UNIVERSITY OF YORK

MASTERS THESIS

Augmented Reality Debugging System for Robot Swarms

Author:
Alistair JEWERS

Supervisor:
Dr. Alan MILLARD

A thesis submitted in fulfillment of the requirements for the degree of Master of Engineering

in the

Department of Electronic Engineering

May 7, 2017

Declaration of Authorship

I, Alistair JEWERS, declare that this thesis titled, "Augmented Reality Debugging System for Robot Swarms" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

University of york

Abstract

Faculty Name

Department of Electronic Engineering

Master of Engineering

Augmented Reality Debugging System for Robot Swarms

by Alistair JEWERS

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

Dedication...

Chapter 1

Introduction

1.1 Background

Recent years have seen rapid development in robotics technology due to the constantly increasing availability of computing power, reductions in the cost of hardware such as digital sensors and actuators, and developments in the application of artificial intelligence to robot control. This has lead to robots being used to perform increasingly complex tasks and solve ever more difficult problems. Many new areas of robotics research have emerged as a result, as researchers strive to find new and better ways to apply this technology, entering into problem domains once thought to be impossible for robots. Whole new robotics paradigms have been created as the standard model of a single, complex, expensive robot has been questioned, opening the door for cooperative robots, multi-robot systems, and more specifically swarm robotics.

Studies into the self-organising behaviour of social insect colonies, and the development of mathematical models based on these behaviours, led to the development of a field of research referred to as Swarm Intelligence (SI). The aim of these models is to determine how large numbers of individual agents are able to solve problems collectively, with each agent using only local information, and without any centralised control. Swarm Robotics developed from a desire to apply these concepts in practice to real world problem solving. Dorigo et al. describe swarm robotics as '*the study of how to design groups of robots that operate without relying on any external infrastructure or on any form of centralized control ...* [where] *the collective behaviour of the robots results from local interactions between the robots and between the robots and the environment*[1]'. Swarm robotics has since emerged as a promising area of research for solving problems which would be infeasibly difficult or expensive for a conventional robotics approach.

1.2 Project Context

Developing and debugging robotics behaviours has always been a challenging task. Whilst traditional software is run in a purely digital environment with a tightly controlled set of inputs and outputs to and from the physical world, robots must interact constantly with the physical world in order to satisfy their intended purpose. Robots are therefore subject to a much wider array of inputs and outputs, and to a huge number of changing variables within their environment at any given time. Often these variables and inputs are continuous in nature, rather than the discreet inputs more commonly used by traditional computers. This makes detecting, reproducing and correcting specific faults in a robot's behaviour significantly harder than in traditional software. A large number of continuous inputs leads to a theoretically infinite set of possible input configurations, and can therefore make reproducing the exact conditions under which a fault occurred extremely difficult if not impossible. Another of the main difficulties comes from the layers of abstraction between the real world, the robot, and the human developer. There is a potential disconnect between the robot's interpretation of the world and the reality of the world itself. Inaccuracies in this interpretation can be caused by any number of issues, including sensor hardware problems as well as software bugs. This can cause erroneous behaviour that might be wrongly attributed to a bug in the robot's behavioural code or decision making, rather than its perception. This issue can be compounded by the fact that the human operator's knowledge of the robot's interpretation of the world might also be inaccurate or incomplete. Figure 1.1 shows these different layers of information abstraction when dealing with a robotic system. The arrow highlighted in red shows where many of these abstraction related debugging difficulties occur. Retrieving human readable information from a robot in a timely manner whilst it is running is often non-trivial, and what the robot sees and what the human operator thinks the robot sees may differ significantly.

This problem is made significantly more complex when working with multi-robot systems, and especially swarm robotics. Introducing multiple robots multiplies the number of potential variables and increases the amount of information required to describe the system, hence both the number of points where a bug may be occurring and the amount of information the operator needs in order to locate it are also increased. The decentralised nature of swarm robotics systems further exacerbates this problem through the lack of a single, central control point, where information for the whole system can be retrieved.

Traditional software debugging tools are predominantly text based, and often require that program execution is paused in order to examine internal values. These tool are therefore often insufficient for robotics applications, as the quantities of data produced by a robot's sensors, and used to make its decisions, can be too large and vary too rapidly for any sensible textual representation. Furthermore pausing

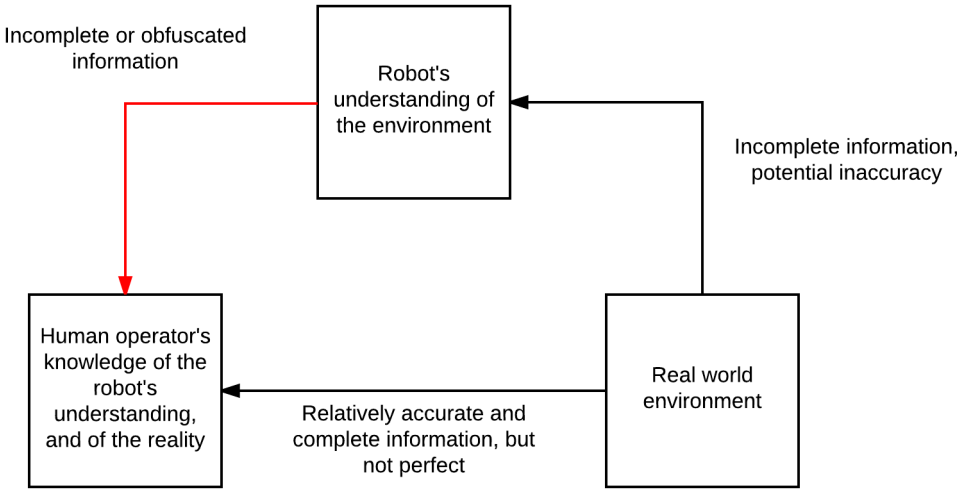


FIGURE 1.1: Layers of information abstraction in robotics debugging.

a robots execution can be undesirable, especially if it is operating in an active real world environment. Once again swarm robotics and multi-robot systems compound on this problem by increasing the amount of data involved, making textual representations less likely to be sufficient, and by involving multiple data sources, which traditional debugging tools are rarely designed to handle. New tools and techniques are therefore needed to retrieve data from these robotic systems and present it in a manner that allows humans to understand and process it effectively.

Virtual and augmented reality technologies have seen significant progress in the last few years, and are beginning to emerge into a large number real-world applications. Augmented reality in particular offers a promising new method for interacting with robots, and may help to overcome some of the limitations of traditional debugging tools. An augmented reality system works by capturing a view of a real world environment through a camera, and augments it with computer generated graphical elements. These graphical elements or 'virtual objects' are usually positioned within the space such that they either appear to be real, physical objects, or relate to other physical objects within the space. Augmented reality systems can therefore create 'hybrid-' or 'mixed-reality' environments, where users are able to interact with real world objects and virtual, computer generated ones simultaneously. This shows particular promise when combined with robotics, as a mixed-reality space is one that can be shared and understood by both humans and robots. The virtual elements of an augmented reality environment are simply representations of digital data, and this data can be readily understood by a digital system such as a robot. By creating tools which effectively utilise augmented reality techniques and mixed-reality environments in conjunction with robots, it should be possible to broaden the human-robot communication channel, introducing novel ways for humans and

robots to communicate and share information, with implications for all aspects of human-robot interaction. Considering debugging specifically, it is easy to imagine how converting a robot's data such as sensor readings to graphical representations, correctly positioned to reflect their spatial significance, could improve a human operator's ability to identify problems with a robot's perception and behaviour.

1.3 Project Concept

This project focuses on trying to mitigate some of the problems discussed in the previous section by improving a human operator's access to system information, thus improving the timeliness with which bugs in a swarm robotics system can be identified, located and fixed. This means designing and implementing a system capable of collecting information from multiple sources and presenting it all in one place, in a human readable manner, in real time. The information sources to be used include the individual robots themselves, as well as a live video feed of the robots and their environment. The project also attempts to incorporate ideas and techniques related to augmented reality in order to represent data retrieved from a robotic system in ways that can be more intuitively understood by a human operator than purely textual and numerical representations.

This project attempts to create a software application and associated wireless data transaction format capable of presenting a user with a single, coherent, and highly readable interface through which they can view relevant information about a swarm and it's constituent robots in real time. This includes the use of a video based tracking system to monitor robot positions, and provide the user with a view of the robots' environment. This view can then be augmented with graphical representations of relevant elements of the retrieved robot data, such as sensor readings. The robots will communicate data to the computer running the application wirelessly. The initial target robot platform is the widely used *e-puck* robot [2], equipped with a Linux extension board and WiFi adapter. The e-puck platform is discussed in greater detail in chapter 5. The diagram in figure 1.2 gives a logical representation of the proposed system architecture, in terms of its component parts, including the e-puck robots, tracking camera, and the application's host computer. This report describes the design, implementation and testing of this system, and includes details of the steps undertaken to evaluate its effectiveness. Some portions of this report appeared previously in a similar form in an *Initial Report* document, and are included here for completeness, with minor alterations.

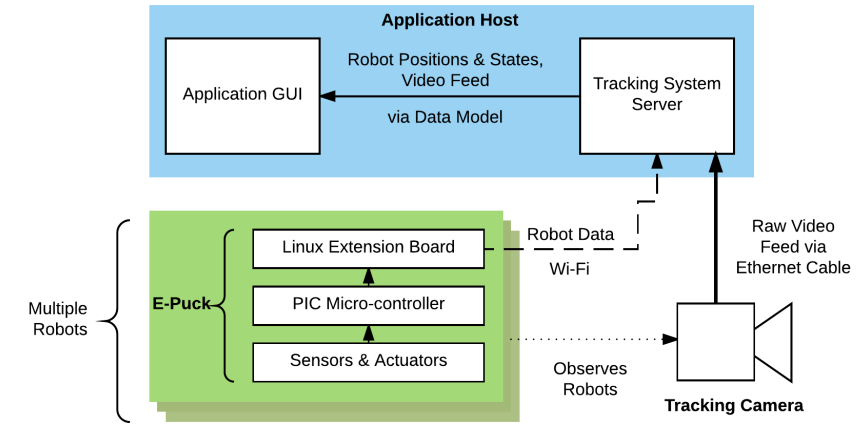


FIGURE 1.2: The proposed system architecture.

1.4 Aim and Objectives

Having gained an understanding of the context for this project in section 1.2, and outlined the concept for the system in section 1.3, the project aim can be summarised in the following statement:

This project aims to understand the needs of a swarm robotics researcher or system developer when attempting to debug their system, and create a computer application which allows a user to monitor the state and behaviour of a robot swarm system in real time, employing augmented reality techniques where possible, and thus improving the ease and efficiency of the debugging process.

The objectives required to achieve this aim are as follows:

- Utilize existing fiducial marker based tracking technology to track the position of individual robots within a swarm over time.
- Develop an application for presenting the user with a live video feed of a robot swarm, augmented with relevant and spatially situated information relating to the robots, using the data obtained from the tracking system.
- Develop code to allow multiple robots to communicate information regarding their internal state, sensor readings and decision making to the application wirelessly via a network.
- Develop a data model that allows the application to store information received from the robots, and update it as new information arrives.
- Develop code to ascertain higher level data related to the robots, such as recent movement history or state transition history, and add this data to the model.

- Design and implement a user interface which presents the data model to the user in a human readable manner, and performs data fusion on the information provided by the robots and the tracking information.
- Develop the user interface in such a way as to allow the user to filter out information that is not currently relevant, and to contrast and compare information related to specific robots.
- Design and implement the system in a modular way so as to allow for relatively simple integration with other swarm robotic platforms and tracking systems in future extensions.

1.5 Functional Specification

When developing software of any kind it is common practice to define a *software specification* prior to starting development. This specification describes the functionality required in the software in order for it to satisfy its purpose. The specification presented here is separated into core and secondary requirements. Core requirements are considered essential to the satisfactory delivery of the software. Secondary requirements are desirable but not strictly necessary, and will be satisfied where possible, given the time constraints of the project.

Core Requirements:

1. Must be comprised of a PC application.
2. Must be capable of receiving data related to the state of multiple robots.
3. Must be capable of receiving positional data for the same set of robots.
4. Must be capable of receiving a live video feed of the robots in their environment.
5. Must collate received data and present it to the user in a combined graphical form.
6. Must present auxiliary, non-spatial data to the user in textual or other forms.
7. Must update in approximately real time.
8. Must at minimum support the e-puck robot platform.

Secondary Requirements:

1. Should use a modularised structure.
2. Should exchange data between the robot platform and the application using a platform-agnostic, extensible protocol.

3. Should provide a basis for interoperability with a number of robotics platforms.
4. Should allow the user to configure the displayed data.
5. Should employ a model-view-controller (MVC) software architecture.
6. Could provide the user with ways to configure and display custom data types.
7. Could allow the user to compare data on two or more specific individual robots.
8. Could calculate and display swarm-level meta-data and statistics.
9. Could generate log files of robot activity over a user defined period.

1.6 Report Structure

This report begins with a survey of the existing literature relevant to the project topic. Individual pieces of research and writing with relevance to some area of the project are highlighted. The project plan is then outlined, with details of the tasks to be undertaken and the time allocated for each, as well as the risk assessment made at the start of the project. This is followed by a short statement on ethics. The hardware to be used in the project is then examined in detail, with information about the target robot platform, the e-puck, and details of the camera and tracking system. The results of an initial survey of some potential users of the system is then presented, and the effect of these results on the implementation are summarised. The design process is then described in detail, including both the structural design of the software architecture and the design of the user interface. The next section gives details of the implementation of the software, with descriptions of how all the key software components function, and how they interact with one another. This is followed by a description of the testing processes used to verify the software and the results, and details of how the system's effectiveness was evaluated. The penultimate section focuses on possible future work that could be carried out to improve the system. Finally the conclusion looks first at the system, discussing its effectiveness, shortcomings, and place in the wider field of swarm robotics, and then goes on to examine the successes and failures of the project itself.

Chapter 2

Literature Review

2.1 Overview

This section presents a review of some of the literature from the field of swarm robotics, including some general summaries of the field's fundamentals, as well as a number of specific pieces of swarm robotics research with particular relevance to this project. It also presents a number of pieces of research from other, related areas, such as human-robot interaction and robotics-focused augmented reality research. The results of this literature survey informed the project direction significantly, and formed the basis for many of the design and implementation decisions made later on. It is also presented with the aim of providing the reader with the base of knowledge required to better understand the project. The literature covered in this section can be separated into several broad topics, each informing a different element of the project work.

Firstly an understanding of the fundamental concepts of Swarm Robotics, and to a lesser extent Swarm Intelligence, was deemed key to producing an application that is useful in practice, and will help a reader to better understand the purpose and aims of the project. A number of key publications related to the core concepts of swarm robotics are presented in Section 2.2. A deep understanding of the technical details of specific swarm systems, such as specific behavioural algorithms or implementation details, is not a priority for understanding this project, as the application aims to be more broadly applicable to a wide range of swarm systems. Emphasis has instead been placed on understanding the general classification of swarm robotic systems, relevant problem domains, and recurring concepts, so that the system might better serve researchers in the field.

This project focuses on a piece of software which forms an interface between a human operator and a robot swarm. A relevant area of research is therefore Human-Swarm Interaction (HSI). Research in this area focuses on the different ways in which humans and robot swarms can interact, the different roles humans take whilst interacting with robot swarms, and the best practices for facilitating this interaction

given different aims, and different user roles (developer, researcher, end user, etc.). The two key challenges of HSI are control - how best to allow a human operator to direct the behaviour of a decentralised swarm - and monitoring - how to retrieve data from a swarm and present it in a useful, human readable manner. This project is primarily related to debugging robot swarm behaviours, and therefore the literature presented from this area focuses on the monitoring side of HSI. An overview of the relevant Human-Swarm Interaction literature is presented in Section 2.3.

'*Debugging*' is the name given to the process of fixing problems or issues (commonly referred to as bugs) in a piece of software. Well established tool-sets and techniques exist for debugging traditional software, however debugging robotic systems is a fundamentally different and more complex problem, as discussed in section 1.2. Section 2.4 reviews literature related to the challenges of robotics debugging and potential solutions.

As previously discussed, augmented reality technologies have been identified as a potentially powerful tool for facilitating human-robot interaction and communication. Section 2.5 reviews a number of pieces of research relating to the use of AR technologies in conjunction with robotic systems.

Finally specific pieces of literature which have significant common ground with this project are reviewed, in an attempt to assess the existing work in this specific area, and establish the advances offered by the system created in this project. A number of pieces of research have investigated different ways to monitor and debug single robot and multi-robot systems, using a range of the technologies and concepts discussed throughout this literature review, including AR technology and HSI techniques. These similar works are summarised and reviewed in section 2.6, with a specific focus on systems which employ graphical, AR-style techniques to a debugging context.

2.2 Swarm Intelligence and Swarm Robotics

Sahin [3] presents a summary of the key concepts of swarm robotics, and attempts to offer a coherent description of the topic. He notes that a key difference from other multi-robot systems is the lack of centralised control, and the idea that desired behaviour should emerge from simple local interactions between robots, and between the robots and their environment. He also notes some of the key motivators behind Swarm Robotics research, stating that a swarm robotics system would ideally exhibit "*robustness*", "*flexibility*" and "*scalability*" [3]. Robustness refers to the swarm's ability to continue to function should one or more individual swarm members suffer a failure of some kind. Flexibility refers to the swarm's ability to adapt to changes in the environment without the need for re-programming. Scalability describes the

idea that a swarm should be functional at a range of sizes, and that ideally the number of robots in the swarm could be increased or decreased depending on the demands of the task. These descriptors should be taken into account by the design of the system presented in this project. In order for the system to be able to work well with robust, scalable swarms it should adapt easily to variation in the number of robots being monitored. Newly detected robots should therefore be incorporated seamlessly. The video feed component of the system will allow a user to see the environment the swarm is interacting with and observe how robots respond to changes within it, and therefore judge the extent to which the swarm exhibits flexibility.

Sahin [3] goes on to describe several classes of application for which Swarm Robotics systems might be well suited. Tasks that cover a region could benefit from a swarm's ability to distribute physically in a space according to need. Dangerous tasks could benefit from the relative dispensability of individual robots in the swarm; should one be damaged or destroyed the swarm could continue to function, and it would be less costly that the loss of a single, complex, expensive robot. Tasks requiring scalability are good candidates, as discussed before, and tasks that require redundancy are also highlighted, as swarm systems should have the ability to degrade gracefully, rather than suffering a single catastrophic failure. Through this generalisation of the application areas, insight can be gained into the kinds of work swarm robotics researchers are likely to be doing, and this should inform the design of the application. Overall Sahin's paper [3] provides a coherent, succinct overview of the field, and although it is now over a decade old the concepts covered remain relevant.

The book '*Swarm Intelligence: From Natural to Artificial Systems*', by Bonabeau, Dorigo and Theraulaz [4], provides in its introductory chapter a good overview of the biological concepts and animal behaviours which inspired much of the research that led to the creation of the field of swarm intelligence. Fundamental concepts such as self-organisation and decentralised control are discussed. In order to be a useful tool in the swarm robotics research space, it is important that the system developed during this project does not violate these core principles of the swarm robotics paradigm. Therefore the system should not facilitate low level control of the robots or allow for forms of communication and data exchange that might invalidate the self-organising, decentralised nature of the swarm's behaviour. The later chapters [4] provide a detailed look at several key insect behaviours, and how mathematical models and algorithms can be derived to mimic them. An understanding of these behaviours and models can offer insight into what information the application might need to expose to allow a user to validate the correct operation of a swarm behaviour based on these concepts. A number of these algorithms focus or rely on data related to the position and movement of individuals within the swarm, and the swarm as a whole. The system developed during this project should reflect the importance positional data and provide features which aid the user in interpreting

it.

In a more modern work, Brambilla et al. [5] focus heavily on the engineering practicalities of designing, implementing and testing swarm robotic systems. The authors then apply this focus as a means by which to classify and critique a large body of swarm robotics research, noting that although much work has been carried out regarding the design and analysis of swarm behaviours, other areas including maintenance and performance measurement are heavily lacking in research contributions. Debugging can be considered a maintenance task, and the system developed in this project has potential in both maintenance and performance measurement applications. It is hoped therefore that this work might contribute to this area of the field, by investigating through implementation the practicalities of a generalised swarm maintenance and observation software application. The authors [5] later note whilst concluding that human-swarm interaction remains an open issue, and will be key to realising functional, real-world swarm robotic systems. They identify a number of works related to HSI, almost all of which focus on the task of controlling a robot swarm, and involve different methods by which a user might insert data into the swarm system. Little consideration appears to be given, both in this paper and throughout the literature, to the task of monitoring a swarm, and best practices for retrieving information in a manner that is useful to a human operator.

2.3 Human Swarm Interaction

In their paper '*Human Interaction with Robot Swarms: A Survey*' [6] Kolling at el. begin by noting the lack of research into methods for interfacing humans and robot swarms. They suggest that real-world applications for swarm robotics systems are now within reach, and that discovering effective methods for allowing humans to control and/or supervise swarms is a key barrier to realising these systems. The paper [6] provides a detailed analysis of human swarm interaction from a number of different perspectives. Of relevance to this project is the statement on page 15 that "*Proper supervision of a semiautonomous swarm requires the human operator to be able to observe the state and motion of the swarm, as well as predict its future state to within some reasonable accuracy*" [6]. This statement lends credence to the aims of this project, as swarm supervision and swarm debugging are highly comparable tasks; both involve observing the swarm whilst performing its task and determining the validity of the behaviour observed. The system developed in this project should allow the state of the swarm, including the internal state of individual robots, to be observed simultaneously with the physical positions and motions of the robots within their environment. The paper [6] goes on to suggest that by observing the swarm over time the human operator will be able to provide '*appropriate control inputs*'. In the case of this application, rather than providing control input, the human operator

will be seeking to identify faults, and provide appropriate corrections to the system, however the concept of state visualisation remains relevant.

Rule and Forlizzi [7] present a thorough examination of the complexities of human robot interaction (HRI) when dealing with multi-robot (and multi-user) systems. Much of the paper focusses on control methods, which are not directly applicable to this project, however section 2.4 titled *Salience of Information* discusses the task of designing interfaces for displaying information about multi-robot systems to a human operator in a manner which is both information dense and rapidly understandable. The authors note that the use of colour has been shown to improve interface readability [8], and that the brain has been shown to process text faster than images [9], hence complex icons should be avoided. These ideas should be incorporated into the design of the application user interface for this project. A range of different designs could be explored, including finding a balance between the amount of information displayed graphically, and the amount displayed textually, and deciding whether to use colour to differentiate between individual robots, or to differentiate between different types of data, or a combination of both.

The authors [7] then go on to use '*contextual inquiry*' interviews with robot operators to determine a set of questions which, if answered, should allow for robust robot operation. Of relevance to this project are the four questions in the *human-robot* category. These are 1) "*What mode, state, and environment is the robot in and how will this affect my commands?*", 2) "*Which robot needs my attention?*", 3) "*What is each robot accomplishing?*", 4) "*What can I accomplish with this robot?*". Providing the user with answers to questions two, three and four is beyond the scope of this project, however the system should provide the user with an answer to question one, by allowing access to state and environment information simultaneously. Once again although the authors [7] consider only a robot control perspective, these questions remain broadly relevant to a debugging perspective also. For example, question one could be rephrased as *What mode, state, and environment is the robot in, and is this having the correct effect on it's behaviour?* to better fit the use case of this project. When discussing their results, specifically the appropriate level of display salience for robot state awareness in section 4.8 [7], the authors note that users wanted the ability to "*select which aspects of robot information to view at any one time*". They also state that users wanted basic overview information, with the ability to access a more detailed view specific to one robot when "troubleshooting". This configurable, layered approach to information display and access should be incorporated into the user interface design for the system.

The authors [7] do not state how they determined that the set of "*essential situation awareness questions*" presented ensured robustness. The contextual inquiry method is potentially subjective, suggesting that answering only these questions may not guarantee full awareness. Other factors which a system user could benefit from an awareness of should be considered. For example the authors mention

the robot's mode, state and environment, but do not include the robot's sensor data. Furthermore care must be taken when applying the results of this work to swarm robotic systems, as it was not produced with swarms specifically in mind. Therefore applying the results within, without also considering needs specific to swarm robotics, could lead to too strong a focus on the actions of individual robots, and a lack of focus on the overall behaviour of the swarm.

Podevijn, O'Grady and Dorigo [10] present a novel method for obtaining feedback from an active robot swarm. The authors propose that information sent back from a self-organised swarm to an operator should itself be self-organised. They note a number of motivations for this approach. Firstly, should every robot simultaneously report information regarding its own "*local worldview*", the authors suggest [10] that the operator would be over-burdened with an excess of information, and would therefore need further tools of some kind to parse this data effectively. They also suggest that the required communication infrastructure would increase the overall system complexity, and the hardware requirements of each robot. The volume of communication traffic for a large swarm might prove impossible for a network to handle. Their work [10] demonstrates a number of possible methods by which a swarm of robots might report information in a self organised manner, in the hope of combining and reducing the information that needs to be reported. Whilst completing a grouping task, coloured LEDs were used to indicate visually which group each robot was a part of. This can be seen in figure 2.1. Another proposed mechanism [10] would use the formation of the swarm to indicate information about their behaviour, for example when moving collectively in one direction the robots would arrange into an arrow formation indicating the direction of movement. This mechanism was proposed only in concept, and not implemented on a real swarm.

This self organised approach to information reporting [10] has a number of issues. Firstly, the additional behaviour required to report information in a self organised manner may negatively impact the original desired behaviour of the swarm. In the case of the arrow formation example, having the robots move into a readable formation might disrupt other position based behaviour. Implementing these information-reporting behaviours might also add significant complexity to the robots behavioural code, further increasing the difficulty of the already complex task of implementing a swarm behaviour. Furthermore, from a debugging perspective, this kind of high level, aggregate data reporting does not allow access to the low level internal state information that can be essential to isolating the cause of an observed issue. The communication infrastructure requirements of a system that allows a large number of robots to communicate individually with the host are a concern, however in practice robot swarms tend to be comprised of numbers of robots which are not un-manageable given modern networking technology. Standard WiFi networks are

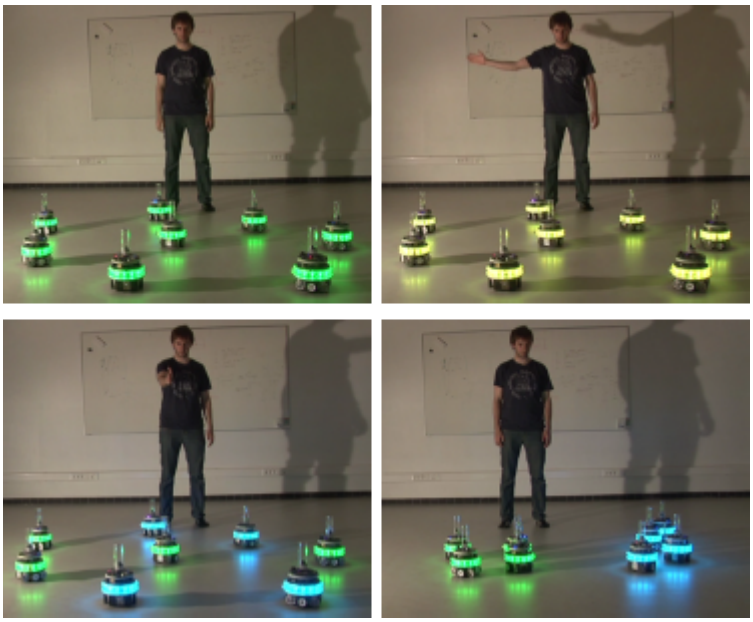


FIGURE 2.1: Self-organised, colour-based feedback during a selection and grouping task. [10]

capable of supporting relatively high numbers of connected devices, whilst developments in mesh-network technologies driven by the emerging Internet of Things sector have made it possible for networks of very large numbers of devices to be achieved with relatively modest hardware [MESH REFERENCE]. In addition, the complexity increase due to greater networking and communication requirements is introduced as a known quantity; these technologies already exist and have standardised implementations, whereas implementing additional swarm behaviours to report information adds unknown complexity, as a reliable, formalised method for developing swarm behaviours is still an open problem. Finally the authors assertion that the volume of data received from a swarm of individually reporting robots would overwhelm an operator is potentially true, however a centralised host receiving all of the reported data is far better positioned to process and condense this information into a form that the user can handle, without sacrificing on detail. This can also provide the user with access to variable levels of information, giving them greater control overall. In summary the concept of self-organised feedback as presented in [10] shows promise for situations where relatively simple, high level information is required to be directly apparent to a system user. However in practice the added complexity at the behavioural level, and the potential to interfere with other desired behaviour, may well outweigh the benefits of reduced communications complexity and infrastructure. By creating a generalised platform for swarm robot information reporting, this project could establish communications infrastructure requirements as a known, fixed quantity, which can potentially be re-used across multiple swarms.

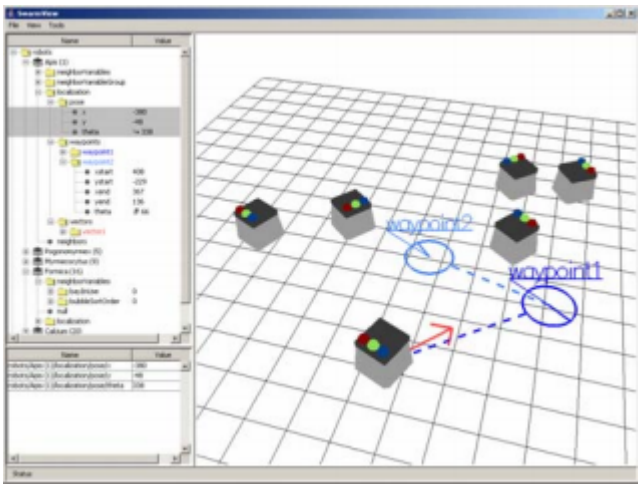


FIGURE 2.2: A screenshot of the *SwarmCraft* GUI, developed by McLurkin et al. [11]

McLurkin et al. [11] discuss the practical considerations of interfacing a human operator with their large swarm of 112 robots. The relevant portion of this work discusses the use of "*utility software for centralized development and debugging*". The authors note the success of graphical interfaces based on *Real Time Strategy* (RTS) video games for centralized data collection and display. Their *SwarmCraft* graphical user interface (GUI), shown in figure 2.2, can display information received regarding individuals, groups, or the whole swarm in a number of graphical forms. The authors note that one potential issue is keeping the code for displaying the data and the code on the robot side in sync, such that the GUI can always correctly display the received data. The idea of a standardised data transfer format investigated by this project could provide a potential solution to this issue. The authors go on to discuss the problem of using monitors to display the robots' internal state information when debugging group behaviours, as a user must continuously switch between watching the swarm and looking at the monitor for detailed information. The authors use a 'global output' system of LEDs and sounds to solve this issue. This output is however inherently limited by the amount of information that can be represented using the small number of LEDs, and the quality of information acquired from a swarm of robots all generating sound at once. This approach also requires specific hardware to be present on each of the robots, such as the LEDs and a speaker. Augmented reality techniques therefore offer a potentially more powerful solution; by superimposing the information that would traditionally be found by looking at the monitor onto a view of the robots themselves, the user can access all available information without looking away from the swarms activity. Furthermore the requirement for the global output hardware is removed, and the user does not need to learn how to correctly interpret these somewhat cryptic output systems, as an AR solution could display information in any form necessary, including text.

2.4 Robotics Debugging

Collet and MacDonald [12] describe in detail the difficulties in debugging robotics systems. The authors identify that the difficulties in developing and debugging robotics applications when compared to traditional software arise from either the environment of the robot - which will often be "*uncontrolled*" and "*dynamic*" - or from the mobile nature of the robot. Because the environment a given robot operates in is a real world space, the level of control that can be exerted over it by the researcher or operator is inherently limited [12]. The environment may therefore change over time, exhibit imperfections, and include other time-varying elements. A robot is a physical actor and will likely experience dynamic change in its sensor readings and its relationship to the environment over time. This is especially true for mobile robots, whose position and orientation will change over time. The behaviour of the robot often largely depends on these highly variable factors, and therefore replicating a given behaviour exactly becomes almost impossible. The authors go on to state that difficulties in debugging often arise from "*the programmer's lack of understanding of the robot's world view* [12]". It can therefore be extrapolated that for a multi robot system such as a robot swarm this problem would be exacerbated. Each robot will have its own perception of the environment, which will differ based on differences in the robots positions and orientations as well as variations in the instrumentation of each robot. For a multi-robot system the programmer is required to have an understanding of not just one but multiple world views, adding yet more potential for error and inaccuracy, and further obscuring bugs or behavioural issues that the programmer is trying to diagnose. This work [12] suggests that developers will need specific tools which enhance their understanding of the robots' world view in order to develop effectively for swarm robotic platforms. This need forms the mandate for the majority this project. Collet and MacDonald [12] go on to discuss the use of augmented reality to satisfy this need, and present an augmented reality based software tool for this purpose, which is discussed in Section 2.5.

Gumbley and MacDonald [13] identify one of the core issues in debugging robotic software using traditional debugging software to be the assumption of a "deterministic, suspendable environment". This assumption rarely holds true for robots, due to their existence in a real world environment. The authors note that traditional debugging concepts such as breakpoints pause code execution, but cannot for obvious reasons also pause a robot's environment. This allows the robots environment to change whilst execution is paused, and therefore affecting the robot's behaviour. In the case where a breakpoint has been inserted to try to isolate the cause of a previously observed fault, this change in behaviour may also stop the fault from occurring. The authors [13] go on to consider a number of possible methods by which a developer can obtain information without pausing execution, including live data extraction which is the method chosen in this project. They note that adding code

to extract information without pausing execution has the potential to affect robot behaviour. This is a salient point, and should be considered when implementing the robot-side portion of this project, where care must be taken to minimize the affect data reporting has on the execution of the robot's actual behaviour. Issues of this nature could be caused by the data reporting code taking too long to execute, thus disrupting robot behaviour. Keeping the data reporting code lightweight and efficient should therefore be a priority.

2.5 AR and Robotics

Augmented Reality is the term used to describe the process of super-imposing 'virtual' objects and elements, in the form of computer generated graphics, on to a live video image of the real world [14]. The general aim of any augmented reality system is to give the impression that the virtual objects being rendered are actually situated within the users real environment, and to allow the user to interact with them as such. True AR is sometimes defined as requiring the use of a Head-Mounted Display, as this leads to the much more immersive qualities of Virtual Reality, from which AR is derived. However this is not a universal requirement, and in one of the most widely cited surveys of the topic, Azuma [14] provides a definition which states only three requirements of an AR system; the combination of virtual and real content, real time interaction, and 3D rendering. Figure 2.3 shows the example image used in Azuma's [14] survey. It includes a view of a real table, augmented by virtual furniture.

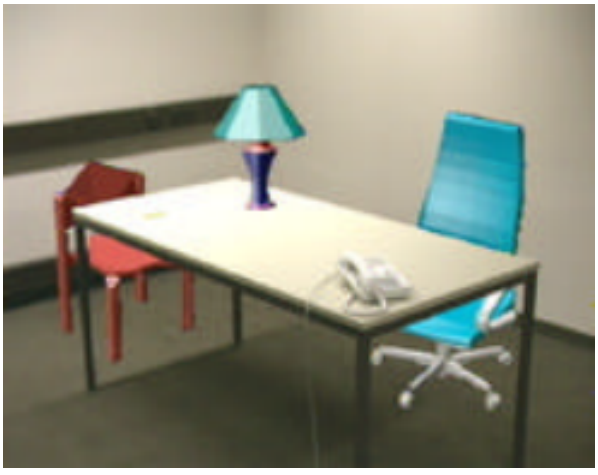


FIGURE 2.3: A simple example of an AR lamp and chairs augmenting a real table. [14]

Augmented reality presents a powerful tool for debugging robots as it allows information gathered by a robot about an environment to be superimposed onto a real world view of that environment, such that discrepancies and inconsistencies

FIGURE 2.4: The Microsoft HoloLens head mounted disaplay.

between this information and the real world become inherently obvious. More generally augmented reality can be used to create a shared space between a human and a robot. Even as early as 1997, Azuma [14] notes that augmented reality has a potential application in robot path planning. One limitation of Azuma's survey is its age, as the two decades since have seen rapid development in computing power and computer graphics, as well as augmented reality hardware, with consumer head mounted displays such as the *Microsoft HoloLens* beginning to emerge into use within real world applications in the late 2010s [15], shown in figure 2.4. A more recent survey of the topic from 2014 [16] agrees with many of Azuma's earlier definitions, but provides many examples of new applications of AR technologies. The author notes the power of AR as a new paradigm for human computer interaction, and the wealth of potential application areas, including robotics.

Milgram et al. [17] discuss the different communication formats used to interface between humans and robots, grouping them into "*continuous*" and "*discrete*" formats. For any communication involving a spatial or temporal component, the process of converting to and from a discrete format in order to transmit this information is an unnecessary burden. Both humans and robots use the continuous spatial dimensions, and humans have an inherent, instinctive understanding of physical things expressed in three dimensions. The authors [17] therefore identify that augmented reality provides an excellent means of supporting the communication of spatial information. Their paper focuses on the combination of stereoscopic displays and computer generated graphics to allow for more intuitive control of robotic systems. In the case of this project the concept is reversed; robots reporting spatial information for validation by a user should do so in a format which is inherently continuous such as a graphical visualisation in AR, rather than one that is discrete such as text-based numerical output. This should in theory reduce the time required for a human to process the information. The authors note [17] that the ideal system

utilises both discrete and continuous formats where appropriate to best communicate the required information, and is ergonomically designed to allow the user to make use of both easily and intuitively.

Like much of the literature surveyed, this paper [17] is focused primarily on robot control, rather than observation and monitoring. In spite of this much of the content of the paper remains applicable. Since the paper was written, just over twenty five years ago, major advancements have been made in virtually every area mentioned, including the quality and precision of robotic systems, their cognitive, perceptive and decision making abilities, augmented reality technologies and robotic autonomy. Because of this, some of the contents of the paper have fallen out of date. Specifically, the assumption that robots lack the level of autonomy required to survey their environment and then form and execute a series of steps to carry out a relatively high level task, such as "find and go to object Q"[17], is no longer necessarily true. A number of modern robots possess sufficient sensing capabilities, processing power and cognitive programming to perform such tasks based on high level commands. This does not however de-value the AR methods discussed within, and given the increased complexity and sensing capabilities of modern robots, AR based methods of interaction actually have more potential than ever. The paper however makes no mention of the potential for an AR system to report data from a robot's sensors visually, which may be attributed to the technological limitations of the time rather than an oversight by the authors.

Collet and MacDonald [12] suggest that augmented reality tools can address and mitigate some of the robotics debugging issues discussed in section 2.4 by superimposing graphical representations of the robot's understanding of the environment on top of a live view of the environment itself [12]. Hence the programmer is able to see how the robot has interpreted the environment, and identify inconsistencies. The authors describe the image of the real world environment as the "*ground truth*" against which the robot's view can be compared and contrasted [12]. Figure 2.5 shows a visual example of this technique, where the data received from the robot's sonar sensors is converted to spatially situated 3D shapes and superimposed over the live image, and can therefore be verified visually by the user.

The application developed during this project closely follows this paradigm; allowing the user to identify bugs by comparing the robot's knowledge of its environment and its decision making factors (collectively referred to as its state) with a view of the environment, in real time. The application aims to apply this concept specifically to swarm robotics systems, and therefore must allow the user to compare the states of multiple robots with the environment simultaneously. From the perspective of each robot in the swarm, the other robots will form part of the environment, therefore the application must take this into account when displaying the information. Because of the large increase in information from a single-robot system to a multi-robot one, it becomes important that the application provide a way for the

FIGURE 2.5: Sonar sensor data visualisation in Collet and MacDonald's system [12].

user to filter what information is displayed, allowing them to focus on the primary aspect under test. Filtering also allows the user to compare and contrast specific robots against one another by filtering out information related to other robots, or by displaying in more detail information related to the robots of interest.

2.6 Similar Work

Ghiringhelli et al. [18] present a system for augmenting a video feed of an environment containing a number of robots with real time information obtained from each of the robots. This is similar in concept to the system described by Collet and MacDonald [12], but is designed specifically to target a multi-robot system. The authors identify the ability to overlay spatial information exposed by the robots on to the video feed in real time, in the form of situated graphical representations, as the most important debugging feature of the system. Figure 2.6 shows a spatially situated overlay of data exposed by robot thirty two, in the authors system [18]. A viewer is able to immediately verify the validity of the robot's world view from this image by comparing the blue overlay to the image beneath. Each robot features a coloured LED blinking a unique coded pattern to enable tracking, and the system uses homography techniques to map between the robots' frame of reference and the camera's. The project proposed in this report intends to use a simpler approach, with position and orientation tracking achieved through the use of the AruCo [19] marker-based tracking system, and a birds-eye view position for the camera to simplify mapping by effectively reducing the space to a 2D approximation. The tracking

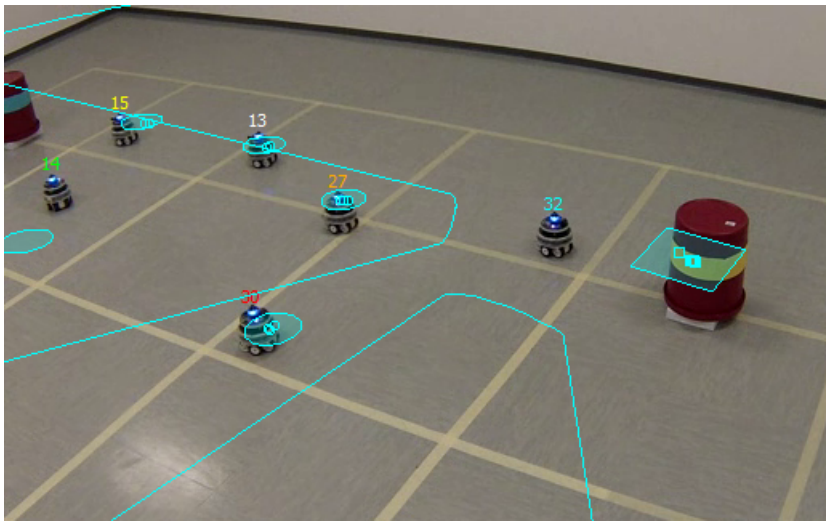


FIGURE 2.6: Example of spatially situated data overlayed on a live image in [18].

LEDs are active components, and therefore require specific hardware to be available on the robots in order for them to be tracked by the system. This limits its generalisability. In contrast, using a passive, marker based technique means that no specific hardware is required, and the tracking could in theory be performed on any robot to which a marker can be attached.

Daily et al. [20] present a technique for retrieving information from a swarm of robots, and displaying this to a user in a "world-embedded" manner using augmented reality techniques. One of the stated aims of this work was the use of minimal communication bandwidth. The authors present [20] an infra-red (IR) LED based solution for both indicating robot position, and communicating a small amount of information, in the form of coded pulse sequences, simultaneously. The head mounted display (HMD) worn by the user decodes this information and superimposes a graphical representation on top of the users view of the robots using a see through optical display. Communicating via IR LED directly to the user's HMD removes the need for communication infrastructure, however it requires the user to have a direct line-of-sight to the robots in order to receive information. The data rate that can be achieved via this method is also heavily limited. However in the use case demonstrated by the authors [20], involving the display of a gradient in the direction of a target known to the swarm, the system is effective. The hardware requirements for this system are quite specific, and unlikely to be available on most robotics platforms, however the power of this kind of spatially situated or world-embedded data visualisation is clear.

2.7 Summary

The system developed during this project seeks to combine a number of existing ideas and techniques used previously into a more generalised,

Chapter 3

Project Plan

This project was completed between Monday 16th January and Thursday 18th May, 2017. A well defined breakdown of the tasks required to complete this project, and an organised plan for completing these tasks, was instrumental in ensuring that the project could be completed in the available time. This chapter gives details of this work breakdown, and the timing considerations. It also includes details of a number of potential risks identified at the start of the project, and information on how these were mitigated where possible. It should be noted that a significant majority of work involved in this project was software development based, and - as with almost all modern software development - accurately predicting the time required to implement every piece of code was a virtually impossible task. The development process inevitably leads to the discovery of unforeseen issues and a deeper understanding of the problem constraints, which in turn requires the allocation of project time to be re-evaluated. Hence wherever possible an '*agile*' methodology and approach was employed. This included frequent re-assessment of the remaining work and the feasibility of individual features. The agile methodology, and its application within this project, is discussed in more detail in section 3.4.

3.1 Work Breakdown

At the start of the project the software related work, including both development and testing stages, was divided into logical tasks. These tasks are shown in tables 3.1 and 3.2 respectively. The time required to complete each task was estimated based on prior experience of software development work, and these approximate timings are listed in the tables. Other tasks, not related specifically to the software development, are listed in table 3.3.

TABLE 3.1: Development tasks.

Task	**Objective**	**Approximate Time**
Read and Understand Existing Code	To understand existing code related to the tracking camera and networking on the e-pucks.	14 Days (Alongside other development)
Establish Development Environment and Toolchain	To enable organised development by establishing a tool set and workflow.	3 Days
Learn to Re-Program e-puck Robots	To understand the cross compilation process for the e-pucks.	2 Days
Outline Software Architecture	To design a coherent code structure in order for code to remain organised and modular.	2 Days
Design General User Interface	To create a high level design of the basic UI and implement a skeleton framework of this UI.	3 Days
Incorporate Tracking Camera Code	To incorporate existing low-level code for acquiring images from the tracking camera and performing tag detection.	2 Days
Implement Tracking Camera Controller	To implement code to create a layer of abstraction between the application code and the tracking code.	2 Days
Implement Wireless Data Receive	To implement code to allow the application to receive data wirelessly.	3 Days
Determine Robot Data Types	To establish an initial set of data types that will be supported by default, and a packet format for these.	2 Days
Design and Implement Data Model	To design the back end data model of the application and implement it in code.	6 Days
Implement Mapping Received Data to Model	To implement code to store received robot and tracking data in the application data model.	3 days

Implement Basic Visualiser	To implement code for displaying the video feed and augmenting it with basic geometric primitives.	5 Days
Design UI Data Representation	To establish a design for the representation of the different data types.	2 Days
Implement Graphical and Textual Data Visualisation	To implement code to convert the data in the data model into relevant visualisations.	10 Days
Implement Data Visualisation Filtering	To implement code to allow the user to filter out unnecessary visualisation elements.	5 Days
Implement Robot Data Comparison	To implement code to allow the user to compare the data of specific robots.	3 Days

TABLE 3.2: Testing tasks.

Task	**Objective**	**Approximate Time**
Continuous Integration Testing	To continually test newly implemented features with the system as a whole during the implementation process.	Throughout development
Manual Verification Testing	To verify the correct operation of the software through manual testing. Specific focus on the UI.	10 Days
Verification Fixes and Changes	To make the necessary changes to correct issues identified in the verification testing.	5 Days
Final Fixes and Changes	Some leeway time is available to make any final changes or fixes based on the results of the user evaluation sessions.	Remaining time

TABLE 3.3: Other tasks.

Task	**Objective**
Initial Report	To produce an initial report in the early stages of the project, outlining the preliminary research completed and the project plan at this stage.
Create Initial User Survey	To create a survey to be answered by potential users of the system such as robotics researchers to gauge interest levels for the proposed system and specific individual features.
Distribute Initial User Survey and Collate Results	Distribute the survey and collate and analyse the responses.
Create a System Evaluation and User Testing Plan	To devise a plan for evaluating the effectiveness of the system including a detailed description of the user testing procedure.
User Evaluation Sessions	To evaluate the system by allowing a number of users to use it in a structured evaluation session.

3.2 Timing and Plan

Having estimated the time required for each development task, a plan for completing the project within the available time frame was devised, and can be found as a Gantt chart in appendix **??**. The plan comprised three main phases; a preliminary phase of research and design work, the main implementation phase, and a final testing and evaluation phase. Where possible, tasks were organised to leave weekend days free, in an attempt to provide a more manageable schedule, but also to allow some slippage time each week for any overrunning tasks.

3.3 Risk Analysis and Mitigation

Engineering projects of are kinds are subject to wide range of risks which may prevent their successful completion. A common technique to reduce the chances of an unsuccessful project is to analyse the potential risks involved in a project before work begins, and attempt to mitigate these risks wherever possible. For this project the following risks where identified, and mitigation steps taken.

1. Failure to complete the work in the available time.

The primary source of risk within this project was time. Software development is an inherently time consuming process, with most implementation tasks suffering from a degree of uncertainty, and the time frame for the project was relatively short. Therefore the major risk to be considered was the potential features to not be implemented due to a lack of available time, and therefore for the system to fall short of its stated aims and objectives. A number of steps were taken to mitigate this risk. Firstly, the features of the system required to satisfy the core aims were assessed, and the task schedule was organised such that a 'minimum viable product' (MVP) would be completed as quickly as possible. The MVP describes the smallest possible set of features which can be implemented such that the system still satisfies its core objectives. The second step taken to mitigate the time-risk was the inclusion of slippage time in the project plan. This extra time was worked into the plan to absorb any overrun in individual, reducing the chances of a single overrunning task having a knock-on effect on the rest of the plan. This slippage time was introduced in two places; by arranging tasks to leave weekend days free, as previously discussed, but also by planning for a flexible testing phase, where time previously allocated for testing could be re-allocated for development if this was deemed necessary. The knock-on affect of reduced testing was determined to be an undesirable but acceptable consequence.

2. Loss of development computer due to a hardware issue.

Another source of potential risk in this project was the development hardware. The majority of the development work was carried out on a personal laptop, running a Linux operating system in a virtual environment. Any damage to this computer could have lead to the loss of important code, and interrupted the development process while a new machine was found. In order to mitigate the potential risk of code loss all code was stored in an online repository, and the latest code was uploaded to this repository frequently. The development environment required to compile the project code was also set up on the tracking server which would be running the final application. This was done so that the code could be tested on the target environment during development, but this also partially mitigated the timing risk involved in the loss of the main development machine, as work could be continued temporarily on the tracking server while a replacement machine was found and set up.

3. Loss of key system component due to a hardware issue.

All electronic hardware is susceptible to breakage, faults and malfunctions. The system proposed by this project includes a number of hardware elements working in tandem, and the loss of any single component could disrupt the implementation and testing phases of the project until a replacement could be found. In the case of the robots this risk was mitigated by the fact that a reasonable number of robots were available, and a broken or malfunctioning robot could be replaced without much

disruption. This was especially true if the Linux extension board was still operating correctly, as this could easily be moved to another standard e-puck. Mitigation by redundancy in this manner was however not an option for some of the larger, more expensive components of the system. The YRL possessed only one tracking camera, so should it become faulty or damaged this would disrupt the project greatly whilst it was fixed or replaced. However, due to the fact that the tracking camera required no moving parts, was not moved during the course of the project, and had shown no signs of unreliability in the past, the chances of a fault occurring or the camera being damaged were considered to be quite low. Therefore, although this was a potentially high-impact risk, its low-probability meant that it could be accepted without mitigation. The tracking server itself was considered more likely to suffer a fault, and this risk was partially mitigated by the fact that another server was available in the YRL that could potentially have served as a replacement.

4. Significant, un-addressed bugs or issues in the software.

Bugs are a risk inherent in all software development, and if un-detected or un-addressed can cause the 'finished' software to be unstable or functionally flawed. In order to mitigate the chances of a major bug going undetected a thorough testing phase was included in the project plan. During this time testing methodologies were applied to the software in an attempt to detect and identify any bugs in the software. These could then be fixed, or if a fix was not possible, mitigated in some way.

3.4 Application of Agile Methodologies

A number of the key principles of agile development methodology were embraced during this project. The first was the idea that functional, working software should always be a priority, and should be the primary measure of progress. The software for this project was therefore developed incrementally, and was verified to still be in a usable state after the addition of each new element or feature. This also tied into the MVP approach discussed previously, as after each incremental addition the software could undergo a cursory evaluation to determine if it yet satisfied the stated aims.

The second agile development principle that guided this project was the idea that reacting to changes in requirements or changes in the feasibility of features is more important than strict adherence to a plan. Whenever an issue arose that changed the estimated time required to implement, or brought into question the feasibility of a specific feature, the necessity of implementing that feature was reconsidered in light of the new information. Features that were determined to be likely to require too much time when compared to their value were discarded in favour of other key features that could be realised more quickly. In practice relatively few issues arose

during the development of this project, and most features were able to implemented as planned, within the original estimated time frames.

One further principle from the agile development methodology that also had an effect on the project was the principle of customer collaboration and continual requirement capture. Although this project did not have a customer in the traditional sense, the researchers within the YRL who responded to an initial survey regarding the software acted as potential customers, guiding the design and implementation of the system with their input. The priority of certain features were adjusted following their comments, with the hope that this would lead to a more useful and practical implementation of the system.

Chapter 4

Statement of Ethics

The ethics considerations affecting this project were minimal, as the work did not involve other humans, animals or potentially unethical practice, and the system does not have direct applicability to military or defence work, or other potentially unethical uses. The system has some potential for indirect use in military or defence work - for example in debugging a military swarm system - however this is an extremely unlikely scenario, and the potential is no greater than for any other robotics software tool. The evaluation process for the project did involve human participants and data collection, and the steps taken to ensure this participation was ethically sound are discussed in section 4.1.

4.1 Human Participant Consideration

4.1.1 Initial User Survey

The initial user survey was a short questionnaire created to gauge the interest of potential users in the system as a whole and individual features. Participants provided responses voluntarily, and the responses were anonymous by default. Participants were optionally given the chance to add their name and email address, so that they might be contacted regarding the later user testing if they were interested. This data was stored in a password protected fashion. This identifying data is not presented as part of this report, and the response data is presented in an aggregated fashion, ensuring anonymity.

4.1.2 User Testing and Evaluation Sessions

The final user testing and evaluation involved an observed testing session, where participants used the system in context whilst under observation, and a final questionnaire. Participants took part voluntarily, and consented to both the observation process and to the use of their questionnaire responses in an anonymous fashion. The data collected does not include any personal or private information, and is purely related to each participants experience and opinion of the system. Once again the questionnaire data is presented in an aggregated fashion to preserve anonymity. Anecdotal information obtained through observation does not include any identifying information about the participant, and is also fully anonymous.

Chapter 5

E-Puck Robot Platform

5.1 Overview

The e-puck robotic platform, created by a team at the *Ecole Polytechnique Fédérale de Lausanne* [2], is a small, relatively inexpensive, multi-purpose robotic platform designed for education and research pursuits regarding robotics and multi-robot systems. The platform is widely used in swarm robotics research, featuring in a number of publications. The e-puck was chosen as the first target platform for this system for a number of reasons. Firstly this was one of the platforms available in suitable numbers in the York Robotics Laboratory (YRL) at the University of York, where the practical work for this project was carried out. Secondly the platform's wide use in swarm robotics research helps to show the broad applicability of the system, and better demonstrates its value when compared to a less widely used or bespoke platform. Finally the platform's extensible design meant that it could be equipped with a Linux extension board, a configuration frequently used at the YRL. This made the use of WiFi for wireless data transfer feasible, and was a large part of the reason for the e-pucks choice. This section provides details of the e-puck robot's hardware, including processor, sensors and actuators, as well as details of the configuration used for this project, including the Linux extension board. Figure 5.1 shows the e-puck robot, equipped with the Linux extension board on top.

5.2 Processor

The e-puck features a dsPIC 30F6014A microprocessor, designed and manufactured by Microchip Technology Inc. This is a general purpose 16-bit CPU, with relatively low performance by modern standards. The processor features 68 I/O pins, which are connected to the e-pucks various peripherals. Due to the use of theLinux extension board discussed in section 5.5, which features a more powerful ARM processor, the primary use of the PIC processor in the e-puck configuration for this project is to interface with the e-puck's hardware. This involves receiving commands from the

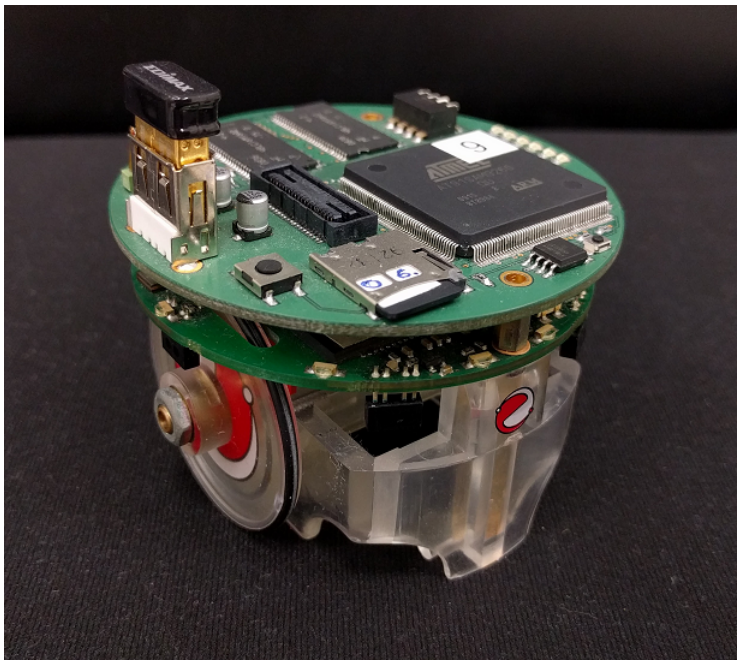


FIGURE 5.1: The e-puck robot, equipped with a Linux extension board.

ARM processor and sending appropriate control signals to the robot's actuators, as well as reading the robot's sensors and passing the retrieved sensor data back to the ARM processor. Prior to the start of this project members of the YRL had already developed a library of low level code allowing the PIC to function in the hardware interface role as described, controlled through the UART serial port. This firmware code was used as-is on the e-puck PIC controllers throughout the project.

5.3 Actuators

The e-puck robot features two wheels, independently actuated by two step motors. The wheels have a diameter of approximately 41mm. The motors can rotate the wheels at an approximate maximum speed of 1000 steps per second in either direction, where 1000 steps is one full revolution. The robot also features a ring of 8 red LEDs around the edge of the main circuit board.

5.4 Sensors

The e-puck robot features a number of different sensor sets, of which only some are used in this project. A set of 8 IR proximity sensors are arranged around the circumference of the robot, with four positioned on the forward hemisphere, two positioned at right angles to the forward direction (one on either side) and two more positioned

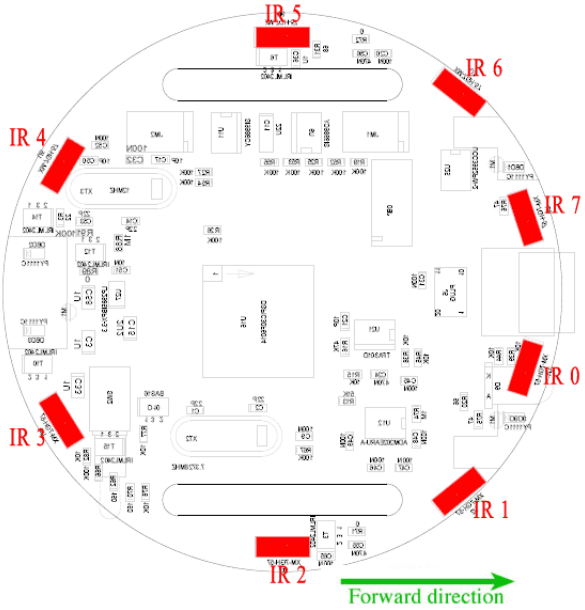


FIGURE 5.2: The layout of the IR sensors on the e-puck robot.

on the backward hemisphere at roughly 45 degrees either side of the backward direction. Figure 5.2 shows this layout. The IR sensors can be used in two modes - active and passive. In active mode the sensor emits an IR pulse and measures the IR strength of the reflection, whereas in passive mode the sensor simply samples the IR strength without emitting a pulse. The passive mode can therefore be used to get a 'background' IR reading, which can be compared to the active reading to improve accuracy. The IR sensor is of particular interest to this project as it is a frequently used tool when working with robots, especially robot swarms.

The robot also features three microphones, a 3 axis accelerometer and a camera. Due to the bandwidth required to use the microphones and camera they were considered a low priority for this project.

5.5 Linux Extension Board

For this project the e-pucks were fitted with an extension board featuring a 32-bit ARM9 processor running a modified Linux operating system [21], developed by Wenguo Liu, and Alan F.T. Winfield. In this configuration the ARM processor, an Atmel AT91, takes charge of the high level robot control logic, as well as any intensive data processing operations. The dsPIC processor is then used to control the low level actuator and sensor control, running in parallel with the ARM processor and communicating via UART. The extension board provides a USB port, and for this project a WiFi adapter was connected to each robot. The controller code running on

each robot could then make use of the standard IP network layer protocol, and the standard transport protocols TCP and UDP.

Chapter 6

Video Tracking System

In order to implement the augmented reality element of the system, and satisfy the related objectives, a live video feed of the swarm was needed. A method for tracking the positions of each individual robot in the swarm within each video frame in this feed was also required, in order to correctly position the graphical overlays. Prior to the start of this project infrastructure had been put in place at the YRL for performing this kind of video-based tracking task, in the form of a machine vision camera placed above an 'arena'. Figure 6.1 shows the arrangement of the machine vision camera used for robot tracking, and the robot arena. Figure 6.2 shows a photo of the robot arena and tracking camera set up at the YRL. Ongoing research being carried out by members of the YRL had already made use of this camera and arena set up, in conjunction with image processing software for tracking fiducial markers within the image, to achieve robot tracking with good accuracy and relatively high frame-rate.

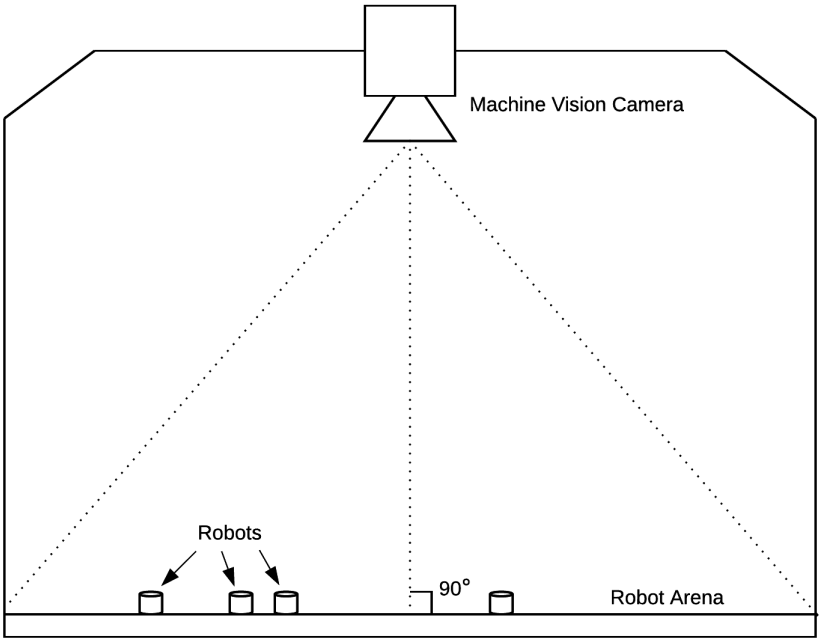


FIGURE 6.1: Arrangement of the tracking camera over the robot arena.

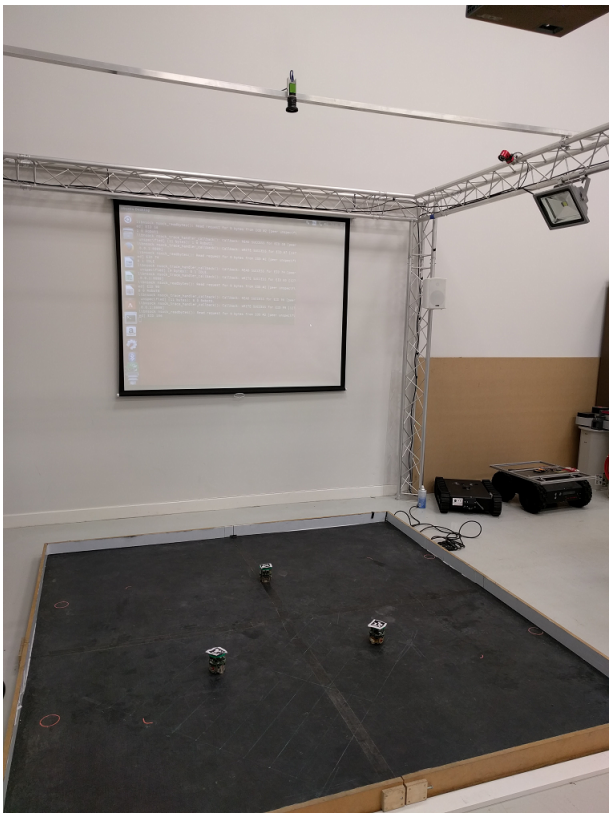


FIGURE 6.2: The robot arena set up at the YRL, with tracking camera visible.

The image processing in these ongoing research efforts was performed using the 'ARuCo[19]' fiducial marker based tracking system, discussed further in section 6.2. It was determined that incorporating this existing infrastructure into this project would be the quickest way to achieve an operational tracking system, allowing work to focus on the novel aspects of this project sooner.

6.1 Camera

The camera used in the aforementioned tracking set-up is a JAI Go 5000C-PGE colour, area-scan camera, shown in figure 6.3. It features a 1-inch, 5-megapixel CMOS sensor, with a maximum resolution of 2560 by 2048 pixels, capable of capturing 22 frames per second at full resolution, and up to 163.5 frames per second with reduced resolution and colour quality. The camera also features a global shutter, meaning it captures the full area of the image simultaneously, as opposed to a rolling shutter which captures portions of the image sequentially. This camera is well suited to marker tracking applications for a number of reasons. The high resolution means that even small markers, or markers that are relatively far from the

FIGURE 6.3: The JAI-Go 5000C-PGE camera [22].

camera will be captured with sufficient detail to be tracked and identified. The relatively high frame-rate means that the tracked positions can be recalculated regularly, and the video will appear smooth. The use of a global shutter avoids issues present in rolling shutter cameras, where a moving marker appears broken due to the time difference between the capture of two areas of the image and the movement of the marker during this time.

The camera is connected to a server rack using Gigabit Ethernet cable, which is required to support the high bandwidth output of the camera due to its resolution and frame rate. This cable also provides power to the camera. A high resolution lens is fitted, with a relatively wide range of possible focal ratios from $f/1.4$ to $f/16$, allowing the camera to be adjusted to work well in a range of light conditions. With the lens included the camera has approximate dimensions of $29 \times 29 \times 100$ mm, and a weight of 246 grams, making it a very compact, lightweight solution.

The image data from the camera is transmitted in accordance with the '*GigE Vision*' interface standard [23]. This standard was first introduced in 2006, and is designed for transmitting video from high-performance industrial cameras over Ethernet networks. The 'Common Vision Blox' (CVB) machine vision programming library is used to map the data from the GigE format to the OpenCV image format, so that it can be processed by an application.

6.2 ARuCo Tracking System

Developed by a team from the Computing and Numerical Analysis Department at Cordoba University in Spain, the ARuCo tag generation and detection system [19] is a powerful fiducial marker creation and tracking tool. It comprises an algorithm for

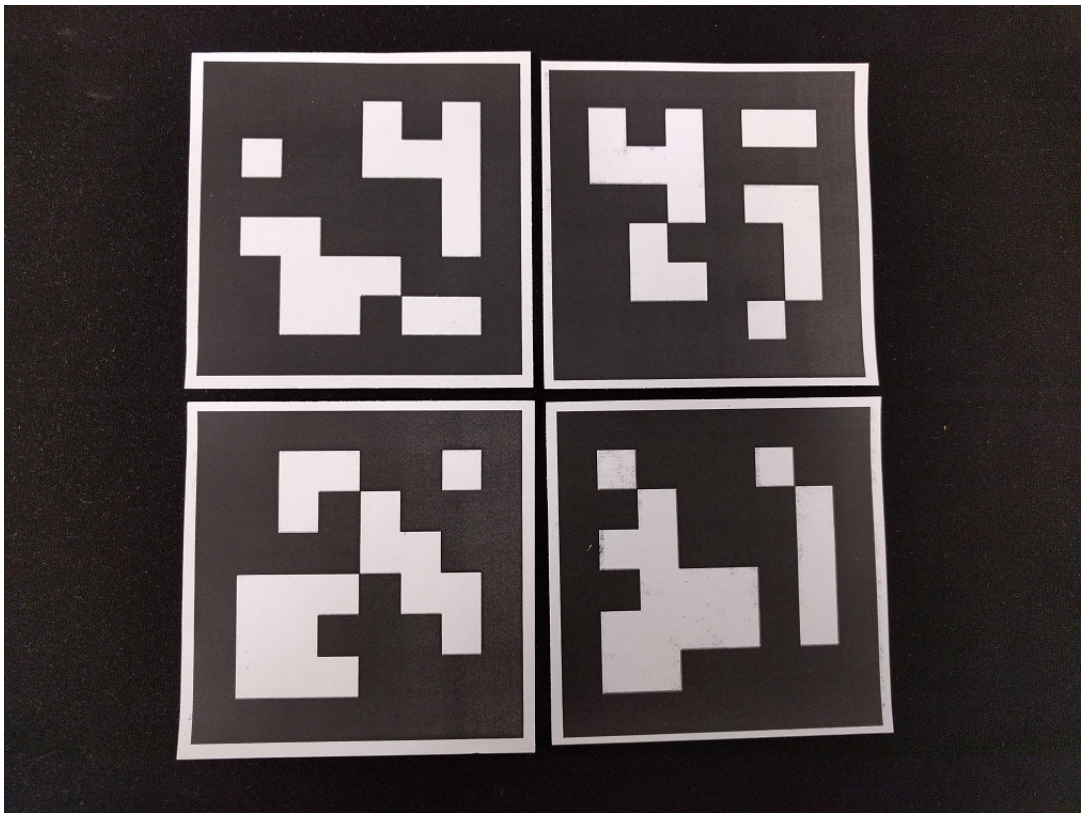


FIGURE 6.4: Four markers generated by the ARuCo fiducial marker generation and detection system.

producing a 'dictionary' of square, black and white, coded markers, and a method for automatically detecting these markers in a given image. These markers can be easily printed using a standard inkjet printer, and attached to surfaces and objects. Figure 6.4 shows four possible marker variations on standard printer paper. The stated applications for the ARuCo system include augmented reality, machine vision, and robot localisation. One of the main benefits of this system over other fiducial marker systems is the execution speed. By first using edge-detection methods to isolate the outlines of potential markers in the image, the system can eliminate a large portion of the image before applying the more complex processing to identify and differentiate individual markers [19]. This makes the algorithm extremely efficient, and therefore makes it possible for the ARuCo system to be run in real time, even with relatively modest computational power. In a conventional use case the orientation of the camera can be calculated based on the positions of the corners of a marker, given that the marker's orientation is known. In this use case the reverse is true, the camera's position is fixed, and therefore the orientation of the robot can be determined based on the position and orientation of the corners of its marker, relative to the camera.

Each of the e-puck robots used in this project were assigned an ID number, and a dictionary of ARuCo markers was generated to match. The markers were affixed to

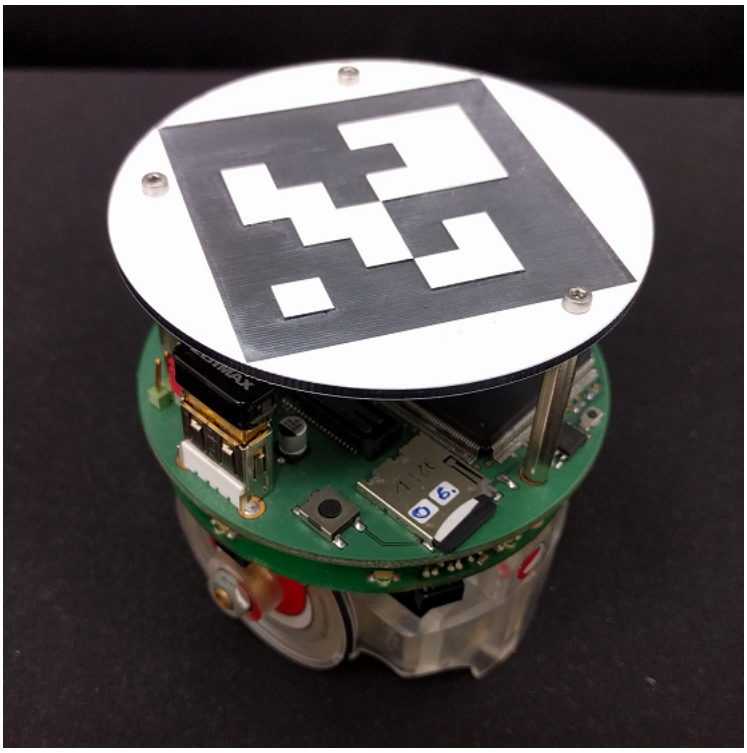


FIGURE 6.5: A laser printed ARuCo marker mounted on top of an e-puck robot.

the top of the robots, oriented to match their forward directions. Figure **??** shows a laser printed ARuCo marker mounted on top of an e-puck robot. Some cursory preliminary tests of the tracking system confirmed that the markers could be accurately and reliably detected in the camera images, at a decent frame rate. Further detail of the integration of the ARuCo marker tracking system into the application developed during this project is given in section 9.6.

Chapter 7

Initial User Survey

7.1 Overview

In order to determine how best to implement the system, and to ensure that the finished system was useful in practice, a survey was carried out at the YRL. The respondents were all actively engaged in robotics work, either in a research capacity or as a technician, and had some experience and understanding of swarm robotics specifically. The survey aimed to determine if a level on interest existed for the proposed system, and then ascertain which specific features were most desired. This would go on to influence design choices and inform priorities during development.

7.2 Data

Question 1: Do you believe that a system for displaying internal robot data for a swarm of robots in real time would be useful when debugging swarm robotics behaviours and/or conducting swarm robotics experiments?

Answer	Votes	Percentage
Yes	4	80
No	1	20
No Opinion	0	0

Question 2: Do you believe that such a system would benefit from the inclusion of an 'augmented reality' component - whereby data retrieved from the robots could be displayed in graphical forms, overlaid on a live video feed of the robots themselves?

Answer	Votes	Percentage
Yes	5	100
No	0	0
No Opinion	0	0

Question 3: Please rate each of the following potential features based on your opinion of their importance or usefulness to the system proposed, on a scale of one to five, where one indicates a feature is not important or useful, and five indicates a feature is very important or useful.

On the questionnaire the values of one to five were qualified as '*Not important or useful*', '*Unlikely to be important or useful*', '*Neutral*', '*Somewhat important or useful*' and '*Very important or useful*' respectively.

Feature	1	2	3	4	5
Real time video feed of the robots and their environment	0	0	0	0	5
Augmentation of the video feed with the position and orientation of each robot	0	0	1	1	3
Augmentation of the video feed with each robots identifier (ID or name)	0	0	0	1	4
Augmentation of the video feed with spatially situated sensor data, represented in a graphical form	0	0	0	2	3
The ability to enable and disable individual elements of the video feed augmentation	0	0	0	2	3
The ability to customise individual elements of the video feed augmentation (colour, size, etc.)	0	0	2	1	2
Displaying the robots internal state and a history of recent state transitions	0	0	0	3	2
Displaying raw sensor data (e.g. IR) in textual format	0	0	0	4	1
Displaying sensor data in plotted graph formats	0	0	1	3	1
Support for displaying unspecified user data in textual form	0	0	1	2	2
The ability to compare two or more specific robots within the swarm	0	1	1	1	2
Logging received data and events to a text or CSV file	0	0	0	1	4

Question 4: Please briefly describe any additional features you believe would be useful, based on your experiences working with swarm robotics systems.

- "macro-level behavioural data on the swarm; e.g., number of robots in different behavioural states (color coded accordingly)."
- "I think the system (already potentially very useful) could be improved/expanded further to add the option of more post-processing/extraction of data. The ability to create video of the over-lay, and perform statistical analysis on the complete run, would begin to turn the system in not only a very useful debugging system but a complete package allow researchers to gather high-quality, publication ready analysis of swarm experiments."

Question 5: Which of the following aspects of the robot data do you believe should be made available by the application to aid in debugging and testing? Tick all that apply. Please add any more you may think of.

Data Type	Votes
Position	4
Orientation / direction	4
Position change over time (recent path tracking)	4
Internal state machine state	3
Internal state transition history	3
IR sensor values	4
Distance between robots	3
Robot ID	4
Other	2

Additional responses:

- "Option to include user-defined data so if a certain controller implements a timer that facilitates a state transition might be useful to see the value of that timer whilst debugging to compare with state transitions"
- "A simple API to add user-defined variables/statuses"

Please add any additional comments you have about the proposed application in relation to your experiences working with swarm robotics systems.

- "A client-server model between back-end (camera) and remote client would potentially be very useful; also the system should be flexible and not reliant on a specific camera/server setup. Would be very interesting to see if it will work on a R-Pi 3 + camera combination, as this would allow for portable tracking setups."
- "All real-time information is useful!"

7.3 Analysis

The positive response to question one indicates a reasonable level of interest in the system amongst those surveyed. The similarly positive response to question two adds more weight to the idea that graphical debugging tools have the potential to be particularly useful in a robotics context, as established during the literature review. The responses to these two questions indicate that interest exists for the system, and that it is worthwhile implementing it. This satisfies the first aim of the survey.

The remainder of the survey focuses on establishing which features are most desirable, and the results were used when considering implementation priorities. The response to question three indicates that the majority of the core features were thought to be potentially useful, especially those related to the video feed and overlay. Tertiary features such as customising the colours and sizes of the overlays showed less interest. This was as expected, as these kinds of features do not aid directly in the debugging process. Considerable interest was expressed in the ability to log data and events, a feature which was not considered a major priority at the outset. Conversely, the ability to compare two robots was the only feature to receive a vote lower than neutral, despite being initially thought a key feature of the system. As a result the implementation of logging was moved up to a main priority, and the comparison feature was reduced to non-essential.

The respondents were then given the chance to optionally suggest additional features in question four. The first of the two responses suggests 'macro-level behavioural data', a concept considered during the project's inception. It was not included in the initial plan or survey partly because at the outset it was not clear what form the feature would take, or whether it would be feasible in the time frame, and partly because it was seen as a feature related more to analysing swarm experiments and results rather than debugging. As a result of this answer displaying macro level swarm data was considered a desirable but not essential feature, to be implemented if time allowed. The second response offers a broader vision for the system as a whole. This report agrees with the observation of the systems potentially to become a complete package for analysing swarm experiments and extracting data, however the majority of the features mentioned are considered beyond the scope of this project. This includes video extraction and post processing of data. These are however considered in chapter 11 regarding future work, as they present significant potential expansions of the system.

Question five attempts to establish which specific data types are most desirable in the system. Note that one respondent did not answer this question at all, leading to the lower vote counts. None of the data elements listed received a significant deficit in votes, suggesting that all the data types listed should be included. Respondents were asked to optionally suggest other data types, and the two responses

received both independently identified user-defined data or 'variables' as a desirable data type. The inclusion of custom user defined data was a planned part of the system from the beginning, and is mentioned in question 3, but these responses confirmed that it should be a priority feature of the system.

The final section gave respondents the opportunity to add any additional comments about the system. The first response notes that designing the system in a way which does not make it 'reliant on a specific camera/server setup' would improve its usability. This thinking matches the stated objective of making the system in a modular way that is easily extensible with different camera set-ups and different robots.

7.4 Issues and Shortcomings

This survey provided a useful tool for gathering a general impression of relevant opinions regarding the project and the system. However it also had a number of issues in its execution which might somewhat diminish the value of the data. Firstly due to the highly specific nature of the desired participants, the sample size is extremely small. Care must therefore be taken in analysing too deeply the results; for example any statistical analysis performed would likely be flawed. Another issue is that the two larger multiple choice questions present a relatively small selection of possible features and data types, based on those already planned or considered for implementation. This report therefore aims to use the results in an indicative, holistic manner to guide implementation, rather than quantitatively define the feature set.

Chapter 8

Design

This section describes the design of the system, and gives details of the reasoning behind some of the design decisions. This design work was done largely prior to implementation, with some elements of the design being re-factored during the implementation phase in adherence to the agile development methodology being followed. The design process was broken down into a number of key areas. Firstly the design of the software architecture, including the breakdown of the different components and the path of data through the system. This served as a road map during the implementation stage. The second key area was the user interface design. This involved determining how the main application should appear to the user, how best to provide the user with access to the various features, and then creating a template of the window and component layout required to achieve this. Graphical representations were also designed for each of the data types to be displayed in the visualiser.

8.1 Software Architecture Design

The guiding principles of the software architecture design were 'Object Oriented Programming' (OOP) practices, and the 'model-view-controller' (MVC) software architecture pattern. OOP [24] is an extremely widespread concept in modern software development theory. It states that code should be organised into units based on individual functionalities, commonly referred to as classes. Each class collects together the data that describes an object and the routines to perform actions with and on that data. A class then acts as a template, and a new instance of the class can be instantiated each time an object of that type is needed. OOP aims to make code easier to understanding and maintain, reduce code duplication, and increase re-usability and modularity. Designing software in an OOP fashion is standard practice for most modern programming tasks, and modern languages are often designed around OOP concepts. C++ was selected as the development language for this application as it offers much of the low level control and efficiency of the C language, whilst also supporting OOP practices natively. The majority of the existing software infrastructure in the YRL had also been implemented in C++, hence following suit would

help with maintainability and integration in the future. Considering the project's requirements for interfacing with low level hardware such as the tracking camera (via a driver) and the robots (via network sockets), and for performing image processing, the speed and low level capabilities of C++ also seemed beneficial. Higher level languages such as Java were considered, as they offered a number of different benefits such as better portability and automatic resource management, but were ultimately deemed less suitable.

The MVC software architecture design pattern is another widespread concept in software development theory. It primarily relates to the programming of applications with user interfaces. The three words that give the pattern its name - model, view and controller - define the three 'layers' into which code components are organised. The model refers to the application's data, and includes all of the information that defines an application's current state. The view refers to the code used to produce the user interface from the data in the model layer. It acts as the method by which the user 'views' the data, thus getting its name. Finally the controller layer acts as the intermediary between the two, retrieving data from the model and processing it if necessary before passing it to the view for display. The controller also responds to input and changes the data in the model accordingly. This includes data input via the user through the view, as well as data from other sources. In the case of this application these other sources include the peripherals, such as the tracking camera, and the robots via the WiFi network. Adhering to an MVC pattern helps to keep code structured and organised, making it easier to understand, maintain and extend. It ensures that state data is not maintained or duplicated within the UI code, and that only one, true copy of the application data exists, stored in the model layer.

With these two principles in mind, the software design process could begin. First the application was broken down into individual components based on the functional specification. The following key areas were identified:

- Code to handle communicating with the camera and retrieving the image data.
- Code to perform the robot tracking.
- Code to handle the networking, including receiving data from the robots.
- Code defining a model to store the robot data in.
- Code to enter new data into the model.
- Code to augment the video feed based on the stored data.
- Code to display the video and augmentations feed to the user.
- Code to respond to user input via the user interface
- Code to store user settings

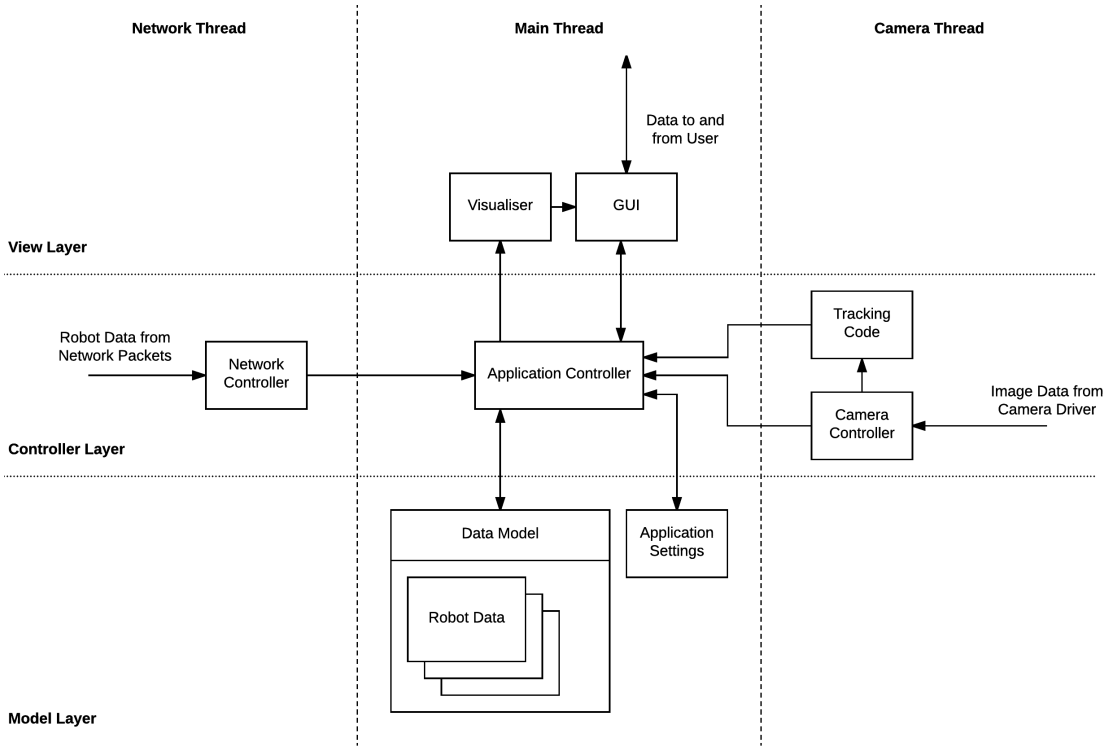


FIGURE 8.1: A diagram of the software architecture design and data flow.

Once separated into functional components, the required code blocks could be organised into a structure, and the data path of the application examined. Figure 8.1 shows this structural arrangement, with boxes for each of the functional objects and arrows indicating data flow. The three layers of the MVC design pattern are shown by the vertical partitioning. The horizontal partitioning is used to show another key design consideration - threading. In order to maximise performance and ensure responsiveness, functionality which has the potential to 'block' execution whilst waiting for a result or response should be run in a separate thread. This application was therefore designed with three threads in mind. The main thread handles the core of the application, including all data model access and GUI operations. The network thread handles communicating with the robots via WiFi. It was anticipated that this networking would involve low level socket code, which meant the potential for blocking socket-read operations, therefore requiring a separate thread. The camera thread handles reading the machine vision camera and performing the tag tracking using the ARuCo library. It was anticipated that the camera read operation could block until the next frame was available in the camera driver's buffer. Tracking the robots in the image using the ARuCo library also had the potential to be CPU intensive, so keeping this off the main thread was considered a potential performance benefit.

As can be seen in figure 8.1, the software architecture is structured around a

central application controller. The data from all other components of the application flows through this central component, which routes it to where it needs to go. It is also responsible for updating the visualiser and GUI with new data when necessary, and acting on the user input signals received through the UI. The other key controller layer components - the network controller and the camera controller - exhibit data flow in only one direction. The design of these components was therefore relatively simple, as they needed only to perform the following loop of tasks:

1. Receive data from external source.
2. Process data as necessary, extracting required information.
3. Notify the application controller of the newly available data.
4. Wait for new data to become available.

The remaining components are more complex, and required greater software design considerations. The design of the data model is discussed in section 8.1.1. The design of the visualiser code is discussed in section 8.1.2.

8.1.1 Data Model

The purpose of the data model component is to store all of the data required to describe the applications current state. This includes all data related to each of the robots being tracked by the system. In order to achieve this in an object oriented manner the data model itself was designed to be comprised of a number of smaller components in a hierarchical structure. The main data model object maintains a collection of smaller objects, each containing the data related to a single robot. These in turn maintain a collection of different data objects relating to the robot's state and sensor data. Figure 8.2 illustrates this hierarchical data model design using the IR sensor data of a single robot as an example.

This follows object oriented programming practices by defining a single class to describe a robot, from which a new robot data object can be instantiated each time the system begins tracking a new robot. When data is received regarding a robot the system is already aware of, it can simply update the correct existing robot data object with the new data.

8.1.2 Visualiser

The purpose of the visualiser component is to generate and display the augmented video feed to the user. The component therefore receives the latest camera image and the most up to date robot data, generating the graphical overlays based on a combination of the robot data and the current settings for each visualisation. It then

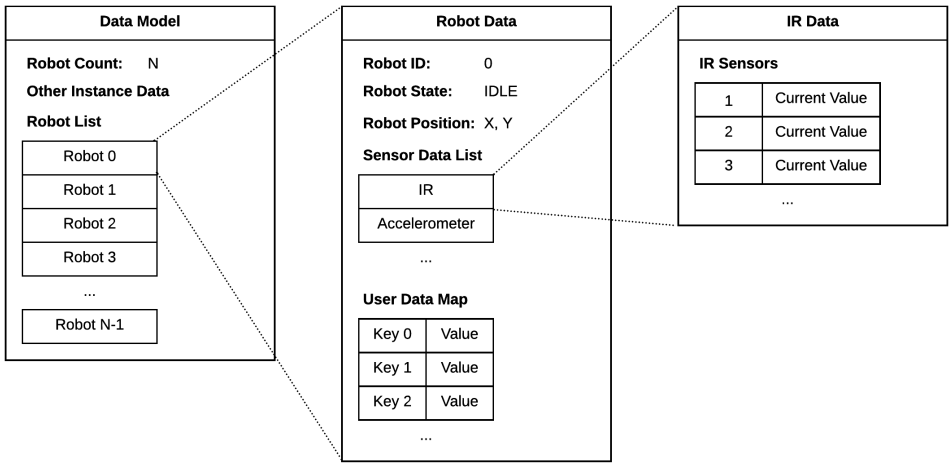


FIGURE 8.2: A diagram of the data model design.

displays the latest image, with the graphical overlays applied, to the user by rendering it as a single image within the appropriate GUI element. This rendering process occurs each time a new video frame is read from the camera, hence both the video and the overlays should update regularly and respond rapidly to changes in the robot data.

In order to further modularise the visualiser code, it was broken down into a number of smaller components. For each type of data visualisation a separate component was defined, which describes how the visualisation for that data type is generated, based on a single robot data object. The main visualiser component could then be designed to generate the graphical overlays iteratively, by stepping through the available set of robot data objects, and for each one step through the different visualisation components, generating the overlay for each combination. The overlays can then be combined and displayed as a single image. Figure 8.3 describes this process diagrammatically.

8.2 User Interface Design

The second main area of consideration during the design phase was the Graphical User Interface (GUI). It was determined that creating a well designed, intuitive interface would be essential to satisfying the objective of providing data in a 'human readable' form. Having real time data would be useless if the user cannot parse the data displayed sufficiently quickly, due to a poorly designed or confusing UI. The first decision made was to try and keep the interface familiar to a computer user, through the use of standard, widely understood user interface elements. There exists a well defined 'language' in computer interface design, using constructs such as

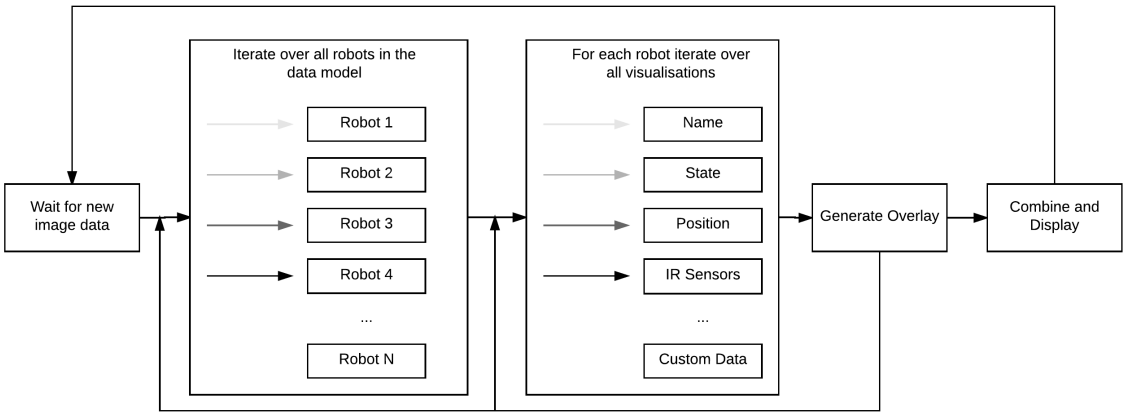


FIGURE 8.3: A diagram describing the design of the visualiser rendering process.

windows, panels, tabs, buttons, text fields and other elements which have well understood functions. It was thought that basing the user interface design on this well established standard would minimize the time for a new user to become accustomed to the system. The next step was determining how many panels would be necessary for the intended functionality to be possible, and how best to lay these out within the window and organise the smaller elements within each panel. Three main panels were determined to be necessary. The first would display the video feed and visual overlay, the key component of the application. A second panel would be used to display a list of the robots known to the system, so that they could be selected without obscuring the visualiser. Finally a third panel would be used to display more detailed information about the selected robot in a number of different tabs. Figure 8.4 shows the basic layout design for these main panels.

The visualiser panel, highlighted in blue, takes primary place in the layout. In order to maximise the readability of the augmented video feed it was determined that this panel should occupy as much space as possible. A number of tabs allow the user to change the focus of the panel, giving access to various settings, including settings for the visualiser and camera. The robot list panel is highlighted in green on the right hand side. This requires less width, as it's main function is to display a list of the known robots, and allow the user to select one. Again tabs within the panel allow the user to access functionality related to the robots, such as settings for the network connection. Finally the data panel is highlighted in yellow at the bottom of the layout. Each of the tabs provide more detailed info on a specific type of data collected about the selected robot, as well as a tab for a general overview, and a console-style log of application events. It was noted that when displaying data in this panel it should be formatted to maximise the use of the available space. This means using the full width of the window and limiting the height to avoid excessive scrolling. The design also includes a toolbar at the top of the window, another

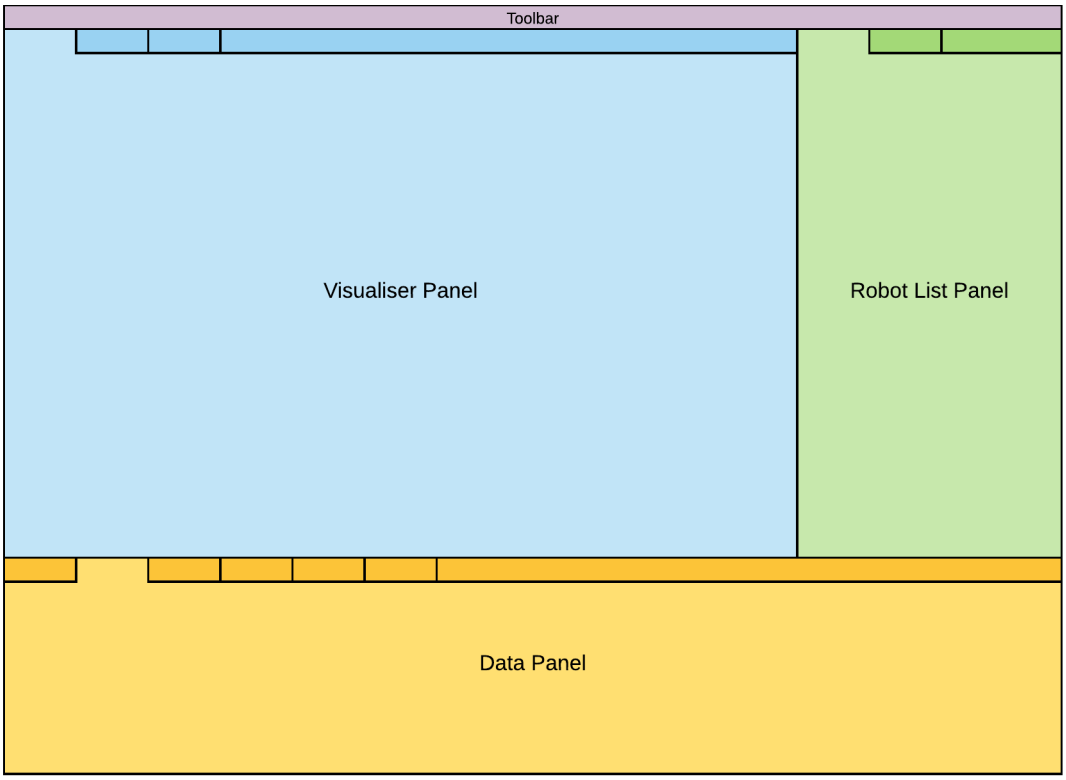


FIGURE 8.4: The design of the basic user interface layout.

standard feature of window-based software applications. The toolbar is provided to allow quick access to useful functionalities.

Figure 8.5 shows how this layout might look when filled with some example content. The robot list identifies the two robots being tracked by the system, and shows that robot 1 is selected. The visualiser component shows an example of how the video image might be augmented with graphical overlays generated from data regarding the two robots. These overlays include indicators for each robots position and orientation, as well as text showing the name and state of the selected robot. The dotted line shows the recent path taken by the robot, which is an example of one potential type of data visualisation. Colour is also used to differentiate the two robots. The data view at the bottom of the window shows some example data for the selected robot, which might be seen on a general 'overview' tab.

Time was also spent creating more detailed designs for some of the individual tabs within the main panels. Each tab within the data panel needed to display a different data type, and therefore required a unique layout and design. Figure 8.6 shows the initial designs for four of the key tabs, using both textual and graphical approaches to data representation. Layout 1 shows the overview tab design, as seen previously. Layout 2 shows the design for the console tab, which displays messages

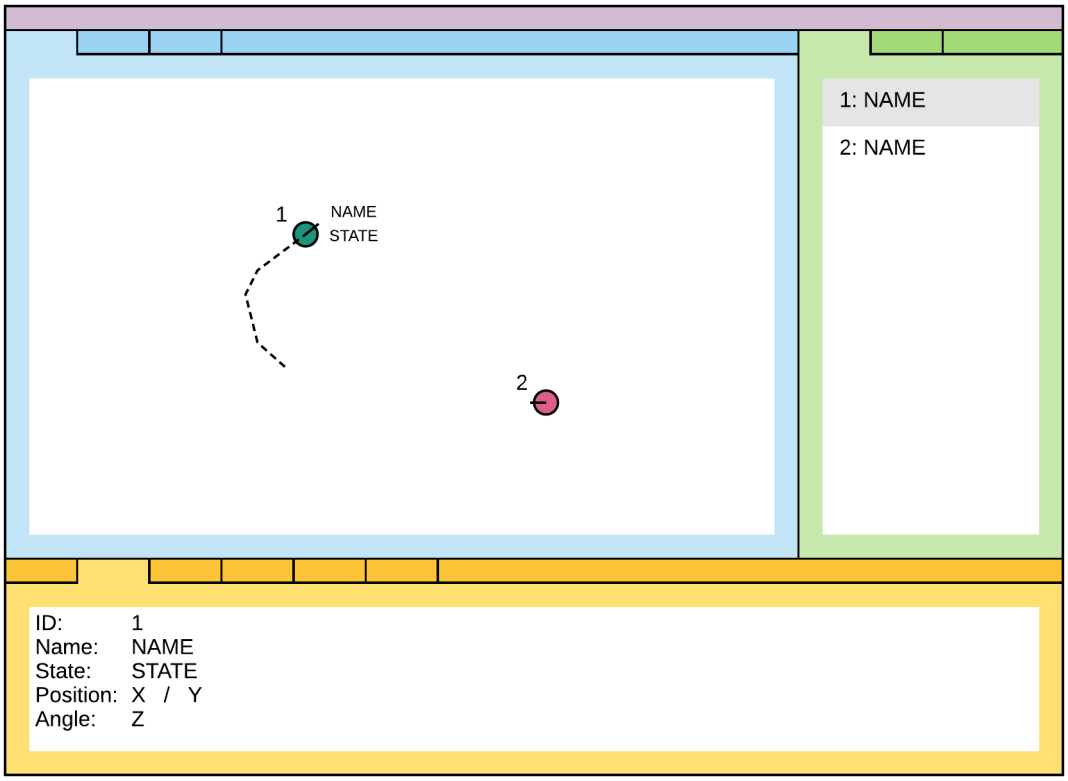


FIGURE 8.5: The basic UI design, filled with some example content.

about the application sequentially. Layout 3 shows the state tab, which offers additional information about the state of the selected robot, beyond simply its current state. The first box displays a list of all of the robot's known states, and the second box displays a list of the robot's recent state transitions, including the original state, the new state and the time the transition occurred. Finally layout 4 shows one possible design for displaying the robot's IR sensor data, using a bar graph to give a relative, comparable impression of each sensors value, and the numerical displays beneath to provide the actual sensor values. Figure 8.7 shows the design for the settings tab of the visualiser panel. The purpose of this tab is to provide access to general settings related to the visualiser, and specific settings for each of the data visualisation types. The actual settings shown in the design are representative. General settings are changed using basic form controls such as tick boxes. Settings for specific visualisations can be changed using extra pop-up windows accessed by double clicking the relevant visualisation in the list. In figure 8.7 the IR data visualisation settings are shown as an example.

As well as designing layouts for some of the more complex individual UI elements, designs were also created for some of the graphical overlays that would be

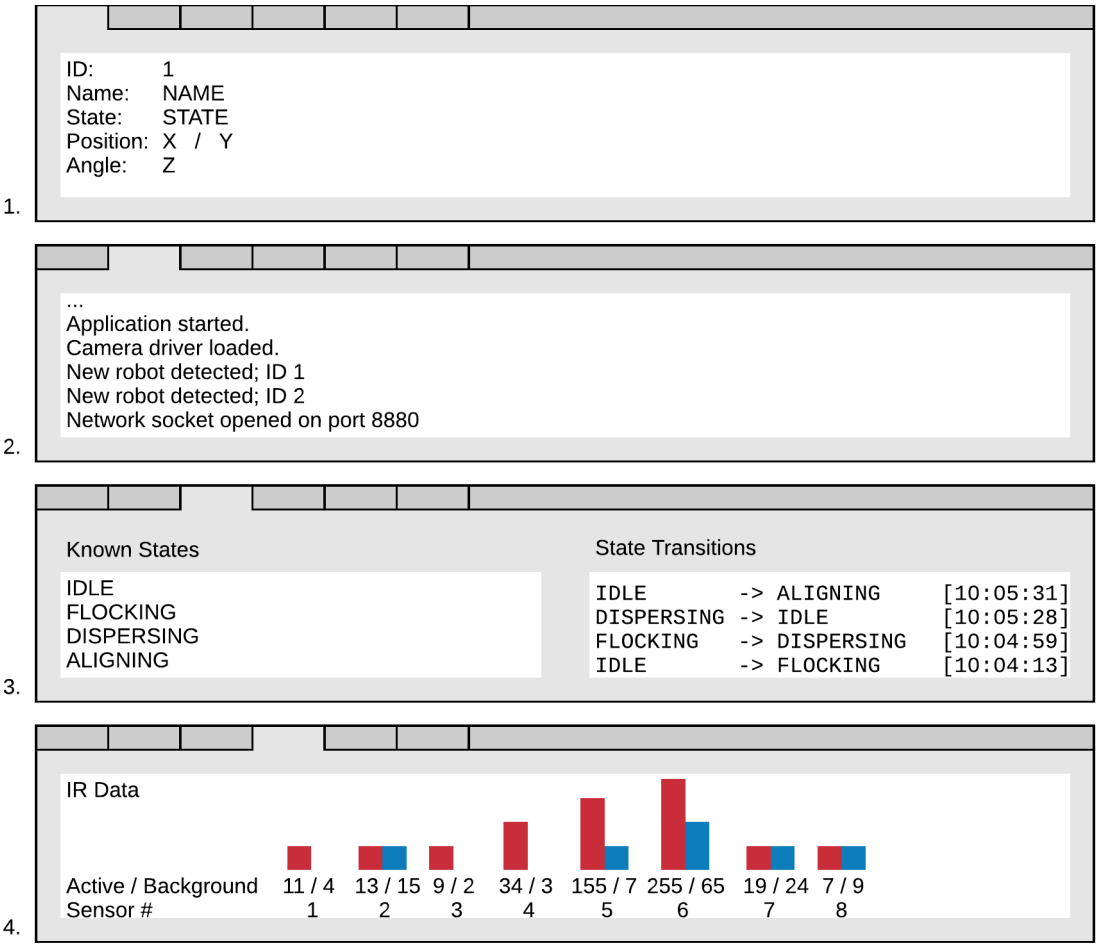


FIGURE 8.6: The initial UI designs created for some of the tabs within the data panel.

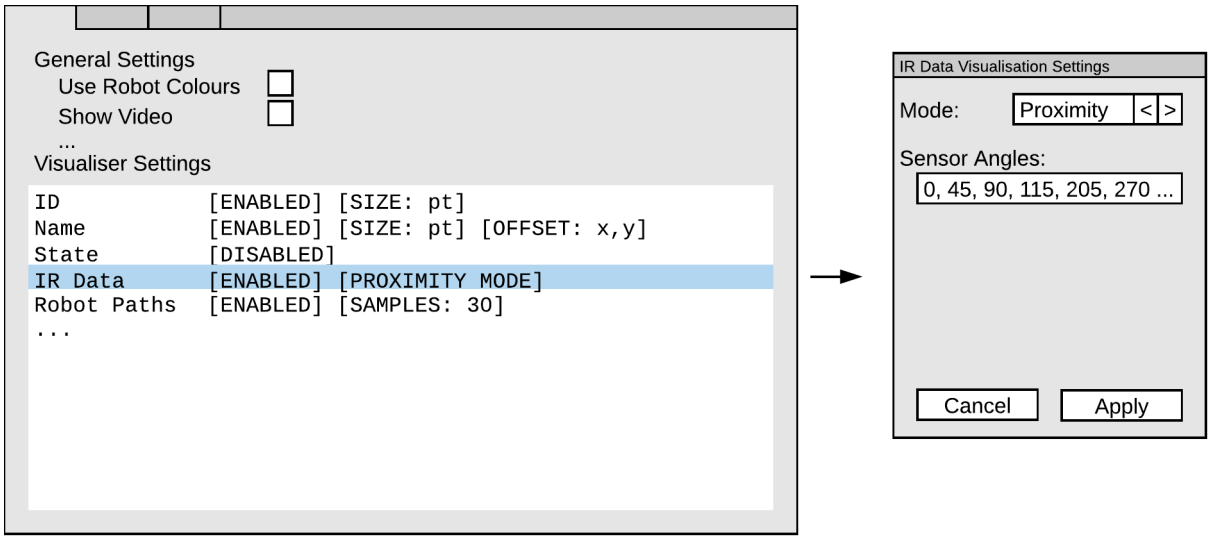


FIGURE 8.7: The initial UI design for the visualiser settings tab, and pop-up visualisation settings windows.

shown within the visualiser. The best way to visualise the data could not be determined prior to implementing the visualiser, hence a number of variations for the overlays of each type were generated. These could then be tried in practice to determine which were the most effective. Figure 8.8 shows a selection of the overlay designs for the key data visualisation types, with the data visualisation shown in orange. Row 1 shows the representational 'robot' (a grey circle with an ARuCo tag on top) with no overlay applied, for reference purposes. Row 2 shows three options for the position overlay, including circular and square outline designs, and a filled circle design. It was thought that the filled circle might be easier for a user to see, but had the downside of obscuring the robot, whereas the outline designs might be more difficult to see due to the thinness of the lines. Row 3 shows three options for the direction overlay. The first uses a simple line on top of the robot, from its center outward, indicating its direction. The second develops on the first with an arrow, which might potentially add clarity, but would be more difficult to render at small sizes. The third also uses an arrow, rendered slightly away from the robot. This arrow could rotate to match the robots orientation, but maintain the same positional offset, or it could simply rotate with the robot about its center point. Offsetting the arrow makes the shape clearer as it does not overlap the robot, however in practice it could overlap other overlay elements, reducing the clarity of both. Row 4 shows three options for the positioning of text overlays, including above and below the robot, and offset both vertically and horizontally. Text overlays are unlikely to be readable if rendered directly on top of the robot, and will always be prone to overlapping other elements if rendered with an offset. Row 5 shows two possible IR data overlay designs. The first uses lines to display the sensor values, where the lines are

oriented to face in the direction of the sensors, and their lengths vary with the relevant sensor value. If the lines were to very inversely with their related sensor values this could be used as an approximate proximity display. The second design shows a 'heat map' style overlay, where the values of the sensors are indicated by boxes that change colour in relation to the relevant sensor value. Finally row 6 shows three variations of the robot path overlay design, including smooth and stepped lines as well as solid and dotted variants. A smoother path line would likely require more data to be stored, which might be a disadvantage.

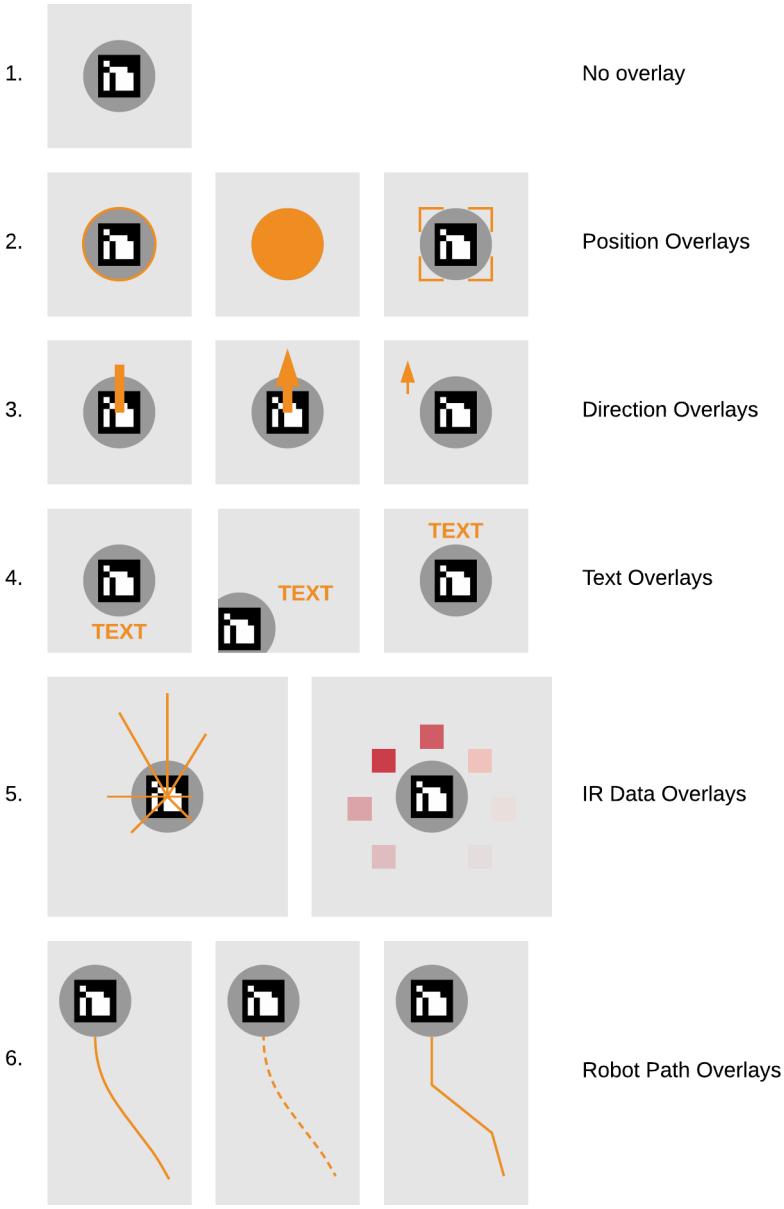


FIGURE 8.8: A selection of the designs for visualiser overlays, for a number of data types.

Chapter 9

Implementation

This section gives details of the implementation phase of the project, including descriptions of how parts of the system function, information regarding the development process and explanations for some of the key decisions made regarding the implementation.

9.1 Overview

The system was implemented closely following the design laid out in chapter 8, and is comprised of two parts; the main computer software application (referred to henceforth as 'the application') and a collection of software routines designed to be included within the code on the robots, which form an 'application programming interface' (API) . This API (referred to as the 'robot side code' or 'robot side API') provides the developer with functions to send data from the robot back to the application, and contains routines for handling the networking requirements to achieve this, and for correctly formatting the data. Details of this robot side API are provided in section 9.11. Both parts of the system implementation are fully independent. The application can be run on its own and receive data from any source, provided that this data correctly follows the format outlined in section 9.8. The robot side code could be used to send data to another host, provided that the robot uses the correct target IP address and port number, and again provided that the host can correctly interpret the data format. The application is the much larger of the two system parts, and therefore will be the focus of the majority of this implementation chapter.

Figure 9.1 shows the user interface for the application as it is seen at start-up. This can be compared to figures 8.4 and 8.5 to see the relationship to the user interface design. The key features of the application can be seen in this image. The visualiser component shows the video feed, augmented with information about the three visible robots, including position, direction, ID number, and the selected robot's name and current state. The robot list panel is visible on the right hand side, showing the

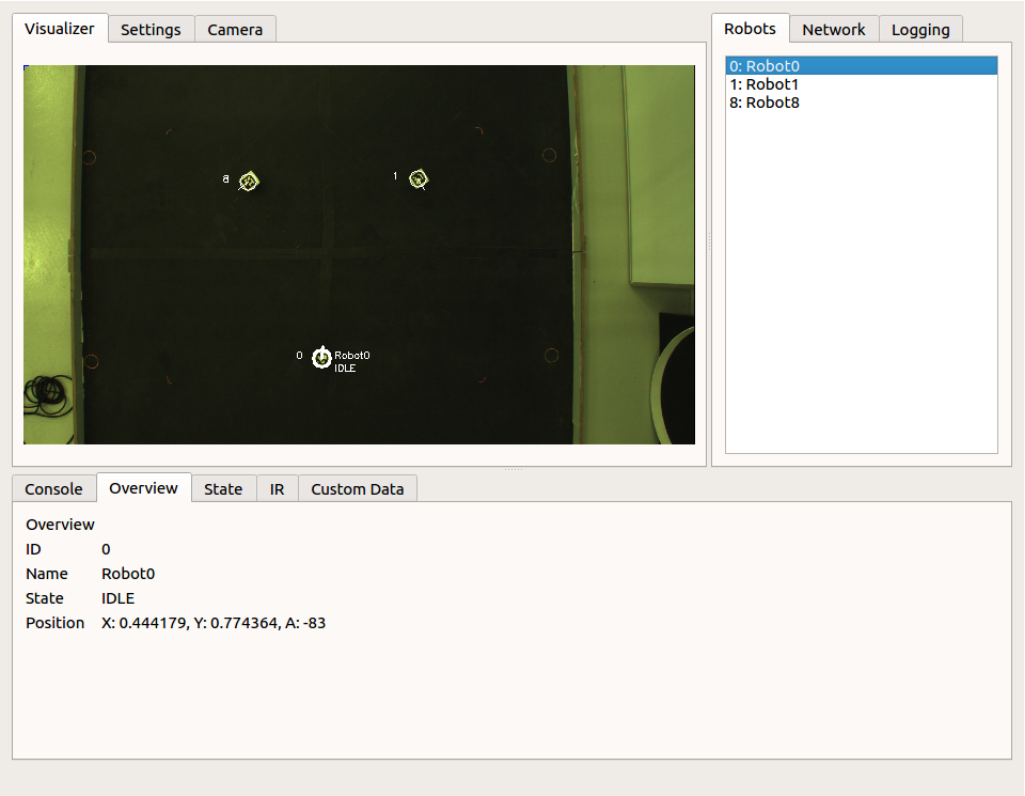


FIGURE 9.1: The user interface for the application.

IDs and names of the known robots, and which of them is currently selected. Finally the data panel can be seen at the bottom of the application, currently set to the overview tab, providing summary information about the selected robot. More detailed information about the implementation of the UI itself can be found in section 9.9.

9.2 Application Framework Selection

Developing computer software applications often involves implementing a large amount of the same low level functionality, regardless of the application. This includes low level back end functionality such as event management and dissemination, standard constructs such as timers and threads, as well as standard user interface elements such as windows, menus, panels, lists, tables, buttons and many more. It is clear that these functionalities are independent of the purpose of the application being developed, and implementing them from scratch for each new application would require a huge amount of time, and therefore be extraordinarily inefficient. For this reason the vast majority of modern software is created using some kind of application programming framework. The purpose of these frameworks is to provide the common, low level functionality in the form of an API, which the developer

can then leverage to develop their specific application. The API will usually include a number of classes to define the common UI elements, and a hierarchical tree- or node-based structure the developer can populate with these elements to create their desired UI. Most computer applications are event driven; when the user provides some kind of input such as a click or a key press an event is generated, and the event generated will be different depending on which UI component the user was interacting with. This event then needs to be disseminated through the application in order to trigger the correct functionality. Most application programming interfaces will handle the generation and dissemination of these events, allowing the developer to '*register*' code to be executed in response to specific events. The developer can therefore simply focus on implementing the functionality that is unique to their application.

Most modern application frameworks contain a wide variety of features in addition to those responsible for the UI and event management. These can include everything from low level components such as timers, input/output (I/O) interfaces, and networking components, to higher level multimedia handlers such as video and audio players, and rich HTML viewers. Frameworks also might include classes for creating data models which can be easily mapped to their more complex user interface elements such as responsive tables.

Selecting an appropriate framework to use as the basis for this application was one of the first steps in the implementation process. All of the available frameworks have different benefits and limitations, and target a variety of different platforms, operating systems, and languages. During the design phase it was determined that the application should be implemented in C++, for reasons discussed in section 8.1. This therefore eliminated a number of frameworks, such as the Oracle Application Development Framework which is specific to the Java language, and Microsoft's .NET framework, which is C# specific. The application also needed to run on the server connected to the tracking camera, which runs a Linux operating system. This therefore ruled out any windows specific framework, including one of the most widely used C++ frameworks, the Microsoft Foundation Class Library (MFC).

9.2.1 The Qt Framework

It was determined that the '*Qt*' application framework was the most suitable for this application, as it provides support for cross-platform compilation, including Linux, and is implemented natively in C++. A number of factors, in addition to the target platform and language, influenced this decision. The framework is widely used, and therefore has a well-tested, refined, and mature API, with a good body of documentation available. It provides a comprehensive library of classes for a range of common application functionalities, including GUI front-end and low level back-end components. The framework also includes built in support for multi-threading,

and a structured '*signals and slots*' system for sending and receiving event notifications, and moving data between components and across threads. For an application such as this, with a number of external data sources in addition to the user input and a multi-threaded design, this was determined to be a highly beneficial feature which could reduce development complexity significantly. Furthermore previous experience with Qt outside of this project had been positive, and meant a smaller learning curve would be necessary to get started developing the application. For non-commercial projects Qt is available free of charge, making it a good fit for an academic project.

The following primary features of the Qt application framework were used within this project:

- Standard GUI components and layout management classes used to create the majority of the user interface.
- Event management system to handle user input events.
- '*QTimer*' component to implement timers and recurring events, such as camera polling.
- '*QThread*' API to partition the application components into threads.
- '*Signals and Slots*' system for inter-component and inter-thread signalling and data transfer.
- '*QPainter*' component for rendering custom user interface elements.

9.3 Application Structure

The application has been implemented closely following the software architecture design outlined in section 8.1. Figure 9.2 shows the structure of the application at a class level following implementation, with the arrows showing the flow of data. All of the main classes are displayed, with some of the minor classes omitted for clarity. Comparing this with the software architecture design we can see that the *MainWindow* class encapsulates the functionality of the Application Controller block, forming the core of the application and routing data between the other components. This class is instantiated when the application begins, and performs all of the set up operations. This involves constructing and initialising the user interface based on the layout defined in *mainwindow.ui*, creating instances of the main components, connecting the related signals and slots of each component, and moving the components to their appropriate threads. The output signals generated by the *DataThread* class when new robot data arrives, and the *CameraController* class when new tracking data is available, are routed to the *DataModel* class's *new data* input slot. Similarly the

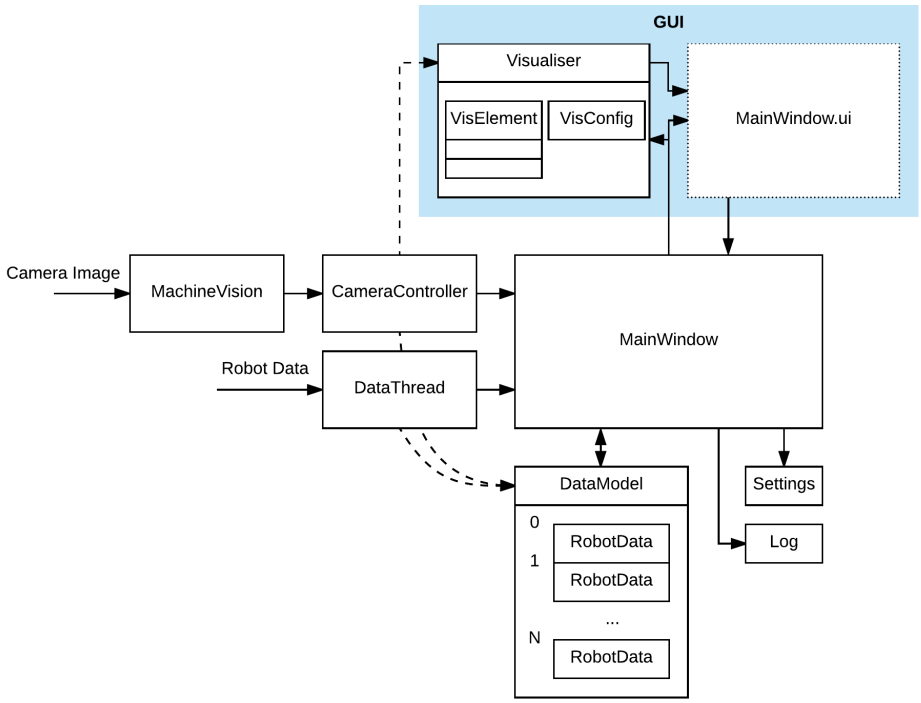


FIGURE 9.2: The structure of the classes within the application. Arrows represent the data path.

signal generated by the *CameraController* class when a new video frame has been acquired is routed to the *Visualiser* class's *image data* slot. In figure 9.2, these signal/slot connections are represented by dotted line arrows. Singleton instances for the *Settings* and *Log* classes are also created during initialisation, which allow the settings and logging functions to be accessed from anywhere in the application. Following initialisation, whilst the application is running, the *MainWindow* class is responsible for handling user input events sent from the UI, modifying the other classes as necessary depending on the input event received. Finally when the application is closed the *MainWindow* class is responsible for tearing down the other components. This involves stopping any active timers, stopping the various threads, and releasing all allocated memory.

The GUI portion of the software architecture design is encapsulated into the UI description file *mainwindow.ui*, which then connects back to the *MainWindow* class through Qt's UI event management system. Section 9.9 provides details of the user interface implementation. The visualiser portion of the GUI is encapsulated into the *Visualiser* class, which handles generating the graphical overlays, applying them to the latest video image, and displaying the resulting augmented video image. Generating the overlays involves a number of smaller classes, as shown in figure 9.2. Section 9.10 provides more detail about the implementation of the visualiser class and its sub-classes. The camera controller and tracking code components are encapsulated in the *CameraController* and *MachineVision* classes respectively, discussed

further in section 9.6, and the network controller component is implemented in the *DataThread* class, which is discussed in section 9.7. Finally the data model component is implemented in the *DataModel* class, making use of the smaller *RobotData* class. This implementation is discussed in section 9.5.

9.4 Source Code

The system source code is implemented in two groups of files; a collection of C++ source and header files which make up the main application, and a smaller collection of C++ source and header files which handle the robot-side portion of the system. In addition to its source files the main application also relies on a number of other files which are used by the Qt system to define the user interface layout, and manage the build process. Table 9.1 details the names and purposes of all files within the main application. Table 9.2 details the files that make up the robot side API.

TABLE 9.1: Source code and tertiary files that make up the main application.

File	Purpose
main.cpp	The entry point for the application. Instantiates the MainWindow class.
mainwindow.cpp, .h	The core class, controls the set up and tear down processes, handles responding to UI events within the main window and routing data throughout the system.
mainwindow.ui	Describes the user interface layout in an XML-like format. Used by the Qt framework to construct the UI.
datamodel.cpp, .h	The top level class encapsulating the full data model. Maintains a list of *RobotData* objects.
robotdata.cpp, .h	A class encapsulating the data of a single robot, including ID, name, position, path, state, sensor data and all custom user data.
datathread.cpp, .h	This class contains all routines for receiving data from the robots via wifi, and is runs on a thread of its own.
cameracontroller.cpp, .h	The high level class encapsulating the routines and data related to reading the tracking camera.
machinevision.cpp, .h	A lower level class encapsulating routines for interfacing with the camera hardware.
visualiser.cpp, .h	This class encapsulates the visualiser GUI component, and is implemented to conform the Qt framework by extending the QWidget class. Contains routines for generating and applying the video augmentations based on the current robot data and visualiser settings.

viselement.h	Contains an abstract class definition for a single visualiser settings element. These elements are used to define how specific elements of the video augmentation are rendered, and also contain settings and variables relevant to this task.
vis*.cpp, .h	Classes beginning with the '*vis*' prefix derive from the *VisElement* abstract class and contain routines for rendering the visualisation for one type of data. The latter part of the class name identifies the relevant data type.
irdataview.cpp, .h	A custom GUI object, derived from QWidget, which displays the raw IR sensor data as a bar graph in the data window.
settings.cpp, .h	Encapsulates the general application settings and provides functions for changing their values. Implemented as a singleton class to allow access from anywhere in the application.
log.cpp, .h	Encapsulates the functionality for recording events and data in log files. Implemented as a singleton class to allow access from anywhere in the application.
util.cpp, .h	Contains static utility functions used in various places throughout the application code.
*settingsdialog.cpp, .h	Classes ending with the '*settingsdialog*' suffix describe dialog windows for adjusting the settings related to the visualisation of specific data types, identified by the first part of the class name.
robotinfodialog.cpp, .h	Defines a dialog window which displays meta information about a specific robot, and provides controls to change the robot's display colour and delete its data from the data model.
addidmappingdialog.cpp, .h	Defines a dialog window which can be used to add a non-standard ID mapping.
testingwindow.cpp, .h	Defines a dialog window for running and displaying the results of the data model unit tests.
appconfig.h	Contains pre-processor definitions for controlling the inclusion of specific code segments.
SwarmDebug.pro	Used by the Qt framework to build the application. Lists the necessary code files and libraries.

TABLE 9.2: Source code files that make up the robot side API.

File	Purpose
debug_network.cpp	This file encapsulates networking functionality for communicating with the debugging system. Contains routines for sending data of specific types, as well as for sending raw packets.
debug_network.h	Header file for the debugging system network interface. Also contains definitions for data type identifiers.

9.5 Data Model

The data model forms a core element of the back end of the application. For each robot being tracked by the system, the data model is required to store the information related to that robot. This is done in a structured manner, such that other parts of the application can query specific data within the model easily. The contents of the data model are updated whenever new data arrives, and are consulted when rendering the user interface. Section 8.1.1 describes the design of the data model and it's hierarchical structure. The implementation follows this design closely, with the *DataModel* class contained in *datamodel.cpp / .h* encapsulating the top level data model container, and the individual robot data object encapsulated in the *RobotData* class in *robotdata.cpp / .h*.

The *DataModel* class uses a standard C++ '*vector*' container to maintain a list of *RobotData* objects. Each of these *RobotData* objects describes one robot that is currently known to the system. The list is ordered based on the robots ID's, and is sorted after each new insertion. The vector container was chosen as it does not require a fixed size, and can be sorted and iterated efficiently. The *DataModel* class provides convenience functions for data retrieval functionality, such as retrieving the data object for a given robot based on ID number. It also provides a function for entering new data into the model, which inherits the properties of a '*slot*' within the Qt framework, allowing it to be called from other threads. Each new data packet received from the network is routed to this slot, as well as position data obtained from the tracking system. All data is supplied in the form of a string in one of the packet formats described in section 9.8. Code within the data model class then handles interpreting the string, determining its purpose and source robot, separating out the data content, and updating the relevant robot within the model. Whenever data arrives from a previously unknown robot, this code handles creating a new *RobotData* object, and adding it to the list.

The *RobotData* class encapsulates the data for a single robot. This involves storing a number of different data points, in a variety of different formats. The class then acts as a relatively simple container for this data, providing functions for retrieving and changing values. Table 9.3 outlines the data points contained within the class.

TABLE 9.3: The contents of the *RobotData* class.

Data Point	Type	Description
Robot ID	Integer	The numerical ID of the robot, used by the tracking system and when transmitting data.
Robot Name	String	The name associated with this robot. Set by the user when programming the robot, and reported in watchdog packets.
State	String	The current state of the robot.
Known States	List of strings	A list of all states the robot has previously reported.
Position	2D Vector	The current position of the robot expressed as a proportional coordinate vector.
Angle	Integer	The current angle of the robot in degrees.
Colour	OpenCV Scalar	The colour used for this robot in the visualiser, if colours are enabled, expressed as an OpenCV Scalar struct in RGB format.
IR Data	Array of Integer	The most recent IR sensor readings for this robot. One value per sensor.
Background IR Data	Array of Integer	The most recent background IR sensor readings. One value per sensor.
Custom Data	Key Value Map	All custom user data. Stored as a map of key value pairs.

In addition to this data, the class also maintains a list of recent state transitions, and a short term history of the robot's position. The state transition list uses a custom structure to store the state before the transition, the state after the transition, and the time the transition occurred. A fixed size array of these custom structures is maintained and updated each time a state transition occurs, acting as a first-in first-out (FIFO) queue. The position history is stored in a similar fashion, using a fixed size array of coordinate pairs, which is updated every Nth position update. This interval can be configured by the user, with a lower interval giving a higher resolution but a shorter history, and vice versa for a higher interval. Functions are provided to retrieve data from these queues when needed.

9.6 Video Feed and Tracking System

Functionality for acquiring images from the machine vision camera, and running the ARuCo tag tracking algorithm, is encapsulated in the *CameraController* and *MachineVision* classes, which are implemented in *cameracontroller.cpp / .h* and *machinevision.cpp / .h* respectively. Video is retrieved from the camera one frame at a time, and can therefore be thought of as a sequence of discreet images. A call to the camera driver to obtain the next image might block execution whilst it waits for the image to become available. For this reason these classes are run on a separate thread dedicated to camera functionality, in order to maximise application performance and ensure responsiveness. The *CameraController* class handles the higher level operations such as running a timer to periodically poll for the next image, supplying the correct dimensions for the image, and converting the image and tracking data into formats which can be passed back to the main thread and used in the UI and data model respectively. The application threading is handled through the use of the Qt framework's *QThread* API, and communication between threads utilises the framework's '*signals* and *slots*' feature, which allows components on different threads to send and receive data in a managed, thread-safe manner. The *CameraController* class therefore utilises two signals; one for emitting the camera image data, and another for emitting the robot position data. At initialisation time the application's core class, *MainWindow*, connects these signals to matching slots within the *Visualiser* and *DataModel* classes respectively.

The *MachineVision* class handles the lower level operations related to the camera and the tracking system, including setting up the camera driver, retrieving and resizing individual images from the camera, and running the ARuCo tag detection algorithm. The ARuCo software is implemented as an additional component of the OpenCV image processing library (discussed below), and provides a function to run the tag detection algorithm on a given image, for a given dictionary of tags. For each tag detected in the image that matches one in the chosen dictionary, the ARuCo algorithm returns the pixel coordinates of the four corners of the tag. The *MachineVision* class includes code to average these four coordinates, acquiring a central pixel coordinate, which is then converted to a 'proportional' coordinate; two numbers between 0 and 1 which represent the horizontal and vertical components of the position as a proportion of the full height and width of the image respectively. This ensures that the robot positions can still be correctly displayed after the image has been resized, without having to maintain information related to the resizing operation. The angle the robot is facing is also calculated. This is done by first calculating the position of the fourth corner on a coordinate axes an where the third corner is positioned at the origin, and then applying an arctangent function to this coordinate. This angle is then converted to degrees and stored as an integer in order to reduce complexity, as a precision greater than one degree was not deemed necessary.

9.6.1 Image Processing Library

A number of the components within the application are required to manipulate image data. The image data acquired from the tracking camera by the CameraController and MachineVision classes must be stored in a format that can be processed by the ARuCo tag detection algorithm. This image data must then be passed to the visualiser component, which renders the graphical overlays on top. Selecting a suitable image processing library was one of the early steps in the implementation process. The ARuCo tag detection algorithm is implemented as an extension to the OpenCV image processing library, so this seemed like the obvious choice. However the Qt framework supports its own image data format and drawing classes, so the image could be converted to this format once the tracking data had been extracted. OpenCV is a widely used, feature rich, powerful image processing library, and ultimately it was decided that all image manipulation should make use of it where possible. IT was hoped that this would improve maintainability in the future, as OpenCV is more widely used, and also improve portability; if the code was ever to be ported away from the Qt framework, the image processing components would require less change.

9.7 Networking

A key element of the system is the wireless retrieval of information from the robots. This required the implementation of networking functionality within both the application and the robot side API. As mentioned previously the Linux extension boards on the e-puck robots feature a WiFi adapter, so WiFi was selected as the target wireless networking technology to implement this functionality with. The next step was to look at the networking requirements in more detail, and decide on which transport layer protocol to use.

The requirements for the networking portion of the system were relatively simple, and can be summarised as follows:

1. Must utilize a WiFi network.
2. Must allow a large number of sources to transmit data to a single host.
3. Must allow for frequent transmission of small packets of data.

WiFi networks utilise the standard Internet Protocol (IP) network layer protocol. There are two commonly supported transport layer protocols which run on top of IP, the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP is a managed and delivery-error checked protocol, and therefore guarantees that packets will be transmitted in the correct order, with lost packets

being retransmitted. This adds overheads such as acknowledgements to the protocol, and requires an established 'connection' in order to function correctly. TCP also operates a queueing system, whereby packets for transmission are sometimes held until a number of them are ready, and can therefore be grouped together and sent. UDP, by contrast, does not error check the delivery of packets, making no guarantees that a packet will be received, or that packets will be received in the correct order, removing the need for an established connection and reducing the overheads involved. Packets can therefore be sent from any application to any target IP address and port on the network, without first establishing a connection with another application. Packets are also sent immediately, with no queueing system in place. It was determined that UDP would be the most suitable for this system, for a number of reasons. The connection requirements of TCP would require the application to form a connection with each robot prior to transmitting data, which would add unnecessary complexity. Using UDP also ensured that the packets were transmitted immediately, reducing the potential for latency in the system. The lack of delivery checking was not considered an issue, as the robots would be transmitting updates frequently enough that a single lost packet would not cause a significant issue.

The networking functionalities of the application are encapsulated in the *DataThread* class, found in *datathread.cpp / .h*. This class is run on a dedicated thread to ensure that potentially blocking operations do not impact application performance. The class contains routines for dealing with the low level network requirements, such as establishing a socket through which to receive data packets from the robots, and continually listening on this socket for new data. Figure 9.3 shows the operation of the *DataThread* class as a flow diagram. The '*Packet Listening Started*' signal, highlighted in green, is generated when the user presses the 'Start Listening' interface button in the networking tab. All socket operations are implemented using structures and definitions from the standard C++ networking libraries. Data received from the robots is passed to the main application thread through a Qt signal, using the Qt signals and slots interface mentioned previously. The main application class, *MainWindow*, connects this signal to the appropriate slot in the *DataModel* class at initialisation time. The application side networking can be configured by the user in the *network* tab of the right-hand panel of the user interface. Here the user is able to enter a desired port number on which to receive data, and can start and stop the application listening for packets on this port by pressing the '*start/stop listening*' button.

For the robot side API, the networking functionality is implemented in a similar way. The initialisation function establishes a target socket based on a supplied IP address and port. This should match the IP address of the computer or server running the main application, and the port chosen by the user within the main application. When any of the functions for sending specific data packets are called, the data is sent to this target socket as a UDP packet. Section 9.8 discusses the format of the data within these packets. All socket operations are once again implemented using

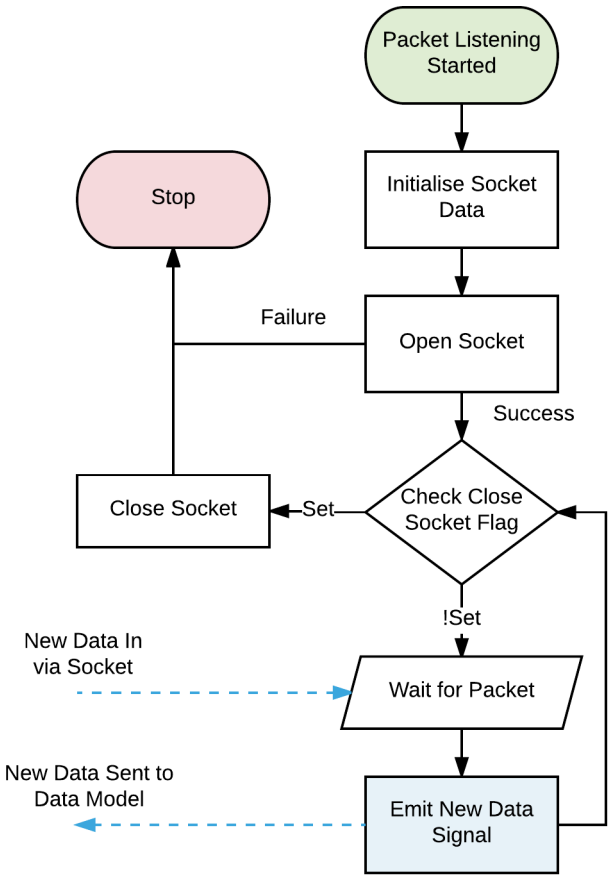


FIGURE 9.3: A flow diagram showing the sequence of operations carried out by the *DataThread* networking class.

the standard C++ networking libraries to increase portability.

9.8 Data Transfer Format

In order for the application to interpret and use data received from the robots, a common format for exchanging this data needed to be defined. Both sides of the communication link then need to use this format correctly when constructing and de-constructing packets. A number of different options were considered for achieving this, ranging from super-lightweight custom packet formats using the minimum number of bytes, to established existing solutions such as the JSON data interchange standard. The primary concerns when making this decision were a desire to minimise any overhead in terms of extra code needed on the robot side, as the robots have limited memory, and to ensure the format remained as simple as possible so that future extensions to the system, such as implementations for other robots, could be programmed with relative ease. It was ultimately decided not to use JSON, to avoid the need for any additional code libraries to be stored in the robot's memory, and to instead use a custom, simple, string-based packet format. All data to be transmitted from the robot to the application is therefore converted to a string which is then transmitted in the data packet. As well as containing the data, the string must identify the robot and describe the type of data within. The format for these strings is defined as three sections separated by space characters. The first section contains the numerical ID of the robot sending the packet. The second contains a number identifying the type of data contained in the packet. The last section contains the packet data, and has a variable format, depending on the packet's type. Figure 9.4 gives a visual representation of this format. Table 9.4 describes the purpose of each packet type, and describes the format of the 'packet data' section for each.

FIGURE 9.4: The general format for each data packet.

TABLE 9.4: The format of the data section and the purpose of each packet type.

Watchdog Packet	
Type ID	0
Format	[ROBOT NAME]
Purpose	Sent periodically to inform the application that the robot is still active. Also contains the robot's name, as should be displayed in the application.
Example	"6 0 Robot_6"
State	
Type ID	1
Format	[CURRENT STATE]
Purpose	Informs the application of the robots current state.
Example	"6 1 IDLE"
Position	
Type ID	2
Format	[X POSITION] _ [Y POSITION] _ [ANGLE]
Purpose	Provides the application with a robot's position and orientation. This data is not sent by the robot, but instead comes from the tracking code.
Example	"6 2 0.23 0.682 110"
IR	
Type ID	3
Format	[SENSOR 1 DATA] _ [SENSOR 2 DATA] _ ... [SENSOR N DATA]
Purpose	Contains a robot's infra-red sensor readings. Each sensor value is separated by a space, and the packet can contain as many values as the robot has IR sensors.
Example	"6 3 101 93 115 112 103 98 365 2850"
Background IR	
Type ID	4
Format	[SENSOR 1 DATA] _ [SENSOR 2 DATA] _ ... [SENSOR N DATA]
Purpose	Contains a robot's background infra-red sensor readings. Formatted the same as the standard IR data packet.
Example	"6 4 95 87 99 110 89 98 103 82"
Message	
Type ID	5
Format	[MESSAGE STRING]
Purpose	This packet allows any general message to be sent from the robot to the application, and will be displayed in the application console and recorded in the logs.
Example	"6 5 Robot entering hibernation mode"
Custom Data	

Type ID	6
Format	[KEY] _ [VALUE]
Purpose	Contains a piece of custom user data, in the form of a key value pair.
Example	"6 6 DebugCounter 540"

9.8.1 Constraints

For a number of the packet types constraints had to be placed on the input data, both in terms of format and in terms of the range of valid values. The space character is used as a delimiter to separate portions of the packet, hence in all cases except the message packet a space character cannot appear in the middle of any of the data. This means that robot names and states cannot include spaces, nor can custom data keys or values. The message packet type is an exception, and will treat any characters following the second space which marks the end of the header data as a single string. This string then forms the message content. For the watchdog, state, position and custom data packet types, the correct number of space-separated data portions must be included, or the string is invalid. This means one, one, three and two data portions respectively. In the case of the IR data a minimum of one data portion must be included, but there is no upper limit. Any portions above the number of supported sensors will be ignored. The numerical data in the position packet must be two floating point values (X and Y position), followed by one integer value (angle). A floating point value can be interpreted from an integer (the decimal point is therefore not necessary), however the integer angle value cannot be interpreted from a floating point data portion, and this will cause the packet to be marked as invalid and ignored by the application. The IR data values must be positive integers in the range of 0 to 4095. Values outside of this range will be clamped to the closest value that satisfies this requirement. Floating point values will cause the packet to be marked as invalid and ignored.

9.9 User Interface

In order to implement the user interface the designs shown in section 8.2 had to be realised using the UI component classes provided by the Qt application framework. The '*QtCreator*' tool was used to arrange all the basic elements of the UI, and generate an XML-based descriptor file (*mainwindow.ui*) which describes this layout. When the application is initialised this file is parsed and used to populate the user interface. Events generated by the interface are linked to functions within the *MainWindow* class using the signals and slots system. Code was then implemented within each of these functions to execute the correct actions based on the input received.

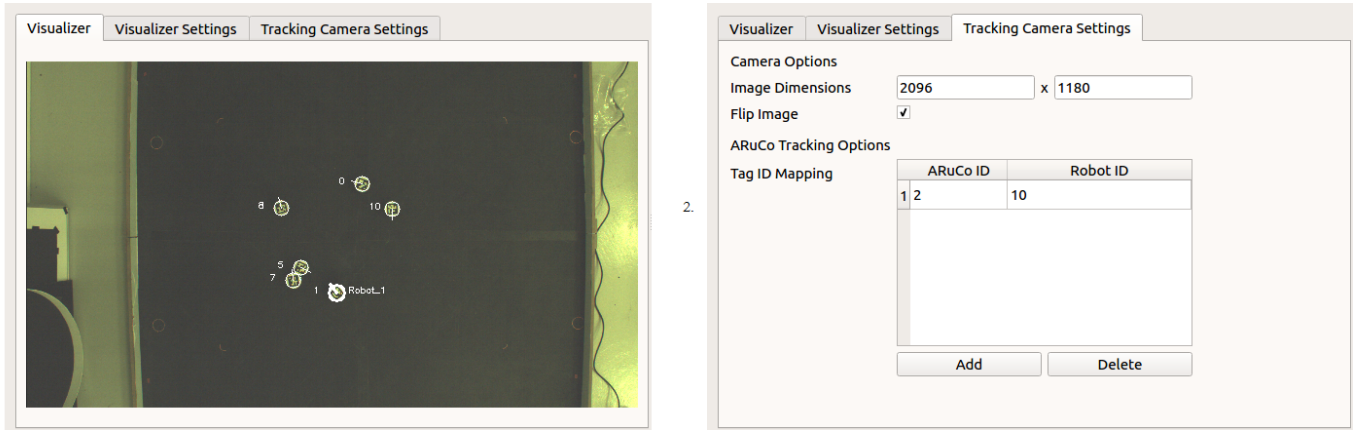


FIGURE 9.5: The visualiser panel, with two of the different tabs shown.

The user interface was implemented following the three panel layout outlined during the design phase. The panels are divided by splitters which can be dragged by the user to adjust the size of each panel to suit their screen. When reduced past a certain minimum size the panels will be minimized completely, freeing up more UI space for the other panels. This allows the user to hide the robot list panel or data panel in favour of a larger visualiser display. Each panel features a tab controller component to allow the user to switch between multiple views. This was necessary to provide sufficient space for the required features and all the various settings. The visualiser panel has three tabs, the first showing the main visualiser view, the second providing access to the settings related to the visualiser, and the third providing access to the settings related to the camera and tracking system. The visualiser and its settings are discussed in section 9.10. The camera settings tab allows the user to input the image dimensions of their camera output, so that the visualiser can display this at the correct aspect ratio. It also allows the user to set up mappings between specific ARuCo tag and robot IDs. This is useful if the ID number of the tag attached to a robot does not match the ID number the robot is using to report data. By applying a mapping the user can instruct the system to apply tracking data for a specified ARuCo tag ID to the robot entry within the data model with a different specified robot ID. Figure 9.5 shows two of the tabs within the visualiser panel; the left hand image shows the actual visualiser tab, whilst the right hand image shows the camera settings tab. The visualiser settings tab is shown in section 9.10.

The robot list panel also features three tabs. The first displays the robot list, and allows the user to select from the robots currently known to the system. By selecting a robot the user makes it the current focus of the application. This can cause the visualiser to display more visualisations for the selected robot depending on the visualiser settings. It will also make the selected robot the focus of the information shown in the data panel. The second tab provides controls for the user to configure the networking functionality, including a text box to enter the desired port number,

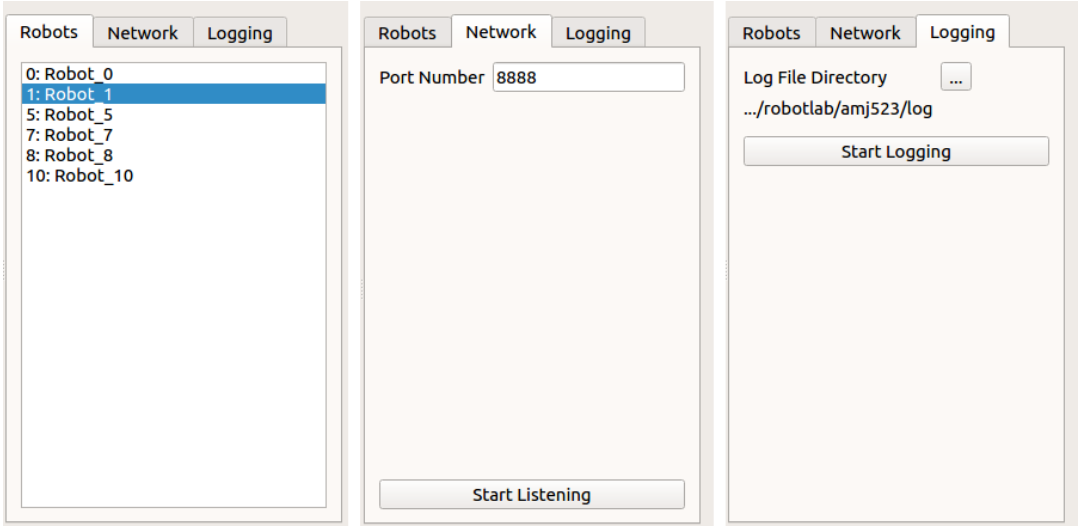


FIGURE 9.6: The robot list panel, showing all three tabs.

and a button to start and stop listening for data packets. The third tab provides controls for the user to configure the data logging functionality, including a button which opens a file browser for setting the directory path where logs should be stored, and a button to start and stop the data logging. Figure 9.6 shows the three tabs within the robot list panel.

The data panel features five tabs. The first tab displays a simple text based console, which reports messages regarding the application itself, as well as any messages received from the robots in message packets. The messages are displayed sequentially, with the most recent message always being added as a new row at the bottom of the console. The second tab is the overview tab, which provides a summary of the selected robots key data. The third tab is the state tab, which displays two lists of state information regarding the selected robot. The first lists all of the robots known states, and the second lists recent state transitions, including the state before and after the transition, and the time at which the transition occurred. This should help the user to determine if a robot is moving through its states correctly, and check that it is not changing states out of order to too frequently. The fourth tab displays the selected robot's IR sensor data in the form of a bar graph. This includes two bars for each IR sensor, one for the active sensor reading and one for the background reading, with colour being used to differentiate between the two. The numerical values of both are also displayed below the bar, so that the user can ascertain the actual values if necessary. Finally the fifth tab displays the custom user data for the selected robot. This data is organised into a table showing the custom data keys in the first column, and the values for each respective key in the second column. All five tabs update their data in real time, in response to received packets. This means the user is kept up to date with the latest information immediately. Figure 9.7 shows the five data panel tabs.

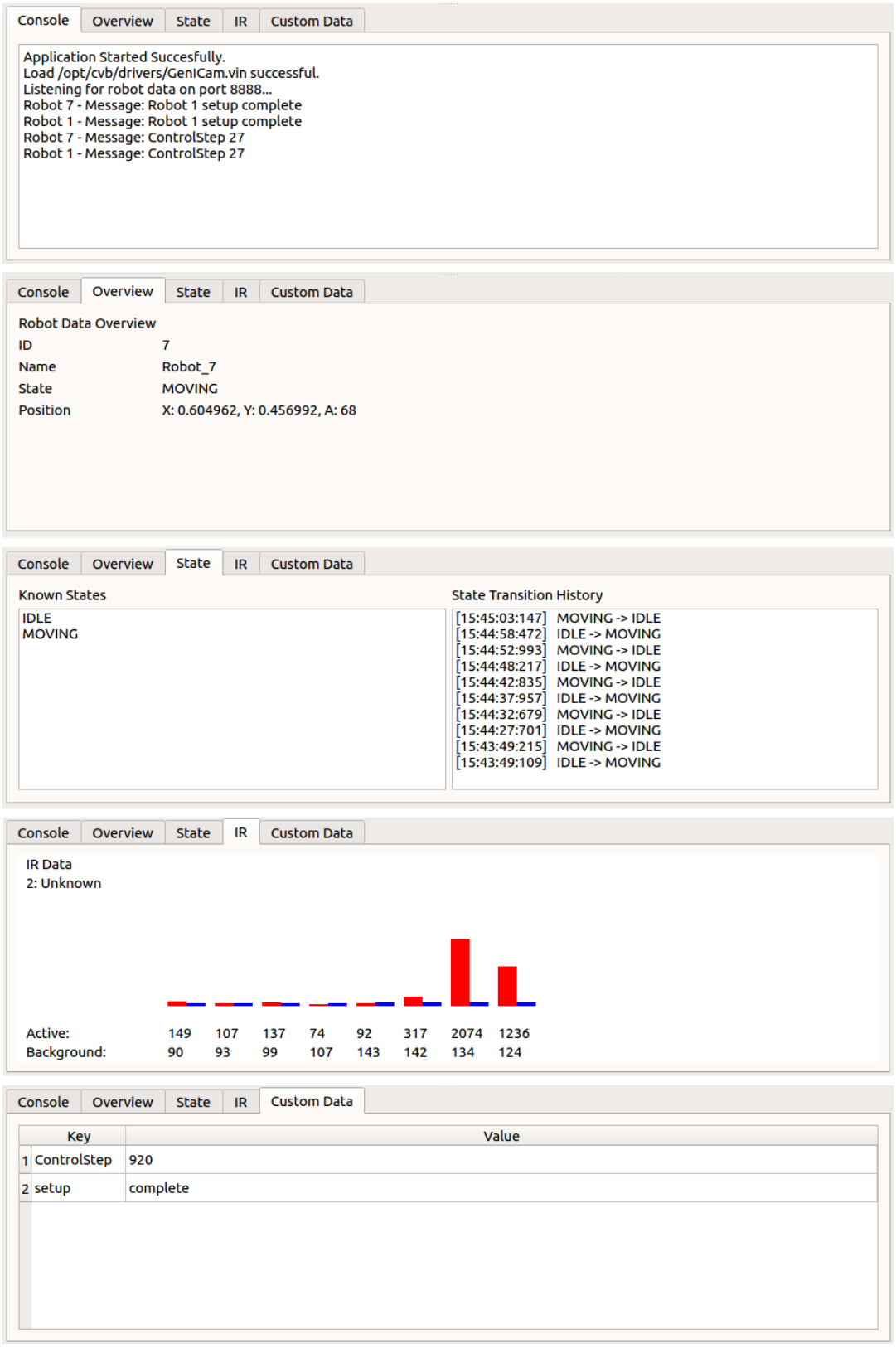


FIGURE 9.7: The data panel, showing all five tabs.

In addition to the main user interface, a number of extra 'dialog' windows are used to provide the user with access to various settings. For each of the data visualisations which feature settings beyond a simple enable/disable , a dialog window class was implemented. The names of these classes take the form **SettingsDialog*, where * is replaced by the visualisation type. The files containing definitions for these classes follow the same naming format, **settingsdialog.cpp / .h*. Dialog windows are a commonly used tool within applications programming. They act as pop-up windows which usually require the user to either confirm or cancel some action or change. In this case the dialog windows present controls for changing specific visualisation settings, and the user can then either apply their changes or cancel them using the standard accept/reject buttons at the bottom of the window.

9.10 Visualiser

The 'visualiser' is the name given to the custom user interface component that renders the augmented video feed. Implementing this component was key to satisfying the portion of the project aims related to augmented reality, and it forms one of the most visible elements of the system. The main visualiser component is defined in the *Visualiser* class (*visualiser.cpp / .h*), and a number of extra classes are used to define the associated settings and routines for visualising specific data types (*VisConfig, VisID, VisName, VisState, VisPosition, VisDirection, VisProximity, VisPath and VisCustom*). Section 9.6 describes the process of retrieving images from the camera and tracking the robots. The image data then arrives at the visualiser via a Qt slot function. At this stage the image is augmented based on the data in the data model, by iterating over the list of robots, and for each one iterating over the list of data visualisations, calling the render function for each. These render functions take the image and the current robot's data as arguments, and then add the relevant graphical representation to the image using the drawing functions within the OpenCV image processing library. This process is implemented following the design outlined in section 8.1.2, and figure 8.3.

It was decided that an individual class would be implemented for each type of data visualisation. Each class derives from the abstract class *VisElement*, which defines the general outline of a data visualisation class, and then each specific implementation defines how to generate the graphical overlay for that data type. The *VisConfig* class then stores a collection of all the *VisElement*-derived objects, which can be iterated through to render each one. The use of an abstract class was necessary in order for the different visualisation element classes to be stored in the same collection. The overall aim of this implementation method was to make the visualisation process simpler to manage, and to follow object oriented practices, making it easier for new visualisation types to be added without modifying the underlying

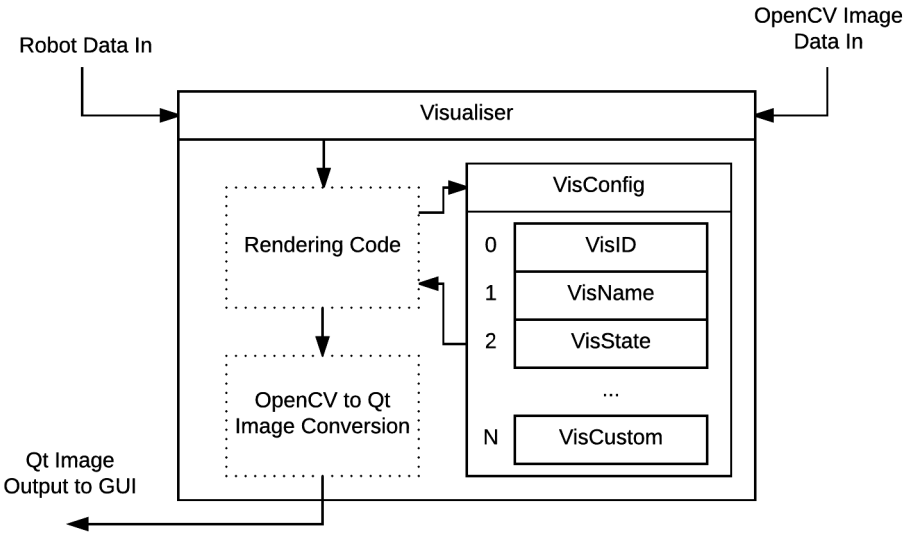


FIGURE 9.8: The path of data through the visualiser when generating each frame.

system. This also allows each data visualisation object to maintain its own settings, so that the visualiser component itself need not be aware of the details of the configuration of each data visualisation element. Instead each data visualisation element simply checks it's own configuration when its render function is called.

The rest of the code in the *Visualiser* class relates to embedding the OpenCV image within a Qt UI widget, and tertiary functionality such as detecting clicks within the image frame, and retrieving the frame's size. The OpenCV to Qt image conversion is done by instantiating a QImage instance directly from the internal pixel data of the OpenCv image. This QImage is then drawn onto the widget. Figure 9.8 shows the flow of data through the visualiser when rendering a frame.

9.10.1 Data Visualisations

The visualiser supports a number of specific data visualisations:

1. Highlighting robot position.
2. Indicating robot 'orientation' by showing forward direction.
3. Displaying the ID of a robot as text.
4. Displaying the name of a robot as text.
5. Displaying the current state of a robot as text.
6. Displaying a graphical representation of a robot's IR sensor data.

FIGURE 9.9: The different data visualisations as implemented.

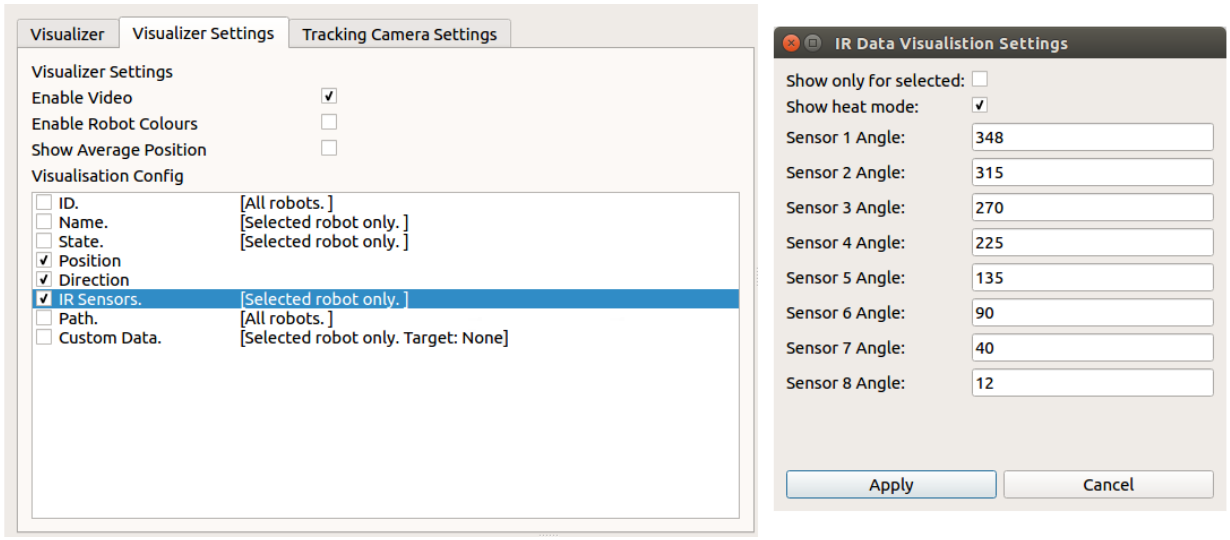
7. Displaying the recent path a robot has taken as a line behind the robot.

8. Displaying a specific piece of custom data related to the robot as text.

The designs for these visualisations described in 8.2 and shown in figure 8.8 were tested, and the most effective selected for the final implementation. Figure 9.9 shows the implemented visualisations. The text based data is rendered adjacent to the robot, on the right hand side, where even relatively long strings will not overlap the robot's other visualisations. The exception to this is the ID number, which is rendered above and to the left, as this is unlikely to be long enough to cause overlapping issues. Row 1 of figure 9.9 shows all of the text-based visualisations individually, followed by the full set combined. Robot position is indicated using an outlined circle. This was chosen over the filled circle or square designs as it was the least intrusive on other elements of the image. Direction is indicated by a line, starting from the robots center and extending outwards in its forward direction. This was selected

over the arrow based designs, as rendering the arrows at small enough sizes was not feasible. Row 2 of figure 9.9 shows the robot position and orientation visualisations, alone and combined. For the IR data visualisation, both designs were implemented, and the user can select which to use. The first renders a line for each of the sensors, extruding out from the center of the robot at the angle of the respective sensor. The length varies inversely and non-linearly with the value of the sensor, in an attempt to approximate a proximity measurement. The second mode, referred to as the 'heat map' mode, renders a number of small boxes around the robot, positioned to match the sensors. The colour and size of each box varies to indicate the sensor value. The choice to provide both was made because the line based visualisation is useful in certain scenarios when testing the response of individual sensors, but the box-based 'heat map' visualisation provides a more immediately understandable indication of the sensor values. Row 3 of figure 9.9 shows the two types of IR data visualisation. The robots recent path is visualised as a trail of line segments. One of the original designs proposed a dotted line, but in the end this proved to be less visually clear than a solid line. The smoothness of the line was also considered in the design, but during implementation it became clear that the smoothness of the line was inherently related to the number of samples. A trade-off therefore had to be made between the smoothness and accuracy of the line, and its length. A setting was added to allow the user to vary the sampling interval for the line, thus allowing them to choose between a longer but less accurate and more jagged path, or a shorter but smoother and more accurate one. Row 4 of figure 9.9 shows the path visualisation. Finally row 5 shows two possible combinations of the visualisations. In the first the robots ID, position and path are shown, as well as its IR data in heat map mode. In the second the robots ID, name, state, position, direction and path are shown.

Each visualiser element can be enabled and disabled via the settings tab. Some of the visualisations have more complex settings, which can be accessed be double clicking the specific element in the visualisations list, also in the settings tab. Figure shows visualiser settings interface and the various dialog windows. For the majority of the visualisations the user has the option to set them to render only for the selected robot, or for all the robots. The IR data visualisation also has settings to switch between the proximity and heat map modes, and to set the angles of the sensors. The path visualisation has a setting for the sampling interval, as previously mentioned. Finally the custom data visualisation allows the user to input the key string for the data point they want displayed. Figure 9.10 shows the visualiser settings tab, as well one of the dialog window for accessing detailed visualisation settings. This dialog is showing the settings for the IR data visualisation.



1.

FIGURE 9.10: The visualiser settings tab, and one of the visualisation settings dialog windows.

9.11 Robot Side API

The robot side API is encapsulated by the *DebugNetwork* class, defined in '*debug_network.cpp / .h*'. The contents of the *DebugNetwork* class are described in table 9.5.

TABLE 9.5: The contents of the *DebugNetwork* class the forms the robot side API.

Variables		
Name	Type	Purpose
socket_ready	Boolean	Indicates whether the socket has been created successfully, and it ready to be used.
sock_in	Struct	A structure defining the target socket.
sock_fd	Integer	Identifies the socket on the local machine (robot).
robot_id	Integer	The numerical ID of this robot. Set at initialisation time. Inserted to the header of each packet to identify the sender.
Functions		
Name	Return Type	Purpose
init	Void	Initialises the class by setting up the socket and setting the ID for this robot. Requires a port number, a target host IP address and the robot ID as arguments.
destroy	Void	Shuts down the socket.
sendData	Void	Sends a given string as a raw data packet.
sendWatchdogPacket	Void	Constructs and sends a watchdog packet. Requires the robot name to be provided as an argument.
sendStatePacket	Void	Constructs and sends a state packet. Requires the current state to be provided as an argument string.
sendIRDataPacket	Void	Constructs and sends an IR data packet. Requires the data be arranged into an array, and takes a pointer to the first element and a size value as parameters. Also takes an extra boolean parameter which can be set to true to indicate that this packet should be identified as backgroud IR data.

sendLogMessage	Void	Constructs and sends a message packet to display a message in the application console and logs. The message string is supplied as an argument.
sendCustomData	Void	Constructs and sends a custom data packet. The key value pair are supplied to this function as two argument strings.
getRobotID	Integer	Returns the numerical robot ID, as currently set.
setRobotID	Void	Sets the numerical robot ID based on an integer argument.

In order to utilize this API a user should modify their robot controller code to include the *debug_network.h* header file, call the *init* function at start-up time, and then call the various packet and data functions whenever necessary. The user is therefore in charge of how frequently data is transmitted, and can decide whether to simply send updates to the application when the robot's data changes, or to transmit a fixed quantity of data every control step. It was potentially possible for the API to handle all data reporting in the background, without requiring the user to specify when to transmit data in the controller code, however this approach was not taken for a number of reasons. Firstly it limits the control and flexibility available to the user. Each swarm system is different, and may have different requirements, hence the API should allow the developer to make decisions regarding data reporting which are right for their specific case. It would also limit the portability of the API, as a more automatic system would likely have to hook into the lower level robot drivers, meaning much greater changes would be necessary to port the code to a different robot. Furthermore by allowing the user to have complete control over when and how frequently data is transmitted they are able to manage the amount of traffic they are putting on their network, potentially mitigating congestion issues with very large swarms. Finally allowing users to control the moments within the code when data is reported to the debugging system was deemed to be a more intuitive system, especially as many developers are already familiar with the concept of using 'print' statements within code to display debugging messages in a console.

Chapter 10

Testing and Evaluation

Software testing is the practice of defining a number of processes which can be applied to a piece of software in order to verify its correct operation. Each of these processes may involve a number of steps, and is usually described as a single test. The results of each test are normally recorded, and used to determine if any bugs or issues remain in the software. Once the implementation phase of this project was complete it was important to thoroughly test the software in order to verify its correct implementation and operation, and identify any remaining issues that needed to be fixed or mitigated. Some testing was also carried out throughout the implementation phase, in order to verify the correct operation of individual components as they were completed, and to ensure that different components would work correctly together. This process of testing during development is referred to here as continuous integration testing, and was done to reduce the risk of issues stacking up and becoming layered or entrenched as development continued. A more rigorous testing process was then applied once the implementation was complete, and included testing of both the user interface and the system back end, as well as testing the system as a whole.

Once the testing processes had been completed, and any issues addressed, an evaluation process was undertaken to determine whether the system was useful in practice, and to gauge what benefits it provided the user, whether it succeeded in meeting the aims of the project, and how it might be improved in the future. The evaluation process involved carrying out observed trials with a number of potential users of the system, as well as a questionnaire completed by participants after having used the system. This section gives details of the different testing stages and the evaluation process.

10.1 Continuous Integration Testing

The modular design of the software made the continuous integration testing process relatively simple. After a software component or a specific functionality was

implemented, it was tested informally by supplying relevant data or control inputs and verifying the correct result. The existing modules and functionality were then also re-tested. This helped to check that the newly implemented functionality had not created an issue elsewhere. Where possible each module was also tested in isolation, prior to being connected to any related modules, to avoid issues becoming compounded. For example the *MachineVision* class was tested alone, by verifying that it could retrieve a single image from the camera and track the robots in said image, before it was connected to the higher level *CameraController* component. This process was completed in a relatively informal manner, as the main focus at the time was implementation, however it was still a useful technique and helped to ensure that bugs were caught early if possible. For a larger, more complex application, being developed by a team rather than an individual, integration testing should be applied in a more formal manner. For this application however it was deemed more important to focus the relatively limited time on the actual implementation.

10.2 Manual User Interface Testing

The purpose of any graphical user interface is to present information to a human user and collect their input. It is therefore important that user interfaces be tested manually by a human user, as automated testing methods are often not sufficient for or not capable of verifying that information is displayed legibly and correctly, and that user input functions properly. The user interface of the system is one of the most important parts of the project, and therefore a thorough manual user interface testing process was undertaken.

10.2.1 Method

The general approach taken to the user interface testing can be summarised as follows:

1. Separate UI into individual elements.
2. State the purpose and required functionality of each elements.
3. Define a general test strategy for a single UI element as a series of checks.
4. Identify any special case components, and define different test strategies where necessary.
5. Apply the relevant strategy to each interface element in turn.

The user interface was separated into the elements described in table 10.1. Special case elements requiring different test strategies are highlighted in bold.

TABLE 10.1: Individual user interface elements that require testing.

Element	Test Strategy
Visualiser Panel Tab System	A
Visualiser	**B**
Visualiser Settings Tab	A
Camera Settings Tab	A
Robot List Panel Tab System	A
Robot List Element	A
Network Settings Tab	A
Logging Settings Tab	A
Data Panel Tab System	A
Console Data Tab	A
Overview Data Tab	A
State Data Tab	A
IR Data Tab	A
Custom Data Tab	A
Individual Visulisation Settings Dialogs	A

The following test strategies were then devised for the different element categories.

Test Strategy A - Standard UI Elements:

1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose.
2. Examine all text within the element. Check for errors in both meaning and spelling.
3. Verify that all components within the element which perform actions in response to user input operate correctly.
4. Verify that all components respond quickly to user input.
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc).
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled.
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible.
8. Verify that the element updates promptly when responding to changes in data.

Test Strategy B - Visualiser:

1. Verify that the video image is displayed correctly.
2. Verify that user clicks within the visualiser space are located correctly, at a number of different window sizes.
3. Verify that robots can be selected by clicking on their location in the visualiser image.
4. For each data visualisation type:
   (a) Define a set of input data and the expected representation of this data.
   (b) Supply the input data.
   (c) Verify the representation is as expected.
   (d) Check that the visualisation is clear and any text is legible.
   (e) Repeat for multiple sets of input data.
   (f) Verify that integrity is maintained with extreme data, corner cases and zero data, wherever possible.
   (g) Verify that integrity is maintained at a range of window sizes, within reasonable limits.

10.2.2 Results

For each user interface element the appropriate test strategy, as specified in table 10.1, was carried out. The results of this testing can be found in appendix A.1. The testing highlighted no major problems with functionality, but did identify a number of smaller issues, mostly related to aesthetics and usability.

10.2.3 Fixes Implemented

The following fixes were implemented to address the issues identified during the manual user interface testing:

- The settings tabs in the visualiser panel were renamed to to better reflect their purpose.
- Long directory paths in the logging tab were truncated to only display the final 25 characters.
- The ordering of messages in the console were reversed, and now read from top to bottom in order.

- The redundant heading was removed from the overview tab.
- Entries in the state transition list were reformatted to clearly separate the time-stamp from the states.
- The IR data tab was reorganised to be more space efficient, and display data more clearly. Headings and labelled were added to further improve clarity.
- Code was added to flip the video image, a setting was added to the camera settings tab to enable and disable this feature.
- Robot IDs rendered in the visualiser were positioned slightly closer to the robots postiion.
- The IR data visualisation in proximity mode was adjusted to have a shorter maximum line length, and to use a non-linear mapping to improve the proximity approximation.
- The IR data visualisation in heat mode was adjusted to use white as the base colour for clarity, changing to red and increasing in size with increasing sensor value.
- An upper limit was added to the robot path sampling interval.

10.3 Data Model and Back End Unit Testing

The next large code component requiring testing was the data model, which formed the majority of the application back-end code. Testing the data model manually, by inputting data packets and verifying the correct insertion of data into the model, was deemed to be too time consuming. This manual approach also posed another problem, in that the contents of the data model could only be examined through the user interface, which meant that the data model testing would be inherently coupled to the UI. This would make it harder to determine the source of any bugs found, as they might be related to the data model or the UI. Avoiding this kind of interdependency is part of the reason for adopting an object oriented approach to the software design and implementation. In order to avoid this issue, the back-end was tested using an automated, 'unit-testing' based approach.

Unit testing is a commonly used technique in professional software testing, and involves writing specific pieces of code which test individual 'units' of code. These test cases will manipulate the unit in some way, using the data and functions it exposes. Then the test case will assert a fact that should be true after the manipulation, such as a comparison of a data point within the unit and the value it should have following the given operations. Each test might include many of these assertion statements, and the test only passes provided that all assertions are determined to

be true. In order to apply this technique to this system an extra class - '*TestingWindow, testingwindow.cpp / .h*' - was added to the application. Alongside this class a number of individual test case functions were added. Each of these functions was written to test the data model in a specific way. Table 10.2 lists the test case functions and their purposes.

TABLE 10.2: Test cases used to unit-test the data model.

Test Case	Purpose and Method
Robot Insertion Test	Tests whether data objects for new robots are inserted into the model correctly. Supplies a packet of each possible type, using a new robot ID each time. After each packet asserts that the number of robots stored in the model has increased by one. Supplies another set of packets, this time reusing the existing robot IDs. Asserts after each packet that the number of robots stored in the model has not increased.
Name Data Test	Tests whether data describing the name of a robot, received in watchdog packets, is inserted into the data model correctly. Supplies three watchdog packets, each with a different robot ID and robot name. Asserts that the data model now contains three robots, and that their name data matches the names entered in the packets. Supplies three new watchdog packets for the same set of robot IDs, with different names. Asserts that the data for each of the robots has been updated to include the new name data.
State Data Test	Tests whether data describing the current state of a robot, received through state packets, is inserted into the data model correctly. Supplies three state packets, each with a different robot ID and state. Asserts that the data model now contains three robots, and that their state data matches the states entered via the packets. Supplies three new state packets for the same set of robot IDs, with different states. Asserts that the state data for each of the robots in the model now matches the new states.

Position Data Test	Tests whether packets describing the current position and orientation of a robot are correctly parsed and the data stored correctly in the data model. Supplies three position packets, each with a different robot ID and different position values. Asserts that the data model now contains data for three robots. Asserts that this data matches the values in the packets for x-position, y-position and angle individually. Supplies three more position packets for the same set of robot IDs with new position and angle values. Asserts that the data in the model for each robot has been updated to reflect the new values. Floating point number assertions are done using a tolerance comparison, with a tolerance of 1×10^{-7}.
IR Data Test	Tests whether packets describing a robot's infra red sensor values are correctly parsed and the data correctly stored in the data model. Supplies an IR data packet, with each sensor reading containing a different value. Asserts that the IR data in the model matches each of the values in turn. Supplies another IR data packet with new, unique values. Asserts that the IR data in the model now matches each of the new values. This process is repeated for packets of background IR data type.
Custom Data Test	Tests whether packets describing custom data key value pairs are correctly parsed and the data correctly stored in the data mode. Supplies three custom data packets, each with a different robot ID and a different value, for a single key. Asserts that the data model now contains data for three robots, and that each robot has a custom data entry for the given key. Asserts that the value for each of these entries matches the value supplied in the relevant packet. Supplies three new packets for the same set of robots with a new key and a new value. Asserts that each robot now contains custom data for the second key, and that the values match those supplied in the packets. Supplies three more packets using the original key, with new values. Asserts that the values for the original key for each robot have been updated to match the values in the latest packet set.

Position History Test	Tests whether position data supplied for a robot is correctly sampled and stored in the position history portion of the data model. Sets the position history sampling rate to 2. Supplies twenty position packets, all attributed to the same robot ID. Asserts that the position history now contains ten entries. Asserts that the x and y-positions values for each of these entries match the values in every second packet, in reverse order. Floating point number assertions are done using a tolerance comparison, with a tolerance of 1×10^{-7}.
State History Test	Tests whether state the state transition history data is correctly formed from a sequence of state packets. Supplies five state packets, each attributed to the same robot ID and containing different states. Asserts that the state transition history now contains five entries. Asserts that the entries describe the correct state transitions, as described by the packets, in reverse order.
Bad Data Test	Tests whether badly formed data packets are correctly rejected by the data model. Supplies a number of correctly formed data packets, each with a different robot ID, and asserts that the data model now contains the correct number of robot entries. Supplies a number of invalid and malformed data packets, distributed between the already used robot IDs and several new IDs. Asserts that the number of robots in the model has not changed, and that the data in each of the existing robots has not changed.

The TestingWindow class provides an additional user interface window, which can be opened by selecting the '*Testing Window*' option from the developer menu on the main tool-bar. This window displays a list of the available tests, a text area for displaying test results, and controls for running either a single test or the full set. Each time a test is run the class instantiates a new data model object, performs the operations for the test in question, displaying the steps involved as text in the results window. For each assertion the text describes what is being asserted and states whether the result was true or false. The overall result of each test is then stated at the end, and the data model object is destroyed. This therefore allows any developer working on the system to open this window at any time whilst the application is running and verify that the full set of tests still passes. This can be done whenever changes are made to the back end code, and gives the developer a degree of certainty that the data model is still functioning correctly. New tests can be

easily added by implementing a new test function and adding it as a test case. The interface for this test window is shown in figure .

[TEST WINDOW UI SCREENSHOT]

10.3.1 Results

In its final state following this project the data model was implemented such that all of the stated test cases passed successfully when run. This indicated that the data model was implemented to a satisfactory standard, and that if data was supplied in correctly formatted packets, it would be correctly stored in the model. Other application functionality that relied on the data model could therefore be used and tested with the assumption that the model was operating correctly.

10.3.2 Issues with this Approach

The unit testing approach suffers from a number of issues. First and foremost the results of the tests are only as good as the tests themselves - meaning that any bugs in the code of each test could be misinterpreted as bugs in the software itself. To mitigate this the tests are designed to be as logically simple as possible, and perform the smallest number of operations necessary to achieve the behaviour under test. This minimises the chances of a mistake in the implementation of the test code.

This approach also relies on the developer writing the tests correctly determining and entering the required result for each assertion statement. One example of this issue is in floating point comparison. The accuracy with which a floating point variable can describe a number is inherently limited due to the way a floating point variable is constructed. In almost all cases a floating point number will differ from the exact value it is attempting to represent by some small amount. This can present an issue when performing comparisons between the contents of a floating point variable and an exact value, leading to false negatives. In order to avoid this all floating point comparisons have been done using a threshold, rather than a direct comparison. This technique asserts that when the expected value is subtracted from the variable being tested, the modulus of the result is less than some very small tolerance value, indicating that the value in the variable is within an acceptable range of the expected value. A tolerance value of 1×10^{-7} was used in all test cases, as this was deemed to indicate sufficient accuracy for the data in this system. To give some perspective to this number note that the x-position value of a robot is stored as a floating point value between 0 and 1, representing a portion of a physical distance of approximately two and a half meters. A discrepancy of +/- 1×10^{-7} therefore equates to an error of $2.5\mu m$. Hence this tolerance value indicates more than adequate accuracy.

10.4 Verification and Validation Testing

Verification and validation (V&V) testing is the process of determining whether the software implemented meets the high level requirements, and whether the individual functionalities operate correctly and without bugs. This was the last step in the testing process, and served to test the system as a whole. The approach taken was to consider the requirements of the system, as stated in the functional specification in section 1.5, and determine whether each requirement is satisfied by the functionality implemented. Where possible each functionality will be tested with relevant data and control inputs. The requirements, and the tests carried out for each are given below.

Core Requirements

C1. Must be comprised of a PC application.

- Verify the software runs on a PC and/or server.
- Verify that the software has the general appearance of a software application.

C2. Must be capable of receiving data related to the state of multiple robots.

- Program a number of robots to send state data through the system, and verify that it is reflected by the application.

C3. Must be capable of receiving positional data for the same set of robots.

- Program the robots to move along a number of different paths in view of the system, and verify that their positioned are correctly tracked by the application.

C4. Must be capable of receiving a live video feed of the robots in their environment.

- Run the application and verify that the video feed is displayed.
- Introduce a number of objects to the environment space and verify that they are seen in the video feed.

C5. Must collate received data and present it to the user in a combined graphical form.

- Program the robots to report a number of different data types, including sensor data, and verify that this can all be displayed by the application simultaneously.

C6. Must present auxiliary, non-spatial data to the user in textual or other forms.

- Program the robots to report name, state and custom data and verify that this is correctly received and displayed in the application.

C7. Must update in approximately real time.

- Program the robots to report data periodically, and verify that the data within the application updates in real time.
- Program the robots to move around in view of the system, and verify that the video shows their movement in approximately real time.

C8. Must at minimum support the e-puck robot platform.

- The previously stated tests should be performed using e-puck robots.

Secondary Requirements

S1. Should use a modularised structure.

- Verifying this through standard testing is not feasible. See section 9.3 for details of the modular implementation.

S2. Should exchange data between the robot platform and the application using a platform-agnostic, extensible protocol.

- Use a platform other than the e-puck robot to transmit packets to the application. Verify that the packets are received and interpreted correctly regardless of the source.

S3. Should provide a basis for interoperability with a number of robotics platforms.

- Verifying this requires another robot platform to be integrated into the system. This is outside the scope of the project. However sections 9.8 and 9.11 indicate how the system has been implemented with portability in mind, and suggest that interoperability should be feasible and relatively easy to achieve.

S4. Should allow the user to configure the displayed data.

- Program the robots to report data of all supported types. Verify that the visualiser settings allow for display configuration.

S5. Should employ a model-view-controller (MVC) software architecture.

- Verifying this through standard testing is not feasible. See section 8.1 for details of the MVC based architecture design.

S6. Could provide the user with ways to configure and display custom data types.

- Program the robots to report a variety of textual and numerical custom data. Verify that this data is received, stored and displayed correctly.
- Verify that the custom data overlay can correctly display any of the received data.

S7. Could allow the user to compare data on two or more specific individual robots.

- Due to time constraints, and a lack of interest in this feature during the initial survey, it was deemed low priority, and ultimately not implemented.

S8. Could calculate and display swarm-level meta-data and statistics.

- Verify that the average position of the tracked robots is correctly displayed.

S9. Could generate log files of robot activity over a user defined period.

- Program the robots to periodically report data of various types. Verify that pressing the start logging button, and then pressing it again (to stop the logging) some time later, results in the generation of a log text file containing entries covering the time period between the presses.
- Verify that setting the log file path directory using the user interface correctly effects where the log file is saved.

10.5 Results

TABLE 10.3: Verification testing results.

Test	**Result**
C1	The application was run successfully on two machines running Linux operating systems. The application has an appearance consistent with the familiarly understood definition of a software application.
C2	Data of all defined types was successfully sent from six robots to the application. The data was correctly displayed within the application.
C3	The positions of six moving robots were correctly tracked simultaneously. The tracking data obtained was correctly displayed in the application both numerically and visually.
C4	The application correctly displays the input video feed. Objects placed in view of the camera are seen in the video feed within the application.

C5	Data received from the robots, including sensor data, was correctly converted to visual representations and displayed in the application.
C6	Data received from the robots was correctly received and displayed in a textual form within the application.
C7	The application updated the data displayed to match new data from the robots in approximately real time. The application updated the position and orientation data to match the movement of the robots in approximately real time.
C8	E-puck robots used for all core tests involving robots. E-puck platform is sufficiently supported.
S2	Utility program used to send specific UDP packets to the application via the network, mimicking the behaviour of an arbitrary robot. The system received and interpreted the packets correctly.
S4	The visualisations of the data reported by six robots could be configured using the options available in the visualiser settings menu, to a moderate extent. Full customisation of all aspects of the visualisations was not possible.
S6	Custom data sent from six robots was received and displayed within a table correctly, updating in approximately real time. The custom data visualisation could be set to target any of the specific data points received. The target data point was then correctly displayed overlaid on the video feed. Configuring the display of custom data beyond a textual representation was not possible. Displaying more than one target data point was not possible.
S8	When enabled, the average position display correctly displayed the average position of the currently tracked robots as a mark within the video feed, when tested with three, four and six robots. This data was not provided in numerical form. Other swarm-level data was not provided.
S9	Was able to correctly start and stop logging whilst receiving data from six robots. The generated text file contained appropriate information. The layout of this information was, however, difficult to parse, making the data hard to understand and use. The directory to store the log files could be correctly set using the UI controls.

All core requirement tests, C1 through C8 passed satisfactorily, indicating that the application satisfies the core requirements. Verifying secondary requirements S1, S3 and S5 was not possible using standard testing approaches. Secondary requirement S7 was dropped following the results of the initial user survey, and a reassessment of the feature set. Secondary requirement test S2 passed satisfactorily, without caveats. Tests S4, S6, S8 and S9 all passed conditionally, as the requirement was either partially met, or met to a limited standard. In order to fully satisfy these requirements the following developments should be made:

- S4: Further develop the visualisation configuration options to allow the user to fully configure all visualisations, including sizes, offset positionings, font sizes, line widths, etc.
- S6: Develop the visualisation of custom data to allow the user a number of graphical options. Allow the user to visualise more than one custom data point simultaneously.
- S8: Add a new tab to the data panel for swarm-level data acquired through analysis. Display the average position values in this tab. Define and implement a number of other swarm-level data analyses, and display this data within the tab, and graphically within the visualiser.
- S9: Edit the log file generation to produce structured, comma-separated-value (CSV) format files.

10.6 Evaluation

10.6.1 Method

10.6.2 Results

10.6.3 Analysis

Chapter 11

Future Work

This project has focused on developing the first version of this swarm robotics debugging system. Responses to the initial user survey, and anecdotal comments from members of the York Robotics Laboratory have indicated that there is an interest in seeing the development of this system continue and its capabilities expanded to a number of new areas. This section discusses potential directions that future development could take, and considers how these might benefit users of the system. Some key areas for potential development are as follows:

- Implementation of additional debugging features.
- Implementation of analytical or experiment-based features, such as macro level analysis and data export.
- Expansion of the target platforms and supported hardware.
- Integration with native augmented reality hardware.

11.1 Additional Debugging Features

It is almost impossible to predict all of the possible types of data, or possible ways of visualising this data that might be desired by users of this system, as each swarm behaviour will have its own unique features and needs, and each different robot platform will have a different set of sensors and actuators. Therefore trying to create specific visualisations that cover all the possibilities is simply infeasible. Instead an approach that might yield much more flexible and useful results could be to incorporate a scripting language, and an API for the visualiser, allowing users to write small simple scripts which provide definitions for custom visualisations. The user could then save these scripts and select them from within the application, and the visualiser would run the scripts with data from the data model to generate the custom visualisations. These scripts could be written in a language such as *Python*, or *LUA*, and then called from within the application using a framework such as '*Boost.Python*'. One downside to this approach would be the requirement for users to

have some knowledge of programming, however considering the target users of the system are robotics developers and researchers, it seems unlikely that they would not be familiar with programming concepts. A well designed API could help to keep the requirements for these visualisation scripts simple, reducing the burden on the user. A visualisation scripting system of this nature would have the added benefit of allowing scripts to be shared, potentially leading to collaborative and faster, more effective discovery of useful ways to visualise robotic data.

11.2 Platform Development

The system could be further developed to become a full swarm robotics platform, used not only for debugging swarm behaviours but also for running swarm experiments and collecting experimental data. A number of additional features would be key to making this step forward. Firstly an organised method for managing experiment 'runs', coupled with a more robust data logging system, could be implemented. This would allow the user to specify when an experiment run was taking place, and record and organise data related to each experiment run. This data could then be used when analysing the results of the experiment. The user could also control which elements of the data should be recorded, to ensure relevance to the experiment being run. The system could also be extended to support bi-directional communication, in order to send simple commands to the robots. This could include commands to start, stop, pause and restart specific behaviours or experiments. When coupled with the improved data logging and experiment organisation this could allow a user to start and stop experiment runs without having to interact directly with each robot, allowing them all to be started simultaneously. This bi-directional communication could also send other types of commands to the robots, potentially experiment-specific, user-defined ones, that might give high level directives to control swarm behaviour.

The addition of higher level analytical features, focusing on the behaviour of the overall swarm rather than the individual robots, could also contribute to the system's use as a swarm robotics platform. These could analyse useful swarm-level parameters such as aggregate position over time, and distributions such as spread and skew. Behaviour significantly different from the average could also potentially be highlighted. This is often the kind of data that is used to judge whether a swarm is behaving correctly. This extra analysis could be coupled with new visualisations, for example giving a visual indicator of a swarms average position, and tracking its changed over time.

The augmented video feed is one of the key features of the application, and could be extended to allow for video to be recorded directly from this feed and exported. This would allow a user to watch back experiment runs and replay specific sections,

perhaps gaining insight into specific interactions or trends within the swarms behaviour. Exported video could also be used for demonstration purposes, with the graphical augmentations adding context for the viewer.

11.3 Expansion of Target Platforms

Wherever possible this system has been implemented with portability in mind, with the aim of allowing different robots to be easily incorporated into the system. The use of ARuCo tags to achieve robot tracking means this portion of the system is fully independent of the robots, requiring no specific hardware, only simple printed paper tags as a minimum. The robot side code has also been designed to be as portable as possible, however it still has two main requirements which are fairly specific; the robots must be running linux, and they must be able to connect to a WiFi network. WiFi connectivity is not always common amongst smaller robot platforms, with many instead utilising Bluetooth to achieve wireless communications. Therefore implementing Bluetooth communication support within the application, alongside WiFi, and adding Bluetooth support to the robot side API could significantly increase the number of robotic platforms on which the system could be used.

Currently the system supports only one specific machine vision camera. A major challenge in the future development of this system will be to find a way to support multiple different camera set ups, with potentially bespoke drivers and APIs, in a way which requires minimal or ideally no changes to the code. One way to achieve this might be to extend the application to be able to receive the video feed via a network, leaving it up to the user to implement a system which obtains the video from their specific camera set up and sends it to the application. This would also allow the application to be run on a machine without a physical connection to the camera hardware. One of the main barriers to this approach, and the reason it was not included in this implementation, is the high bandwidth required to transmit the video feed.

11.4 Integration with Augmented Reality Hardware

In recent years there has been rapid development in augmented reality technology, with the technology beginning to find its way into real world applications. Dedicated hardware platforms such as Microsoft HoloLens, Google Glass, and Google Cardboard have given consumers a first taste of real, perspective based, head mounted, augmented reality. By contrast, the version of augmented reality used in this project is simplistic, and fairly limited. In the future, a combination of a swarm robotics tool such as the system developed in this project, and dedicated AR hardware, could

lead to much more immersive and effective human-robot interaction. The data visualisation could be developed to support full 3D, and utilise the motion tracking abilities of these hardware platforms,and the orientation calculation features built into the ARuCo system, to render the augmentations from any perspective. This report firmly believes that, considering the immense potential of augmented reality systems when coupled with robotics, the use of AR-based tools when working with robots will become commonplace.

11.5 Integration with Tablet Application

Another project being completed at the University of York, simultaneously with this project, has implemented a similar swarm debugging system as an Android app for a tablet computer. The main aim of this approach is to allow the user to be more mobile whilst using the system, so that they can interact with the swarm in a hands on fashion whilst also having immediate access to internal data from the robots. In the future the tablet application could be integrated with the application developed in this project, to form a single platform, with the tablet acting as a satellite terminal through which the user can access the system.

Chapter 12

Conclusion

12.1 Overview

12.2 Evaluation Against Aims and Objectives

12.3 System Limitations

- Requires code to be added to the robot behaviour to report information, and therefore behaviour may change, especially if this code is removed after debugging.

12.4 Use Within the York Robotics Laboratory

One of the key aims of this project was to create a system which is practical, usable, and offers real benefit to swarm robotics developers and researchers. In this capacity it is hoped that the application will continue to be used within the YRL to aid in swarm robotics research and development efforts, and that the application itself will see further development to build upon its core functionality in some of the ways previously mentioned.

12.5 Value Within the Swarm Robotics Space

Appendix A

Test Results

A.1 Manual UI Testing

Visualiser Panel Tab System	
Purpose	Allows access to the different tabs within the visualiser panel.
Required Functionality	Must display the names of each different tab. Must allow the user to click on the tabs and thus change between them. The required tabs are the visualiser tab, the visualiser settings tab and the tracking camera settings tab.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose.	
The tab bar appears to be visually correct. All required tabs are present and visible.	
2. Examine all text within the element. Check for errors in both meaning and spelling.	
The spelling of each tab name is correct. The meaning of each tab name is relatively clear, although 'Settings' is ambiguous, and only related to the visualiser through position and context. Consider renaming this tab to clarify its purpose.	
3. Verify that all components within the element which perform actions in response to user input operate correctly.	
Tab selection operates correctly. Clicking any of the tabs changes the panel to display the contents of that tab. This satisfies the required functionality.	
4. Verify that all components respond quickly to user input.	
The tab changes immediately when clicked.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc).	
Repeatedly and rapidly changing tabs does not have any detrimental affect on the application.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled.	
n/a.	

7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible. The tabs are not affected by window resizing. All of the tabs fit within the minimum size of the panel. If the panel is reduced below this size it is minimized, and the tabs are hidden as intended.
8. Verify that the element updates promptly when responding to changes in data. n/a.

Fixes Required	Consider changing the text on the visualiser settings tab from 'Settings' to something more informative to improve clarity.

Visualiser Settings Tab	
Purpose	Displays the current settings for the visualiser and allows the user to change them.
Required Functionality	Must correctly display the current visualiser settings. Must allow the user to adjust the general visualiser settings, and access dialog windows for changing the settings of specific visualisations.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose. Panel layout appears to be correct. Controls are included to modify all of the required settings.	
2. Examine all text within the element. Check for errors in both meaning and spelling. All spelling is correct, and the meaning of all text is clear.	
3. Verify that all components within the element which perform actions in response to user input operate correctly. All the settings controls work correctly. Double clicking on any of the visualiser config elements in the list displays the appropriate settings dialog, if one exists.	
4. Verify that all components respond quickly to user input. All controls respond immediately to input.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc). Rapidly enabling and disabling settings multiple times has no detrimental affect. Disabling and re-enabling the robot colours setting causes the robots to be re-assigned colours randomly, which might be undesired behaviour.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled. Information regarding the current settings is readable, correctly arranged and clearly labelled. Detailed settings for each visualiser element are described in text next to the element name.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible. All text fits within the minimum panel width, and scroll bars are presented when the height becomes too small to display the full visualiser config element list. Resizing is handled gracefully.	
8. Verify that the element updates promptly when responding to changes in data. Changes to the general settings and the visualiser settings are reflected in the panel immediately.	
Fixes Required	Investigate alternative methods for assigning robot colours.

Camera Settings Tab	
Purpose	Displays the current settings for the tracking camera and allows the user to change them.
Required Functionality	Must correctly display the current camera settings. Must allow the user to adjust the camera settings. Must allow the user to adjust the tracking system settings, and apply/remove mappings between specific tracking tag IDs and robot IDs.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose. Panel layout appears to be correct. Controls are included to modify all of the required settings. A table and associated controls are included to allow the ID mapping to be modified.	
2. Examine all text within the element. Check for errors in both meaning and spelling. All spelling is correct, and the meaning of all text is clear.	
3. Verify that all components within the element which perform actions in response to user input operate correctly. All the settings controls and the mapping table work correctly.	
4. Verify that all components respond quickly to user input. All controls respond immediately to input.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc). Camera resolution input boxes are input validated to only accept numbers in the range 1 to 10,000. Rapid use of controls has no detrimental affect.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled. Current settings are all correctly displayed.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible. All components fit within the panel's minimum dimensions, and are hidden when the panel is minimized.	
8. Verify that the element updates promptly when responding to changes in data. n/a.	
Fixes Required	None.

Robot List Panel Tab System	
Purpose	Allows access to the different tabs within the robot list panel.
Required Functionality	Must display the names of each different tab. Must allow the user to click on the tabs and thus change between them. The required tabs are the robot list tab, the network settings tab and the logging tab.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose.	
The tab bar appears to be visually correct. All required tabs are present and visible.	
2. Examine all text within the element. Check for errors in both meaning and spelling.	
The spelling of each tab name is correct. The meaning of each tab name is clear.	
3. Verify that all components within the element which perform actions in response to user input operate correctly.	
Tab selection operates correctly. Clicking any of the tabs changes the panel to display the contents of that tab. This satisfies the required functionality.	
4. Verify that all components respond quickly to user input.	
The tab changes immediately when clicked.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc).	
Repeatedly and rapidly changing tabs does not have any detrimental affect on the application.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled.	
n/a.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible.	
When the panel width is reduced such that the tabs do not fit, small scroll arrows are added to allow the user to scroll along the tabs, therefore remaining accessible even when the window or panel size is reduced.	
8. Verify that the element updates promptly when responding to changes in data.	
n/a.	
Fixes Required	None.

Robot List	
Purpose	Displays a list of all the robots for which the system has data, allowing the user to select them.
Required Functionality	Must display a list of all the known robots, identifying them by both ID and name. Must allow the user to select any of the robots, causing the other parts of the interface to target the newly selected robot.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose. The list displays correctly. Contains all know robots. Clearly shows current selection.	
2. Examine all text within the element. Check for errors in both meaning and spelling. n/a.	
3. Verify that all components within the element which perform actions in response to user input operate correctly. Robots can be selected correctly by clicking, and changing the selection causes the rest of the application to update to the new focus.	
4. Verify that all components respond quickly to user input. The application responds to a new selection immediately.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc). Rapidly changing the selected robot has no detrimental effect on the application.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled. For each robot the correct ID number and name are displayed, as defined in the data model. The the number of robots exceeds the available space in the list a scroll bar is added.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible. The list is unaffected by window resizing, adding a scroll bar when necessary at small sizes.	
8. Verify that the element updates promptly when responding to changes in data. Changes in the data model are reflected immediately, including robots being added, removed and changing name.	
Fixes Required	None.

Network Settings Tab	
Purpose	Allows the user to configure settings related to the network communication with the robots.
Required Functionality	Must display the current network settings. Must allow the user to change the network settings. Must allow the user to start and stop the data thread which listens for packets.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose.	
The panel appears visually correct, with the necessary components visible.	
2. Examine all text within the element. Check for errors in both meaning and spelling.	
All spelling correct and all meanings clear.	
3. Verify that all components within the element which perform actions in response to user input operate correctly.	
The port number can be changed correctly, and is input-validated to reject non-numerical input. The button for starting and stopping the data thread operates correctly.	
4. Verify that all components respond quickly to user input.	
All components respond immediately.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc).	
Rapidly and repeatedly pressing the start/stop listening button appears to have no detrimental effect. The port number entry box rejects numbers below 1, but does not have a maximum value, as some computers support extremely high port numbers.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled.	
n/a.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible.	
Controls remain usable at the full range of interface sizes.	
8. Verify that the element updates promptly when responding to changes in data.	
n/a.	
Fixes Required	None.

Data Logging Tab	
Purpose	Allows the user to configure the data logging functionality.
Required Functionality	Must display the current data logging settings. Must allow the user to change the data logging settings. Must allow the user to start and stop the data logging.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose. The panel appears correct, with all required elements present.	
2. Examine all text within the element. Check for errors in both meaning and spelling. Spellings correct and meanings clear. The log file directory path can be extremely long, and might overrun the available width of the panel. Long paths should be truncated.	
3. Verify that all components within the element which perform actions in response to user input operate correctly. All components operate correctly. The log file path can be set using a file dialog.	
4. Verify that all components respond quickly to user input. All components respond immediately.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc). Rapid use of the start/stop button has no detrimental effect. Invalid file selections are rejected by the dialog.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled. n/a.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible. All controls fit within the minimum size of the panel. See 2. for note on the log file path length.	
8. Verify that the element updates promptly when responding to changes in data. n/a.	
Fixes Required	Add code to handle long directory paths gracefully.

Data Panel Tab System	
Purpose	Allows access to the different tabs within the data panel.
Required Functionality	Must display the names of each different tab. Must allow the user to click on the tabs and thus change between them. The required tabs are: Console, Overview, State, IR data, Custom data.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose.	
The tab bar appears to be visually correct. All required tabs are present and visible.	
2. Examine all text within the element. Check for errors in both meaning and spelling.	
The spelling of each tab name is correct. The meaning of each tab name is clear.	
3. Verify that all components within the element which perform actions in response to user input operate correctly.	
Tab selection operates correctly. Clicking any of the tabs changes the panel to display the contents of that tab. This satisfies the required functionality.	
4. Verify that all components respond quickly to user input.	
The tab changes immediately when clicked.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc).	
Repeatedly and rapidly changing tabs does not have any detrimental affect on the application.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled.	
n/a.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible.	
All tabs fit within the minimum window width.	
8. Verify that the element updates promptly when responding to changes in data.	
n/a.	
Fixes Required	None.

Console Tab	
Purpose	Displays a text based console which reports messages regarding application events and messages from the robots.
Required Functionality	Must contain a text console. Must display messages regarding the application and messages from the robots. Must display messages in order. Must identify the source of all robot messages. Must update immediately when a message is received from either source.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose. The console is visually correct, however the most recent message is displayed on the top line, counter-intuitively.	
2. Examine all text within the element. Check for errors in both meaning and spelling. The text within the element comes from the messages, hence cannot be checked here.	
3. Verify that all components within the element which perform actions in response to user input operate correctly. Messages are presented correctly and in order. Robot messages are identified correctly.	
4. Verify that all components respond quickly to user input. n/a.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc). Long messages and large numbers of messages are handled gracefully, and remain readable using the automatic scroll bars.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled. All messages are readable and clearly labelled.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible. Resizing teh window has no detrimental effect on the console.	
8. Verify that the element updates promptly when responding to changes in data. New messages appear immediately.	
Fixes Required	Adjust the ordering so that the most recent message appears on the bottom line, as is the standard for text consoles.

Overview Tab	
Purpose	Displays a summary of the data related to the selected robot.
Required Functionality	Must display data related to the selected robot, including ID, name, state and position/orientation values. Must update immediately when new data is received.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose.	
The tab appears visually correct. The heading text appears redundant as the tab itself already contains the heading.	
2. Examine all text within the element. Check for errors in both meaning and spelling.	
All spelling correct and all meanings clear.	
3. Verify that all components within the element which perform actions in response to user input operate correctly.	
n/a.	
4. Verify that all components respond quickly to user input.	
n/a.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc).	
n/a.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled.	
All of the data points are visible, readable, clear, correctly arranged and labelled.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible.	
All data is visible at the full range of window size.	
8. Verify that the element updates promptly when responding to changes in data.	
Updates to the data are reflected immediately.	
Fixes Required	Remove the overview heading from inside the tab, as it is redundant.

State Tab	
Purpose	Displays information related to the internal state of the robot.
Required Functionality	Must display a list of the selected robot's known states. Must display a list of the selected robots recent state changes, including timing information. Must update immediately when a state change occurs.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose.	
Appears visually correct. Both lists are present.	
2. Examine all text within the element. Check for errors in both meaning and spelling.	
All spelling correct and meanings clear.	
3. Verify that all components within the element which perform actions in response to user input operate correctly.	
Lists operate correctly and ignore selection events.	
4. Verify that all components respond quickly to user input.	
n/a.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc).	
Large numbers of states and transition entries remain readable using scroll bars.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled.	
Known states are readable and clear. Transitions are readable but no always clear, due to the layout of the time-stamp. The clarity could be improved with better formatting.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible.	
resizing has no detrimental effect on the tab. Scroll bars are available when necessary.	
8. Verify that the element updates promptly when responding to changes in data.	
New states and transitions are displayed immediately.	
Fixes Required	Re-format the state transition list entries to improve clarity.

IR Data Tab	
Purpose	Displays the IR sensor data for the selected robot.
Required Functionality	Must provide a visual display of the selected robot's IR sensor values. Must provide a numerical display of the robot's IR sensor values. Must support both active and background values. Must update immediately when new values arrive.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose. The element is lacking clear headings for some of the data points. The bar graph is visually correct. The numerical displays are correct but their current format takes up a lot of space.	
2. Examine all text within the element. Check for errors in both meaning and spelling. All spelling correct. Some headings have unclear meanings.	
3. Verify that all components within the element which perform actions in response to user input operate correctly. n/a.	
4. Verify that all components respond quickly to user input. n/a.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc). Sensor values that are out of range are not displayed.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled. All data is readable and correctly arranged. The IR data is not clearly labelled.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible. Some of the data becomes hidden if the window width is reduced.	
8. Verify that the element updates promptly when responding to changes in data. New data is reflected immediately.	
Fixes Required	Add clear headings and reformat the numerical data to improve clarity. Rearrange the layout so that the bar graph fits within the minimum window width.

Custom Data Tab	
Purpose	Displays custom data reported by the selected robot.
Required Functionality	Must display each custom data point in, including both key and value strings. Must update immediately when new data is received.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose. The tab is visually correct, including a table to display the custom data key/value pairs.	
2. Examine all text within the element. Check for errors in both meaning and spelling. All spelling correct and meanings clear.	
3. Verify that all components within the element which perform actions in response to user input operate correctly. The table cannot be modified using the mouse or keyboard, as intended.	
4. Verify that all components respond quickly to user input. n/a.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc). Long keys and values and large numbers of key/value pairs are handled gracefully with scroll bars.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled. All data is displayed clearly. The columns of the table are clearly labelled.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible. The table scales to fit the available space, providing scroll bars whenever necessary.	
8. Verify that the element updates promptly when responding to changes in data. New and updated custom data is displayed immediately.	
Fixes Required	None.

Individual Visualisation Settings Dialogs	
Purpose	Allow the user to change settings for the data visualisations.
Required Functionality	Must display the settings for the selected visualisation type in a pop-up, modal window. Must allow the user to change these settings. Must provide the user with options for applying or cancelling the changes.
1. Examine UI element visually. Verify that it appears correct. Verify that it contains all elements necessary to satisfy its purpose.	
All settings dialogs appear correctly, containing all the necessary controls to adjust the available settings.	
2. Examine all text within the element. Check for errors in both meaning and spelling.	
All spelling correct and all meanings clear.	
3. Verify that all components within the element which perform actions in response to user input operate correctly.	
All settings controls working correctly.	
4. Verify that all components respond quickly to user input.	
Settings changes take effect immediately after pressing the apply button.	
5. Verify that component actions and functionality do not degrade with extreme use (sustained rapid input, large numbers of input changes, etc).	
All components appear robust to repeated, rapid use. Input fields are validated to reject out of range values.	
6. Verify that all data displayed within the element is visible, readable, correctly arranged and correctly labelled.	
All settings are clearly displayed, labelled and arranged.	
7. Verify that the element behaves sensibly when window resizing occurs, and that it remains usable and data remains visible whenever possible.	
Dialog windows cannot be resized.	
8. Verify that the element updates promptly when responding to changes in data.	
n/a.	
Fixes Required	None.

General Visualiser Functionality	
Purpose	Displays the live video feed and overlays the the data visualisations.
Required Functionality	Must display the video feed. Must render the data visualisations based on the current data. Must allow the user to select a robot by clicking on it within the image.
1. Verify that the video image is displayed correctly.	
The video feed image is displayed correctly, and updates at a decent rate. Due to the mounting orientation of the camera the image appears reversed when compared to the real world view. Settings could be added to allow the user to flip the image.	
2. Verify that user clicks within the visualiser space are located correctly, at a number of different window sizes.	
User clicks are correctly located as shown by the cross-hairs showing the latest click location. Tested for a number of different sizes and image dimensions / aspect ratios, and worked correctly each time.	
3. Verify that robots can be selected by clicking on their location in the visualiser image.	
Robots can be selected by clicking, with a reasonable tolerance. Robots positioned directly adjacent to one another can still be selected correctly. Clicking a robot with ID 10 caused a robot with ID 1 to be selected erroneously.	
Fixes Required	Add setting to allow the user to flip the video image. Robot selection works incorrectly for robots with IDs beginning with the same digit; determine the cause and fix.

Robot ID Visualisation	
Purpose	Overlay the numerical ID of the robot on the video feed.
Required Functionality	Display the numerical ID as a text string adjacent to the related robot.
Settings	On / off (Toggle). Display for selected robot only or all robots (Toggle).
1. Define input data and expected representation.	
Five active robots using ID's 0, 1, 5, 7 and 8. expect to see the ID numbers rendered to the left of each robot, slightly above their center.	
3. Verify the representation is as expected.	
Numerical ID numbers appear next to each robot. The ID is positioned slightly too far to the left of each robot, sometimes overlapping with other visualisations in crowded areas. Settings apply correctly.	
4. Check that the visualisation is clear and any text is legible.	
ID numbers are legible.	
5. Repeat for multiple sets of input data.	
Tested with five robots.	

6. Verify that integrity is maintained with extreme data, corner cases and zero data, wherever possible. Robot IDs higher than 999 result in long strings, which overlap with the robot itself and its position/direction visualisation. Robot swarm of this size are not anticipated however.
7. Verify that integrity is maintained at a range of window sizes, within reasonable limits. IDs display at the same size for all visualiser sizes, therefore remaining legible but occupying excessive space at small sizes. However the visualiser is not usable at such small sizes, so this is not anticipated to cause issues.
Fixes Required Reposition IDs slightly closer to the robot. Potentially allow the user to configure the positioning.

Robot Name Visualisation	
Purpose	Overlay the name of the robot on the video feed.
Required Functionality	Display the name of the robot as a text string adjacent to the related robot.
Settings	On / off (Toggle). Display for selected robot only or all robots (Toggle).
1. Define input data and expected representation. Five active robots reporting the names Robot_0, Robot_1, Robot_5, Robot_7 and Robot_8 in watchdog packets. Expect to see the names rendered to the right of each robot, slightly above their center.	
3. Verify the representation is as expected. Names appear next to each robot. The text is positioned correctly to the right of each robot. Settings apply correctly.	
4. Check that the visualisation is clear and any text is legible. Names are legible.	
5. Repeat for multiple sets of input data. Tested with five robots.	
6. Verify that integrity is maintained with extreme data, corner cases and zero data, wherever possible. Extremely long names are displayed up to the edge of the image.	
7. Verify that integrity is maintained at a range of window sizes, within reasonable limits. Names remain legible for all usable sizes of the visualiser.	
Fixes Required	None.

Robot State Visualisation	
Purpose	Overlay the current state of the robot on the video feed.
Required Functionality	Display the state of the robot as a text string adjacent to the related robot. Update this display whenever the state changes.
Settings	On / off (Toggle). Display for selected robot only or all robots (Toggle).
1. Define input data and expected representation. Five active robots oscillating between STATE1 and STATE2. Expect to see the states rendered to the right of each robot, below the name visualisation.	
3. Verify the representation is as expected. States appear next to each robot. The text is positioned correctly to the right of each robot and below the name text. Settings apply correctly.	
4. Check that the visualisation is clear and any text is legible. States are legible.	
5. Repeat for multiple sets of input data. Tested with five robots, each changing state at different times.	
6. Verify that integrity is maintained with extreme data, corner cases and zero data, wherever possible. Extremely long states are displayed up to the edge of the image.	
7. Verify that integrity is maintained at a range of window sizes, within reasonable limits. States remain legible for all usable sizes of the visualiser.	
Fixes Required	None.

Robot Position Visualisation	
Purpose	Overlay a small circle around the robot's current position.
Required Functionality	Render a circle outline around the robots current position. Use a thicker line to highlight the robot if it is currently selected.
Settings	On / off (Toggle).
1. Define input data and expected representation. Five active robots moving around the arena in different paths. Expect to see a circle rendered around the center of each robot, updating as their positions change.	
3. Verify the representation is as expected. Circles are correctly rendered for all robots, and update correctly over time. Can be correctly toggled on and off.	
4. Check that the visualisation is clear and any text is legible. Circles are clear.	
5. Repeat for multiple sets of input data. Tested with five robots.	
6. Verify that integrity is maintained with extreme data, corner cases and zero data, wherever possible. Circles display correctly for all positions within the image.	
7. Verify that integrity is maintained at a range of window sizes, within reasonable limits. Circles render correctly at all sizes.	
Fixes Required	None.

	Robot Direction Visualisation
Purpose	Overlay a small line indicating the direction the robot is facing.
Required Functionality	Render a line from the center of the robot outwards, in the direction it is facing. Use a thicker line if the robot is selected.
Settings	On / off (Toggle).
1. Define input data and expected representation. Five active robots moving around the arena in different paths. Expect to see a line rendered from the center of each robot outward in the direction it is facing, updating as its orientation changes.	
3. Verify the representation is as expected. Lines are correctly rendered for all robots, and update correctly over time. Can be correctly toggled on and off.	
4. Check that the visualisation is clear and any text is legible. Lines are clear.	
5. Repeat for multiple sets of input data. Tested with five robots.	
6. Verify that integrity is maintained with extreme data, corner cases and zero data, wherever possible. Lines display correctly for all orientations, at all positions within the image.	
7. Verify that integrity is maintained at a range of window sizes, within reasonable limits. Lines render correctly at all sizes.	
Fixes Required	None.

IR Data Visualisation	
Purpose	Overlay a graphical representation of the robots infra-red sensor data.
Required Functionality	IR sensor data represented in one of two modes. In proximity mode, display a line in the direction of each relevant sensor with a length inversely related to the sensors value, indicating an approximation of proximity. In 'heat' mode, display a small box for each sensor adjacent to the robot and positioned to match the sensor layout, that changes colour as the sensor value changes. Uses the position and orientation of the robot to render with the correct position and rotation.
Settings	On / off (Toggle). Display for selected robot only or all robots (Toggle). Proximity or heat mode (Toggle). Angle for each sensor in degrees (Numerical).
1. Define input data and expected representation. Five active robots reporting their IR sensor data whilst an object is placed at varying distances from each of their sensors. Expect to see the graphical representations change to reflect the changing value.	
3. Verify the representation is as expected. IR data is displayed in both modes. The display varies correctly as the sensor values change. Proximity lines change length correctly, but extend much further than necessary for low values. The boxes in heat mode change colour correctly, but are coloured close to black for low values, which does not display well on a dark background.	
4. Check that the visualisation is clear and any text is legible. Both visualisations are clear.	
5. Repeat for multiple sets of input data. Tested with 5 robots across a range of sensor values.	
6. Verify that integrity is maintained with extreme data, corner cases and zero data, wherever possible. Out of range sensor data is not displayed.	
7. Verify that integrity is maintained at a range of window sizes, within reasonable limits. The size of the visualisation does not change with the size of the image. However the proximity lines can only be representative, so their actual size is not important, only the relative variation in that size.	
Fixes Required	Set a more sensible maximum length for the proximity lines. Improve mapping between sensor values and line length (non-linear). Adjust heat mode colour scheme for clarity.

Robot Path Visualisation	
Purpose	Display the robots recent movement in the form of a trail.
Required Functionality	Render the robot's position history as a sequence of line segments. support an adjustable sampling interval.
Settings	On / off (Toggle). Display for selected robot only or all robots (Toggle). Sampling interval (Numerical).
1. Define input data and expected representation. Five active robots moving around the arena along different paths. Expect to see a trail line behind each robot showing the path it has taken.	
3. Verify the representation is as expected. Trails are correctly drawn for all robots, accurate to their approximate path. Varying the sample interval allows for longer, lower resolution and shorter, higher resolution paths. Can be correctly toggled on and off, and set to only display for the selected robot.	
4. Check that the visualisation is clear and any text is legible. Trails are clearly drawn.	
5. Repeat for multiple sets of input data. Tested with 5 robots moving along a number of different paths.	
6. Verify that integrity is maintained with extreme data, corner cases and zero data, wherever possible. Setting an excessively large sampling interval leads to a very long, jagged path, as expected.	
7. Verify that integrity is maintained at a range of window sizes, within reasonable limits. Paths remain accurate for all visualiser sizes, as path point positions are stored as proportional coordinates.	
Fixes Required	Add an upper limit to the sampling interval setting.

Custom Data Visualisation	
Purpose	Display a specific element of the robot's custom data as text.
Required Functionality	Must display the key and the current value for the target data point as text adjacent to the robot, below the state text. Must update whenever the value for that key changes. The user must be able to set the target key.
Settings	On / off (Toggle). Display for selected robot only or all robots (Toggle). Set the target data point (Text input).
1. Define input data and expected representation. Five robots, each reporting custom data for two keys, with the values varying over time. Expect to see the target data key and value displayed as text below the state text.	
3. Verify the representation is as expected. Custom data for the target key is correctly displayed, and updates immediately when new data arrives. The target key can be changed and the visualisation updates to reflect this. Can be toggled on and off correctly.	
4. Check that the visualisation is clear and any text is legible. Text is clear and legible.	
5. Repeat for multiple sets of input data. Tested with 5 robots reporting data for two different keys.	
6. Verify that integrity is maintained with extreme data, corner cases and zero data, wherever possible. If no key is set, no data is displayed. Very long data values are displayed up to the edge of the image. If no data exists for the target key and the selected robot no data is displayed.	
7. Verify that integrity is maintained at a range of window sizes, within reasonable limits. Remains visible at all usable visualiser sizes.	
Fixes Required	None.

Bibliography

[1] M. Dorigo, M. Birattari, and M. Brambilla. "Swarm robotics". In: *Scholarpedia* 9.1 (2014), p. 1463.

[2] F. Mondada et al. "The e-puck, a Robot Designed for Education in Engineering". In: *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions* 1.1 (2009), pp. 59–65.

[3] Erol Sahin. "Swarm Robotics: From Sources of Inspiration to Domains of Application". In: *Swarm Robotics WS 2004*. Ed. by E. Sahin amd W. M. Spears. Berlin: Springer, 2005, pp. 10–20.

[4] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm intelligence: From natural to artificial systems*. Oxford University Press, 1999. ISBN: 0-19-513159-2.

[5] Manuele Brambilla et al. "Swarm Robotics: a Review from the Swarm Engineering Perspective". In: *Swarm Intelligence*. Springer, 2013.

[6] A. Kolling et al. "Human Interaction With Robot Swarms: A Survey". In: *IEEE Transactions on Human-Machine Systems* 46.1 (2016), pp. 9–26.

[7] A. Rule and J. Forlizzi. "Designing interfaces for multi-user, multi-robot systems". In: *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction* (2012), pp. 97–104.

[8] R. E. Christ. "Research for evaluating visual display codes: An emphasis on colour coding". In: *Information design: The design and evaluation of signs and printed materials* (1984), pp. 209–228.

[9] C. Carney and J. L. Campbell. *In vehicle display icons and other information elements: Literature review*. Tech. rep. U.S. Department of Transportation, Federal Highway Administration, 1998, pp. 209–228.

[10] Gaëtan Podevijn, Rehan O'Grady, and Marco Dorigo. "Self-organised feedback in human swarm interaction". In: *Proceedings of the workshop on robot feedback in human-robot interaction: how to make a robot readable for a human interaction partner (Ro-Man 2012)*. 2012.

[11] James McLurkin et al. "Speaking Swarmish: Human-Robot Interface Design for Large Swarms of Autonomous Mobile Robots." In: *AAAI spring symposium: to boldly go where no human-robot team has gone before*. 2006, pp. 72–75.

[12] T. H. J. Collet and A. MacDonald. "An augmented reality Debugging system for mobile robot software engineers". In: *Proceedings 2006 IEEE International Conference on Robotics and Automation* (2006), pp. 3954–3959.

[13] Luke Gumbley and Bruce A. MacDonald. "Realtime Debugging for Robotics Software". In: *Australasian Conference on Robotics and Automation (ACRA)* (2009).

[14] Ronald T. Azuma. "A Survey of Augmented Reality". In: *Foundations and Trends nin Human-Computer Interaction* 8.2-3 (1997), pp. 73–272.

[15] Gurmeet Singh Pandher. *Microsoft HoloLens Preorders: Price, Specs Of The Augmented Reality Headset*. https://www.thebitbag.com/microsoft-hololens-preorders-price-specs-of-the-augmented-reality-headset/137410. Accessed: 2017-05-03.

[16] Mark Billinghurst Adrian Clark and Gun Lee. "A Survey of Augmented Reality". In: *Presence: Teleoperators and Virtual Environments* 6.4 (2014), pp. 355–385.

[17] P. Milgram, D. Zhai S. Drascic, and J. Grodski. "Applications of augmented reality for human-robot communication". In: *Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems '93, IROS '93.* 3 (1993), pp. 1467–1472.

[18] F. Ghirighelli et al. "Interactive Augmented Reality for understanding and analyzing multi-robot systems". In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2014), pp. 1195–1201.

[19] S. Garrido-Jurado et al. "Automatic generation and detection of highly reliable fiducial markers under occlusion". In: *Pattern Recognition* 47.6 (2014), pp. 2280–2292.

[20] Mike Daily et al. "World embedded interfaces for human-robot interaction". In: *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*. 2003.

[21] W. Liu and A. F. T Winfield. "Open-hardware e-puck Linux extension board for experimental swarm robotics research". In: *Microprocessors and Microsystems* 35.1 (2011), pp. 60–67.

[22] Stemmer Imaging. *JAI Go Machine Vision Camera*. 2014. URL: https://www.stemmer-imaging.co.uk/en/products/series/jai-go/ (visited on 04/29/2017).

[23] Stemmer Imaging. *GIGE Vision In Practice*. 2011.

[24] Deborah J. Armstrong. "The Quarks of Object-oriented Development". In: *Communications of the ACM - Next-generation cyber forensics* 49.2 (Feb. 2006), pp. 123–128.