its debugging architecture. However, after extensive evaluation, this approach was deemed unfeasible due to the inability to establish a direct connection between the desired and the Virtual Reality applications. Despite thorough investigation and documentation of this architecture, it became clear that an alternative strategy was needed.

The decision to explore the use of TCP requests and the CORO Behaviour elements from the B-Human codebase emerged as a more promising approach. This idea was initially considered following a tutorial during one of the RoboÉireann weekly meetings but was only pursued after exhausting the possibilities with the debugging architecture.

The new strategy entailed developing a TCP sender to transmit the dual camera feeds via TCP on separate ports and a TCP listener to process incoming commands. The implementation was integrated within the CameraProvider and CoroBehavior components of the existing software, facilitating direct command execution on the robot. Additionally, a new CoroBehavior was crafted to manage all commands and execute motion directives on the Nao Robot, utilizing a switch statement for command interpretation.

To assess the effectiveness of this solution, an initial evaluation was planned within the SimRobot environment, allowing for the observation of latency in camera feed transmission and command execution. This phase aimed to identify areas for performance enhancement and determine the feasibility of the proposed approach. A subsequent evaluation involved testing the solution in a real-world RoboCup scenario with the robot operating alongside teammates controlled by the existing RoboÉireann AI. This stage was intended to ascertain the practicality of incorporating the solution into a competitive RoboCup team setting.

1.5 Metrics

The evaluation of this project is to be conducted through a multi-faceted approach. Firstly, the effectiveness of the application in facilitating the playing of football will be assessed by observing its performance when operated by a human on a field, competing against Nao Robots controlled by conventional AI. This real-world testing aims to highlight the practical utility of the application in enhancing the interactive aspect of the game.

Secondly, the application's behaviour and performance will be scrutinized within the SimRobot environment. This controlled setting allows for a detailed analysis of the application under simulated conditions, offering insights into its responsiveness, accuracy, and reliability in replicating intended movements and actions.

Additionally, a significant component of the evaluation will involve gathering and analysing user feedback regarding the application's interface and overall user experience. This feedback will be instrumental in identifying areas for improvement, particularly in terms of ease of use, intuitiveness of the interface, and the application's efficacy in enabling users to effectively control the robots. Through this comprehensive evaluation strategy, the project aims to refine the application, ensuring it meets the needs and expectations of its users while providing a robust solution for Human-Robot interaction in the context of robot football.

1.6 Project

Achieving a thorough understanding of the RoboÉireann codebase marked a pivotal initial accomplishment within this project. This understanding not only paved the way for identifying potential solutions to the primary challenge but also equipped me with the requisite knowledge to delve deeper into the codebase, thereby facilitating the exploration of various modifications aimed at devising an effective solution.

A notable milestone was reached with the decision to conclude the investigation into the debugging architecture. This decision effectively closed the chapter on the initial method of solution development, recognizing it as unfeasible and acknowledging the need for alternative strategies to address the project's objectives.

A significant breakthrough was achieved in successfully operationalizing the SimRobot environment. Despite initial challenges attributed to compatibility issues with the hardware of a modern machine, a strategic shift to an older computer circumvented these obstacles, enabling the unimpeded running of the application. This accomplishment not only overcame a technical hurdle but also underscored the importance of adaptability in problem-solving. The resolution of these hardware-software compatibility issues highlighted the intricacies of working with cutting-edge technology alongside legacy systems, offering valuable insights into managing technological discrepancies in project development.

Chapter two: Technical Background

Summary

This chapter provides a review of the literature and resources pertinent to the project, including related topics.

2.1 Topic Material

B-Human. (2021). Code Release Documentation. Retrieved from https://docs.b-human.de/coderelease2021/
Was used very early on in the project cycle to investigate the architecture that the RoboÉireann code is based on.

RoboÉireann. (n.d.). Rematch Wiki. GitHub repository. For access, contact Rudi Villing or Ralf Bierig. Retrieved from https://github.com/roboeireann/rematch/wiki
The RoboÉireann Wiki was used for the majority of introduction work and setup for the Nao Codebase. As a disclaimer, this Wiki is within a private repository on GitHub which may require permissions to access. For access contact Rudi Villing or Ralf Bierig.

Dyno Robotics. (2020). Teleoperated Nao-robot through VR-interface. YouTube. https://youtu.be/PUn5A76dlJs?si=eTvEPce5_kcxkFKX
RoboticsAtHsUlm. (2012). Teleoperation of the Robot NAO with a Kinect Camera. YouTube. https://youtu.be/_VbT8a0OJsE?si=M_uJP-WfVyQeBX9L
During the initial research on existing solutions and possibly interesting resources to do with this project these two videos were found, which showed that the scope of the project should be achievable but that the approach would have to differ significantly due to the fact that it would have to work with an existing codebase for the Nao robot that will be responsible for the movement rather than some other proprietary codebase like the people in the videos are using.

Fleetwood Game Design Lab. (2017). AR in Godot with webstream camera. YouTube. https://youtu.be/SiH7Ni9Yu3g?si=NbH0HyBh_4GydcBF
Another interesting video that was found during the research was the above one, which explores the implementation of a webcam feed into Godot, the gaming engine that can handle VR that was tasked to be used for the front end of the application. While this video is outdated, it showed me a good outline of what could be done and used to get the camera feeds from the Nao into the application.

2.2 Technical Material

RoboÉireann. (n.d.). Writing Your First Coro Behaviour. GitHub Wiki. Retrieved from https://github.com/roboeireann/rematch/wiki/Writing-your-first-coro-behaviour

This tutorial was used for the development of the Coro Behaviour responsible for handling requests from the front-end application.

Thread by user Joshua Bakker. (2017). Simple TCP Listener Class. Code Review Stack Exchange. Retrieved from https://codereview.stackexchange.com/questions/147996/simple-tcp-listener-class

Thread by user SPlatten (2020). Two-Way Communication Using Sockets. Stack Overflow. Retrieved from https://stackoverflow.com/questions/64569529/two-way-communication-using-sockets

Bozkurthan. (2019). Simple TCP Server Client C++ Example. GitHub. Retrieved from https://github.com/bozkurthan/Simple-TCP-Server-Client-CPP-Example

Hot Examples (n.d.). QTcpSocket Examples in C++. CPP Hot Examples. Retrieved from https://cpp.hotexamples.com/examples/-/QTcpSocket/-/cpp-qtcpsocket-class-examples.html

These examples were used for the creation of TCP Listener and Emitter within the Nao Codebase. These examples were helpful with understanding how TCP works within C++ as well as showing how to structure the classes to implement them.

Godot Engine Documentation. (n.d.). Networking. Retrieved from https://docs.godotengine.org/en/stable/tutorials/networking/index.html

Godot Engine Documentation. (n.d.). InputEvent. Retrieved from https://docs.godotengine.org/en/stable/tutorials/inputs/inputevent.html

These documentations were used to learn about event handling, input methods as well as networking within Godot.

Chapter three: The Problem

Summary

This chapter clearly explains the technical problem and identifies the requirements and specifications for the solution of the problem.

3.1 Project UML documentation

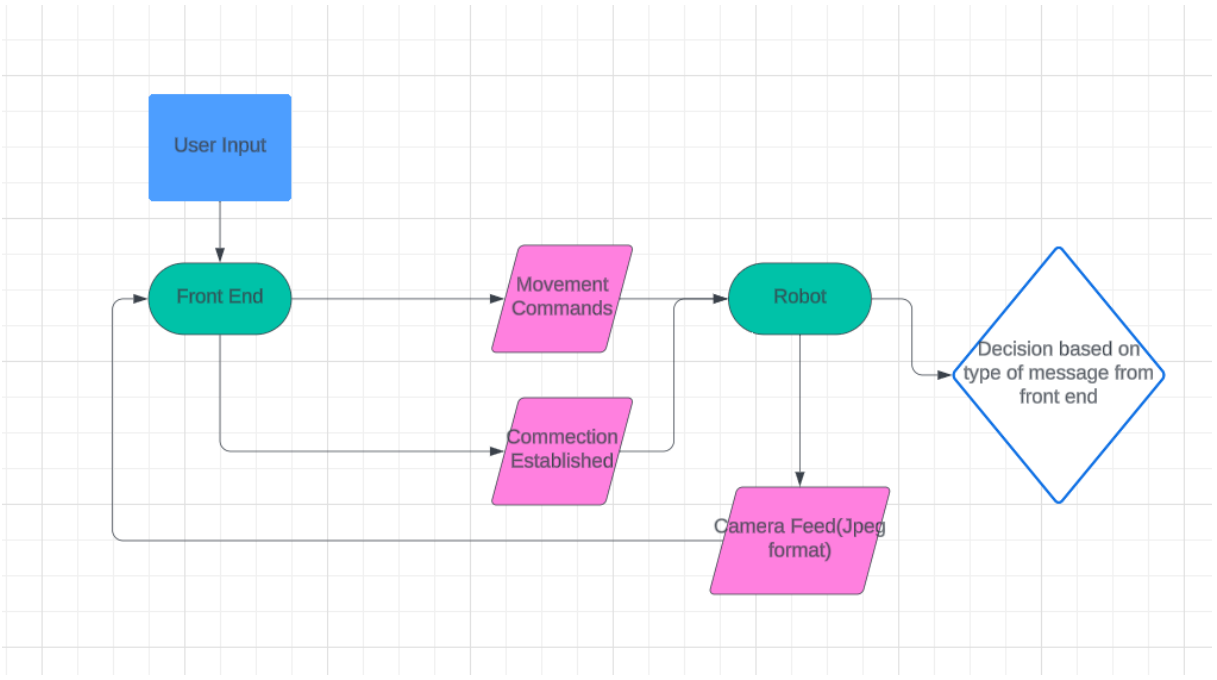


Figure 3-1 Flow chart showing the basic process of the planned System

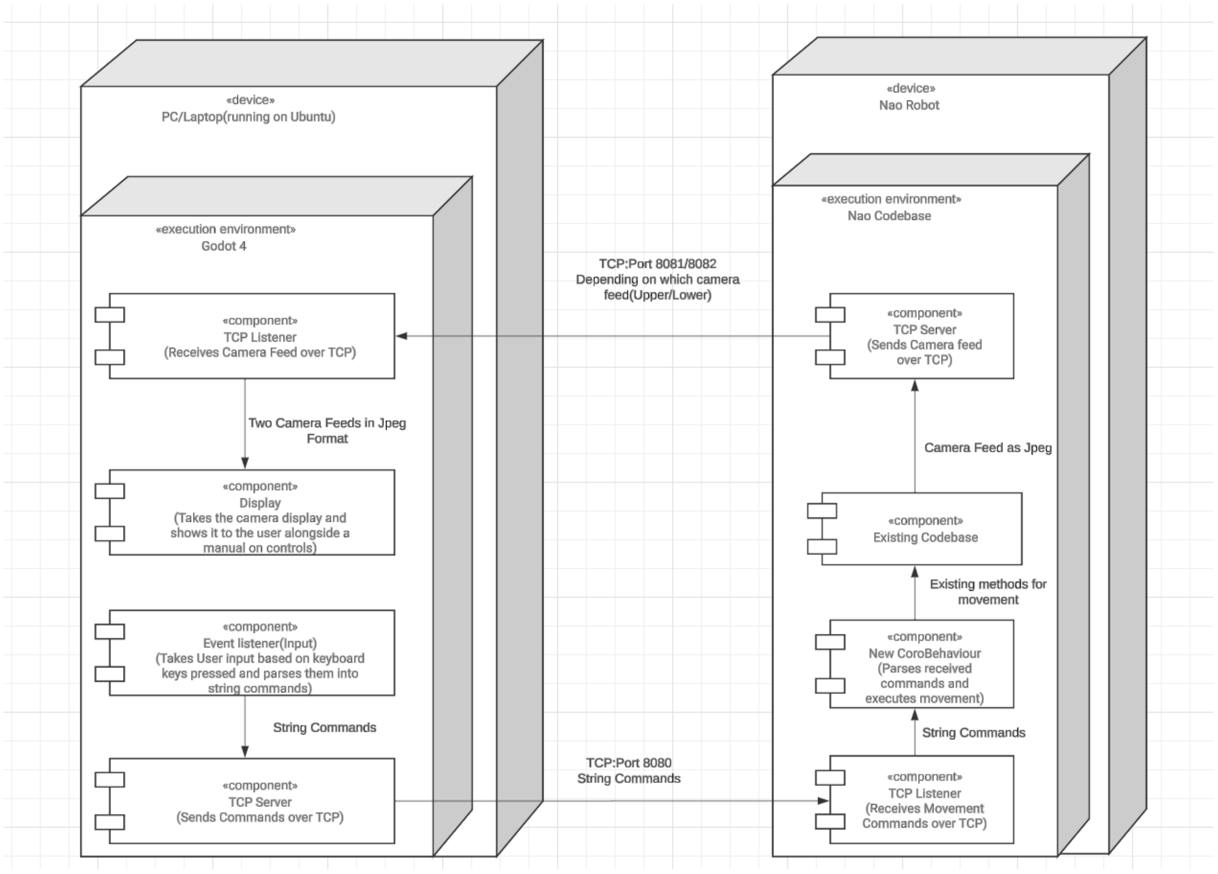

«device»
PC/Laptop(running on Ubuntu)
«execution environment»
Godot 4
«component»
TCP Listener
(Receives Camera Feed over TCP)
Two Camera Feeds in Jpeg Format
«component»
Display
(Takes the camera display and shows it to the user alongside a manual on controls)
«component»
Event listener(Input)
(Takes User input based on keyboard keys pressed and parses them into string commands)
String Commands
«component»
TCP Server
(Sends Commands over TCP)
TCP:Port 8081/8082
Depending on which camera feed(Upper/Lower)
TCP:Port 8080
String Commands
«device»
Nao Robot
«execution environment»
Nao Codebase
«component»
TCP Server
(Sends Camera feed over TCP)
Camera Feed as Jpeg
«component»
Existing Codebase
Existing methods for movement
«component»
New CoroBehaviour
(Parses received commands and executes movement)
String Commands
«component»
TCP Listener
(Receives Movement Commands over TCP)

Figure 3-2 UML Deployment Diagram of the planned System

3.2 Problem analysis

At its core, the proposed solution involves developing an application that enables the manipulation of a robot to execute basic movements such as moving forward, backward, rotating on its Y-axis, and performing a kicking action when a ball is positioned directly in front of it. Although this concept might appear straightforward in theory, the practical implementation into an extensive existing codebase, originally designed to be autonomously controlled by Artificial Intelligence, significantly complicates the task.

This undertaking necessitates the incorporation of several key functionalities: establishing effective communication channels between the front-end application and the Nao Robot, enabling the sharing of camera feeds through these communication channels, and devising a mechanism to control the robot's movements based on commands received from the front-end application. These components are critical for the seamless operation of the system and pose distinct challenges in their integration and execution within the broader framework of the existing software infrastructure.

Chapter four: The Solution

Summary

This chapter delves into the decisions made during the project, discussing, and justifying them in relation to the project specifications. It outlines the rationale behind key strategic choices, their alignment with project goals, and the impact these decisions are expected to have on the project's outcomes. Through this exploration, the chapter aims to shed light on the decision-making process and the considerations that guided the development approach.

4.1 Analytical Work

Figure 3-1 UML Deployment Diagram of the planned System

This outlines a foundational plan for structuring the project's solution, highlighting the essential components such as movement command processing and camera feed sharing. It illustrates the envisioned data flow, aiming to minimize the data transmitted by the robot to conserve its limited battery life and avoid overburdening its computing resources. Given the robot's constrained processing capacity and susceptibility to thread blocking, this plan is designed to ensure that any new computationally intensive features are implemented judiciously to maintain system efficiency.

4.2 Architectural Level

Figure 3-2 UML Deployment Diagram of the planned System.

This detailed presentation delineates the flow of information and the structure of data exchanged among the various components of the system. The diagram specifies the communication ports used for transmitting distinct types of information, emphasizing a streamlined information flow to minimize the computational load on the Nao Robot and ensure seamless integration with the pre-existing codebase. Specifically, it illustrates that the existing camera feed, represented as JPEG images within the current system, will be dispatched to the Godot application through a newly established TCP server. Furthermore, it is explicitly shown that the commands received from the Godot application and its users, formatted as strings, will be interpreted, and applied to trigger predefined movement functions in the Nao Robot, leveraging the capabilities of the existing codebase. This approach highlights a strategic effort to maintain efficiency and responsiveness in robot operations by minimizing data transmission and processing requirements.

4.3 High Level

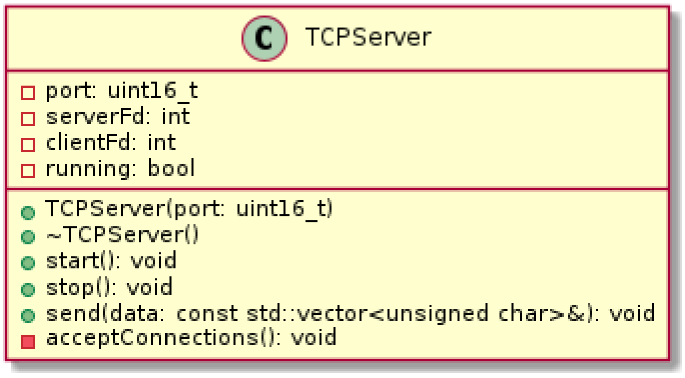

TCPServer
port: uint16_t
serverFd: int
clientFd: int
running: bool
TCPServer(port: uint16_t)
~TCPServer()
start(): void
stop(): void
send(data: const std::vector<unsigned char>&): void
acceptConnections(): void

Figure 4-3-1 UML Class Diagram for the TCPServer Class

This shows the structure of the TCP Server class, including its constructor, destructor, methods (start, stop, send, and acceptConnections), and private attributes (port, serverFd, clientFd, and running). It encapsulates the functionality for setting up a TCP server, accepting client connections, and sending data to clients.

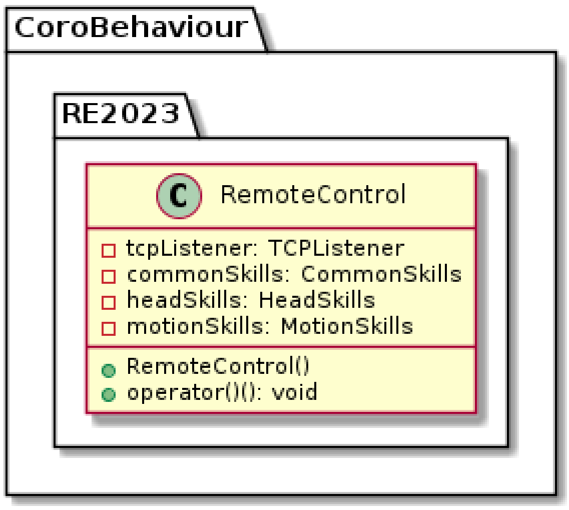


Figure 4-3-2 UML Class Diagram for the RemoteControl Class

This diagram represents the RemoteControl class within the CoroBehaviour::RE2023 namespace. It includes its private members such as the TCPListener for handling remote commands and various skills (commonSkills, headSkills, motionSkills) used to execute these commands. The diagram also highlights the constructor and the operator function, which processes incoming remote control commands.

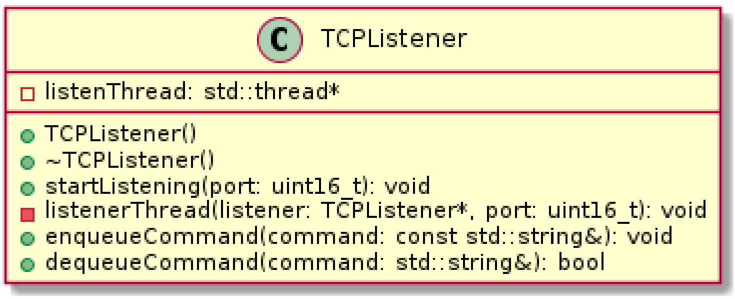

TCPListener
listenThread: std::thread*
TCPListener()
~TCPListener()
startListening(port: uint16_t): void
listenerThread(listener: TCPListener*, port: uint16_t): void
enqueueCommand(command: const std::string&): void
dequeueCommand(command: std::string&): bool

Figure 4-3-3 UML Class Diagram for the TCPListener Class

This diagram displays the structure of the TCPListener class, highlighting its constructor, destructor, methods for starting the listener and managing the command queue, as well as the private attributes and methods for handling incoming connections and commands.

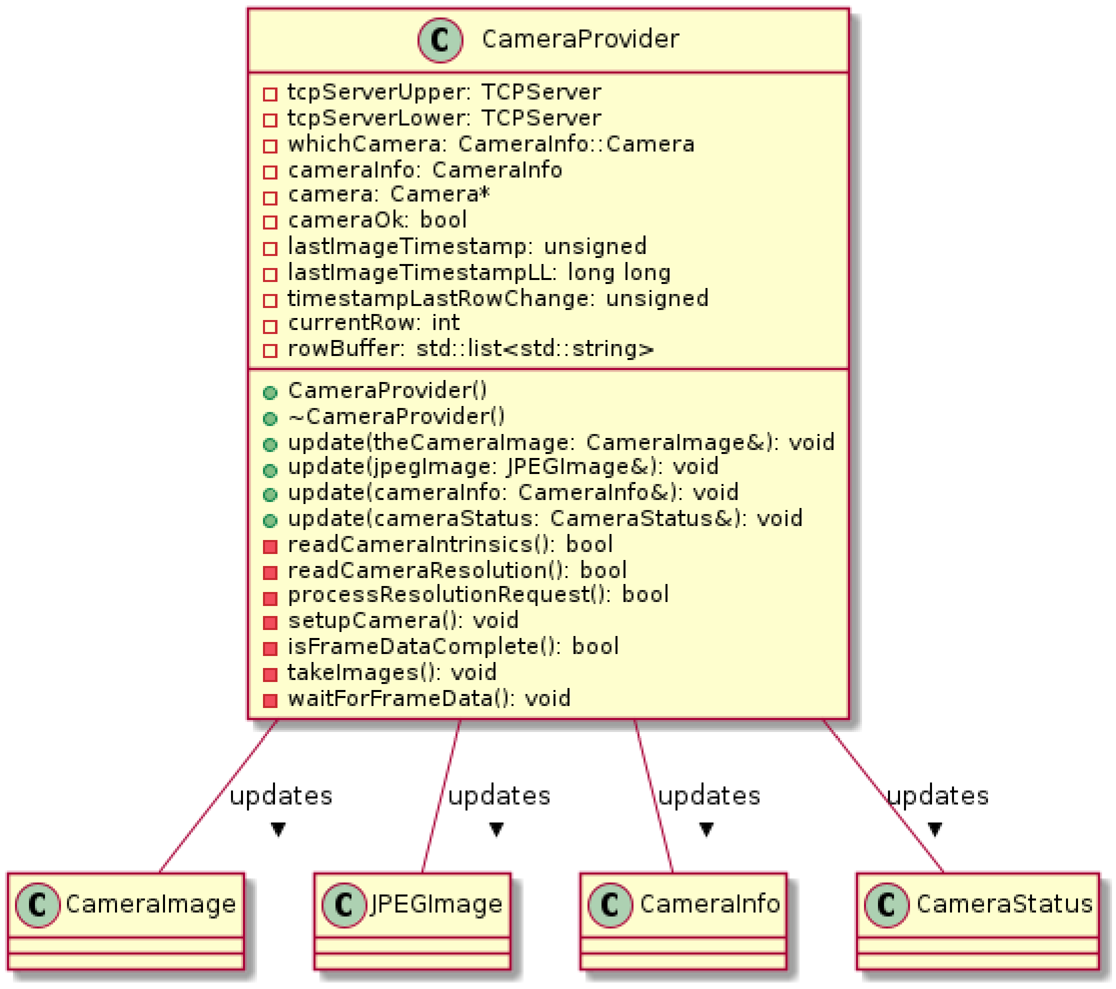

CameraProvider
tcpServerUpper: TCPServer
tcpServerLower: TCPServer
whichCamera: CameraInfo::Camera
cameraInfo: CameraInfo
camera: Camera*
cameraOk: bool
lastImageTimestamp: unsigned
lastImageTimestampLL: long long
timestampLastRowChange: unsigned
currentRow: int
rowBuffer: std::list<std::string>
CameraProvider()
~CameraProvider()
update(theCameraImage: CameraImage&): void
update(jpegImage: JPEGImage&): void
update(cameraInfo: CameraInfo&): void
update(cameraStatus: CameraStatus&): void
readCameraIntrinsics(): bool
readCameraResolution(): bool
processResolutionRequest(): bool
setupCamera(): void
isFrameDataComplete(): bool
takeImages(): void
waitForFrameData(): void
updates ▾
updates ▾
updates ▾
updates ▾
CameraImage
JPEGImage
CameraInfo
CameraStatus

Figure 4-3-4 UML Class Diagram for the altered CameraProvider Class

This diagram illustrates the CameraProvider class, highlighting its attributes, methods, and relationships with classes such as CameraImage, JPEGImage, CameraInfo, and CameraStatus. It captures the essence of the CameraProvider's functionality within the system, including its interactions with cameras and handling of image data.

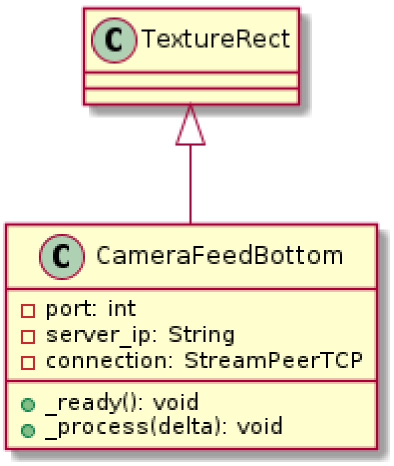


Figure 4-3-5 UML Class Diagram for the CameraFeedBottom Godot Script

This class, CameraFeedBottom, extends TextureRect and is designed to connect to a camera feed on port 8081, defaulting to localhost (127.0.0.1). It attempts to establish a TCP connection using StreamPeerTCP and, upon successful connection, processes and displays the JPEG data received from the camera feed by setting it as the texture of the TextureRect. It is also worth noting that there is a class called CameraFeedTop, this class is of the exact same structure as CameraFeedBottom with the only difference being the specified Port.

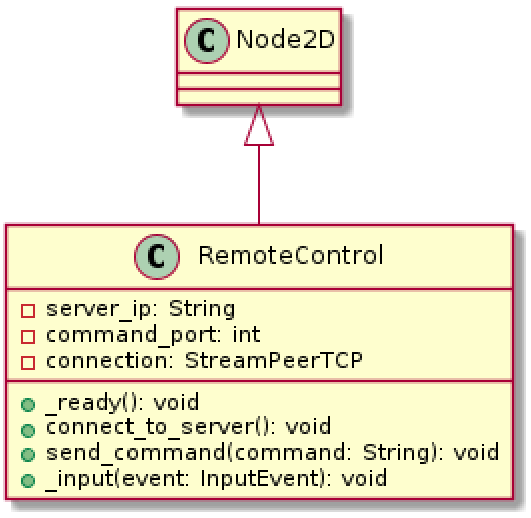

Node2D
RemoteControl
server_ip: String
command_port: int
connection: StreamPeerTCP
_ready(): void
connect_to_server(): void
send_command(command: String): void
_input(event: InputEvent): void

Figure 4-3-6 UML Class Diagram for the RemoteControl Godot Script

This class, RemoteControl, extends Node2D and includes functionalities for connecting to a TCP server, sending commands to it and handling input events to trigger these commands. The script outlines how it connects to the server upon entering the scene, sends commands based on specific key inputs (like Numpad 8 for "walk_forward"), and attempts to reconnect if the connection is lost.

Worth noting is the fact that a UML Diagram of the entire system with all its classes is missing, this is due to the fact that there is no such Diagram for the existing RoboÉireann codebase and manufacturing such a diagram would take a considerable amount of time considering the massive size of the codebase. As a result of this I have decided to only include Diagrams for my contributions to this codebase.

4.4 Implementation

The original implementation plan for this project diverged substantially from the approach that ultimately took shape. Initially, the strategy cantered on leveraging the existing Debugging architecture within the RoboÉireann codebase. This architecture was seen as a promising foundation upon which to build the project's functionalities, specifically due to its established frameworks and potential for facilitating direct control over the Nao Robot's actions.

However, as the project progressed, several challenges emerged with this approach. Key among these was the realization that the Debugging architecture, while robust for its intended purposes, posed possible limitations when it came to adapting it to the problem at hand. Issues related to implementing a connection between the Debugging thread and a front-end application became apparent, leading to a revaluation of the project's technical direction.

The shift away from the Debugging architecture was motivated by these technical hurdles and the necessity for a more flexible and easily adaptable solution. As a result, the project pivoted towards developing a new communication mechanism from scratch, focusing on establishing a reliable communication link between the front-end application and the robot, and ensuring the efficient sharing of camera feeds. This revised approach necessitated a departure from the original plan, embracing a more tailored solution that addressed the specific challenges of Human-Robot interaction within the scope of the project.

This significant alteration in the implementation strategy underscores the dynamic nature of technological development, where initial plans may evolve in response to unforeseen challenges and insights gained through the development process. The move away from the Debugging architecture represents a strategic adaptation, aiming to fulfil the project's objectives through a solution better suited to the complexities of integrating VR technologies with robotic control systems.

It's important to acknowledge that utilizing the Debugging architecture might have represented the most effective approach for addressing the project's challenges. The inherent complexity and the deep integration of this architecture within the RoboÉireann codebase, however, posed significant navigational challenges. These complexities made it daunting to adapt and extend the architecture for the project's specific requirements, particularly for facilitating nuanced Human-Robot interactions.

The strategic shift towards a simpler TCP communication architecture was undertaken as a pragmatic response to these challenges, aiming to streamline the development process. Despite this shift appearing to be a move in the right direction, the project's timeline and the ambitious nature of the task were underappreciated from the outset. This misjudgement led to the project's implementation remaining incomplete by its conclusion. Nevertheless, the work accomplished provides a comprehensive outline of a potentially effective solution. This blueprint lays the groundwork for future development, suggesting that with additional time and resources, the envisioned system could function as intended, bridging the gap between human commands and robotic actions in a seamless and responsive manner.

Chapter five: Evaluation

Summary

This chapter describes the plan to evaluate and verify the Solution as well as how it was not possible to due to the many obstacles encountered along the way.

5.1 Testing

The testing strategy for the solution involved two distinct environments: the SimRobot Environment, which is a simulated virtual environment, and a real-world RoboCup scenario. Given the complexity of the application, the limited timeframe, and the multiplicity of components, user testing was identified as the most feasible approach for both environments.

SimRobot Environment Testing:
The primary objective within the SimRobot Environment was to assess the solution's efficacy in controlling and manipulating the robot. This environment serves as an essential platform for initial testing, thanks to its accessibility and the minimal setup time required. It enables the execution of tests in a short timeframe and allows for the creation of custom scenarios ranging from empty fields to fields populated with AI-controlled robots. This flexibility facilitates comprehensive testing and evaluation of new features and functionalities without the logistical challenges of arranging a physical setup with real robots. The SimRobot Environment was thus designated as the principal venue for testing new additions to the solution, providing an efficient means to observe and refine these functionalities.

Real World RoboCup Scenario Testing:
Testing in a real-world RoboCup scenario, involving a match against AI-operated Nao robots with a team utilizing the solution, offered a unique opportunity to gauge the solution's viability and its potential direction for future RoboCup competitions. This environment was deemed optimal for evaluating the solution's performance in actual gameplay conditions, interacting with both AI-controlled opponents and teammates. It also presented the possibility of exploring enhancements to AI strategies by integrating advanced machine learning techniques into the solution-controlled Nao robot, thereby assessing its ability to surpass the performance of AI-operated counterparts. Ultimately, this real-world testing environment was chosen for its potential to provide a comprehensive evaluation of the solution in a competitive setting and to contribute insights into improving AI efficiency through human-robot collaboration.

5.2 Performance Measure

To evaluate the performance of the solution, two primary metrics were identified: the latency between the front-end application and the robot's responsiveness, and the robot's capability to surpass AI-operated robots in performance.

Latency Measurement:
The latency metric focuses on the delay from when an action is commanded in the front-end application to when the robot executes the action. This measurement is anticipated to be in milliseconds and would ideally be captured using a high-speed camera or a smartphone capable of slow-motion video recording at a consistent frame rate. By counting and analysing the frames from the initiation of an action to its execution, an accurate assessment of latency can be achieved. This quantitative analysis aims to ensure that the solution provides a responsive interface for controlling the robot, minimizing delays that could affect performance in real-time scenarios.

Performance Against AI-Operated Robots:
Evaluating the robot's performance in comparison to AI-operated robots involves several scenario-based assessments. These assessments are designed to explore various aspects of gameplay and strategic interaction:
One-on-One Scenario: This setup pits a single AI-operated robot against a robot controlled using the solution. The goal is to assess how well the human-operated robot can outmanoeuvre and outperform its AI counterpart in a direct competition.

Two-on-Two Scenario (Mixed Teams): In this configuration, each team consists of an AI-operated robot and a robot controlled via the solution. This scenario tests the solution in a cooperative context, examining how effectively a human-controlled robot can work with an AI teammate against an all-AI opponent team.

Two-on-Two Scenario (Solution Teams): This variation features one team entirely controlled by the solution (both robots operated by users) versus an all-AI opponent team. This setup allows for the evaluation of team coordination and communication when both robots are controlled by the solution, providing insights into the potential for strategic teamwork that surpasses AI capabilities.

These performance metrics and scenarios are designed to thoroughly assess the solution's effectiveness, responsiveness, and potential advantages over traditional AI-operated robots, providing a comprehensive understanding of its capabilities and areas for improvement.

5.3 Software Design Verification

If the application functions as expected, a critical step would be to assess the efficiency of the solution's architecture. This involves several key measures:

Latency between Perception and Display: Evaluating the efficiency of the architecture would first involve measuring the latency between the real-world imagery captured by the Nao robot and its subsequent display on the front-end application. This measure is crucial for understanding the real-time performance of the system, ensuring that what the user sees and reacts to on the front-end closely mirrors the actual scenario the robot encounters.

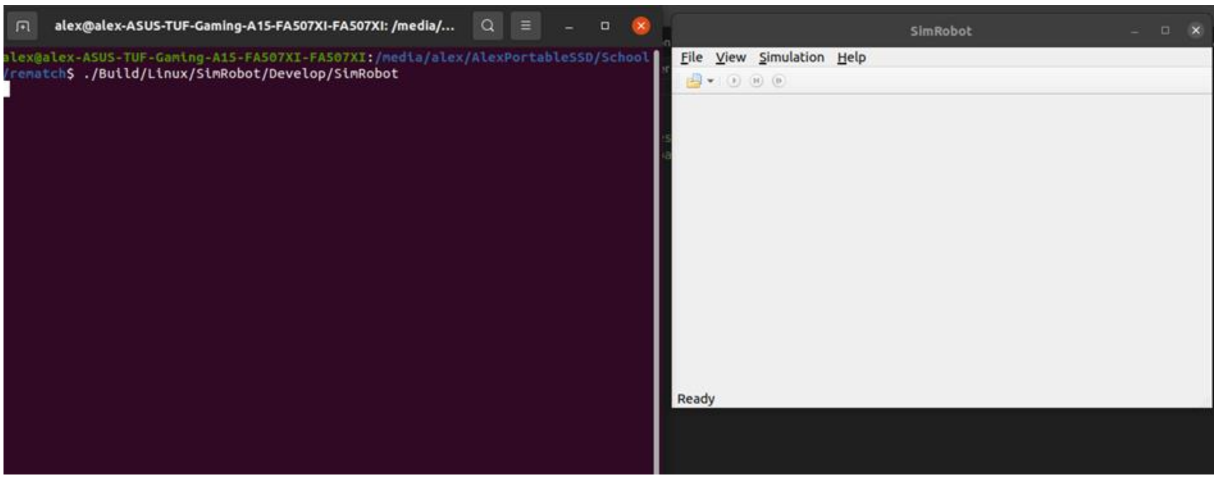
Response Time to User Commands: Another vital aspect of evaluating the architecture's efficiency is to measure the time it takes for the robot to initiate an action following a user's command input in the front-end application. This responsiveness is essential for effective control, particularly in scenarios requiring precise timing and quick reflexes, such as in competitive sports or tasks demanding high accuracy.

Comparison with RoboCup 2024 Approaches: While it may not be feasible before the submission of the Final Year Project, attending the RoboCup 2024 event could provide invaluable insights into the solution's efficiency and innovation. Discussing and comparing the solution's architecture with the approaches of different teams facing a similar challenge—controlling the Nao robot by a user instead of AI—would offer a broader perspective on the solution's effectiveness and areas for improvement. This engagement could reveal alternative strategies, optimizations, and innovations that could enhance the solution or inspire future developments.

These evaluations aim to thoroughly assess the solution's architecture in terms of real-time performance and responsiveness, as well as its comparative innovation and effectiveness within the broader context of human-controlled robotics, as exemplified in international competitions like RoboCup.

5.4 The Testing that was performed and the obstacles faced

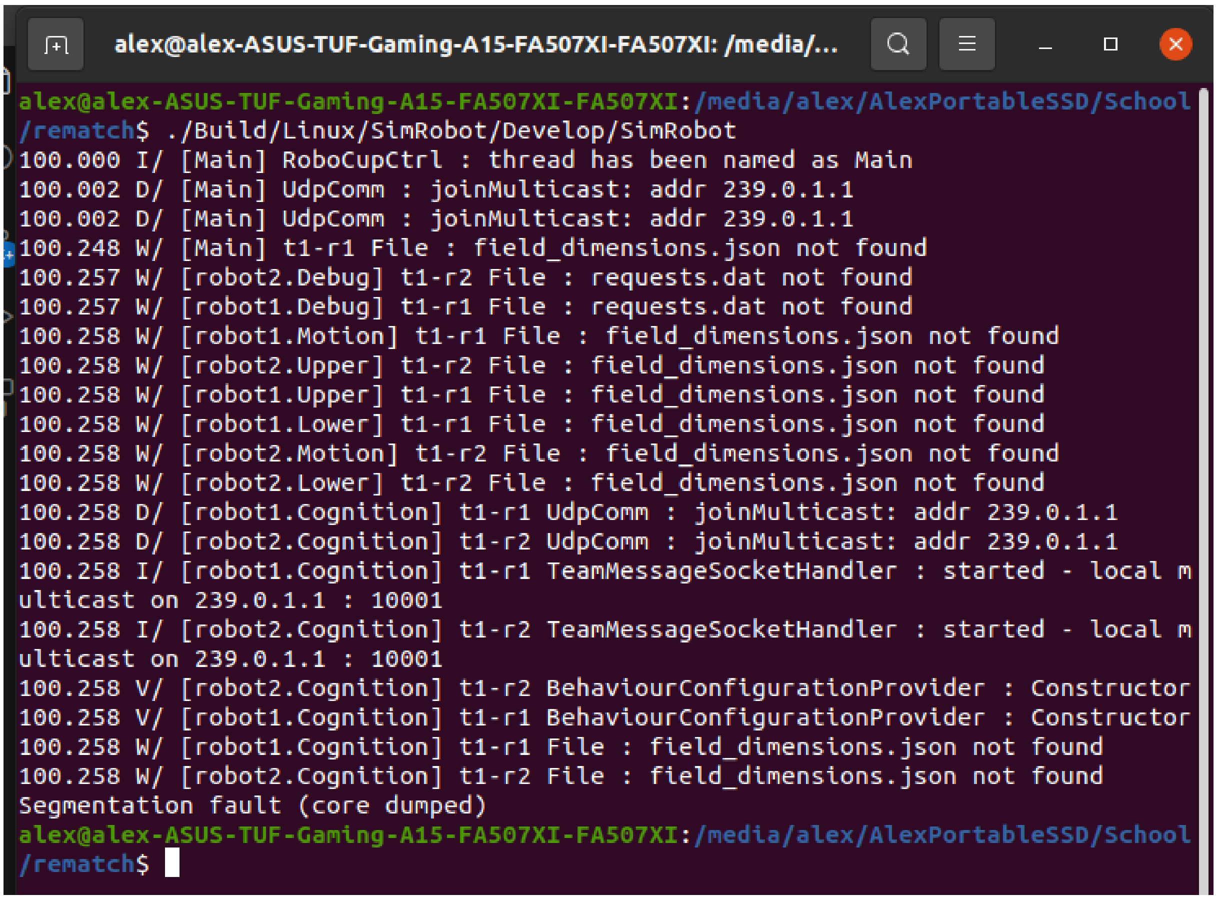
During the development of this project, the testing strategy was designed to evaluate the solution in both a simulated environment (SimRobot) and real-world RoboCup scenarios, focusing primarily on user testing due to the solution's complexity and the project's time constraints. However, the solution remained theoretical and untested within the deadline, due to incomplete development and unresolved issues. Significant obstacles were encountered, including difficulties integrating with the existing Debugging architecture and adapting the SimRobot testing environment to specific hardware requirements, which led to a switch from a modern Asus laptop to an older Lenovo model for compatibility reasons.


alex@alex-ASUS-TUF-Gaming-A15-FA507XI-FA507XI: /media/...
SimRobot
File View Simulation Help
Ready

```
alex@alex-ASUS-TUF-Gaming-A15-FA507XI-FA507XI:/media/alex/AlexPortableSSD/School
/rematch$ ./Build/Linux/SimRobot/Develop/SimRobot
```

Figure 5-4-1 This shows the SimRobot environment launched before a testing scene is started up.(Asus Laptop)

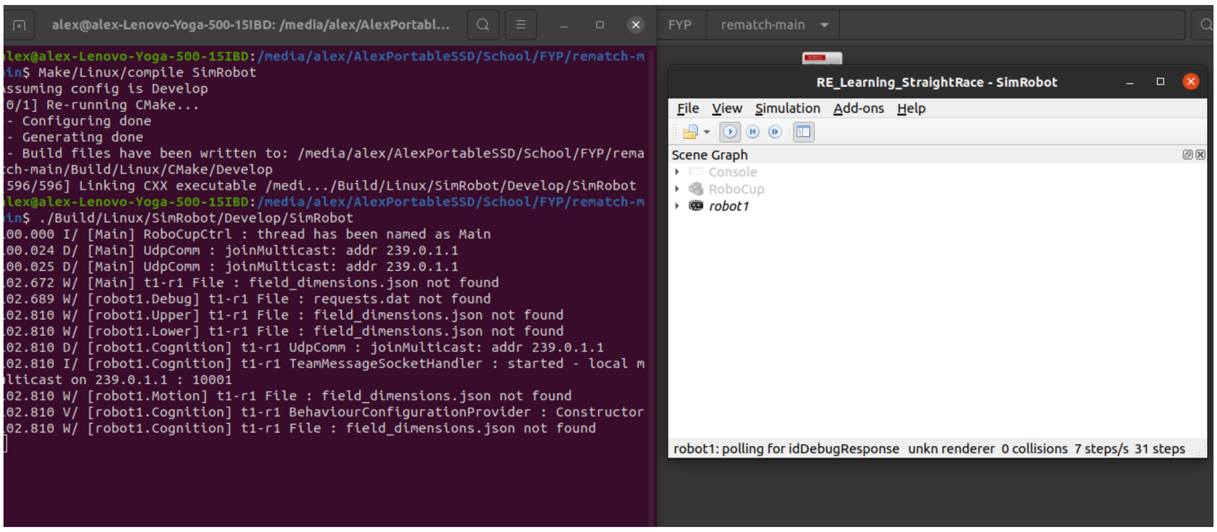
This Figure illustrates that the SimRobot was initially built and started as intended, following the tutorials from the RoboÉireann team.


alex@alex-ASUS-TUF-Gaming-A15-FA507XI-FA507XI: /media/...

```
alex@alex-ASUS-TUF-Gaming-A15-FA507XI-FA507XI:/media/alex/AlexPortableSSD/School
/rematch$ ./Build/Linux/SimRobot/Develop/SimRobot
100.000 I/ [Main] RoboCupCtrl : thread has been named as Main
100.002 D/ [Main] UdpComm : joinMulticast: addr 239.0.1.1
100.002 D/ [Main] UdpComm : joinMulticast: addr 239.0.1.1
100.248 W/ [Main] t1-r1 File : field_dimensions.json not found
100.257 W/ [robot2.Debug] t1-r2 File : requests.dat not found
100.257 W/ [robot1.Debug] t1-r1 File : requests.dat not found
100.258 W/ [robot1.Motion] t1-r1 File : field_dimensions.json not found
100.258 W/ [robot2.Upper] t1-r2 File : field_dimensions.json not found
100.258 W/ [robot1.Upper] t1-r1 File : field_dimensions.json not found
100.258 W/ [robot1.Lower] t1-r1 File : field_dimensions.json not found
100.258 W/ [robot2.Motion] t1-r2 File : field_dimensions.json not found
100.258 W/ [robot2.Lower] t1-r2 File : field_dimensions.json not found
100.258 D/ [robot1.Cognition] t1-r1 UdpComm : joinMulticast: addr 239.0.1.1
100.258 D/ [robot2.Cognition] t1-r2 UdpComm : joinMulticast: addr 239.0.1.1
100.258 I/ [robot1.Cognition] t1-r1 TeamMessageSocketHandler : started - local m
ulticast on 239.0.1.1 : 10001
100.258 I/ [robot2.Cognition] t1-r2 TeamMessageSocketHandler : started - local m
ulticast on 239.0.1.1 : 10001
100.258 V/ [robot2.Cognition] t1-r2 BehaviourConfigurationProvider : Constructor
100.258 V/ [robot1.Cognition] t1-r1 BehaviourConfigurationProvider : Constructor
100.258 W/ [robot1.Cognition] t1-r1 File : field_dimensions.json not found
100.258 W/ [robot2.Cognition] t1-r2 File : field_dimensions.json not found
Segmentation fault (core dumped)
alex@alex-ASUS-TUF-Gaming-A15-FA507XI-FA507XI:/media/alex/AlexPortableSSD/School
/rematch$ 
```

Figure 5-4-2 This shows the SimRobot environment crashed after a testing scene is started.(Asus Laptop)

This figure illustrates the instance when the SimRobot environment failed upon loading a scene, leading to a crash. Subsequent investigation attributed the crash to a device-specific issue related to the OpenGL and OpenCL libraries, which triggered a segmentation fault. Despite efforts to resolve the issue, including a clean system reinstall, testing various graphics drivers, and experimenting with different scenes, the persistent problem necessitated a transition to an alternative hardware setup to continue the project's development.



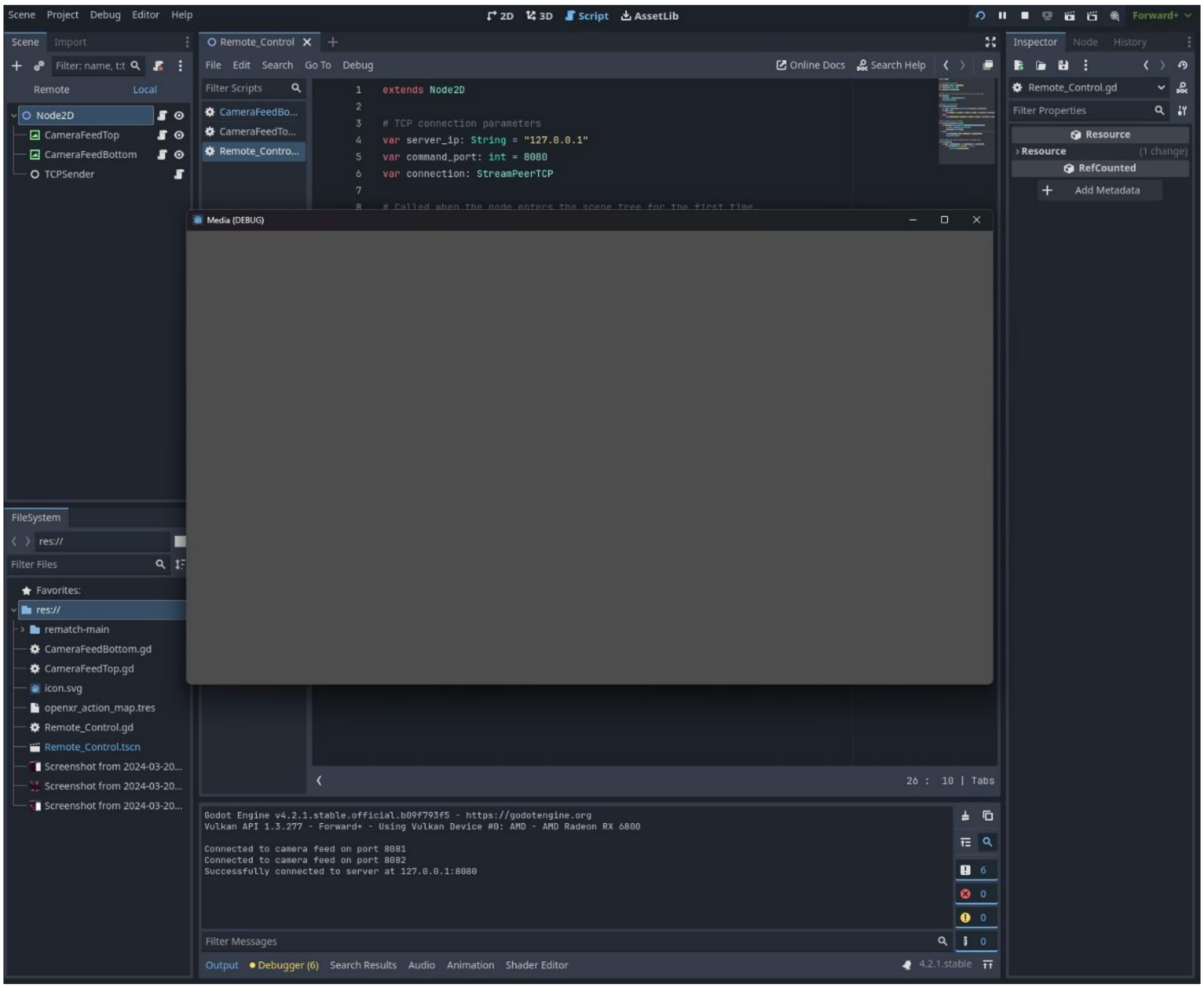
```
alex@alex-Lenovo-Yoga-500-15IBD: /media/alex/AlexPortabl...
lex@alex-Lenovo-Yoga-500-15IBD:/media/alex/AlexPortableSSD/School/FYP/rematch-m
in$ Make/Linux/compile SimRobot
ssuming config is Develop
0/1] Re-running CMake...
- Configuring done
- Generating done
- Build files have been written to: /media/alex/AlexPortableSSD/School/FYP/rema
ch-main/Build/Linux/CMake/Develop
596/596] Linking CXX executable /medi.../Build/Linux/SimRobot/Develop/SimRobot
lex@alex-Lenovo-Yoga-500-15IBD:/media/alex/AlexPortableSSD/School/FYP/rematch-m
in$ ./Build/Linux/SimRobot/Develop/SimRobot
00.000 I/ [Main] RoboCupCtrl : thread has been named as Main
00.024 D/ [Main] UdpComm : joinMulticast: addr 239.0.1.1
00.025 D/ [Main] UdpComm : joinMulticast: addr 239.0.1.1
02.672 W/ [Main] t1-r1 File : field_dimensions.json not found
02.689 W/ [robot1.Debug] t1-r1 File : requests.dat not found
02.810 W/ [robot1.Upper] t1-r1 File : field_dimensions.json not found
02.810 W/ [robot1.Lower] t1-r1 File : field_dimensions.json not found
02.810 D/ [robot1.Cognition] t1-r1 UdpComm : joinMulticast: addr 239.0.1.1
02.810 I/ [robot1.Cognition] t1-r1 TeamMessageSocketHandler : started - local m
lticast on 239.0.1.1 : 10001
02.810 W/ [robot1.Motion] t1-r1 File : field_dimensions.json not found
02.810 V/ [robot1.Cognition] t1-r1 BehaviourConfigurationProvider : Constructor
02.810 W/ [robot1.Cognition] t1-r1 File : field_dimensions.json not found
```

```
robot1: polling for idDebugResponse  unkn renderer  0 collisions  7 steps/s  31 steps
```

Figure 5-4-3 This shows the testing suite working as intended.(Lenovo Laptop)

This figure demonstrates the successful operation of the testing suite on an older Lenovo laptop, following a switch from a more recent device that encountered compatibility issues. Initial problems with outdated libraries, stemming from an older Ubuntu installation on the Lenovo, were resolved through a clean system install, leading to the testing environment performing flawlessly.

Additionally, the figure reveals an issue in the final version of the Nao Codebase, where it freezes at step thirty-one, visible in the bottom-right corner. This suggests potential thread-blocking within the code developed for the Nao Robot, despite efforts to avoid such issues. The suspected culprit is the CameraProvider component, tasked with transmitting large volumes of data as JPEG images, which is inherently computationally intensive. Given the project's time constraints and the priority placed on finalizing the report, further troubleshooting of this specific issue was deprioritized.



```
extends Node2D

# TCP connection parameters
var server_ip: String = "127.0.0.1"
var command_port: int = 8080
var connection: StreamPeerTCP

# Called when the node enters the scene tree for the first time.
```

```
Godot Engine v4.2.1.stable.official.b09f793f5 - https://godotengine.org
Vulkan API 1.3.277 - Forward+ - Using Vulkan Device #0: AMD - AMD Radeon RX 6800

Connected to camera feed on port 8081
Connected to camera feed on port 8082
Successfully connected to server at 127.0.0.1:8080
```

Figure 5-4-4 This shows the front-end compiling without errors.

This figure highlights the successful compilation of the final version of the front-end application, indicating that while the overall solution has not reached completion, the front-end code is nearing its final stages. The application compiles smoothly, exhibiting no errors or operational issues such as freezing, underscoring the progress made in developing the software's core functionality. The primary task remaining in this domain is the development of the User Interface (UI), which will enhance the application's usability and interaction capabilities, marking a critical step towards the project's completion.

Chapter six: Conclusion

Summary

This chapter examines the implications and outcomes of the project, focusing on both the immediate impacts and potential future effects. It highlights the key results, such as technological advancements, enhancements in human-robot interaction, and contributions to the robotics field. The analysis aims to outline how the findings could shape further research, development approaches, and applications in robotics and adjacent areas.

6.1 Contribution to the state-of-the-art

While the project's contributions may not redefine the cutting edge, they venture into previously uncharted territories and could significantly aid future endeavours in related fields. The comprehensive documentation of the Nao Robot's Debugging architecture stands as a valuable resource for anyone seeking to comprehend, utilize, or enhance this aspect for similar objectives. Moreover, the detailed description of the solution's architecture provided in this document offers a foundational blueprint and guidance for individuals tackling comparable challenges. Understanding the hurdles encountered during this project's development is equally essential, offering insights that could help others circumvent or navigate similar issues more effectively.

6.2 Results discussion

The outcomes associated with this solution remain theoretical, attributed to the incomplete status of the solution by the project deadline, alongside existing issues that necessitate resolution prior to conducting empirical tests. Despite this, the project introduces a comprehensive framework for testing that could prove invaluable for any entity interested in assessing analogous solutions. This framework not only provides a detailed methodology for evaluation but also sets expectations for those considering this project as a foundational guide or blueprint for similar endeavours.

6.3 Project Approach

The strategy employed for this project was initially marked by enthusiasm and confidence, facing a novel challenge without sufficient caution. Early research appeared promising, leading to a sense of rapid progress. However, difficulties with the Debugging architecture soon emerged, causing confusion and setbacks. This initial approach, while ambitious, underestimated the complexity of the task, especially when pursued without extensive experience in robotics and C++ programming.

A significant portion of the project's timeline was dedicated to an approach based on the Debugging architecture. This direction might have been fruitful under the guidance of a seasoned programmer in robotics and C++. However, as challenges mounted, it became evident that the project was more daunting than anticipated, transforming an eager endeavour into an uphill battle. The diminishing timeframe, coupled with unexpected obstacles, gradually eroded morale.

Complications with the SimRobot testing environment and the solution itself consumed considerable time and effort, further derailing the project's progress. Additionally, a lack of proficiency in C++ and robotics, alongside distractions from external projects and assignments, compounded the difficulties faced.

In retrospect, the initial optimism was overshadowed by numerous challenges, leading to a shift towards a more reactive, "saving the ship" approach. As the project lagged behind the desired progress, tasks began to accumulate, rendering the completion of key components, such as the front-end application and a fully functional solution, unfeasible. Furthermore, issues introduced by new modifications to the RoboÉireann codebase, particularly those affecting scheduling on the robot, exacerbated the situation.

Ultimately, the journey of this project underscores the importance of adaptability, realistic planning, and the acknowledgment of one's limitations in the face of ambitious technological projects.

6.4 Future Work

The future work surrounding this project harbours substantial potential for further exploration and development, given the current solution's preliminary state and the absence of empirical testing to validate its efficacy. Several avenues present themselves for advancement:

Completion and Refinement of Existing Components: The current solution, while incomplete, lays a foundation that could be significantly enhanced. Future efforts could focus on finalizing and refining these components to achieve a more polished and effective system.

Empirical Validation and Comparative Analysis: With the solution yet to be tested, future work should prioritize empirical evaluation to ascertain its viability. Moreover, the upcoming 2024 RoboCup challenge will likely introduce a plethora of solutions addressing similar objectives. Analysing these forthcoming solutions could provide valuable benchmarks for assessing the technical decisions made in this project and inspire improvements or alternative strategies.

Leverage 2024 RoboCup Insights: The disclosure of various solutions from the 2024 RoboCup challenge will offer a rich repository of knowledge and strategies. Accessing and studying these solutions will facilitate a more informed approach to tackling the project's objectives, potentially simplifying the process of devising a viable solution by emulating successful aspects of these forthcoming models.

Revisiting the Original Plan with Enhanced Expertise: A long-term opportunity involves revisiting the project's initial strategy, particularly the exploration of the Debugging Architecture within the RoboÉireann codebase, after acquiring a deeper understanding of Robotics and C++. This approach, armed with increased technical proficiency and insights from future developments in the field, may ultimately yield the most effective solution without necessitating extensive alterations to the existing codebase.

In essence, the path forward for this project is rich with opportunities for innovation, learning, and improvement. By building on the groundwork laid, engaging with future developments in the robotics community, and expanding technical knowledge, there exists a strong potential to realize a solution that not only meets but exceeds the initial aspirations of this endeavour.

References

B-Human. (2021). Code Release Documentation.
Retrieved from https://docs.b-human.de/coderelease2021/

RoboÉireann. (n.d.). Rematch Wiki. GitHub repository. For access, contact Rudi Villing or Ralf Bierig.
Retrieved from https://github.com/roboeireann/rematch/wiki

Dyno Robotics. (2020). Teleoperated Nao-robot through VR-interface. YouTube.
https://youtu.be/PUn5A76dlJs?si=eTvEPce5_kcxkFKX

RoboticsAtHsUlm. (2012). Teleoperation of the Robot NAO with a Kinect Camera. YouTube.
https://youtu.be/_VbT8a0OJsE?si=M_uJP-WfVyQeBX9L

Fleetwood Game Design Lab. (2017). AR in Godot with webstream camera. YouTube.
https://youtu.be/SiH7Ni9Yu3g?si=NbH0HyBh_4GydcBF

RoboÉireann. (n.d.). Writing Your First Coro Behaviour. GitHub Wiki. Retrieved from
https://github.com/roboeireann/rematch/wiki/Writing-your-first-coro-behaviour

Thread by user Joshua Bakker. (2017). Simple TCP Listener Class. Code Review Stack Exchange.
Retrieved from https://codereview.stackexchange.com/questions/147996/simple-tcp-listener-class

Thread by user SPlatten (2020). Two-Way Communication Using Sockets. Stack Overflow. Retrieved from https://stackoverflow.com/questions/64569529/two-way-communication-using-sockets

Bozkurthan. (2019). Simple TCP Server Client C++ Example. GitHub. Retrieved from
https://github.com/bozkurthan/Simple-TCP-Server-Client-CPP-Example

Hot Examples (n.d.). QTcpSocket Examples in C++. CPP Hot Examples. Retrieved from
https://cpp.hotexamples.com/examples/-/QTcpSocket/-/cpp-qtcpsocket-class-examples.html

Godot Engine Documentation. (n.d.). Networking. Retrieved from
https://docs.godotengine.org/en/stable/tutorials/networking/index.html

Godot Engine Documentation. (n.d.). InputEvent. Retrieved from
https://docs.godotengine.org/en/stable/tutorials/inputs/inputevent.html

Appendices

Appendix 1 Schematic of the hardware associated with this project.

For the implementation and testing of this project, the required hardware setup includes:

Laptop or PC with Ubuntu 20.04 LTS: The initial choice was an Asus Tuf A15 (2023), which later was switched to an older Lenovo Yogabook. The change was necessitated by device-specific issues that hindered the effective utilization of the SimRobot environment on the newer Asus model.

Godot 4 Software: For developing and running the front-end application, Godot 4, a versatile and open-source game engine, is required. This software supports the creation of the user interface and interaction logic for the project.

Nao Robot (Optional): For conducting real-world tests, a Nao Robot equipped with the developed codebase is recommended. While not essential for all stages of development, having access to the robot is crucial for final testing and validation of the project's functionalities in practical scenarios.

This hardware setup ensures that the project can be developed, tested, and demonstrated under conditions that closely mimic real-world usage, thereby facilitating a comprehensive evaluation of its performance and usability.

Appendix 2 Code developed for this project.

The full code for this project can be found in the following Repository:
https://github.com/AlexAmI123/FYP
as well as in the attachment to this Report.

All code from this point is either fully developed for the purpose of this project, or code that needed to be altered to serve some purpose for it.

```
/**

* @file RemoteControl.h

*

* This file implements a behaviour that aims to control the robot remotely

*/


#pragma once


#include "TCPListener.h"

#include "Modules/BehaviorControl/CoroBehaviour/CoroBehaviourCommon.h"

#include "Modules/BehaviorControl/CoroBehaviour/Skills/CommonSkills.h"

#include "Modules/BehaviorControl/CoroBehaviour/Skills/SoundSkills.h"



namespace CoroBehaviour

{

  namespace RE2023

  {

    CRBEHAVIOUR(RemoteControl)

    {

      CRBEHAVIOUR_INIT(RemoteControl) {

        // Start the listener

        tcpListener.startListening(8080);

      }
```

```
void operator()(void) {
std::string command;
CRBEHAVIOUR_BEGIN();

while (true) {
   if (tcpListener.dequeueCommand(command)) {
      // Process commands
      if (command == "walk_forward") {
         // Trigger walking forward
         motionSkills.walkAtRelativeSpeed(Pose2f(0.f, 1.f, 0.f));
      }
      // Trigger crab walking left
      // Trigger crab walking right
      // Trigger rotate clockwise
      // Trigger rotate anticlockwise
      // Trigger kick ball
      //etc
      if (command == "end_remote") {
         // End Remote control
         break;
      }
   }
   CR_YIELD(); break;
}

// CRBEHAVIOUR_END();
}

private:
// DEFINES_PARAMS(RemoteControl,
            // {,
            // (unsigned)(5000) walkForwardMs,
            // });
            TCPListener tcpListener;
            CommonSkills commonSkills  {env};
            HeadSkills headSkills {env};
            MotionSkills motionSkills {env};
        };
    } // RE2023
} // CoroBehaviour
```

```
/**
* @file TCPListener.cpp
*/
#include "TCPListener.h"
#include <netinet/in.h>
#include <cstring>
#include <unistd.h>
#include <iostream>

// Constructor for the TCPListener class. It initializes the listenThread pointer to nullptr.
TCPListener::TCPListener() : listenThread(nullptr) {}

// Destructor for the TCPListener class. Ensures the listening thread is joined properly and deallocates
memory.
TCPListener::~TCPListener() {
    // Check if listenThread is not null and joinable, then join it.
    if (listenThread && listenThread->joinable()) {
        listenThread->join();
    }
    // Free the memory allocated for the listenThread.
    delete listenThread;
}

// Method to start listening on a specified port. It spawns a new thread to handle incoming connections.
void TCPListener::startListening(uint16_t port) {
    // Allocate a new thread for listening to incoming connections and pass the listenerThread static
method as the thread function.
    listenThread = new std::thread(listenerThread, this, port);
}

// Static method to handle incoming connections. This method is intended to run on a separate thread.
void TCPListener::listenerThread(TCPListener* listener, uint16_t port) {
    int server_fd, client_socket;
    struct sockaddr_in address;
    int opt = 1; // Used to set socket options.
    int addrlen = sizeof(address); // Length of the address data structure.
    char buffer[1024] = {0}; // Buffer for reading incoming data.

    // Create a socket for IPv4 with TCP.
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Set socket options to allow reuse of local addresses.
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt,
sizeof(opt))) {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }

    // Configure the address structure for the server socket.
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY; // Listen on all interfaces.
    address.sin_port = htons(port); // Convert the port number to network byte order.

    // Bind the server socket to the specified port.
    if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections, with a maximum backlog of 3 pending connections.
    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    // Accept a connection. The client_socket represents the connection to the client.
    if ((client_socket = accept(server_fd, (struct sockaddr*)&address, (socklen_t*)&addrlen)) < 0) {
        perror("accept");
        exit(EXIT_FAILURE);
    }

    // Continuously read data from the client socket until there's no more data.
    while (read(client_socket, buffer, 1024) > 0) {
        // Enqueue the read command into the listener's command queue.
        listener->enqueueCommand(std::string(buffer));
        // Clear the buffer for the next read operation.
        memset(buffer, 0, 1024);
    }

    // Close the server socket once done.
    close(server_fd);
}

// Method to add a command to the internal command queue, using mutex for thread safety.
void TCPListener::enqueueCommand(const std::string& command) {
    std::lock_guard<std::mutex> lock(commandQueueMutex); // Ensure thread-safe access to the
command queue.
    commandQueue.push(command); // Add the command to the queue.
}

// Method to remove and return the next command from the queue, if available.
```

```
bool TCPListener::dequeueCommand(std::string& command) {
    std::lock_guard<std::mutex> lock(commandQueueMutex); // Ensure thread-safe access to the
command queue.
    if (commandQueue.empty()) {
        return false; // Return false if the queue is empty.
    }
    command = commandQueue.front(); // Get the command at the front of the queue.
    commandQueue.pop(); // Remove the command from the queue.
    return true; // Return true to indicate a command was dequeued.
}
```

```
/**
* @file TCPListener.h
*/
#pragma once
#include <queue>
#include <mutex>
#include <string>
#include <thread>

// Declaration of the TCPListener class.
class TCPListener {
public:
    // Default constructor: Initializes a new instance of the TCPListener class.
    TCPListener();

    // Destructor: Responsible for cleaning up resources, such as joining threads.
    ~TCPListener();

    // Starts listening on a specified port for incoming TCP connections.
    // port: The port number on which the server will listen for incoming connections.
    void startListening(uint16_t port);

    // Attempts to dequeue a command from the internal queue of received commands.
    // command: A reference to a string where the dequeued command will be stored if available.
    // Returns true if a command was successfully dequeued, or false if the queue is empty.
    bool dequeueCommand(std::string& command);

private:
    // A static member function that represents the thread's entry point for listening to incoming
connections.
    // This function is static because the thread callback needs a static function or a free-standing
function.
```

```
    // listener: A pointer to the TCPListener instance that spawns the thread, used to access instance members.
    // port: The port number to listen on for incoming connections.
    static void listenerThread(TCPListener* listener, uint16_t port);


    // A queue to store the commands received from the TCP connections.
    std::queue<std::string> commandQueue;


    // A mutex to protect access to the commandQueue, ensuring thread safety.
    std::mutex commandQueueMutex;


    // A pointer to a std::thread object that will be used to run the listenerThread function in a separate thread.
    std::thread* listenThread;


    // Enqueues a command received from a TCP connection into the commandQueue.
    // This function is private because it is intended to be called only from within the listenerThread function
    // or other member functions of the TCPListener class.
    // command: The command to enqueue.
    void enqueueCommand(const std::string& command);
};
```

```
/**
 * @file CameraProvider.cpp
 *
 * This file implements a module that handles the communication with the two
 * cameras. This implementation starts a separate thread to avoid blocking
 * calls to slow down processing.
 *
 * @author Colin Graf
 * @author Thomas Röfer
 *
 * Any changes made for the purpose of the 2024 Final Year Project are commented as:
 * // Project Contributions here
 */

#include "CameraProvider.h"
#include "Platform/SystemCall.h"
#include "Platform/Thread.h"
#include "Platform/Time.h"
#include "Tools/Settings.h"
#include "Tools/Streams/InStreams.h"
#include "Tools/Debugging/Stopwatch.h"
#include "Tools/Debugging/Annotation.h"
#include <cstdio>

MAKE_MODULE(CameraProvider, infrastructure);

thread_local CameraProvider* CameraProvider::theInstance = nullptr;
#ifdef CAMERA_INCLUDED
Semaphore CameraProvider::performingReset = Semaphore(1);
bool CameraProvider::resetPending = false;
#endif
```

```
CameraProvider::CameraProvider() :
    // Project Contributions here
    tcpServerUpper(8081), // Port for the upper camera images
    tcpServerLower(8082), // Port for the lower camera images
    whichCamera(Thread::getCurrentThreadName() == "Upper" ?  CameraInfo::upper :
CameraInfo::lower),
    cameraInfo(whichCamera)
{
  VERIFY(readCameraIntrinsics());
  VERIFY(readCameraResolution());

  theInstance = this;
  setupCamera();

  tcpServerUpper.start();
  tcpServerLower.start();

#ifdef CAMERA_INCLUDED
  headName = Global::getSettings().headName.c_str();
  headName.append(".wav");
  thread.start(this, &CameraProvider::takeImages);
  takeNextImage.post();
  imageTaken.wait();
#endif
}

CameraProvider::~CameraProvider()
{
#ifdef CAMERA_INCLUDED
  thread.announceStop();
  takeNextImage.post();
  thread.stop();
  tcpServerUpper.stop();
  tcpServerLower.stop();

  if(camera)
    delete camera;
#endif
  theInstance = nullptr;
}

void CameraProvider::update(CameraImage& theCameraImage)
{
#ifdef CAMERA_INCLUDED
  unsigned timestamp = static_cast<long long>(camera->getTimestamp() / 1000) > static_cast<long
long>(Time::getSystemTimeBase())
                       ? static_cast<unsigned>(camera->getTimestamp() / 1000 -
Time::getSystemTimeBase()) : 100000;
  if(camera->hasImage())
  {
    theCameraImage.setReference(cameraInfo.width / 2, cameraInfo.height, const_cast<unsigned
char*>(camera->getImage()), std::max(lastImageTimestamp + 1, timestamp));

    if(theCameraImage.timestamp - timestampLastRowChange > 3000)
    {
      timestampLastRowChange = theCameraImage.timestamp;
      currentRow = Random::uniformInt(1, cameraInfo.height-2);
      rowBuffer.clear();
    }
    std::string res = MD5().digestMemory(reinterpret_cast<unsigned
char*>(theCameraImage[currentRow]), cameraInfo.width * sizeof(CameraImage::PixelType));
    rowBuffer.push_front(res);
    int appearances = 0;
    for(auto i = rowBuffer.begin(); i != rowBuffer.end(); ++i)
      if(*i == res && ++appearances > 25)
      {
        OUTPUT_ERROR("Probably encountered a distorted image!");
        camera->resetRequired = true;
        return;
      }
  }
  else if(theCameraImage.isReference())
  {
    theCameraImage.setResolution(cameraInfo.width / 2, cameraInfo.height);
    theCameraImage.timestamp = std::max(lastImageTimestamp + 1, timestamp);
  }

  if(whichCamera == CameraInfo::upper)
  {
    DEBUG_RESPONSE_ONCE("module:CameraProvider:doWhiteBalanceUpper") camera-
>doAutoWhiteBalance();
    DEBUG_RESPONSE_ONCE("module:CameraProvider:readCameraSettingsUpper") camera-
>readCameraSettings();
  }
  else
  {
    DEBUG_RESPONSE_ONCE("module:CameraProvider:doWhiteBalanceLower") camera-
>doAutoWhiteBalance();
    DEBUG_RESPONSE_ONCE("module:CameraProvider:readCameraSettingsLower") camera-
>readCameraSettings();
  }
  lastImageTimestampLL = camera->getTimestamp();
  ASSERT(theCameraImage.timestamp >= lastImageTimestamp);
  lastImageTimestamp = theCameraImage.timestamp;
#else
  theCameraImage.setResolution(cameraInfo.width / 2, cameraInfo.height);
  theCameraImage.timestamp = Time::getCurrentSystemTime();
#endif // CAMERA_INCLUDED
}

void CameraProvider::update(JPEGImage& jpegImage)
{
  jpegImage = theCameraImage;
  // Project Contribution Here
  // Retrieve the JPEG data
  const std::vector<unsigned char>& jpegData = jpegImage.getAllocator();

  // Determine which TCP server to use based on whichCamera
  TCPServer& server = (whichCamera == CameraInfo::upper) ? tcpServerUpper : tcpServerLower;

  // Send the JPEG data over TCP
  server.send(jpegData);
}

void CameraProvider::update(CameraInfo& cameraInfo)
{
  cameraInfo = this->cameraInfo;
}

void CameraProvider::update(CameraStatus& cameraStatus)
{
  if(!cameraOk)
  {
    if(cameraStatus.ok)
    {
      ANNOTATION("CameraProvider", "Could not acquire new image.");
      SystemCall::playSound("sirene.wav");
      SystemCall::say("Camera broken");
    }
#ifdef NDEBUG
    else if(!SystemCall::soundIsPlaying())
    {
      SystemCall::playSound("sirene.wav");
      SystemCall::say("Camera broken");
    }
#endif
  }

  cameraStatus.ok = cameraOk;
}

bool CameraProvider::readCameraIntrinsics()
{
  InMapFile stream("cameraIntrinsics.cfg");
  bool exist = stream.exists();
  if(exist)
    stream >> cameraIntrinsics;
  return exist;
}

bool CameraProvider::readCameraResolution()
{
  InMapFile stream("cameraResolution.cfg");
  bool exist = stream.exists();
  if(exist)
    stream >> cameraResolutionRequest;
  return exist;
}

bool CameraProvider::processResolutionRequest()
{
  if(SystemCall::getMode() == SystemCall::Mode::physicalRobot
     && theCameraResolutionRequest.resolutions[whichCamera] != lastResolutionRequest)
  {
    lastResolutionRequest = theCameraResolutionRequest.resolutions[whichCamera];
    switch(theCameraResolutionRequest.resolutions[whichCamera])
    {
      case CameraResolutionRequest::noRequest:
        return false;
      case CameraResolutionRequest::defaultRes:
        if(!readCameraResolution())
          cameraResolutionRequest.resolutions[whichCamera] = whichCamera == CameraInfo::upper
                                                   ? CameraResolutionRequest::w640h480
                                                   : CameraResolutionRequest::w320h240;
        return true;
      case CameraResolutionRequest::w320h240:
      case CameraResolutionRequest::w640h480:
      case CameraResolutionRequest::w1280h960:
        cameraResolutionRequest.resolutions[whichCamera] =
theCameraResolutionRequest.resolutions[whichCamera];
        return true;
      default:
        FAIL("Unknown resolution.");
        return false;
    }
  }
  else
    return false;
}

void CameraProvider::setupCamera()
{
  // set resolution
  switch(cameraResolutionRequest.resolutions[whichCamera])
  {
    case CameraResolutionRequest::w320h240:
      cameraInfo.width = 320;
      cameraInfo.height = 240;
      break;
    case CameraResolutionRequest::w640h480:
      cameraInfo.width = 640;
      cameraInfo.height = 480;
      break;
    case CameraResolutionRequest::w1280h960:
      cameraInfo.width = 1280;
      cameraInfo.height = 960;
      break;
    default:
      FAIL("Unknown resolution.");
      break;
  }

  // set opening angle
  cameraInfo.openingAngleWidth = cameraIntrinsics.cameras[whichCamera].openingAngleWidth;
  cameraInfo.openingAngleHeight = cameraIntrinsics.cameras[whichCamera].openingAngleHeight;
  // set optical center
  cameraInfo.opticalCenter.x() = cameraIntrinsics.cameras[whichCamera].opticalCenter.x() *
cameraInfo.width;
  cameraInfo.opticalCenter.y() = cameraIntrinsics.cameras[whichCamera].opticalCenter.y() *
cameraInfo.height;

  // update focal length
  cameraInfo.updateFocalLength();

#ifdef CAMERA_INCLUDED
  ASSERT(camera == nullptr);
  camera = new NaoCamera(whichCamera == CameraInfo::upper ?
                         "/dev/video-top" : "/dev/video-bottom",
                         cameraInfo.camera,
                         cameraInfo.width, cameraInfo.height, whichCamera == CameraInfo::upper,
                         theCameraSettings.cameras[whichCamera],
theAutoExposureWeightTable.tables[whichCamera]);
#else
  camera = nullptr;
#endif
}

bool CameraProvider::isFrameDataComplete()
{
#ifdef CAMERA_INCLUDED
  if(resetPending)
    return false;
  if(theInstance)
    return theInstance->camera->hasImage();
  else
#endif
    return true;
```

```
}


void CameraProvider::takeImages()
{
#ifdef CAMERA_INCLUDED
  BH_TRACE_INIT(whichCamera == CameraInfo::upper ? "UpperCameraProvider" :
"LowerCameraProvider");
  Thread::nameCurrentThread("CameraProvider");
  thread.setPriority(11);
  unsigned imageReceived = Time::getRealSystemTime();
  while(thread.isRunning())
  {
    if(camera->resetRequired)
      resetPending = true;
    if(resetPending)
    {
      delete camera;
      camera = nullptr;
      performingReset.wait();
      if(resetPending)
      {
        NaoCamera::resetCamera();
        SystemCall::playSound(headName.c_str());
        SystemCall::say("Camera reset");
        resetPending = false;
      }
      setupCamera();
      performingReset.post();
      continue;
    }
```

```
takeNextImage.wait();

if(camera->hasImage())
  camera->releaseImage();

// update resolution
if(processResolutionRequest())
{
  delete camera;
  camera = nullptr;
  setupCamera();
}

while(!camera->hasImage())
{
  cameraOk = camera->captureNew(maxWaitForImage);

  if(!cameraOk)
  {
    BH_TRACE_MSG("Camera broken");
    break;
  }
}

if(camera->hasImage())
  imageReceived = Time::getRealSystemTime();
else if(Time::getRealTimeSince(imageReceived) > resetDelay)
{
  delete camera;
  camera = new NaoCamera(whichCamera == CameraInfo::upper ?
                         "/dev/video-top" : "/dev/video-bottom",
                              cameraInfo.camera,
                              cameraInfo.width, cameraInfo.height, whichCamera == CameraInfo::upper,
                              theCameraSettings.cameras[whichCamera],
theAutoExposureWeightTable.tables[whichCamera]);
      imageReceived = Time::getRealSystemTime();
    }

    if(camera->hasImage())
      camera->setSettings(theCameraSettings.cameras[whichCamera],
theAutoExposureWeightTable.tables[whichCamera]);

    imageTaken.post();

    if(camera->hasImage())
      camera->writeCameraSettings();
  }
#endif
}

void CameraProvider::waitForFrameData()
{
#ifdef CAMERA_INCLUDED
  if(theInstance)
  {
    theInstance->takeNextImage.post();
    theInstance->imageTaken.wait();
  }
#endif
}
```

```
/**
 * @file CameraProvider.h
 *
 * This file declares a module that handles the communication with the two
 * cameras. This implementation starts a separate thread to avoid blocking
 * calls to slow down processing.
 *
 * @author Colin Graf
 * @author Thomas Röfer
 *
 * Any changes made for the purpose of the 2024 Final Year Project are commented as:
 * // Project Contributions here
 */

#pragma once
// Project Contributions here
#include "TCPServer.h"
#include "Platform/Camera.h"
#include "Platform/Semaphore.h"
#include "Platform/Thread.h"
#include "Representations/Configuration/AutoExposureWeightTable.h"
#include "Representations/Configuration/CameraIntrinsics.h"
#include "Representations/Configuration/CameraResolutionRequest.h"
#include "Representations/Configuration/CameraSettings.h"
#include "Representations/Infrastructure/CameraImage.h"
#include "Representations/Infrastructure/CameraInfo.h"
#include "Representations/Infrastructure/CameraStatus.h"
#include "Representations/Infrastructure/FrameInfo.h"
#include "Representations/Infrastructure/JPEGImage.h"
#include "Tools/Module/Module.h"
```

```
#include "Tools/Math/Random.h"
#include "Tools/Md5.h"
#include "Tools/RingBuffer.h"

class NaoCamera;

MODULE(CameraProvider,
{,
  USES(AutoExposureWeightTable),
  REQUIRES(CameraResolutionRequest),
  REQUIRES(CameraSettings),
  REQUIRES(CameraImage),
  PROVIDES_WITHOUT_MODIFY(CameraImage),
  PROVIDES(FrameInfo),
  PROVIDES(CameraInfo),
  PROVIDES(CameraIntrinsics),
  PROVIDES(CameraStatus),
  PROVIDES_WITHOUT_MODIFY(JPEGImage),
  DEFINES_PARAMETERS(
  {,
    (unsigned)(1000) maxWaitForImage, /** Timeout in ms for waiting for new images. */
    (int)(2000) resetDelay, /** Timeout in ms for resetting camera without image. */
  }),
});

class CameraProvider : public CameraProviderBase
{
  static thread_local CameraProvider* theInstance; /**< Points to the only instance of this class in this
thread or is 0 if there is none. */

  TCPServer tcpServerUpper;
  TCPServer tcpServerLower;
  CameraInfo::Camera whichCamera;
  NaoCamera* camera = nullptr;
  CameraInfo cameraInfo;
  CameraIntrinsics cameraIntrinsics;
  CameraResolutionRequest cameraResolutionRequest;
  CameraResolutionRequest::Resolutions lastResolutionRequest =
CameraResolutionRequest::defaultRes;
  volatile bool cameraOk = true;
#ifdef CAMERA_INCLUDED
  static Semaphore performingReset;
  static bool resetPending;
  RingBuffer<std::string, 120> rowBuffer;
  unsigned int currentRow = 0, timestampLastRowChange = 0;
  std::string headName;
  unsigned int lastImageTimestamp = 0;
  unsigned long long lastImageTimestampLL = 0;
#endif

  Thread thread;
  Semaphore takeNextImage;
  Semaphore imageTaken;

  /**
   * This method is called when the representation provided needs to be updated.
   * @param theCameraImage The representation updated.
   */
  void update(CameraImage& theCameraImage) override;

  void update(CameraInfo& cameraInfo) override;
  void update(CameraIntrinsics& cameraIntrinsics) override {cameraIntrinsics = this-
>cameraIntrinsics;}
```

```
  void update(CameraStatus& cameraStatus) override;
  void update(FrameInfo& frameInfo) override {frameInfo.time = theCameraImage.timestamp;}
  void update(JPEGImage& jpegImage) override;

  bool readCameraIntrinsics();
  bool readCameraResolution();

  bool processResolutionRequest();

  void setupCamera();

  void takeImages();

public:
  CameraProvider();
  ~CameraProvider();

  /**
   * The method returns whether a new image is available.
   * @return Is an new image available?
   */
  static bool isFrameDataComplete();

  /**
   * The method waits for a new image.
   */
  static void waitForFrameData();
};
```

```
/**
* @file TCPServer.cpp
*/
#include "TCPServer.h"
#include <sys/socket.h>
#include <unistd.h>
#include <cstring>

// Constructor for the TCPServer class. Initializes the server with the specified port number.
// The server and client file descriptors are initialized to -1 to indicate they are not in use.
// The server's running state is initialized to false.
TCPServer::TCPServer(uint16_t port) : port(port), serverFd(-1), clientFd(-1), running(false) {}

// Destructor for the TCPServer class. It ensures that the server is stopped and resources are released
properly.
TCPServer::~TCPServer() {
    stop();
}

// Starts the server: creates a socket, binds it to the specified port on any available interface,
// listens for incoming connections, and starts a thread to accept connections.
void TCPServer::start() {
    // Create a socket for the server using IPv4 and TCP.
    serverFd = socket(AF_INET, SOCK_STREAM, 0);

    // Define the server address and port.
    sockaddr_in address{};
    address.sin_family = AF_INET; // Address family: IPv4
    address.sin_addr.s_addr = INADDR_ANY; // Listen on any network interface.
    address.sin_port = htons(port); // Convert port number to network byte order.
    // Bind the server socket to the specified address and port.
    bind(serverFd, reinterpret_cast<sockaddr*>(&address), sizeof(address));

    // Start listening for incoming connections. The backlog parameter is set to 1.
    listen(serverFd, 1);

    // Mark the server as running.
    running = true;

    // Start a new thread to handle incoming connections.
    acceptThread = std::thread(&TCPServer::acceptConnections, this);
}

// Stops the server: stops accepting connections, joins the accept thread,
// closes any open client and server sockets, and marks the server as not running.
void TCPServer::stop() {
    // Mark the server as not running.
    running = false;

    // If the accept thread is running, wait for it to finish.
    if (acceptThread.joinable()) {
        acceptThread.join();
    }

    // If a client is connected, close the client socket.
    if (clientFd != -1) {
        close(clientFd);
        clientFd = -1;
    }

    // Close the server socket if it's open.
    if (serverFd != -1) {
        close(serverFd);
        serverFd = -1;
    }
}

// Sends data to the connected client.
void TCPServer::send(const std::vector<unsigned char>& data) {
    // If a client is connected, send the data.
    if (clientFd != -1) {
        ::send(clientFd, data.data(), data.size(), 0);
    }
}

// Accepts incoming client connections.
// accepts new connections until the server is stopped. It handles one connection at a time.
void TCPServer::acceptConnections() {
    while (running) {
        // Accept a new connection
        clientFd = accept(serverFd, nullptr, nullptr);

        // If accepting a client connection fails, stop the server.
        if (clientFd < 0) {
            running = false;
        }
    }
}
```

```
/**
* @file TCPServer.h
*/
#pragma once
#include <netinet/in.h>
#include <string>
#include <thread>

// Declaration of the TCPServer class.
class TCPServer {
public:
    // Constructor: Initializes a new instance of the TCPServer with a specific port.
    // port: The TCP port number on which the server will listen for incoming connections.
    TCPServer(uint16_t port);

    // Destructor: Cleans up resources when a TCPServer object is destroyed. This includes stopping the
server if it is running.
    ~TCPServer();

    // Starts the server: Sets up the TCP socket, binds it to the specified port, listens for incoming
connections,
    // and launches a thread to handle these connections asynchronously.
    void start();

    // Stops the server: Stops the server from accepting new connections, closes any active connection,
and cleans up resources.
    void stop();

    // Sends data to the connected client.
    // data: A vector of unsigned chars representing the data to be sent to the client.
    void send(const std::vector<unsigned char>& data);
private:
    // The port number on which the server listens for incoming connections.
    uint16_t port;

    // File descriptor for the server socket. Used to manage the listening socket.
    int serverFd;

    // File descriptor for the client socket. Represents a socket for communication with the connected
client.
    int clientFd;

    // A thread object for accepting connections. This thread runs the acceptConnections method,
    // allowing the server to handle incoming connections without blocking the main thread.
    std::thread acceptThread;

    // A boolean flag indicating whether the server is currently running. This is used to control the loop
    // in the acceptConnections method and to manage the server's running state.
    bool running;

    // A private method that continuously accepts incoming connections when the server is running.
    // This method is intended to be executed in a separate thread.
    void acceptConnections();
};
```

```
# CameraFeedBottom.gd

# This script extends TextureRect to display a camera feed from a network source.


extends TextureRect


# Default port and server IP address for the camera feed

var port: int = 8082

var server_ip: String = "127.0.0.1"  # Defaulting to localhost for testing or local network use

var connection: StreamPeerTCP  # Declares a variable to hold the TCP connection


func _ready():

# This method is called when the node is added to the scene and is ready.

connection = StreamPeerTCP.new()  # Initializes the StreamPeerTCP object

var error = connection.connect_to_host(server_ip, port)  # Tries to connect to the specified host and
port

if error != OK:

print("Failed to connect to camera feed on port ", port)  # Prints an error message if the connection
fails

else:

print("Connected to camera feed on port ", port)  # Prints a success message if the connection is
established

set_process(true)  # Enables the _process function to be called every frame


func _process(delta):

# This method is called every frame and is used to update the camera feed.

# Checks if the connection is active and if there are bytes available to read

if connection.get_status() == StreamPeerTCP.STATUS_CONNECTED and
connection.get_available_bytes() > 0:

var jpeg_data = connection.get_var()  # Reads JPEG data from the connection

var image = Image.new()  # Creates a new Image object

var error = image.load_jpg_from_buffer(jpeg_data)  # Attempts to load the JPEG data into the Image
object

if error == OK:
```

```
var texture = ImageTexture.new()  # Creates a new ImageTexture object

texture.create_from_image(image)  # Creates a texture from the loaded image

self.texture = texture  # Updates the TextureRect's texture to display the new image
```

```
# CameraFeedTop.gd

# This script extends TextureRect to display a top camera feed from a network source.


extends TextureRect


# Default port and server IP address for the top camera feed

var port: int = 8081

var server_ip: String = "127.0.0.1"  # Defaulting to localhost for testing or local network use

var connection: StreamPeerTCP  # Declares a variable to hold the TCP connection


func _ready():

# This method is called when the node is added to the scene and is ready.

connection = StreamPeerTCP.new()  # Initializes the StreamPeerTCP object

var error = connection.connect_to_host(server_ip, port)  # Tries to connect to the specified host and
port

if error != OK:

print("Failed to connect to camera feed on port ", port)  # Prints an error message if the connection
fails

else:

print("Connected to camera feed on port ", port)  # Prints a success message if the connection is
established

set_process(true)  # Enables the _process function to be called every frame


func _process(delta):

# This method is called every frame and is used to update the camera feed.

# Checks if the connection is active and if there are bytes available to read

if connection.get_status() == StreamPeerTCP.STATUS_CONNECTED and
connection.get_available_bytes() > 0:

var jpeg_data = connection.get_var()  # Reads JPEG data from the connection

var image = Image.new()  # Creates a new Image object

var error = image.load_jpg_from_buffer(jpeg_data)  # Attempts to load the JPEG data into the Image
object

if error == OK:
```

```
var texture = ImageTexture.new()  # Creates a new ImageTexture object
texture.create_from_image(image)  # Creates a texture from the loaded image
self.texture = texture  # Updates the TextureRect's texture to display the new image
```

```
# Remote_Control.gd
# This script extends Node2D to create a remote control system using TCP communication.
# It's designed to send commands to a server based on user input.

extends Node2D

# TCP connection parameters including the server IP and the port to connect for commands
var server_ip: String = "127.0.0.1"  # The IP address of the server (localhost in this case)
var command_port: int = 8080  # The port on which to send command data
var connection: StreamPeerTCP  # Variable to hold the TCP connection object

# Called when the node is added to the scene tree, initializes the connection
func _ready():
connection = StreamPeerTCP.new()  # Creates a new StreamPeerTCP instance
connect_to_server()  # Calls the function to connect to the server

# Connect to the TCP server using the specified IP address and port
func connect_to_server():
var error = connection.connect_to_host(server_ip, command_port)  # Attempts to connect to the server
if error != OK:
print("Failed to connect to server at %s:%d" % [server_ip, command_port])  # Error handling if
connection fails
else:
print("Successfully connected to server at %s:%d" % [server_ip, command_port])  # Success message
on connection

# Send a command to the TCP server. This function is called with a string command
func send_command(command: String):
if connection.get_status() == StreamPeerTCP.STATUS_CONNECTED:  # Checks if the connection is
still active
var data = command.to_utf8_buffer()  # Converts the command string to UTF-8 byte buffer
connection.put_data(data)  # Sends the data through the connection
```

```
else:

print("Connection lost. Attempting to reconnect...")  # If connection is lost, tries to reconnect

connect_to_server()  # Reconnection attempt


# Input handler for sending commands. This listens for keyboard inputs

func _input(event):

if event is InputEventKey and event.pressed and !event.echo:  # Filters for non-repeated key press events

if event.scancode == KEY_KP_8:  # If the Numpad 8 key is pressed

send_command("walk_forward")  # Sends the "walk_forward" command to the server
```

Appendix 3 UML Class, Use Case, and sequence diagrams for this project.

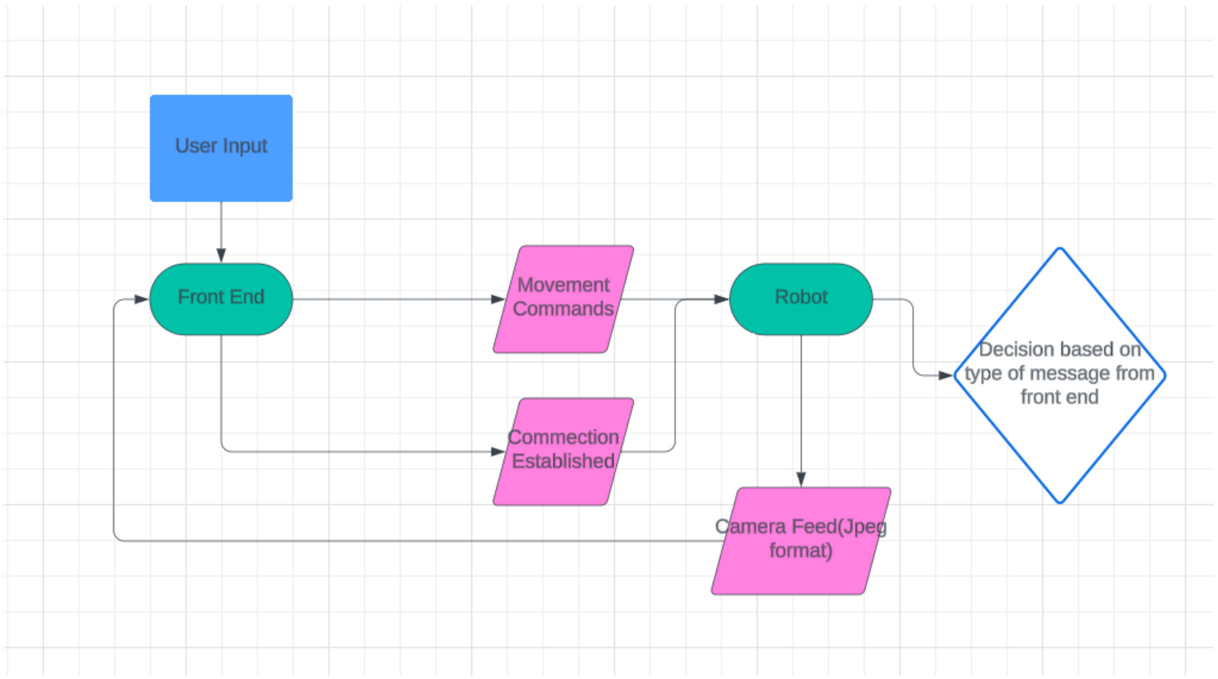


Figure 3-1 Flow chart showing the basic process of the planned System

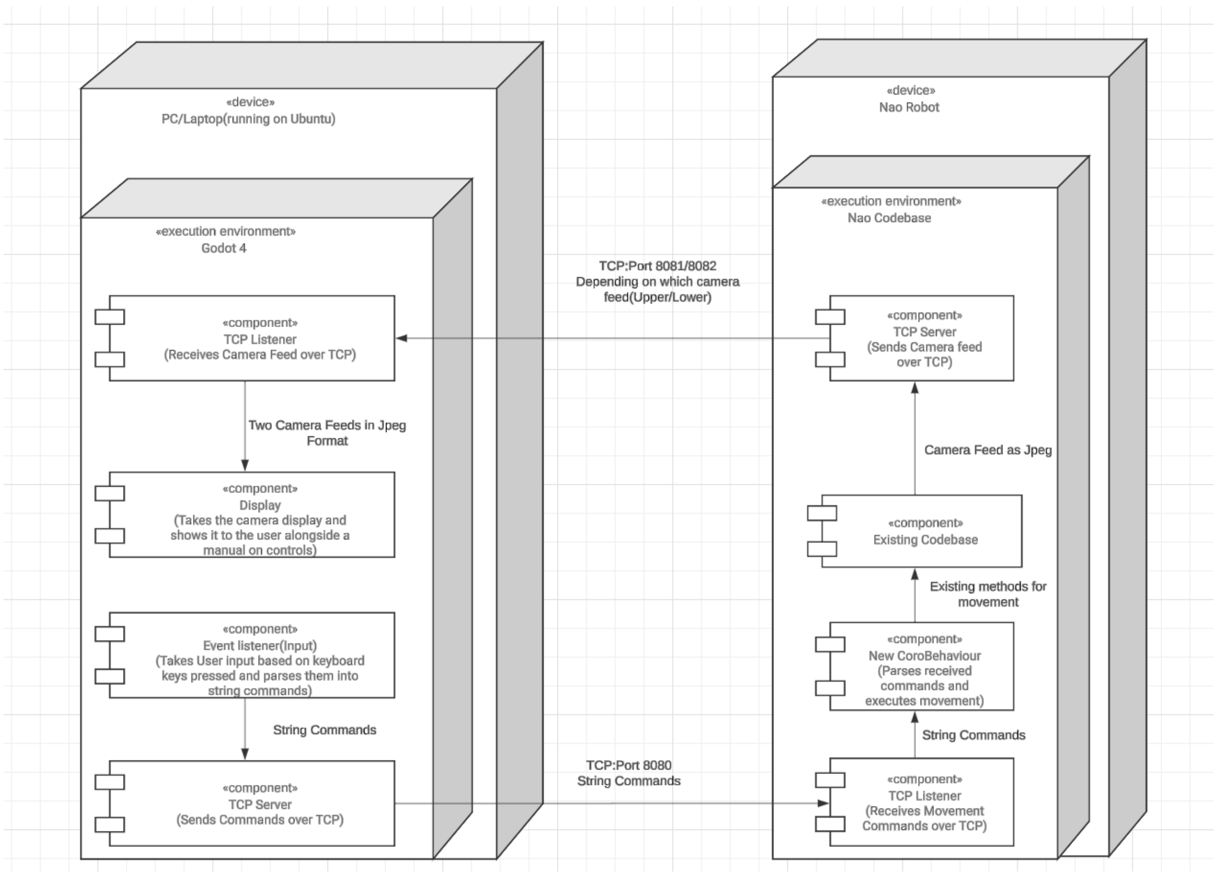

«device»
PC/Laptop(running on Ubuntu)
«execution environment»
Godot 4
«component»
TCP Listener
(Receives Camera Feed over TCP)
Two Camera Feeds in Jpeg Format
«component»
Display
(Takes the camera display and shows it to the user alongside a manual on controls)
«component»
Event listener(Input)
(Takes User input based on keyboard keys pressed and parses them into string commands)
String Commands
«component»
TCP Server
(Sends Commands over TCP)
TCP:Port 8081/8082
Depending on which camera feed(Upper/Lower)
TCP:Port 8080
String Commands
«device»
Nao Robot
«execution environment»
Nao Codebase
«component»
TCP Server
(Sends Camera feed over TCP)
Camera Feed as Jpeg
«component»
Existing Codebase
Existing methods for movement
«component»
New CoroBehaviour
(Parses received commands and executes movement)
String Commands
«component»
TCP Listener
(Receives Movement Commands over TCP)

Figure 3-2 UML Deployment Diagram of the planned System

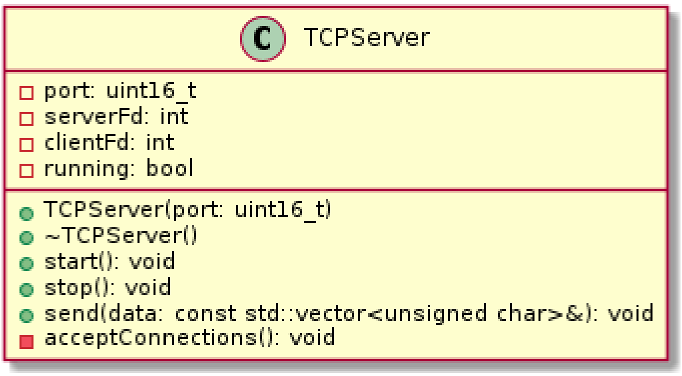


Figure 4-3-1 UML Class Diagram for the TCPServer Class

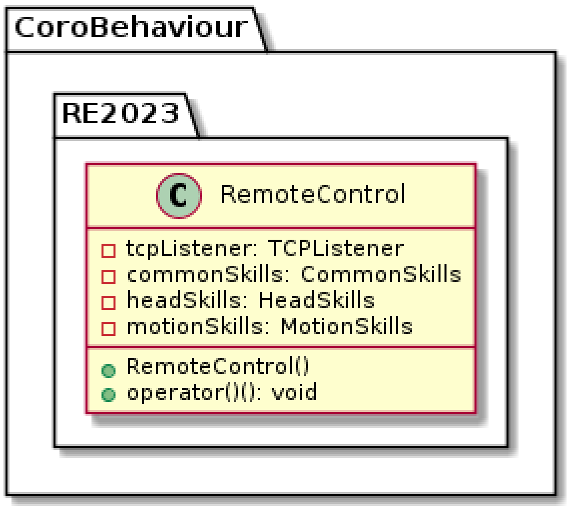


Figure 4-3-2 UML Class Diagram for the RemoteControl Class

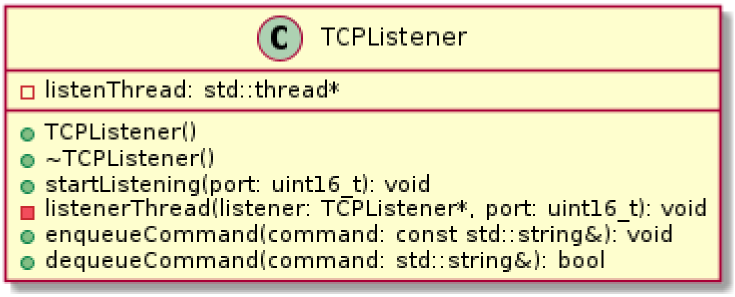

C TCPListener
□ listenThread: std::thread*
● TCPListener()
● ~TCPListener()
● startListening(port: uint16_t): void
■ listenerThread(listener: TCPListener*, port: uint16_t): void
● enqueueCommand(command: const std::string&): void
● dequeueCommand(command: std::string&): bool

Figure 4-3-3 UML Class Diagram for the TCPListener Class

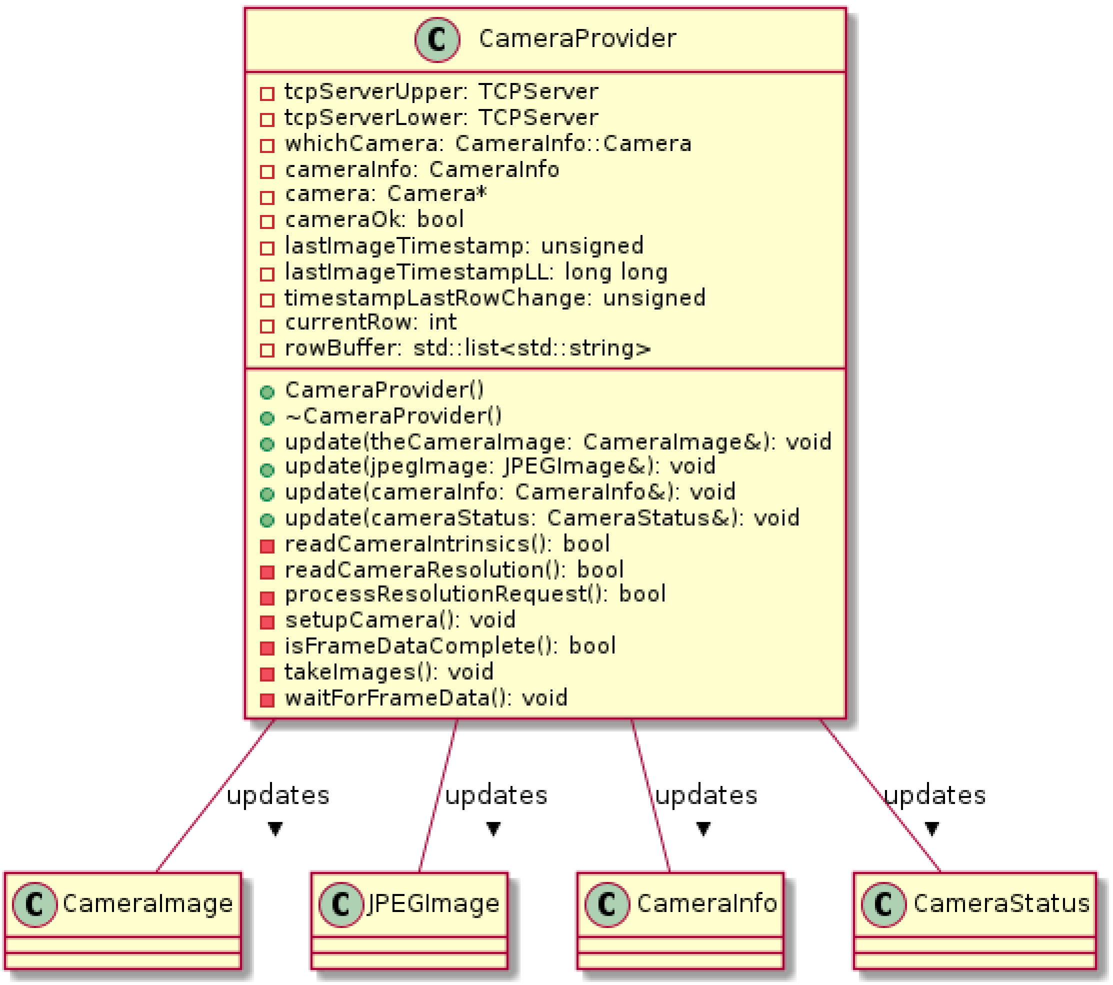

CameraProvider
tcpServerUpper: TCPServer
tcpServerLower: TCPServer
whichCamera: CameraInfo::Camera
cameraInfo: CameraInfo
camera: Camera*
cameraOk: bool
lastImageTimestamp: unsigned
lastImageTimestampLL: long long
timestampLastRowChange: unsigned
currentRow: int
rowBuffer: std::list<std::string>
CameraProvider()
~CameraProvider()
update(theCameraImage: CameraImage&): void
update(jpegImage: JPEGImage&): void
update(cameraInfo: CameraInfo&): void
update(cameraStatus: CameraStatus&): void
readCameraIntrinsics(): bool
readCameraResolution(): bool
processResolutionRequest(): bool
setupCamera(): void
isFrameDataComplete(): bool
takeImages(): void
waitForFrameData(): void
updates ▼
updates ▼
updates ▼
updates ▼
CameraImage
JPEGImage
CameraInfo
CameraStatus

Figure 4-3-4 UML Class Diagram for the altered CameraProvider Class

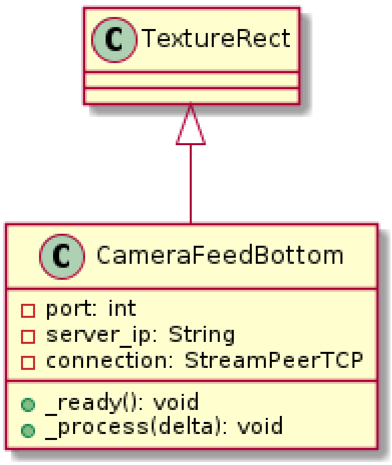


Figure 4-3-5 UML Class Diagram for the CameraFeedBottom Godot Script

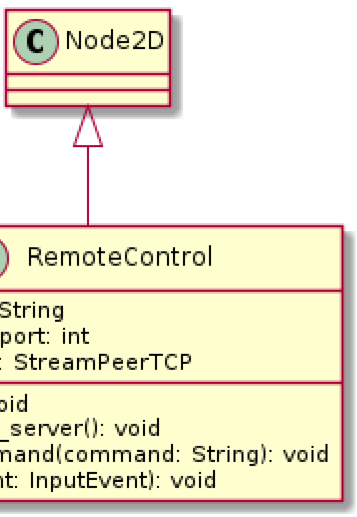
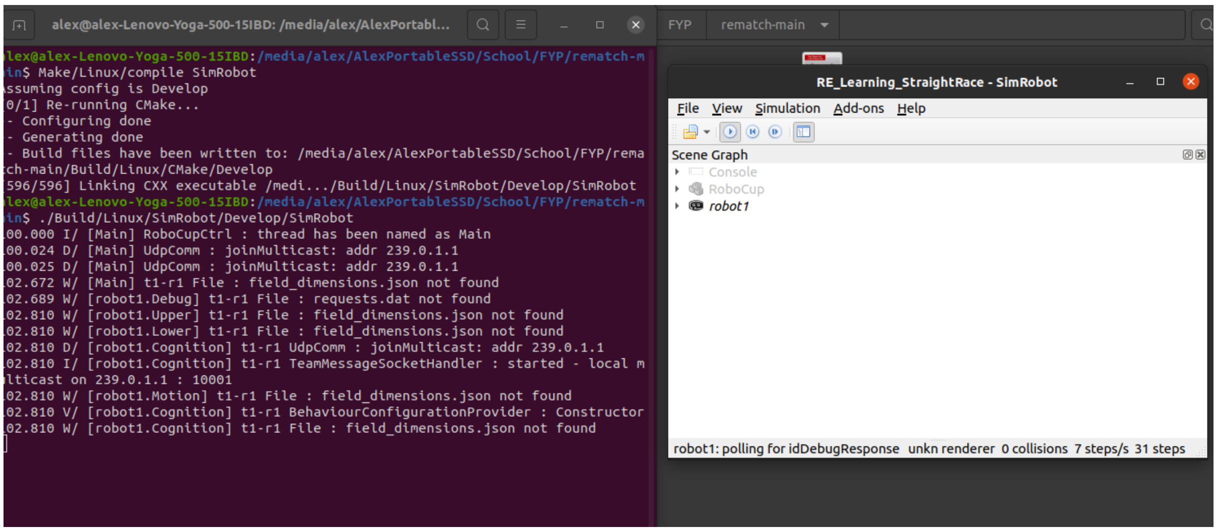

Node2D
RemoteControl
- server_ip: String
- command_port: int
- connection: StreamPeerTCP
+ _ready(): void
+ connect_to_server(): void
+ send_command(command: String): void
+ _input(event: InputEvent): void

Figure 4-3-6 UML Class Diagram for the RemoteControl Godot Script

Appendix 4 Screen shots of the project implementation

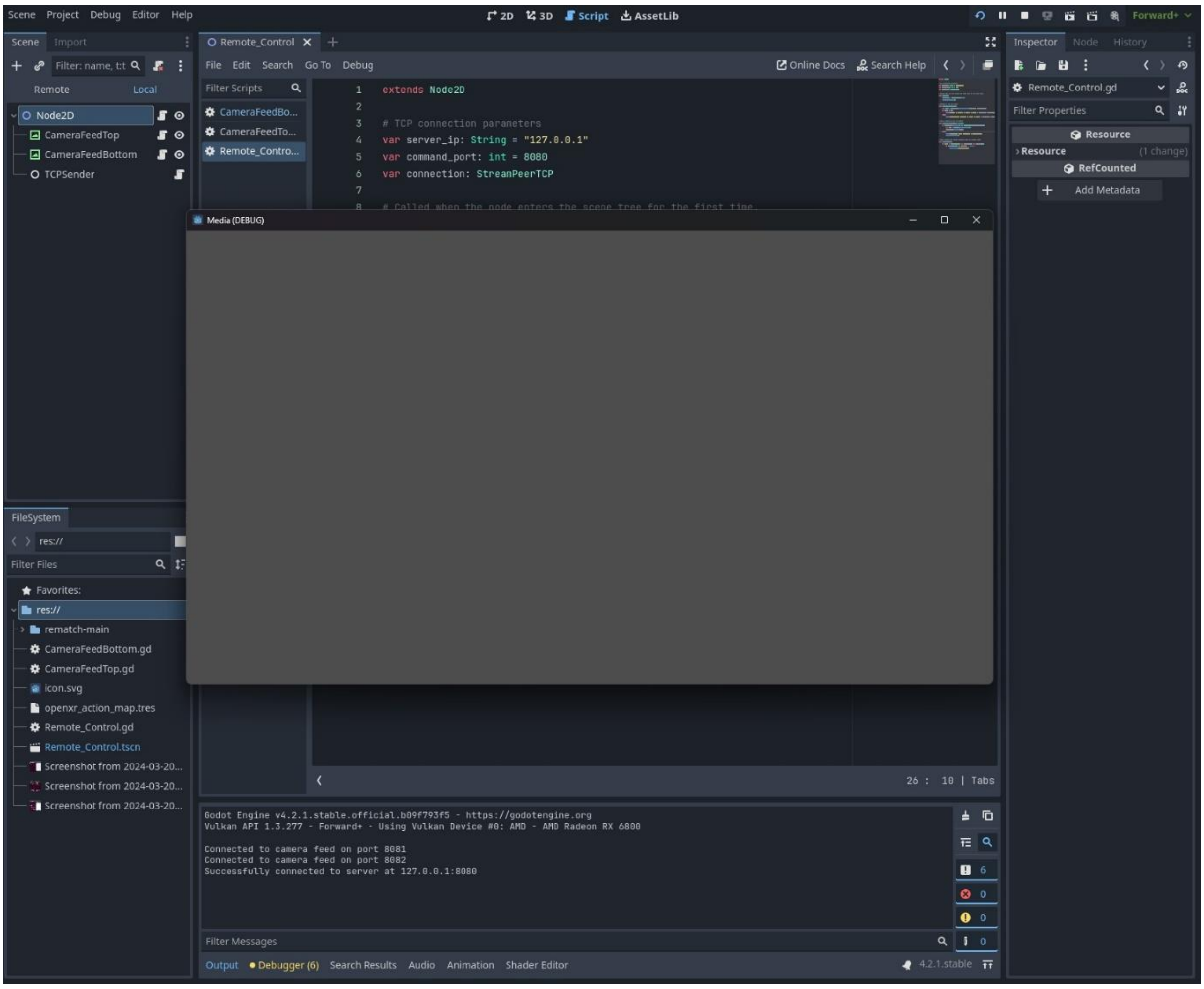


```
lex@alex-Lenovo-Yoga-500-15IBD:/media/alex/AlexPortableSSD/School/FYP/rematch-m
in$ Make/Linux/compile SimRobot
ssuming config is Develop
0/1] Re-running CMake...
- Configuring done
- Generating done
- Build files have been written to: /media/alex/AlexPortableSSD/School/FYP/rema
ch-main/Build/Linux/CMake/Develop
596/596] Linking CXX executable /medi.../Build/Linux/SimRobot/Develop/SimRobot
lex@alex-Lenovo-Yoga-500-15IBD:/media/alex/AlexPortableSSD/School/FYP/rematch-m
in$ ./Build/Linux/SimRobot/Develop/SimRobot
00.000 I/ [Main] RoboCupCtrl : thread has been named as Main
00.024 D/ [Main] UdpComm : joinMulticast: addr 239.0.1.1
00.025 D/ [Main] UdpComm : joinMulticast: addr 239.0.1.1
02.672 W/ [Main] t1-r1 File : field_dimensions.json not found
02.689 W/ [robot1.Debug] t1-r1 File : requests.dat not found
02.810 W/ [robot1.Upper] t1-r1 File : field_dimensions.json not found
02.810 W/ [robot1.Lower] t1-r1 File : field_dimensions.json not found
02.810 D/ [robot1.Cognition] t1-r1 UdpComm : joinMulticast: addr 239.0.1.1
02.810 I/ [robot1.Cognition] t1-r1 TeamMessageSocketHandler : started - local m
lticast on 239.0.1.1 : 10001
02.810 W/ [robot1.Motion] t1-r1 File : field_dimensions.json not found
02.810 V/ [robot1.Cognition] t1-r1 BehaviourConfigurationProvider : Constructor
02.810 W/ [robot1.Cognition] t1-r1 File : field_dimensions.json not found
```

alex@alex-Lenovo-Yoga-500-15IBD: /media/alex/AlexPortabl...
FYP
rematch-main
RE_Learning_StraightRace - SimRobot
File View Simulation Add-ons Help
Scene Graph
Console
RoboCup
robot1

robot1: polling for idDebugResponse unkn renderer 0 collisions 7 steps/s 31 steps

Figure 5-4-3 This shows the testing suite working as intended.(Lenovo Laptop)



```
extends Node2D

# TCP connection parameters
var server_ip: String = "127.0.0.1"
var command_port: int = 8080
var connection: StreamPeerTCP

# Called when the node enters the scene tree for the first time.
```

```
Godot Engine v4.2.1.stable.official.b09f793f5 - https://godotengine.org
Vulkan API 1.3.277 - Forward+ - Using Vulkan Device #0: AMD - AMD Radeon RX 6800

Connected to camera feed on port 8081
Connected to camera feed on port 8082
Successfully connected to server at 127.0.0.1:8080
```

Figure 5-4-4 This shows the front-end compiling without errors.

Appendix 5 User Manual

In order to run the Nao codebase:

1. On a Ubuntu 20.04 System, open a terminal.
2. Install all necessary packages with ‘sudo apt install clang cmake graphviz libasound-dev libglew-dev \ libqt5opengl5-dev libqt5svg5-dev lld llvm net-tools ninja-build \ pigz qtbase5-dev rsync wish xterm xxd’
3. Navigate to the FYP/rematch-main folder.
4. Paste ‘NO_CLION=true Make/Linux/generate’ into the terminal.
5. To build the code run ‘Make/Linux/compile SimRobot Develop’
6. Run the SimRobot using ‘./Build/Linux/SimRobot/Develop/SimRobot’
7. Now to open a scene file. Go to File > Open and choose the scene FYP\rematch-main\Config\Scenes\RELearning\RE_Learning_StraightRace.ros2
8. This should open the scene as intended and the SimRobot should show that there is a Robot like in the screenshot in Appendix 4.

In order to run the Godot 4 Front-end application.

1. On a Windows or Ubuntu open up the Godot 4 Engine.
2. Navigate to Import and from there navigate to the FYP folder, select project.godot and click open.
3. Once the code loads in, click the run button on the top right of the window.
4. This should run without issue and show a window similar to the screenshot in Appendix 4